EOxServer Documentation

Release 0.3.2

Stephan Meissl **Stephan Krause**
Fabian Schindler **Gerhard Triebnig**
Milan Novacek **Arndt Bonitz** **Martin Paces**
Joachim Ungar **Marko Locher** **Christian Schiller**

January 31, 2014

CONTENTS

1 EOxServer Users' Guide **1**
- 1.1 EOxServer Basics . . . 1
- 1.2 Global Use Case . . . 4
- 1.3 Installation . . . 14
- 1.4 Installation on CentOS . . . 17
- 1.5 Service Instance Creation and Configuration . . . 20
- 1.6 Recommendations for Operational Installation . . . 26
- 1.7 Migration . . . 33
- 1.8 Mailing Lists . . . 36
- 1.9 Demonstration . . . 37
- 1.10 EO-WCS Request Parameters . . . 42
- 1.11 EOxServer Operators' Guide . . . 46
- 1.12 The Webclient Interface . . . 62
- 1.13 Identity Management System . . . 69
- 1.14 SOAP Proxy . . . 92
- 1.15 EOxServer Presentations . . . 96
- 1.16 Configuration Options . . . 98
- 1.17 Supported CRSs and Their Configuration . . . 102
- 1.18 Supported Raster File Formats and Their Configuration . . . 103
- 1.19 Asynchronous Task Processing . . . 105
- 1.20 Web Coverage Service - Transaction Extension . . . 108

2 EOxServer Developers' Guide **113**
- 2.1 Basics . . . 113
- 2.2 Core . . . 114
- 2.3 Data Model . . . 114
- 2.4 Plugins . . . 118
- 2.5 Services . . . 118
- 2.6 Data Formats . . . 118
- 2.7 Metadata Formats . . . 118
- 2.8 The *autotest* instance . . . 118
- 2.9 SOAP Proxy . . . 122
- 2.10 Handling Coverages . . . 124
- 2.11 Asynchronous Task Processing - Developers Guide . . . 128
- 2.12 Modules . . . 131
- 2.13 Testing . . . 243

3 EOxServer Requests for Comments **245**
- 3.1 RFC Procedures . . . 245
- 3.2 Writing RFCs . . . 245
- 3.3 RFCs . . . 245

4 License **333**

4.1 EOxServer Open License 333
4.2 EOxServer-Soap Proxy Open License 333

5 Credits **335**

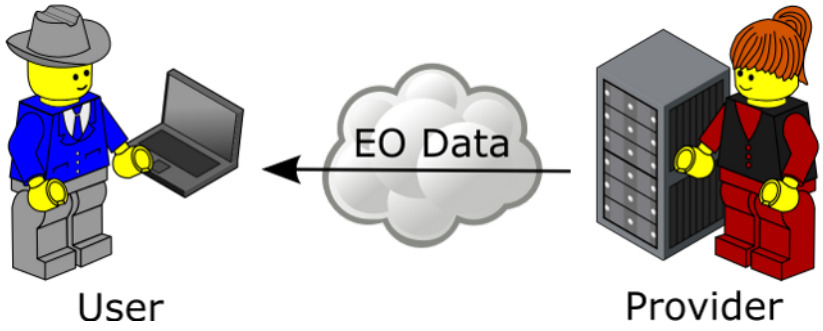
Index **337**

CHAPTER

ONE

EOXSERVER USERS' GUIDE

This section is intended for users of the EOxServer software stack. Users range from administrators installing and configuring the software stack and operators registering the available *EO Data* on the *Provider* side to end users consuming the registered *EO Data* on the *User* side.



Developers needing to know all the nitty-gritty about EOxServer implementation and APIs please refer to the *EOxServer Developers' Guide* (page 113).

1.1 EOxServer Basics

Table of Contents

- EOxServer Basics (page 1)
  - Introduction (page 2)
    - What is EOxServer? (page 2)
    - What are the main features of EOxServer? (page 2)
    - Where can I get it? (page 2)
    - Where can I get support? (page 2)
    - EOxServer Documentation (page 3)
    - Demonstration Services (page 3)
  - Data Model (page 3)
  - Service Model (page 3)
    - Web Coverage Service (page 4)
    - Web Map Service (page 4)

1.1.1 Introduction

What is EOxServer?

EOxServer is an open source software for registering, processing, and publishing Earth Observation (EO) data via different Web Services. EOxServer is written in Python and relies on widely-used libraries for geospatial data manipulation.

The core concept of the EOxServer data model is the one of a coverage. In this context, a coverage is a mapping from a domain set (a geographic region of the Earth described by its coordinates) to a range set. For original EO data, the range set usually consists of measurements of some physical quantity (e.g. radiation for optical instruments).

The EOxServer service model is designed to deliver (representations of) EO data using open standard web service interfaces as specified by the Open Geospatial Consortium[1] (OGC).

What are the main features of EOxServer?

- Repository for Earth Observation data
- OGC Web Services
- Administration Tools
- Web Client
- Identity Management System

Where can I get it?

You can get the EOxServer source from

- the EOxServer Download page[2]
- the Python Package Index (PyPi)[3]
- the EOxServer SVN repository[4]

Additionally the following binary packages are provided:

- Enterprise Linux RPMs from EOX' YUM repository[5]

The recommended way to install EOxServer on your system is to use the Python installer utility pip[6].

Please refer to the *Installation* (page 14) document for further information on installing the software.

Where can I get support?

If you have questions or problems, you can get support at the official EOxServer Users' mailing list users@eoxserver.org[7]. See *Mailing Lists* (page 36) for instructions how to subscribe.

Documentation is available on this site and as a part of the EOxServer source.

[1] http://www.opengeospatial.org
[2] http://eoxserver.org/wiki/Download
[3] http://pypi.python.org/pypi/EOxServer/
[4] http://eoxserver.org/svn/trunk
[5] http://packages.eox.at
[6] http://www.pip-installer.org/en/latest/index.html
[7] users@eoxserver.org

EOxServer Documentation

The EOxServer documentation consists of the

- *EOxServer Users' Guide* (page 1) (which this document is part of)
- *EOxServer Developers' Guide* (page 113) (where you can find implementation details)
- *EOxServer Requests for Comments* (page 245) (where you can find high-level design documentation)

Furthermore, you can consult the inline documentation in the source code e.g. in the Source Browser[8].

Demonstration Services

There is a demonstration service available on the EOxServer site. You can reach it under http://eoxserver.org/demo_stable/ows. For some sample calls to different OGC Web Services, see *Demonstration* (page 37).

1.1.2 Data Model

The EOxServer data model describes which data can be handled by the software and how this is done. This section gives you a short overview about the basic components of the data model.

The term coverage is introduced by the OGC Abstract Specification. There, coverages are defined as a mapping between a domain set that can be referenced to some region of the Earth to a range set which describes the possible values of the coverage. This is, of course, a very abstract definition. It comprises everything that has historically been called "raster data" (and then some, but that is out of scope of EOxServer at the moment).

The data EOxServer originally was designed for is satellite imagery. So the domain set is the extent of the area that was scanned by the respective sensor, and the range set contains its measurements, e.g. the radiation of a spectrum of wavelengths (optical data).

In the language of the OGC Abstract Specification ortho-rectified data corresponds to "rectified grid coverages", whereas data in the original geometry corresponds to "referenceable grid coverages".

The EOxServer coverage model relies heavily on the data model of the Web Coverage Service 2.0 Earth Observation Application Profile which is about to be approved by OGC. This profile introduces different categories of Earth Observation data:

- Rectified or Referenceable Datasets roughly correspond to satellite scenes, either ortho-rectified or in the original geometry
- Rectified Stitched Mosaics are collections of Rectified Datasets that can be combined to form a single coverage
- Dataset Series are more general collections of Datasets; they are just containers for coverages, but not coverages themselves

Datasets, Stitched Mosaics and Dataset Series are accompanyed by Earth Observation metadata. At the moment, EOxServer supports a limited subset of metadata items, such as the identifier of the Earth Observation product, the acquisition time and the acquisistion footprint.

1.1.3 Service Model

Earth Observation data are published by EOxServer using different OGC Web Services. The OGC specifies open standard interfaces for the exchange of geospatial data that shall ensure interoperability and universal access to geodata.

[8] http://eoxserver.org/browser

Web Coverage Service

The OGC Web Coverage Service[9] (WCS) is designed to deliver original coverage data. EOxServer implements three versions of the WCS specification:

- version 1.0.0
- version 1.1.0
- version 2.0.1 including the Earth Observation Application Profile (EO-WCS)

Each of these versions supports three operations:

- GetCapabilities - returns an XML document describing the available coverages (and Dataset Series)
- DescribeCoverage - returns an XML document describing a specific coverage and its metadata
- GetCoverage - returns (a subset of) the coverage data

The WCS 2.0 EO-AP (EO-WCS) adds an additional operation:

- DescribeEOCoverageSet - returns an XML document describing (a subset of) the datasets contained in a Rectified Stitched Mosaic or Dataset Series

For detailed lists of supported parameters for each of the operations see *EO-WCS Request Parameters* (page 42) .

In addition, EOxServer supports the WCS 1.1 Transaction operation which provides an interface to ingest coverages and metadata into an existing server.

Web Map Service

The OGC Web Map Service[10] (WMS) is intended to provide portrayals of geospatial data (maps). In EOxServer, WMS is used for viewing purposes. The service provides RGB or grayscale representations of Earth Observation data. In some cases, the Earth Observation data will be RGB imagery itself, but in most cases the bands of the images correspond to other parts of the wavelength spectrum or other measurements altogether.

EOxServer implements WMS versions 1.0, 1.1 and 1.3 as well as parts of the Earth Observation Application Profile for WMS 1.3. The basic operations are:

- GetCapabilities - returns an XML document describing the available layers
- GetMap - returns a map

For certain WMS 1.3 layers, there is also a third operation available

- GetFeatureInfo - returns information about geospatial features (in our case: datasets) at a certain position on the map

Every coverage (Rectified Dataset, Referenceable Dataset or Rectified Stitched Mosaic) is mapped to a WMS layer. Furthermore, Dataset Series are mapped to WMS layers as well. In WMS 1.3 a "bands" layer is appended for each coverage that allows to select and view a subset of the coverage bands only. Furthermore, queryable "outlines" layers are added for Rectified Stitched Mosaics and Dataset Series which show the footprints of the Datasets they contain.

1.2 Global Use Case

[9] http://www.opengeospatial.org/standards/wcs
[10] http://www.opengeospatial.org/standards/wms

Table of Contents

- Global Use Case (page 4)
  - The General *Provider* View (page 6)
    - Environment & Software Configuration (page 7)
    - Data Registration (page 8)
  - The General *User* View (page 11)
    - Web Browser (page 11)
    - GIS Tool (page 11)

This section describes the global Use Case of EOxServer including concrete usage scenarios as examples.

Figure: "*Parties involved in the EOxServer Global Use Case* (page 5)" introduces the involved parties in this global Use Case.

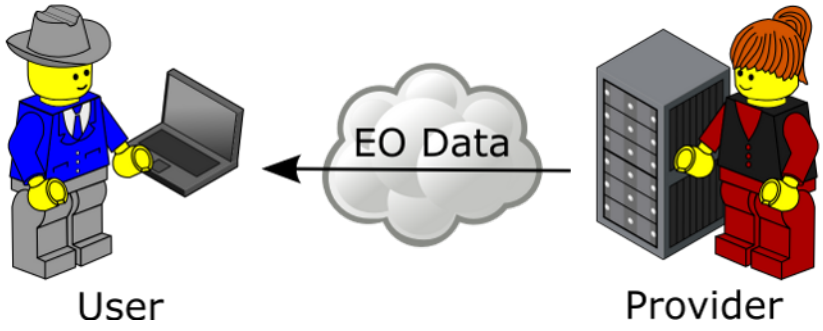


Figure 1.1: *Parties involved in the EOxServer Global Use Case*

On the one side there is a provider of Earth Observation (EO) data. The provider has a possibly huge, in terms of storage size, archive of EO data and wants to provide this data to users. Of course the data provision has to follow certain constraints and requirements like technical, managerial, or security frame conditions but in general the provider wants to reach as many users as possible with minimal efforts.

On the other side there is a user of EO data. The user has the need of certain EO data as input to some processing which varies from simple viewing to complex data analysis and generation of derived data. The user wants to obtain the needed EO data as easily as possible which includes finding the right data from the right provider at the right time at the right location and retrieving it in the right representation e.g. format.

Already from this simple constellation the need for standardized interfaces is evident. Thus EOxServer implements the open publicly available interface standards defined by the Open Geospatial Consortium (OGC). In particular EOxServer contains an implementation of the Web Coverage Service (WCS) including its Earth Observation Application Profile (EO-WCS) and the Web Map Service (WMS) again including its EO extension (EO-WMS).

These interface standards have been chosen to support the new paradigm of "zooming to the data". This means looking at previews of the data rather than searching in a catalogue in order to find the right data. WMS together with its EO extension is used for the previews whereas WCS with its EO extensions is used to download the previously viewed data and metadata. Of course a provider is free to operate a catalogue in parallel including references to the EO-WMS and EO-WCS.

The EOxServer software stack is a collection of Open Source Software designed to enhance a wide range of legacy systems of EO data archives with controlled Web-based access ("online data access") with minimal efforts for the provider. The user not only significantly benefits from the provider's enhanced online data access but also from the client functionalities included in the EOxServer software stack.

In particular the EOxServer software stack provides the following features:

- easy to install
- simple yet powerful web interface for data registration for the provider
- standardized way to access geographic data i.e. via EO-WMS and EO-WCS

- download of subsets of data
- on the fly re-projection, re-sampling, and format conversion
- visual preview of data
- integrated usage of EO-WMS and EO-WCS to view and download the same data
- intelligent automated handling of EO collections and mosaics
- homogeneous way to access different data, metadata, and packaging formats
- homogeneous access to different storage systems i.e. file system, ftp, and rasdaman

These features result in the general benefit for the provider to be more attractive to the user.

The following sub-sections provide details from the provider and user point of view highlighting the possible usage of the EOxServer software stack.

1.2.1 The General *Provider* View

The provider operates an archive of EO data with different ways of actually accessing the data. For simplicity let's assume the data archived in this legacy system can be accessed in two ways. First there is the local access directly to the file system via operating system capabilities. Second there is an online access by exposing certain directories via FTP.

The EOxServer software stack acts as a middle-ware layer in front of the legacy archive system that expands the offered functionality and thus widens the potential user or customer base. The additional functionality compared to plain FTP access includes:

- interoperable online access via a standard interface defined by an accepted international industry consortium
- domain and range sub-setting of coverages allowing to download only the needed parts of a coverage and thus saving bandwidth
- spatial-temporal search within the offered coverages
- on-the-fly mosaicking
- on-the-fly re-projection
- delivery in multiple encoding formats i.e. on-the-fly format conversion
- on-the-fly scaling and re-sampling
- preview via EO-WMS
- embedding of metadata (EO-O&M) adjusted to the actual delivered coverage

Figure: "*Provider View* (page 7)" provides an overview of the provider environment showing the provider's legacy system and the extending EOxServer software stack.

The recommended way for the installation of the EOxServer software stack is to use a host which has direct read access to the data via the file system using operating system capabilities. If this file system is physically located on the same hardware host or if it is mounted from some remote storage e.g. via NFS or Samba doesn't matter in terms of functionality. However, in terms of performance the actual configuration has some impact as big data might have to be transferred over the network with different bandwidths.

The other option is to use the read access via FTP which is a practical configuration in terms of functionality. However, in terms of performance this isn't the recommended configuration because of the need to always transfer whole files even if only a subset is needed. Various caching strategies will significantly improve this configuration, though.

After the installation of all software components needed for the EOxServer software stack there are two main activities left for the provider:

- Configure the environment (e.g. register service endpoint(s) in a web server) and EOxServer (e.g. enable or disable components like services)

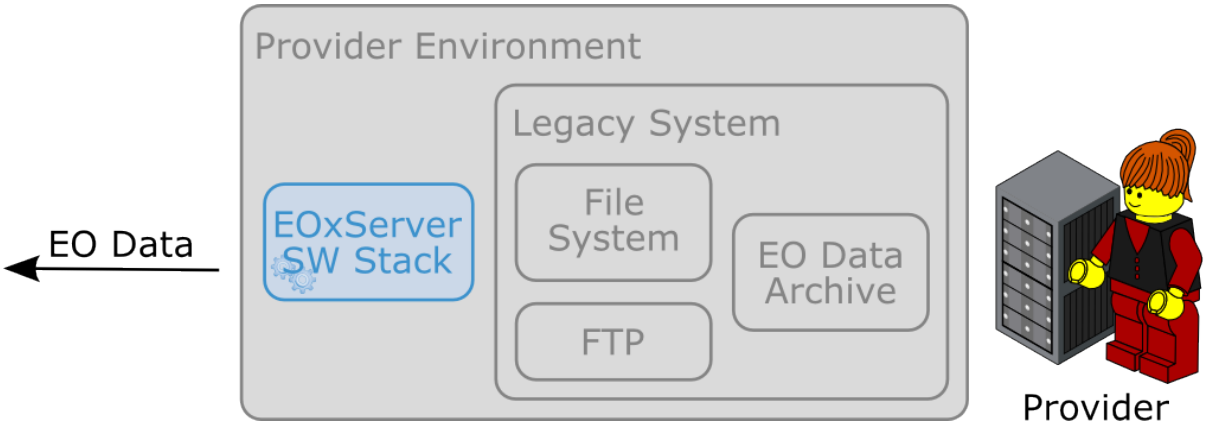


Figure 1.2: *Provider View*

- Register data

Figure: "*Activities to Enhance the Provider's Environment* (page 7)" shows these activities needed to enhance the provider's environment with online data access to the EO data archive legacy system.

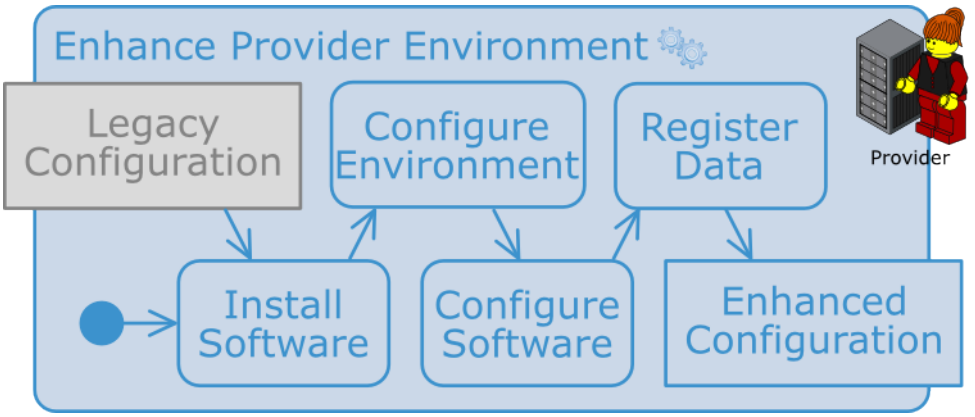


Figure 1.3: *Activities to Enhance the Provider's Environment*

Environment & Software Configuration

The EOxServer software stack consists of the EOxServer, the Identity Management, and the Applications Interface software components.

The Identity Management layer is an optional layer on top of EOxServer. Thus and because its configuration is extensively discussed in section *Identity Management System* (page 69) we skip it here.

The Applications Interface software components are discussed in detail in section The General User View (page 11) below.

As EOxServer is based on Python, MapServer, GDAL/OGR, and Django these software components need to be installed first. The base configuration of EOxServer consists of the generation of an EOxServer instance and registering it in a web server.

The EOxServer instance generation includes the configuration of various parameters like database name, type, and connection info, instance id, paths to logfiles, temporary directories, etc. as well as the initialization of its database. There are two options for the database management system (DBMS). The first is SQLite together with SpatialLite which is a single file DBMS and thus best suited for testing purposes. The second is PostgreSQL together with PostGIS which is a full fledged DBMS with numerous management functionalities and thus best suited for operational environments.

The database itself holds the configuration of components and resources (e.g. is WCS 1.0.0 enabled) as well as the coverage metadata ingested during registration (see section Data Registration (page 8)).

EOxServer can be operated with any web server that supports the Python WSGI standards[11]. For testing and implementation purposes the Django framework directly provides a simple web server. However, in operational environments the recommended deployment of EOxServer is to use the well-known Apache web server[12] together with mod_wsgi[13]. In most cases it will be the easiest, fastest, and most stable deployment choice.

At this point the provider's administrator or operator can actually run the software stack and configure the remainder via EOxServer's admin app. This app is accessed via a standard web browser and, when using Django's internal web server, available at the URL: "http://localhost:8000/admin". Use the user credentials that have been set in the database initialization step.

Figure: "*Admin app - Start* (page 9)" shows the admin app after successful login. On the left side the four modules "Auth", "Backends", "Core", and "Coverages" are shown. "Auth" is the internal Django user management module which is at the moment only used for the admin app itself. "Backends" and "Coverages" are the modules for data registration which is described in section Data Registration (page 8) below.

The "Core" module is used to enable or disable EOxServer components like services. The provider can decide which services and even which versions of which services EOxServer shall expose. A possible configuration is to expose WCS 2.0 and WMS 1.3.0 which are the latest versions but not any older version. In the default database initialization all services are enabled.

Data Registration

The data registration is done via the functionalities provided by the "Backends" and "Coverages" modules of the admin app. Figure: "*Admin app - Start* (page 9)" shows for which data types, or models in Django terminology, instances can be added or changed in these modules. These data types correspond to tables in the database. Only a subset of the full data model (see Figure: "*EOxServer Data Model for Coverage Resources* (page 116))" is shown in the admin app because some are filled automatically upon saving and some are included in the available ones like TileIndex in Stitched Mosaics.

The Dataset Series provides a convenient way to register a complete dataset series or collection at once. Figure: "*Admin app - Add/Change Dataset Series* (page 9)" shows the admin app when changing a Dataset Series instance. The operator has to provide an "EO ID" and an "EO Metadata Entry". All other parameters are optional as can be seen by the usage of normal instead of bold face text. However, in order to actually register coverages either one or multiple "Data sources", consisting of a "Location" e.g. a data directory and a "Search pattern", have to be added. Alternatively, the administrator can decide to register single coverages and link them to the Dataset Series via the "Advanced coverage handling" module (see Figure: "*Admin app - Add/Change Dataset Series Advanced* (page 12)").

Figure: "*Admin app - Add/Change EO Meatadata* (page 10)" shows the screen for adding or changing an EO metadata entry. The operator has to provide the "Begin of acquisition", "End of acquisition", and "Footprint" of the overall Dataset Series in the same way as for any EO Coverage. Calendar, clock, and map widgets are provided to ease the provision of these parameters. Optionally a full EO O&M metadata record can be supplied.

Saving a Dataset Series triggers a synchronization process. This process scans the Locations, e.g. directories and included sub-directories, of all configured Data Sources for files that follow the configured search pattern e.g. "*.tif". All files found are evaluated using GDAL and for any valid and readable raster file a Dataset instance is generated in the database holding all metadata including EO metadata for the raster file. Of course the raster file itself remains unchanged in the file system.

Let's look in more detail at the synchronization process and assume a plain GeoTIFF file with name "demo.tif" was found. The synchronization process extracts the necessary geographic metadata i.e. the domainSet or extent consisting of CRS, size, and bounding box directly from the GeoTIFF file. Where does the metadata come from? In order to retrieve the EO metadata at the moment the process looks for a file called "demo.xml" accompanying the GeoTIFF file. In future this may be expanded to automatically retrieve the metadata from catalogues like the ones the EOLI-SA connects to but for the moment the files have to be generated before the registration.

[11] https://docs.djangoproject.com/en/1.4/howto/deployment/
[12] http://httpd.apache.org
[13] http://code.google.com/p/modwsgi/

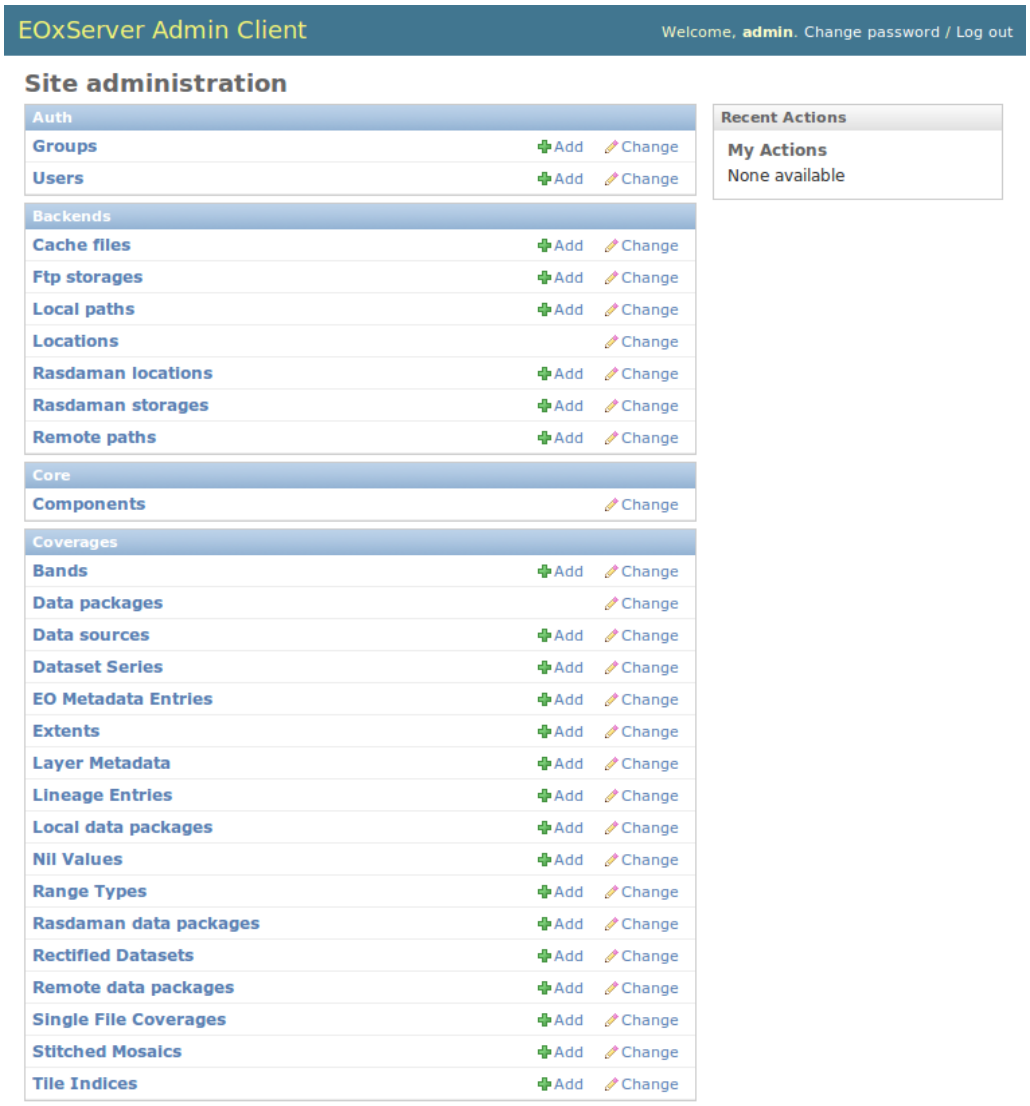


Figure 1.4: *Admin app - Start*

Figure 1.5: *Admin app - Add/Change Dataset Series*

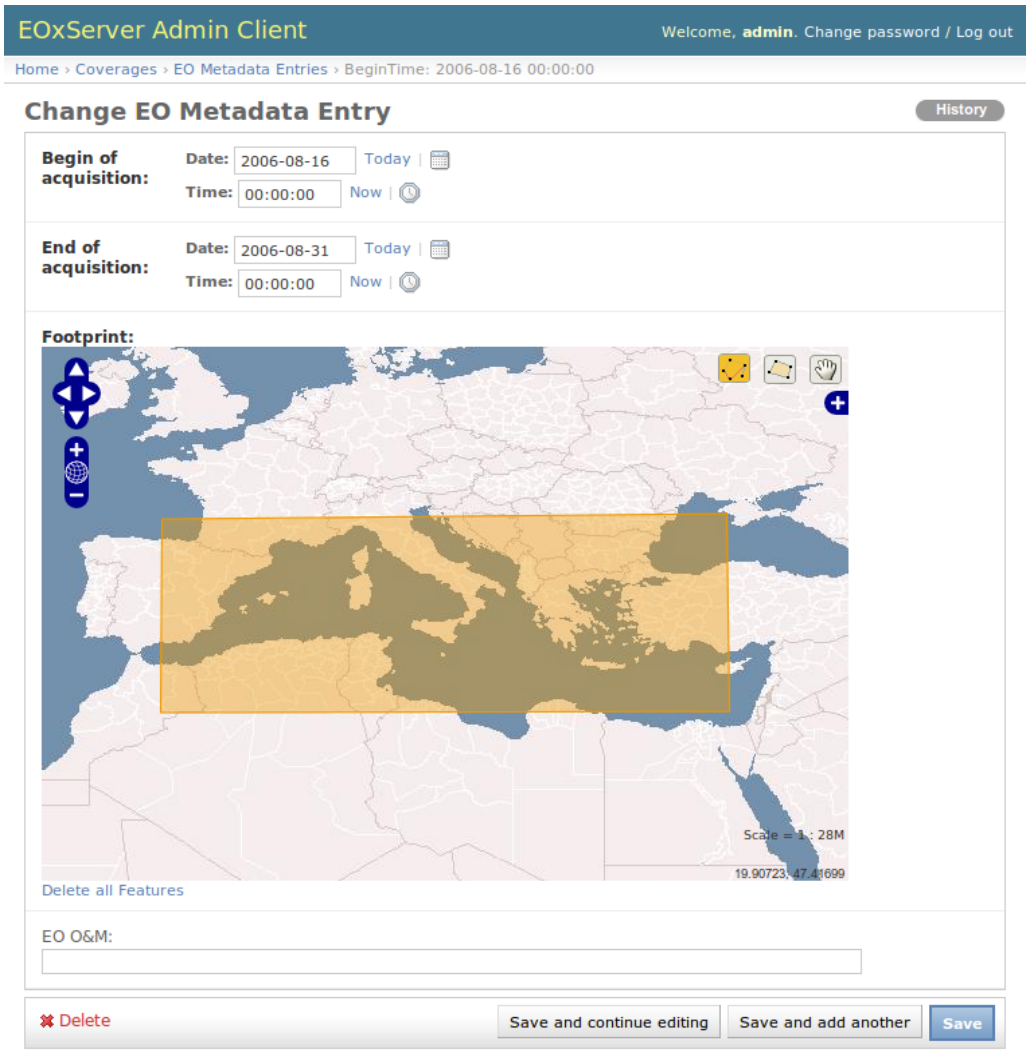


Figure 1.6: *Admin app - Add/Change EO Meatadata*

The content of this file can either be a complete EO-O&M metadata record or a simple native metadata record containing only the mandatory parameters which are: "EOID", "Begin of acquisition", "End of acquisition", and "Footprint". If no "demo.xml" is found the process uses default values which are: file name without extension, current date and time, and full bounding box of raster file. Of course, the synchronization process can be re-run at any time e.g. from a daily, hourly, etc. cronjob.

This configuration is sufficient to bring online a complete EO data archive accessible via the file system.

A comparable synchronization process is available for FTP and rasdaman back-ends as well as for Stitched Mosaics. However, mostly these processes require more complex synchronization steps. For example, via the FTP back-end it is better to not inspect the raster files itself which would mean to completely transfer them but to retrieve the geographic information together with the EO metadata. Please refer to the remainder of this *EOxServer Users' Guide* (page 1) for detailed information and usage instructions.

1.2.2 The General *User* View

The user needs certain EO data as input to some processing. This processing ranges from simply viewing certain parameters of EO data to complex data analysis and generation of derived data. The user has an environment with the software installed needed for the processing. For simplicity let's assume the user has two different software tools installed to process the data. First there is a standard web browser which manages the HTTP protocol and is capable of viewing HTML web pages. Second there is a GIS software which shall be QGis in our example.

Figure: "*User View* (page 13)" shows the user environment and its installed software.

First of all the user needs to find an EO data provider who has data that fit the user's purpose and who offers the data via a mechanism the user can handle. Luckily the user happens to know a provider who is running the EOxServer software stack on an EO data archive holding the required data. Thus the user can decide between several ways how to retrieve the data. Some involve client side components of the EOxServer software stack but because of the strict adherence to open standards various other ways are possible in parallel. However, we'll focus below on two ways involving EOxServer software components.

Web Browser

In the first case the provider offers a dedicated app using EOxServer's Web API. This app consists of HTML and Javascript files and is served via a web server from the provider's environment. In our case the app provides access to one dataset series holding some MERIS scenes over Europe.

Figure: "*Browser app featuring EOxServer's Web API* (page 13)" shows a screen shot of this app. The app implements the paradigm of "zooming to the data" i.e. the user directly looks at previews of the data served via EO-WMS rather than having to search in a catalogue first. After zooming to and therewith setting the Area of Interest (AoI) and setting the Time of Interest (ToI) the user following the download button is presented with the metadata of the included datasets retrieved from the offered EO-WCS. The metadata includes grid, bands, CRS, nil values, etc. of the datasets but also formats, CRSs, and interpolation methods the dataset can be retrieved in. Based on this information the user decides which datasets to download and specifies parameters of the download like spatial sub-setting, band sub-setting, CRS, size/resolution, interpolation method, format, and format specific parameters like compression. The app guides the user to specify all these parameters and downloads only the really needed data to the user's environment. The EO-WCS protocol is used by the app transparently to the user i.e. most of the complexity of the EO-WCS protocol is hidden.

This app shows the benefit of the integrated usage of EO-WMS and EO-WCS for the online data access to the EO data archive.

The Webclient Interface (page 62) section of the documentation provides more details about the Web API.

GIS Tool

Note, that the Python Client API is not yet implemented and only available as concept.

In the second case the user wants to use the full-fledged GIS software tool QGis and thus decides to use the handy EO-WCS plug-in provided by the provider. This plug-in makes extensive use of EOxServer's Python Client API.

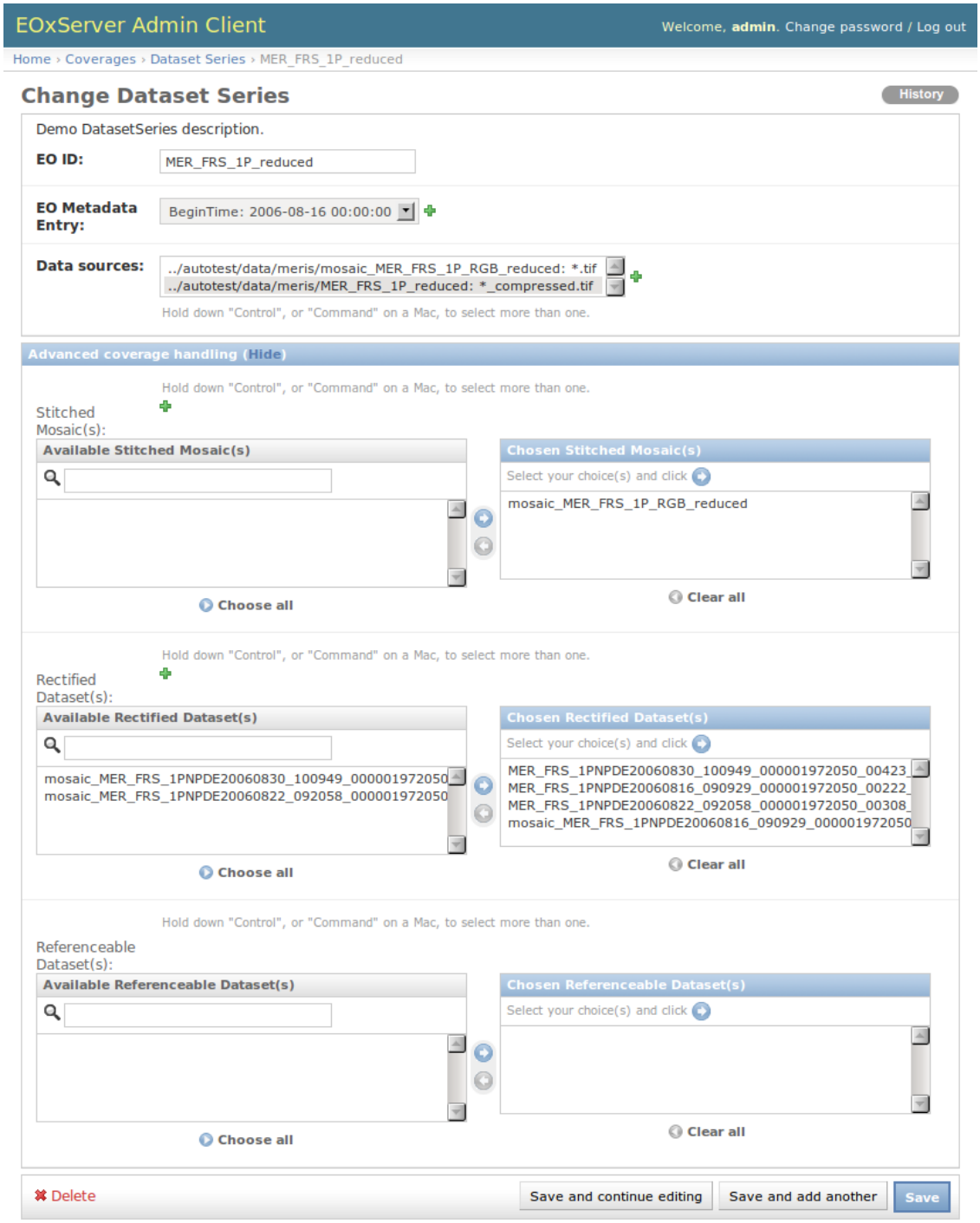


Figure 1.7: *Admin app - Add/Change Dataset Series Advanced*

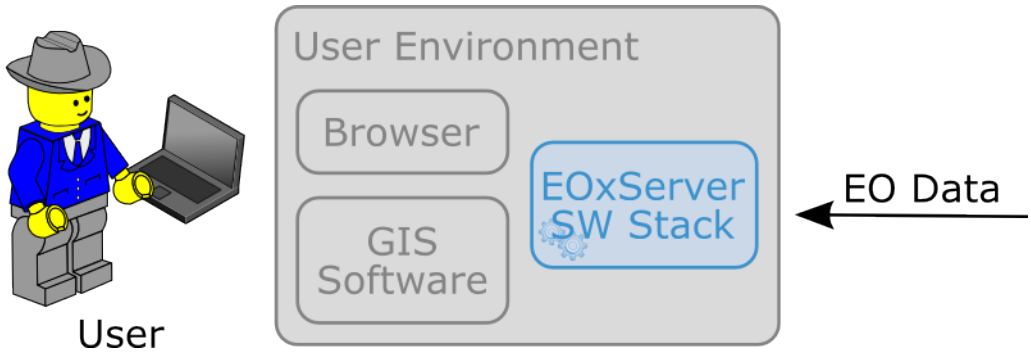


Figure 1.8: *User View*

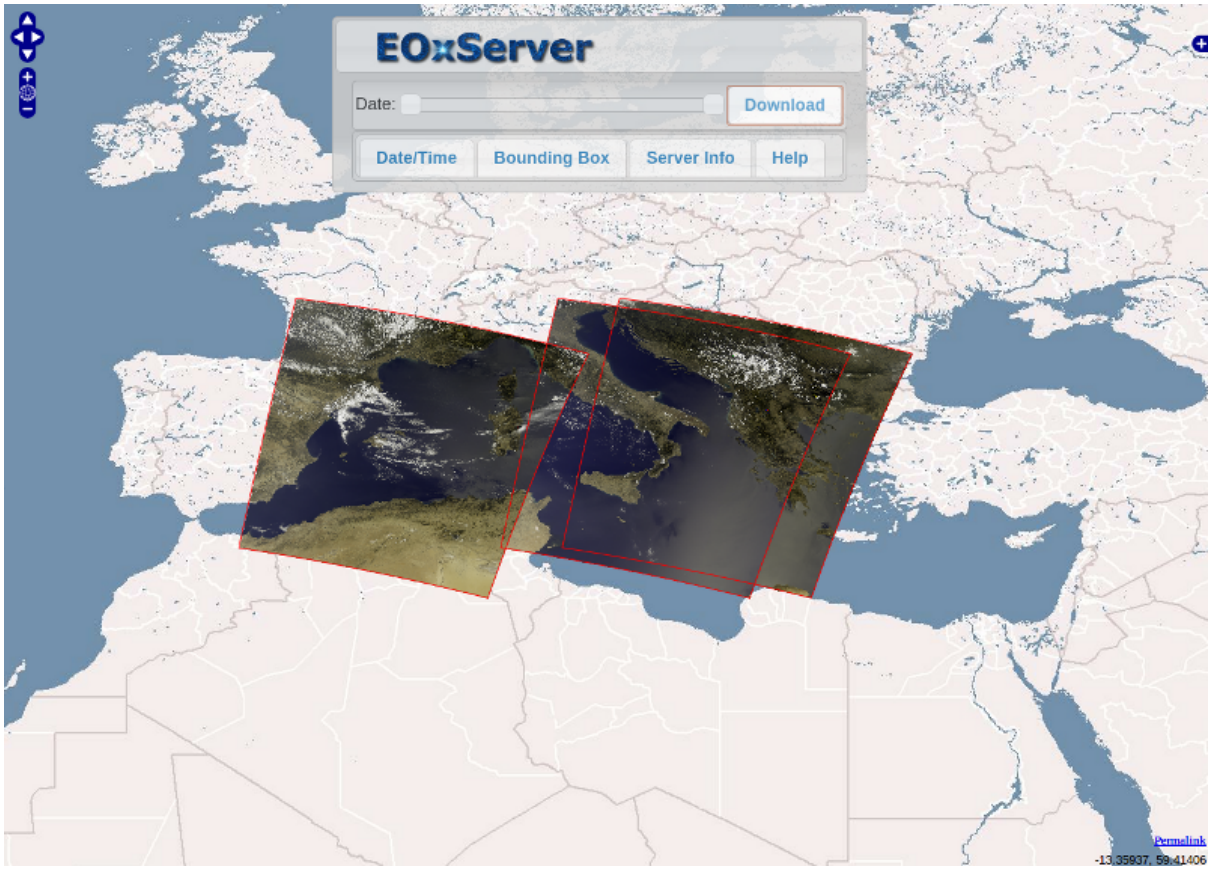


Figure 1.9: *Browser app featuring EOxServer's Web API*

Figure: "*QGis EO-WCS Plug-in featuring EOxServer's Python Client API* (page 14)" shows a screen shot how the usage of the EO-WCS plug-in for QGis might look like. The user first has to connect to the provider's EO-WCS endpoint. Once connected the plug-in retrieves the metadata about the available dataset series and shows them as a list to the user together with the tools to specify AoI and ToI. Metadata of datasets and stitched mosaics might also be retrieved in this step if the provider configured some to be directly visible in the capabilities of the EO-WCS.

The selected dataset series are transparently searched within the set spatio-temporal bounding box and available datasets and stitched mosaics presented to the user. After exploring and setting the download parameters like in the first case the EO-WCS plug-in downloads again only the required data sub-sets. In addition to the previous case the EO-WCS plug-in applies various strategies to limit the data download. For example if a dataset is added to the current list of layers only the currently viewed area needs to be filled with data at the resolution of the screen. In addition the data can be sub-setted to one or three bands that are shown i.e. there's no need to download numerous float32 bands just to preview the data.

With using the EOxServer software stack on the provider side the plug-in includes the possibility to exploit the integrated usage of EO-WMS and EO-WCS. This exploitation includes the displaying of previews in the two steps described above. Another feature is, that the possibly nicer looking images are retrieved from the performance optimized EO-WMS to fill the current view.

Once the user starts some sophisticated processing the plug-in retrieves the required sub-sets of the original data. Again strategies to limit the data download are applied.

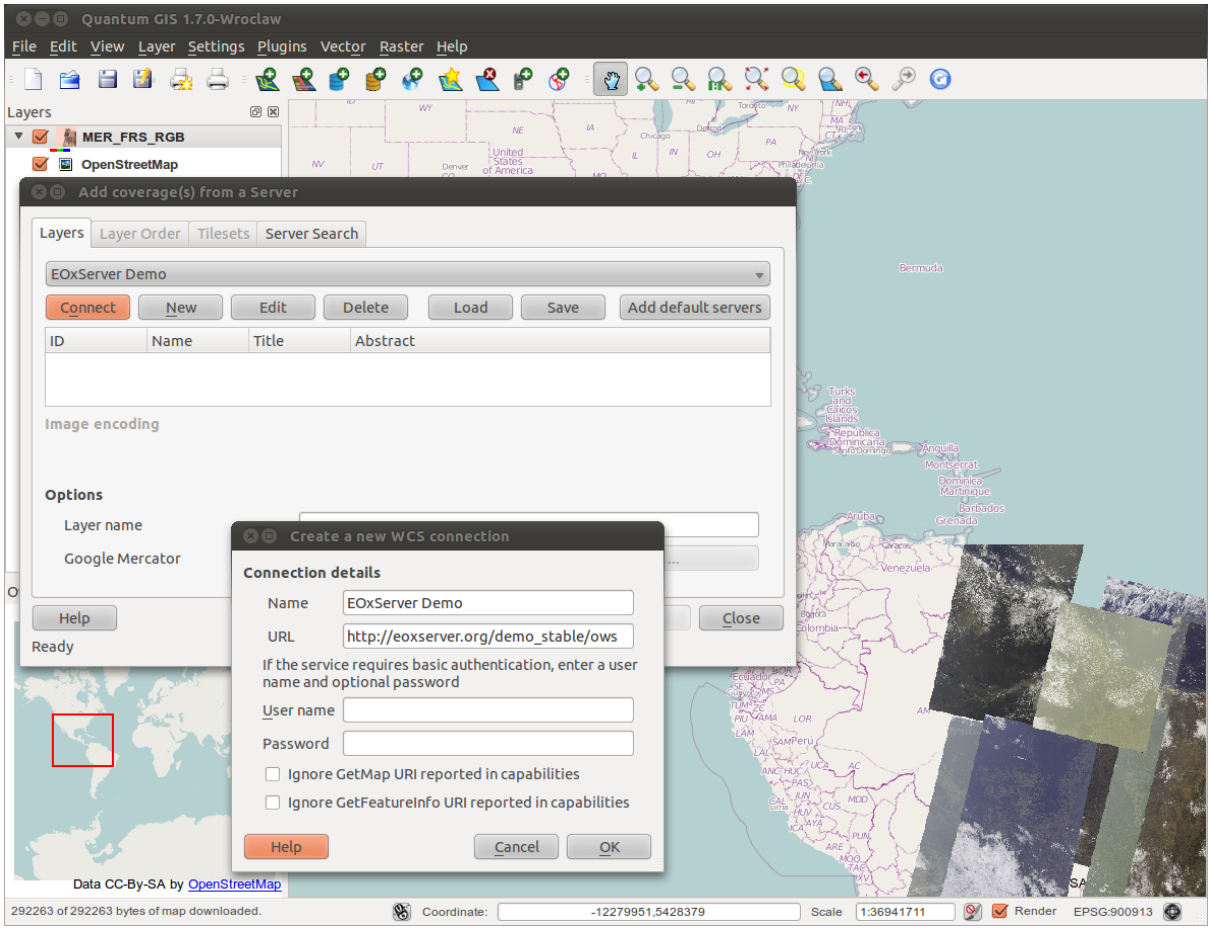


Figure 1.10: *QGis EO-WCS Plug-in featuring EOxServer's Python Client API*

1.3 Installation

Table of Contents

- Installation (page 14)
  - Hardware Requirements (page 15)
  - Dependencies (page 15)
  - Installing EOxServer (page 16)
  - Upgrading EOxServer (page 17)

To use EOxServer it must be installed first. Following this guide will give you a working software installation.

See Also:

- ***Installation on CentOS*** **(page 17)** for specific installation on CentOS.
- ***Service Instance Creation and Configuration*** **(page 20)** to configure an instance of EOxServer after successful installation.
- ***Recommendations for Operational Installation*** **(page 26)** to configure an operational EOxServer installation.

1.3.1 Hardware Requirements

EOxServer has been deployed on a variety of different computers and virtual machines with commonplace hardware configurations. The typical setup is:

- a dual-core or quad-core CPU
- 1 to 4 GB of RAM

The image processing operations required for certain OGC Web Service requests (subsetting, reprojection, resampling) may be quite expensive in terms of CPU load and memory consumption, so adding more RAM or an additional core (for VMs) may increase the performance of the service. Bear in mind however, that disk I/O speed is often a bottleneck.

EOxServer itself requires about 15 MB of disk space. Usually, the data to be served has a magnitude of 10-100 GB or larger. So, this will be the determining factor when choosing the appropriate disk size. Note that for Rectified Stitched Mosaics, EOxServer will generate mosaic tiles from the original data which requires additional disk space up to the space occupied by the composing Rectified Datasets (depending on how much they overlap).

EOxServer itself does not have any GUI other than the Web Administration Client and Web Map Client and thus no graphics support is required on the server.

Running (parts of) the Identity Management System (see *Identity Management System* (page 69)) on the same machine as EOxServer puts additional load on the server. Usually, running the Tomcat server will require about 512 MB of RAM. Note that the different components of the IDM may be deployed on different machines. The additional network latency for checking a remote PDP on every incoming request may have a considerable impact on the performance of the services (in particular WMS), though, and thus it may be preferable to run the PDP on the same machine as EOxServer.

1.3.2 Dependencies

EOxServer depends on some external software. Table: "*EOxServer Dependencies* (page 15)" below shows the minimal required software to run EOxServer.

Table 1.1: EOxServer Dependencies

Software	Required Version	Description
Python	>= 2.5, < 3.0 (>=2.6.5 for Django 1.5)	Scripting language
Django	>= 1.4 (1.5 for PostGIS 2.0 support)	Web development framework written in Python including the GeoDjango extension for geospatial database back-ends.
GDAL	>= 1.7.0 (1.8.0 for rasdaman support)	Geospatial Data Abstraction Library providing common interfaces for accessing various kinds of raster and vector data formats and including a Python binding which is used by EOxServer
GEOS	>= 3.0	GEOS (Geometry Engine - Open Source) is a C++ port of the Java Topology Suite (JTS).
libxml2	>= 2.7	Libxml2 is the XML C parser and toolkit developed for the Gnome project.
lxml	>= 2.2	The lxml XML toolkit is a Pythonic binding for the C libraries libxml2 and libxslt.
MapServer	>= 6.2 (works partly with 6.0)	Server software implementing various OGC Web Service interfaces including WCS and WMS. Includes a Python binding which is used by EOxServer.

The Python bindings of the GDAL, MapServer (MapScript) and libxml2 libraries are required as well.

EOxServer is written in Python[14] and uses the Django[15] framework which requires a Python version from 2.5 to 2.7. Due to backwards incompatibilities in Python 3.0, Django and thus EOxServer does not currently work with Python 3.0.

EOxServer makes heavy usage of the OSGeo[16] projects GDAL[17] and MapServer[18].

EOxServer also requires a database to store its internal data objects. Since it is built on Django, EOxServer is mostly database agnostic, which means you can choose from various database systems. Since EOxServer requires the database to have geospatial enablement, the according extensions to that database have to be installed. We suggest you use one of the following:

- For testing environments or small amounts of data, the SQLite[19] database provides a lightweight and easy-to-use system.
- However, if you'd like to work with a "large" database engine in an operational environment we recommend installing PostgreSQL[20].

For more and detailed information about database backends please refer to Django database notes[21] and GeoDjango installation[22].

Table 1.2: Database Dependencies

Backend	Required Version	Required extensions/software
SQLite	>= 3.6	spatialite (>= 2.3), pysqlite2 (>= 2.5), GEOS (>= 3.0), PROJ.4 (>= 4.4)
PostgreSQL	>= 8.1	PostGIS (>= 1.3), GEOS (>= 3.0), PROJ.4 (>= 4.4), psycopg2 (== 2.4.1)

1.3.3 Installing EOxServer

There are several easy options to install EOxServer:

[14] http://www.python.org/
[15] https://www.djangoproject.com
[16] http://osgeo.org
[17] http://www.gdal.org
[18] http://mapserver.org
[19] http://sqlite.org/
[20] http://www.postgresql.org/
[21] https://docs.djangoproject.com/en/1.4/ref/databases/
[22] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/install/

- Install an official release of EOxServer, the best approach for users who want a stable version and aren't concerned about running a slightly older version of EOxServer. You can install EOxServer either from
  - PyPI - the Python Package Index[23] using pip[24]:

    ```
    sudo pip install eoxserver
    ```

  - or from the EOxServer download page[25] using pip:

    ```
    sudo pip install http://eoxserver.org/export/head/downloads/EOxServer-<version>.tar.gz
    ```

    or manual:

    ```
    wget http://eoxserver.org/export/head/downloads/EOxServer_full-<version>.tar.gz .
    tar xvfz EOxServer-<version>.tar.gz
    cd EOxServer-<version>
    sudo python setup.py install
    ```

  - or binaries provided by your operating system distribution e.g. *CentOS* (page 17).
- Install the latest development version, the best option for users who want the latest-and-greatest features and aren't afraid of running brand-new code. Make sure you have Subversion[26] installed and install EOxServer's main development branch (the trunk) using pip:

  ```
  sudo pip install svn+http://eoxserver.org/svn/trunk
  ```

  or manual:

  ```
  svn co http://eoxserver.org/svn/trunk/ eoxserver-trunk
  cd eoxserver-trunk
  sudo python setup.py install
  ```

If the directory EOxServer is installed to is not on the Python path, you will have to configure the deployed instances accordingly, see *Deployment* (page 23) below.

The successful installation of EOxServer can be tested using the *autotest instance* (page 118) which is described in more detail in the *EOxServer Developers' Guide* (page 113).

Now that EOxServer is properly installed the next step is to *create and configure a service instance* (page 20).

1.3.4 Upgrading EOxServer

To upgrade an existing installation of EOxServer simply add the *–upgrade* switch to your pip command e.g.:

```
sudo pip install --upgrade eoxserver
```

or rerun the manual installation as explained above.

Please carefully follow the *migration/update procedure* (page 33) corresponding to your version numbers for any configured EOxServer instances in case of a major version upgrade.

1.4 Installation on CentOS

[23] http://pypi.python.org/pypi
[24] http://www.pip-installer.org/en/latest/index.html
[25] http://eoxserver.org/wiki/Download
[26] http://subversion.tigris.org/

Table of Contents

- Installation on CentOS (page 17)
  - Prerequisites (page 18)
  - Installation from RPM Packages (page 18)
    - Preparation of RPM Repositories (page 18)
    - Installing EOxServer (page 19)
  - Alternate installation method using *pip* (page 19)
    - Required Software Packages (page 19)
    - Installing EOxServer (page 19)
  - Special *pysqlite* considerations (page 20)

This section describes specific installation procedure for EOxServer on CentOS[27] GNU/Linux based operating systems. In this example, a raw CentOS 6.4 minimal image is used.

This guide is assumed (but not tested) to be applicable also for equivalent versions of the prominent North American Enterprise Linux and its clones.

See Also:

- ***Installation*** **(page 14)** generic installation procedure for GNU/Linux operating systems.
- ***Service Instance Creation and Configuration*** **(page 20)** to configure an instance of EOxServer after successful installation.
- ***Recommendations for Operational Installation*** **(page 26)** to configure an operational EOxServer installation.

1.4.1 Prerequisites

This example requires a running CentOS installation with superuser privileges available.

1.4.2 Installation from RPM Packages

Preparation of RPM Repositories

The default repositories of CentOS do not provide all software packages required for EOxServer, and some packages are only provided in out-dated versions. Thus several further repositories have to be added to the system's list.

The first one is the ELGIS (Enterprise Linux GIS)[28] repository which can be added with the following *yum* command:

```
sudo rpm -Uvh http://elgis.argeo.org/repos/6/elgis-release-6-6_0.noarch.rpm
```

The second repository to be added is EPEL (Extra Packages for Enterprise Linux)[29] again via a simple *yum* command:

```
sudo rpm -Uvh http://download.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
```

Finally EOxServer is available from the yum repository at packages.eox.at[30]. This repository offers current versions of packages like MapServer[31] as well as custom built ones with extra drivers enabled like GDAL[32] and/or with patches applied like libxml2[33]. It is not mandatory to use this repository as detailed below but it is highly

[27] http://www.centos.org/
[28] http://wiki.osgeo.org/wiki/Enterprise_Linux_GIS
[29] http://fedoraproject.org/wiki/EPEL
[30] http://packages.eox.at
[31] http://mapserver.org/
[32] http://gdal.org/
[33] http://xmlsoft.org/

recommended in order for all features of EOxServer to work correctly. The repository is again easily added via a single *yum* command:

```
sudo rpm -Uvh http://yum.packages.eox.at/el/eox-release-6-2.noarch.rpm
```

Installing EOxServer

Once the RPM repositories are configured EOxServer and all its dependencies are installed via a single command:

```
sudo yum install EOxServer
```

To update EOxServer simply run the above command again or update the whole system with:

```
sudo yum update
```

Please carefully follow the *migration/update procedure* (page 33) corresponding to your version numbers for any configured EOxServer instances in case of a major version upgrade.

Further packages may be required if additional features (e.g: a full DBMS) are desired. The following command for example installs all packages needed when using SQLite:

```
sudo yum install sqlite libspatialite python-pysqlite python-pyspatialite
```

Alternatively the PosgreSQL DBMS can be installed as follows:

```
sudo yum install postgresql postgresql-server postgis python-psycopg2
```

To run EOxServer behind the Apache web server requires the installation of this web server:

```
sudo yum install httpd mod_wsgi
```

Now that EOxServer is properly installed the next step is to *create and configure a service instance* (page 20).

1.4.3 Alternate installation method using *pip*

Required Software Packages

The installation via pip builds EOxServer from its source. Thus there are some additional packages required which can be installed using:

```
sudo yum install gdal gdal-python gdal-devel mapserver mapserver-python \
                 libxml2 libxml2-python python-lxml python-pip \
                 python-devel gcc
```

Installing EOxServer

For the installation of Python packages pip[34] is used, which itself was installed in the previous step. It automatically resolves and installs all dependencies. So a simple:

```
sudo pip-python install eoxserver
```

suffices to install EOxServer itself.

To upgrade an existing installation of EOxServer simply add the `--upgrade` switch to your pip command:

```
sudo pip-python install --upgrade eoxserver
```

Please don't forget to follow the update procedure for any configured EOxServer instances in case of a major version upgrade.

Now that EOxServer is properly installed the next step is to *create and configure a service instance* (page 20).

[34] http://www.pip-installer.org/

1.4.4 Special *pysqlite* considerations

When used with spatialite[35] EOxServer also requires pysqlite[36] and *pyspatialite* which can be either installed as RPMs from packages.eox.at[37] (see *Installing EOxServer* (page 19) above) or from source.

If installing from source please make sure to adjust the *SQLITE_OMIT_LOAD_EXTENSION* parameter in `setup.cfg` which is set by default but not allowed for EOxServer. The following provides a complete installation procedure:

```
sudo yum install libspatialite-devel geos-devel proj-devel
sudo pip-python install pyspatialite
wget https://pysqlite.googlecode.com/files/pysqlite-2.6.3.tar.gz
tar xzf pysqlite-2.6.3.tar.gz
cd pysqlite-2.6.3
sed -e '/^define=SQLITE_OMIT_LOAD_EXTENSION$/d' -i setup.cfg
sudo python setup.py install
```

If the installation is rerun the `--upgrade` respectively the `--force` flag have to be added to the `pip-python` and `python` commands in order to actually redo the installation:

```
sudo pip-python install --upgrade pyspatialite
sudo python setup.py install --force
```

1.5 Service Instance Creation and Configuration

Table of Contents

- Service Instance Creation and Configuration (page 20)
  - Instance Creation (page 20)
  - Instance Configuration (page 21)
  - Database Setup (page 22)
  - Deployment (page 23)
  - Data Registration (page 24)

Speaking of EOxServer we distinguish the common EOxServer installation (the installed code implementing the software functionality) and EOxServer instances. An instance is a collection of data and configuration files that enables the deployment of a specific service. A single server will typically contain a single software installation and one or more specific instances.

This section deals with the creation and configuration of EOxServer instances.

See Also:

- ***Installation*** **(page 14)** generic installation procedure for GNU/Linux operating systems.
- ***Installation on CentOS*** **(page 17)** for specific installation on CentOS.
- ***Recommendations for Operational Installation*** **(page 26)** to configure an operational EOxServer installation.

1.5.1 Instance Creation

To create an instance, we recommend to use the `eoxserver-admin.py` script that comes with EOxServer. The script provides the command *create_instance* in order to create an EOxServer instance:

[35] http://www.gaia-gis.it/spatialite/
[36] http://code.google.com/p/pysqlite/
[37] http://packages.eox.at

Usage: `eoxserver-admin.py create_instance [options] INSTANCE_ID [Optional destination directory]`

Creates a new EOxServer instance with name `INSTANCE_ID` in the current or optionally given directory with all necessary files and folder structure. If the `--init_spatialite` flag is set, then an initial sqlite database will be created and initialized.

Options:

-h, --help	show this help message and exit
--init_spatialite	Flag to initialize the sqlite database.

1.5.2 Instance Configuration

Every EOxServer instance has three configuration files:

- `settings.py` - template[38]
- `conf/eoxserver.conf` - template[39]
- `conf/template.map` - template[40]

For each of them there is a template in the `eoxserver/conf` directory of the EOxServer distribution (referenced above) which is copied and adjusted by the *create_instance* command of the `eoxserver-admin.py` script to the instance directory. If you create an EOxServer instance without the script you can copy those files and edit them yourself.

The file `settings.py` contains the Django configuration. Settings that need to be customized:

- `PROJECT_DIR`: Absolute path to the instance directory.
- `DATABASES`: The database connection details. For detailed information see Database Setup (page 22)

You can also customize further settings, for a complete reference please refer to the Django settings overview[41].

Please especially consider the setting of the TIME_ZONE[42] parameter and read the Notes provided in the `settings.py` file.

The file `conf/eoxserver.conf` contains EOxServer specific settings. Please refer to the inline documentation for details.

The file `conf/template.map` contains basic metadata for the OGC Web Services used by MapServer. For more information on metadata supported please refer to the MapServer Mapfile documentation[43].

Once you have created an instance, you have to configure and synchronize the database. If using the *create_instance* command of the `eoxserver-admin.py` script with the `--init_spatialite` flag, all you have to do is:

- Make sure EOxServer is on your `PYTHONPATH` environment variable
- run in your instance directory:

```
python manage.py syncdb
```

Note down the username and password you provide. You'll need it to log in to the admin client.

[38] http://eoxserver.org/browser/trunk/eoxserver/conf/TEMPLATE_settings.py
[39] http://eoxserver.org/browser/trunk/eoxserver/conf/TEMPLATE_eoxserver.conf
[40] http://eoxserver.org/browser/trunk/eoxserver/conf/TEMPLATE_template.map
[41] https://docs.djangoproject.com/en/1.4/topics/settings/
[42] https://docs.djangoproject.com/en/1.4/ref/settings/#std:setting-TIME_ZONE
[43] http://mapserver.org/mapfile/index.html

1.5.3 Database Setup

This section is only needed if the `--init_spatialite` flag was not used during instance creation or a PostgreSQL/PostGIS database back-end shall be used. Before proceeding, please make sure that you have installed all required software for the database system of your choice.

Using a SQLite database, all you have to do is to copy the `TEMPLATE_config.sqlite` and place it somewhere in your instance directory. Now you have to edit the `DATABASES` of your `settings.py` file with the following lines:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.spatialite',
        'NAME': '/path/to/config.sqlite',
    }
}
```

Note: By default the number of SQL variables (SQLITE_MAX_VARIABLE_NUMBER) in SQL is limited to 999. This leads to problems when having inserted 1000 datasets or more. In this case the limit could either be increased or PostgreSQL/PostGIS must be used as a back-end database.

Using a PostgreSQL/PostGIS database back-end configuration for EOxServer is a little bit more complex. Setting up a PostgreSQL database requires also installing the PostGIS extensions (the following example is an installation based on a Debian system):

```
sudo su - postgres
POSTGIS_DB_NAME=eoxserver_db
POSTGIS_SQL_PATH=`pg_config --sharedir`/contrib/postgis-1.5
createdb $POSTGIS_DB_NAME
createlang plpgsql $POSTGIS_DB_NAME
psql -d $POSTGIS_DB_NAME -f $POSTGIS_SQL_PATH/postgis.sql
psql -d $POSTGIS_DB_NAME -f $POSTGIS_SQL_PATH/spatial_ref_sys.sql
psql -d $POSTGIS_DB_NAME -c "GRANT ALL ON geometry_columns TO PUBLIC;"
psql -d $POSTGIS_DB_NAME -c "GRANT ALL ON geography_columns TO PUBLIC;"
psql -d $POSTGIS_DB_NAME -c "GRANT ALL ON spatial_ref_sys TO PUBLIC;"
```

This creates the database and installs the PostGIS extensions within the database. Now a user with password can be set with the following line:

```
createuser -d -R -P -S eoxserver-admin
```

Depending on the configuration of the system used there may be the need to enable access for the user in the `pg_hba.conf`.

In the `settings.py` the following entry has to be added:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': 'eoxserver_db',
        'USER': 'eoxserver-admin',
        'PASSWORD': 'eoxserver',
        'HOST': 'localhost',    # or the URL of your server hosting the DB
        'PORT': '',
    }
}
```

Please refer to GeoDjango Database API[44] for more instructions.

[44] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/db-api/

1.5.4 Deployment

EOxServer is deployed using the Python WSGI interface standard as any other Django application[45]. The WSGI endpoint accepts HTTP requests passed from the web server and processes them synchronously. Each request is executed independently.

In the following we present the way to deploy it using the Apache2 Web Server[46] and its mod_wsgi[47] extension module.

The deployment procedure consists of the following:

- Customize the Apache2 configuration file, e.g. `/etc/apache2/sites-enabled/000-default`, by adding:

```
Alias /<url> <absolute path to instance dir>/wsgi.py
<Directory "<absolute path to instance dir>">
        AllowOverride None
        Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
        AddHandler wsgi-script .py
        Order Allow,Deny
        Allow from all
</Directory>
```

- If using EOxServer < 0.3 customize `wsgi.py` in your EOxServer instance and add:

```
import sys

path = "<absolute path to instance dir>"
if path not in sys.path:
    sys.path.append(path)
```

  - If using Django < 1.4 please copy `TEMPLATE_wsgi.py` from the EOxServer distribution `eoxserver/conf` directory in your instance under the name `wsgi.py` and customize it at the two indicated places.

- Restart the Web Server

As a general good idea the number of threads can be limited using the following additional Apache2 configuration. In case an old version of MapServer, i.e. < 6.2 or < 6.0.4, is used the number of threads **needs** to be limited to 1 to avoid some thread safety issues[48]:

```
WSGIDaemonProcess ows processes=10 threads=1
<Directory "<absolute path to instance dir>">
    ...
    WSGIProcessGroup ows
</Directory>
```

This setup will deploy your instance under the URL `<url>` and make it publicly accessible.

Now that the public URL is known don't forget to adjust the configuration in `conf/eoxserver.conf`:

```
[services.owscommon]
http_service_url=http://<url>/ows
```

Finally all the static files need to be collected at the location configured by `STATIC_ROOT` in `settings.py` by using the following command from within your instance:

```
python manage.py collectstatic
```

Don't forget to update the static files by re-running above command if needed.

[45] https://docs.djangoproject.com/en/1.4/howto/deployment/
[46] http://httpd.apache.org
[47] http://code.google.com/p/modwsgi/
[48] https://github.com/mapserver/mapserver/issues/4369

1.5.5 Data Registration

To insert data into an EOxServer instance there are several ways. One is the admin interface, which is explained in detail in the *Admin Client* (page 51) section.

Another convenient way to register datasets is the command line interface to EOxServer. As a Django application, the instance can be configured using the manage.py[49] script.

EOxServer provides a specific command to insert datasets into the instance, called `eoxs_register_dataset`. It is invoked from command line from your instance base folder:

```
python manage.py eoxs_register_dataset --data-file DATAFILES --rangetype RANGETYPE
```

The mandatory parameter `--data-file` is a list of at least one path to a file containing the raster data for the dataset to be inserted. The files can be in any compliant (GDAL readable) format. When inserting datasets located in a Rasdaman database, this parameter defines the *collection* the dataset is contained in.

Also mandatory is the parameter `--rangetype`, the name of a range type which has to be already present in the instance's database.

For each data file there may be given one metadata file containing Earth Observation specific metadata. The optional parameter `--metadata-file` shall contain a list of paths to these files, where the items of this list refer to the data files with the same index of the according option. A metadata file for each data file is assumed with the same path, but with an *.xml* extension when this parameter is omitted. However, it is only used when it actually exists. Otherwise the data file itself is used to retrieve the metadata values. When this is not possible either, the default values are used as described below or the insertion is aborted.

When inserting datasets located in a Rasdaman database, this parameter is mandatory, since the metadata cannot be retrieved from within the rasdaman database and must be locally accessible.

For each dataset a coverage ID can be specified with the `--coverage-id` parameter. As with the `--metadata-file` option, the items of the list refer to the items of the `--data-file` list. If omitted, an ID is generated using the data file name.

The parameters `--dataset-series` and `--stitched-mosaic` allow to insert the dataset into all dataset series and rectified stitched mosaics specified by their EO IDs.

The `--mode` parameter specifies the location of the data and metadata files as they may be located on a FTP server or in a Rasdaman database. This can either be *local*, *ftp* or *rasdaman*, whereas the default is *local*.

When the mode is set to either *ftp* or *rasdaman* the following options define the location of the dataset and the connection to it more thoroughly: `--host`, `--port`, `--user`, `--password`, and `--database` (only for *rasdaman*). Only the `--host` parameter is mandatory, all others are optional.

The `--default-srid` parameter is required when the SRID cannot be determined automatically, as for example with rasdaman datasets.

For when you explicitly want to override the geospatial metadata of a dataset you can use `--default-size` and `--default-extent`. Both parameters need to be used together and in combination with `--default-srid`. This is required for datasets registered in a rasdaman database or for any other input method where the geospatial metadata cannot be retrieved.

For datasets that do not have any EO metadata associated and want to be inserted anyways, the options `--default-begin-time`, `--default-end-time` and `--default-footprint` have to be used. These meta data values will only be used when no local meta data file is found (remote files are not checked). All three options have to be used in combination, so it is, for example, not possible to only provide the footprint via `--default-footprint` and let EOxServer gather the rest. There is one exception: when only begin and end dates are given, the footprint is generated using the image extent.

With the `--visible` option, all registered datasets can be marked as either visible (`true`) or invisible (`false`). This effects the advertisment of the dataset in e.g: GetCapabilities responses. By default, all datasets are visible.

This is an example usage of the `eoxs_register_dataset` command:

[49] https://docs.djangoproject.com/en/1.4/ref/django-admin/

```
python manage.py eoxs_register_dataset --data-file data/meris/mosaic_MER_FRS_1P_RGB_reduced/*.tif
    --dataset-series MER_FRS_1P_RGB_reduced --stitched-mosaic mosaic_MER_FRS_1P_RGB_reduced -v3
```

In this example, the parameter `--metadata-file` is omitted, since these files are in the same location as the data files and only differ in their extension. Also note that the `--data-file` parameter uses a shell wildcard **.tif* which expands to all files with *.tif* extension in that directory. This funcitonality is not provided by EOxServer but by the operating system or the executing shell and is most certainly platform dependant.

Here is another example including the `--coverage-ids` parameter which overwrites the default ids based on the data file names e.g. because they are not valid `NCNames` which is needed by the XML schemas:

```
python manage.py eoxs_register_dataset --data-files 1.tif 2.tif 3.tif \
    --coverage-ids a b c --rangetype RGB  -v3
```

The registered dataset is also inserted to the given dataset series and rectified stitched mosaic.

Here is the full list of available options:

-v VERBOSITY, --verbosity=VERBOSITY Verbosity level; 0=minimal output, 1=normal output, 2=all output

--settings=SETTINGS The Python path to a settings module, e.g. "myproject.settings.main". If this isn't provided, the DJANGO_SETTINGS_MODULE environment variable will be used.

--pythonpath=PYTHONPATH A directory to add to the Python path, e.g. "/home/djangoprojects/myproject".

--traceback Print traceback on exception

-d, --data-file, --data-files, --collection, --collections Mandatory. One or more paths to a files containing the image data. These paths can either be local, ftp paths, or rasdaman collection names.

-m, --metadata-file, --metadata-files Optional. One or more paths to a local files containing the image meta data. Defaults to the same path as the data file with the ".xml" extension.

-r RANGETYPE, --rangetype=RANGETYPE Mandatory identifier of the rangetype used in the dataset.

--dataset-series Optional. One or more eo ids of a dataset series in which the created datasets shall be added.

--stitched-mosaic Optional. One or more eo ids of a rectified stitched mosaic in which the dataset shall be added.

-i, --coverage-id, --coverage-ids Optional. One or more coverage identifier for each dataset that shall be added. Defaults to the base filename without extension.

--mode=MODE Optional. Defines the location of the datasets to be registered. Can be 'local', 'ftp', or 'rasdaman'. Defaults to 'local'.

--host=HOST Mandatory when mode is not 'local'. Defines the ftp/rasdaman host to locate the dataset.

--port=PORT Optional. Defines the port for ftp/rasdaman host connections.

--user=USER Optional. Defines the ftp/rasdaman user for the ftp/rasdaman connection.

--password=PASSWORD Optional. Defines the ftp/rasdaman user password for the ftp/rasdaman connection.

--database=DATABASE Optional. Defines the rasdaman database containing the data.

--oid, --oids Optional. List of rasdaman oids for each dataset to be inserted.

--default-srid=DEFAULT_SRID Optional. Default SRID, needed if it cannot be determined automatically by GDAL.

--default-size=DEFAULT_SIZE Optional. Default size, needed if it cannot be determined automatically by GDAL. Format: <sizex>,<sizey>

--default-extent=DEFAULT_EXTENT Optional. Default extent, needed if it cannot be determined automatically by GDAL. Format: <minx>,<miny>,<maxx>,<maxy>

--default-begin-time Optional. Default begin timestamp when no other EO- metadata is available. The format is ISO-8601.

--default-end-time Optional. Default end timestamp when no other EO- metadata is available. The format is ISO-8601.

--default-footprint Optional. The default footprint in WKT format when no other EO-metadata is available.s

--visible=VISIBLE Optional. Sets the visibility status of all datasets to thegiven boolean value. Defaults to 'True'.

--version show program's version number and exit

-h, --help show this help message and exit

1.6 Recommendations for Operational Installation

Table of Contents

- Recommendations for Operational Installation (page 26)
  - Introduction EOxServer (page 27)
  - Directory Structure (page 27)
  - User Management (page 27)
    - Operating System Users (page 28)
    - Database User (page 28)
    - Django Sysadmin (page 28)
    - Application User Management (page 28)
  - EOxServer Configuration Step-by-step (page 29)
    - Step 1 - Web Server Installation (page 29)
    - Step 2 - Database Backend (page 29)
    - Step 3 - Creating Users and Directories for Instance and Data (page 30)
    - Step 4 - Instance Creation (page 30)
    - Step 5 - Database Setup (page 31)
    - Step 6 - Web Server Integration (page 32)
    - Step 7 - Start Operating the Instance (page 33)

This section provides a set of recommendations and a step-by-step guide for the installation and configuration of EOxServer as an operational system. This guide goes beyond the basic installation presented in previous sections.

Unless stated otherwise this guide considers installing on CentOS GNU/Linux operating systems although the guide is applicable for other distributions as well.

We assume that the reader of this guide *knows* what the presented commands are doing and he/she understands the possible consequences. This guide is intended to help the administrator to setup the EOxServer quickly by extracting the salient information but the administrator must be able to alter the procedure to fit the particular needs of the administered system. We bear no responsibility for any possible harms caused by mindless following of this guide by a non-qualified person.

See Also:

- *Installation* **(page 14)** generic installation procedure for GNU/Linux operating systems.
- *Installation on CentOS* **(page 17)** for specific installation on CentOS.
- *Service Instance Creation and Configuration* **(page 20)** to configure an instance of EOxServer after successful installation.

1.6.1 Introduction EOxServer

When installing and configuring EOxServer a clear distinction should be made between the common EOxServer installation (the installed code implementing the software functionality) and EOxServer instances. An instance is a collection of data and configuration files that enables the deployment of a specific service. A single server will typically contain a single software installation and one or more specific instances.

While the EOxServer installation is straightforward and typically does not require much effort (see the *generic* (page 14) and *CentOS* (page 17) installation guides) the *configuration* (page 20) requires more attention of the administrator and a bit of planning as well.

Closely related to EOxServer is the (possibly large) served EO data. It should be borne in mind, that EOxServer as such is not a data management system, i.e., it can register the stored data but does neither control nor require any specific data storage locations itself. Where and how the data is stored is thus in the responsibility of the administrator.

EOxServer registers the EO data and keeps only the essential metadata (data and full metadata location, geographic extent, acquisition time, etc.) in a database.

1.6.2 Directory Structure

First, the administrator has to decide in which directory each instance should be located. Each of the EOxServer instances is represented by a dedicated directory.

For system wide installation we recommend to create a single specific directory to hold all instances in one location compliant with the filesystem hierarchy standard[50]:

```
/srv/eoxserver
```

Optionally, for user defined instances a folder in the user's home directory is acceptable as well:

```
~/eoxserver
```

Note: We **strongly discourage** to keep the instance configuration in system locations not suited for this purpose such as `/root` or `/tmp`!

A dedicated directory should also be considered for the served EO data, e.g.:

```
/srv/eodata
```

or:

```
~/eodata
```

1.6.3 User Management

The EOxServer administrator has to deal with four different user management subsystems:

- system user (operating system),
- database user (SQL server),

[50] http://www.pathname.com/fhs/pub/fhs-2.3.html#SRVDATAFORSERVICESPROVIDEDBYSYSTEM

- django user (Django user management), and
- application user (e.g., Single Sign On authentication).

Each of them is described hereafter.

Operating System Users

On a typical mutli-user operating system several users exist each of them owning some files and each of them is given some right to access other files and run executables.

In a typical EOxServer setup, the installed executables are owned by the *root* user and when executed they are granted the rights of the invoking process owner. When executed as a WGSI application, the running EOxServer executables run with the same ID as the web server (for Apache server this is typically the *apache* or *www-data* system user). This need to be considered when specifying access rights for the files which are expected to be changed or read by a running application.

The database back-end has usually its own dedicated system user (for PostgreSQL this is typically *postgres*).

Coming back, for EOxServer instances' configuration we recommend both instance and data to be owned by one or (preferably) two distinct system or ordinary users. These users can by existing (e.g., the *apache* user) or new dedicated users.

Note: We **strongly discourage** to keep the EOxService instances (i.e., configuration data) and the served EO data owned by the system administrator (*root*).

Database User

The Django framework (which EOxSerevr is build upon) requires access to a Database Management System (DBMS) which is typically protected by user-name/password based authentication. Specification of these DBMS credential is part of the service instance *configuration* (page 22).

The sole purpose of the DBMS credentials is to access the database.

It should be mentioned that user-name/password is not the only possible way how to secure the database access. The various authentication options for PosgreSQL are covered, e.g., here[51].

Django Sysadmin

The Django framework provides its own user management subsystem. EOxServer uses the Django user management system for granting access to the system administrator to the low level *Admin Web GUI.* (page 51). The Django user management is neither used to protect access to the provided Web Service interfaces nor to restrict access via the command line tools.

Application User Management

EOxServer is based on the assumption that the authentication and authorisation of an operational system would be performed by an external security system (such as the Shibboleth based *Single Sign On* (page 69) infrastructure). This access control would be transparent from EOxServer's point of view.

It is beyond the scope of this document to explain how to configure a Single Sign On (SSO) infrastructure but principally the configuration does not differ from securing plain apache web server.

[51] http://www.postgresql.org/docs/devel/static/auth-pg-hba-conf.html

1.6.4 EOxServer Configuration Step-by-step

The guidelines presented in this section assume a successful installation of EOxServer and of the essential dependencies performed either from the available RPM packages (see CentOS *Installation from RPM Packages* (page 18)) or via the Python Package Index (see *Alternate installation method using pip* (page 19)).

This guide assume that the sudo[52] command is installed and configured on the system.

In case of installation from RPM repositories it is necessary to install the required repositories first:

```
sudo rpm -Uvh http://elgis.argeo.org/repos/6/elgis-release-6-6_0.noarch.rpm
sudo rpm -Uvh http://download.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm
sudo rpm -Uvh http://yum.packages.eox.at/el/eox-release-6-2.noarch.rpm
```

and then install EOxServer's package:

```
sudo yum install EOxServer
```

Step 1 - Web Server Installation

EOxServer is a Django based web application and as such it needs a web server (the simple Django provided server is not an option for an operational system). Any instance of EOxServer receives HTTP requests via the WSGI interface. EOxServer is tested to work with the Apache[53] web server using the WSGI[54] module. The server can be installed using:

```
sudo yum install httpd mod_wsgi
```

EOxServer itself is not equipped by any authentication or authorisation mechanism. In order to secure the resources an external tool must be used to control access to the resources (e.g., the Shibboleth Apache module or the Shibboleth based *Single Sign On* (page 69)).

To start the apache server automatically at the boot-time run following command:

```
sudo chkconfig httpd on
```

The status of the web server can be checked by:

```
sudo service httpd status
```

and if not running the service can be started as follows:

```
sudo service httpd start
```

It is likely the ports offered by the web service are blocked by the firewall. To allow access to port 80 used by the web service it should be mostly sufficient to call:

```
sudo iptables -I INPUT -m state --state NEW -m tcp -p tcp --dport 80 -j ACCEPT
```

Setting up access to any other port than 80 (such as port 443 used by HTTPS) is the same, just change the port number in the previous command.

To make these **iptable** firewall settings permanent (preserved throughout reboots) run:

```
sudo service iptables save
```

Step 2 - Database Backend

EOxServer requires a Database Management System (DBMS) for the storage of its internal data. For an operational system a local or remote installation of PostgreSQL[55] with PostGIS[56] extension is recommended over the

[52] http://www.centos.org/docs/4/4.5/Security_Guide/s3-wstation-privileges-limitroot-sudo.html
[53] http://www.apache.org/
[54] http://en.wikipedia.org/wiki/Web_Server_Gateway_Interface
[55] http://www.postgresql.org/
[56] http://postgis.net/

simple file-based SQLite backend. To install the DBMS run following command:

```
sudo yum install postgresql postgresql-server postgis python-psycopg2
```

PostgreSQL comes with reasonable default settings which are often sufficient. For details on more advanced configuration options (like changing the default database location) see, e.g., PosgreSQL's wiki[57]

On some Linux distributions like recent RHEL and its clones such as CentOS, the PostgreSQL database must be initialized manually by:

```
sudo service postgresql initdb
```

To start the service automatically at boot time run:

```
sudo chkconfig postgresql on
```

You can check if the PostgreSQL database is running or not via:

```
sudo service postgresql status
```

If not start the PostgreSQL server:

```
sudo service postgresql start
```

Once the PostgreSQL deamon is running we have to setup a database template including the required PostGIS extension:

```
sudo -u postgres createdb template_postgis
sudo -u postgres createlang plpgsql template_postgis
PG_SHARE=/usr/share/pgsql
sudo -u postgres psql -q -d template_postgis -f $PG_SHARE/contrib/postgis.sql
sudo -u postgres psql -q -d template_postgis -f $PG_SHARE/contrib/spatial_ref_sys.sql
psql -d postgres psql -q -d template_postgis -c "GRANT ALL ON geometry_columns TO PUBLIC;"
psql -d postgres psql -q -d template_postgis -c "GRANT ALL ON geography_columns TO PUBLIC;"
psql -d postgres psql -q -d template_postgis -c "GRANT ALL ON spatial_ref_sys TO PUBLIC;"
```

Please note that the `PG_SHARE` directory can vary for each Linux distribution or custom PostgreSQL installation. For CentOS `/usr/share/pgsql` happens to be the default location. The proper path can be found, e.g., by:

```
locate contrib/postgis.sql
```

Step 3 - Creating Users and Directories for Instance and Data

To create the users and directories for the EOxServer instances and the served EO Data run the following commands:

```
sudo useradd -r -m -g apache -d /srv/eoxserver -c "EOxServer's administrator" eoxserver
sudo useradd -r -m -g apache -d /srv/eodata -c "EO data provider" eodata
```

For meaning of the used options see documentation of useradd[58] command.

Since we are going to access the files through the Apache web server, for convenience, we set the default group to `apache`. In addition, to make the directories readable by other users run the following commands:

```
sudo chmod o+=rx /srv/eoxserver
sudo chmod o+=rx /srv/eodata
```

Step 4 - Instance Creation

Now it's time to setup a sample instance of EOxServer. Create a new instance e.g., named `instance00`, using the `eoxserver-admin.py` command:

[57] http://wiki.postgresql.org/wiki/Main_Page
[58] http://unixhelp.ed.ac.uk/CGI/man-cgi?useradd+8

```
sudo -u eoxserver mkdir /srv/eoxserver/instance00
sudo -u eoxserver eoxserver-admin.py create_instance instance00 /srv/eoxserver/instance00
```

Now our first bare instance exists and needs to be configured.

Step 5 - Database Setup

As the first to animate the instance it is necessary to setup a database. Assuming the Postgress DBMS is up an running, we start by creating a database user (replace <db_username> by a user-name of your own choice):

```
sudo -u postgres createuser --no-createdb --no-superuser --no-createrole --encrypted --password <
```

The user's password is requested interactively. Once we have the database user we can create the database for our instance:

```
sudo -u postgres createdb --owner <db_username> --template template_postgis --encoding UTF-8 eoxs
```

Where eoxs_instance00 is the name of the new database. As there may be more EOxServer instances, each of them having its own database, it is a good practice to set a DB name containing the name of the instance.

In addition the PostgreSQL access policy must be set to allow access to the newly created database. To get access to the database, insert the following lines (replace <db_username> by your actual DB user-name):

```
local eoxs_instance00 <db_username> md5
```

to the file:

```
/var/lib/pgsql/data/pg_hba.conf
```

Note: This allows *local* database access only.

When inserting the line make sure you put this line **before** the default access policy:

```
local all all ident
```

In case of an SQL server running on a separate machine please see PosgreSQL documentation[59].

The location of the pg_hba.conf file varies from one system to another. In case of troubles to locate this file try, e.g.:

```
sudo locate pg_hba.conf
```

Once we created and configured the database we need to update the EOxServer settings stored, in our case, in file:

```
/srv/eoxserver/instance00/instance00/settings.py
```

Make sure the database is configured in settings.py as follows:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.contrib.gis.db.backends.postgis',
        'NAME': 'eoxs_instance00',
        'USER': '<db_username>',
        'PASSWORD': '<bd_password>',
        'HOST': '', # keep empty for local DBMS
        'PORT': '', # keep empry for local DBMS
    }
}
```

As in our previous examples replace <db_username> and <bd_password> by the proper database user's name and password.

[59] http://www.postgresql.org/docs/devel/static/auth-pg-hba-conf.html

Finally it is time to initialize the database of your first instance by running the following command:

```
sudo -u eoxserver python /srv/eoxserver/instance00/manage.py syncdb
```

The command interactively asks for the creation of the Django system administrator. It is safe to say no and create the administrator's account later by:

```
sudo -u eoxserver python /srv/eoxserver/instance00/manage.py createsuperuser
```

The `manage.py` is the command-line proxy for the management of EOxServer. To avoid repeated writing of this fairly long command make a shorter alias such as:

```
alias eoxsi00="sudo -u eoxserver python /srv/eoxserver/instance00/manage.py"
eoxsi00 createsuperuser
```

Step 6 - Web Server Integration

The remaining task to be performed is to integrate the created EOxServer instance with the Apache web server. As it was already mentioned, the web server access the EOxServer instance through the WSGI interface. We assume that the web server is already configured to load the `mod_wsgi` module and thus it remains to configure the WSGI access point. The proposed configuration is to create the new configuration file `/etc/httpd/conf.d/default_site.conf` with the following content:

```
<VirtualHost *:80>
    # EOxServer instance: instance00
    Alias /instance00 "/srv/eoxserver/instance00/instance00/wsgi.py"
    Alias /instance00_static "/srv/eoxserver/instance00/instance00/static"
    WSGIDaemonProcess ows processes=10 threads=1
    <Directory "/srv/eoxserver/instance00/instance00>
            Options +ExecCGI FollowSymLinks
            AddHandler wsgi-script .py
            WSGIProcessGroup ows
            AllowOverride None
            Order allow,deny
            allow from all
    </Directory>
</VirtualHost>
```

In case there is already a `VirtualHost` section present in `/etc/httpd/conf/httpd.conf` or in any other `*.conf` file included from the `/etc/httpd/conf.d/` directory we suggest to add the configuration lines given above to the appropriate virtual host section.

The `WSGIDaemonProcess` option forces execution of the Apache WSGI in daemon mode using multiple single-thread processes. While the number of daemon processes can be adjusted the number of threads *must* be always set to 1.

On systems such as CentOS, following option must be added to Apache configuration (preferably in `/etc/httpd/conf.d/wsgi.conf`) to allow communication between the Apache server and WSGI daemon (the reason is explained, e.g., here[60]):

```
WSGISocketPrefix run/wsgi
```

Don't forget to adjust the URL configuration in `/srv/eoxserver/instance00/instance00/conf/eoxserver.conf`:

```
[services.owscommon]
http_service_url=http://<you-server-address>/instance00/ows
```

The location and base URL of the static files are specified in the EOxServer instance's `setting.py` file by the `STATIC_ROOT` and `STATIC_URL` options:

[60] http://code.google.com/p/modwsgi/wiki/ConfigurationIssues

```
...
STATIC_ROOT = '/srv/eoxserver/instance00/instance00/static/'
...
STATIC_URL = '/instance00_static/'
...
```

These options are set automatically by the instance creation script.

The static files needed by the EOxServer's web GUI need to be initialized (*collected*) using the following command:

```
alias eoxsi00="sudo -u eoxserver python /srv/eoxserver/instance00/manage.py"
eoxsi00 collectstatic -l
```

To allow the `apache` user to write to the instance log-file make sure the user is permitted to do so:

```
sudo chmod g+w /srv/eoxserver/instance00/instance00/logs/eoxserver.log
```

And now the last thing to do remains to restart the Apache server by:

```
sudo service httpd restart
```

You can check that your EOxServer instance runs properly by inserting the following URL to your browser:

```
http://<you-server-address>/instance00
```

Step 7 - Start Operating the Instance

Now we have a running instance of EOxServer. For different operations such as data registration see *EOxServer Operators' Guide* (page 46).

1.7 Migration

Table of Contents

- Migration (page 33)
  - Migration from 0.2 to 0.3 (page 33)
    - Disclaimer (page 34)
    - Preparatory steps (page 34)
    - Software upgrade (page 34)
      - Django & GDAL (page 34)
      - EOxServer (page 34)
    - Instance migration (page 35)
    - New configuration options (page 35)

Migrating or upgrading an existing EOxServer instance may require to perform several tasks depending on the version numbers. In general upgrading versions with changes in the third digit of the version number only e.g. from 0.2.3 to 0.2.4 doesn't need any special considerations. For all other upgrades please carefully read the relevant sections below.

1.7.1 Migration from 0.2 to 0.3

From version 0.2 to version 0.3 a lot of development effort has been put into EOxServer. Many new features have been implemented and a couple of bugs are now eradicated.

However, if you already have an instance running EOxServer 0.2, this requires a couple of changes to that instance and enables you to configure some new optional configurations aswell.

Disclaimer

Before trying to upgrade EOxServer please make sure to backup your database. This step depends on the actual DBMS you are using for your instance.

Note: If you do not have a lot of datasets registered, or can easily reproduce the current status of your instance, a complete newly created instance may be more failsafe than trying to migrate your instance.

Warning: Because of changes in the database schema, the migration of referenceable datasets does **not** work. Please re-register them once the instance is migrated/re-created.

Preparatory steps

Before you upgrade your software, you will need to perform a database dump. The dump is required to migrate your registered objects to the new database. It is performed with the following call:

```
python manage.py dumpdata core backends coverages --indent=4 > dump.json
```

Unfortunately in some versions `spatialite` produces some output aswell, which has to be removed from the top of the created `dump.json` file.

Software upgrade

Now you are ready to actually perform the software upgrade.

Django & GDAL

The most notable changes concern our technology base: Django & GDAL. EOxServer now relies on features of Django 1.4, so if you still have Django 1.3 or lower installed, please upgrade to (at least) that version. This step, however, depends on how you installed Django in the first place. With `pip` it should be easy as pie/py:

```
pip install Django --upgrade
```

If EOxServer is installed via pip, the upgrade of Django should be done automagically.

Similar to Django, EOxServer now requires at least version 1.7 of the GDAL library respectively its python bindings. GDAL is not explicitly stated in the EOxServer dependencies to allow custom builds and OS specific installations. So you are required to install the minimum required version on your own, via pip, yum, apt, msi or whatever mechanism you prefer.

Please refer to the *EOxServer Dependencies* (page 15) table for details on dependencies.

EOxServer

The upgrade of EOxServer is quite similar to *Installing EOxServer* (page 16). For `pip` you will need the `-U` (`--upgrade`) option:

```
pip install -U EOxServer==0.3
```

or

pip install -U “svn+http://eoxserver.org/svn/branches/0.3”

Instance migration

Now that you have installed your software, there is a small step to perform which requires manual handling to upgrade your instance to the new version of EOxServer.

Please open the `conf/eoxserver.conf` file within your instance directory and locate the `modules` setting of the `[core.registry]` setting. The list entry `eoxserver.resources.coverages.covmgrs` must be corrected to `eoxserver.resources.coverages.managers`.

Now it is time to re-create your database which is done in three steps: deletion of the old database, creation of a new one, and a synchronization. The deletion and creation of the database depend on the database backend used. For SQLite, for example, only the database file needs to be deleted.

The initialization of the database is done via:

```
python manage.py syncdb
```

The old contents of the database can be restored via:

```
python manage.py loaddata dump.json
```

New configuration options

Since version 0.2 a couple of new configuration options are available, most notably for defining output *formats* (page 103) and *CRSs* (page 102). Please have a look at the relevant sections to see how both are set up.

With Django 1.4, EOxServer allows a much more fine-grained logging mechanism defined in `settings.py`. Details can be obtained from the Django documentation[61]. The following is an example of how the logging is set up by default in new EOxServer instances using version 0.3:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'filters': {
        'require_debug_false': {
            '()': 'django.utils.log.RequireDebugFalse'
        }
    },
    'formatters': {
        'simple': {
            'format': '%(levelname)s: %(message)s'
        },
        'verbose': {
            'format': '[%(asctime)s][%(module)s] %(levelname)s: %(message)s'
        }
    },
    'handlers': {
        'eoxserver_file': {
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'filename': join(PROJECT_DIR, 'logs', 'eoxserver.log'),
            'formatter': 'verbose',
            'filters': [],
        }
    },
    'loggers': {
        'eoxserver': {
            'handlers': ['eoxserver_file'],
            'level': 'DEBUG' if DEBUG else 'INFO',
            'propagate': False,
        },
```

[61] https://docs.djangoproject.com/en/dev/topics/logging/#configuring-logging

```
    }
}
```

Another important feature that was introduced in Django 1.4 is the implicit support of time-zones. This can be activated in `settings.py`:

```
USE_TZ = True
```

For a complete list of changes in Django see the official documentation (1.4[62] and 1.5[63]).

1.8 Mailing Lists

Table of Contents

- Mailing Lists (page 36)
  - Users Mailing List (page 36)
  - Dev Mailing List (page 36)

1.8.1 Users Mailing List

The users mailing list is the primary means for EOxServer users and developers to exchange and discuss ideas and potential software improvements, and to ask questions.

Subscribe at http://eoxserver.org/mailman/listinfo/users/. You can later change your subscription information or unsubscribe from the list at this website too.

Here are some points to remember when posting to the list:

- Search the archive at http://eoxserver.org/pipermail/users/ or http://eoxserver.2316974.n4.nabble.com/EOxServer-Users-f4264995.html for your answer first, people get tired of answering the same questions over and over.
- Before posting subscribe to the list by following the procedure described above.
- Post questions to the list by sending an email message to users@eoxserver.org[64].
- Provide version and configuration information for your EOxServer installation, like relevant snippets of your configuration files.
- Always post your responses back to the whole list, as opposed to just the person who replied to your question.
- Questions to the list are usually answered quickly and often by the developers themselves.

1.8.2 Dev Mailing List

A separate mailing list is available for EOxServer developers. It is meant to be used by individuals working on EOxServer source code and related libraries to discuss issues that would not be of interest to the entire users mailing list.

Subscribe at http://eoxserver.org/mailman/listinfo/dev/. You can later change your subscription information or unsubscribe from the list at this website too.

The archive is located at http://eoxserver.org/pipermail/dev/ or http://eoxserver.2316974.n4.nabble.com/EOxServer-Dev-f4265142.html.

[62] https://docs.djangoproject.com/en/dev/releases/1.4/
[63] https://docs.djangoproject.com/en/dev/releases/1.5/
[64] users@eoxserver.org

1.9 Demonstration

Table of Contents

- Demonstration (page 37)
  - GetCapabilities (page 37)
  - DescribeCoverage (page 38)
  - DescribeEOCoverageSet (page 38)
    - Dataset (page 38)
    - StitchedMosaic (page 39)
    - DatasetSeries (page 39)
  - GetCoverage (page 40)

The EOxServer demonstration is an instantiation of the *autotest instance* (page 118) and is based on the Envisat MERIS sample data available here[65].

The configuration includes one DatasetSeries and one StitchedMosaic both combining the three available datasets:

- DatasetSeries (EOId: MER_FRS_1P_reduced) containing the 3 MERIS sample datasets with all 15 radiance bands encoded as uint16 values
- StitchedMosaic (CoverageId: mosaic_MER_FRS_1P_RGB_reduced) containing the 3 MERIS sample datasets reduced to RGB 8-bit

Note, the data has been reduced from 300m resolution to 3000m.

The demonstration tries to show the usage of all available *EO-WCS request parameters* (page 42).

1.9.1 GetCapabilities

GetCapabilities[66]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=GetCapabilities
```

Interesting parts of the repsonse:

- Advertising EO-WCS:

```
<ows:Profile>http://www.opengis.net/spec/WCS_application-profile_earth-observation/1.0/conf/e
```

- The additional EO-WCS operation:

```
<ows:Operation name="DescribeEOCoverageSet">
    <ows:DCP>
        <ows:HTTP>
            <ows:Get xlink:href="http://eoxserver.org/demo_stable/ows?" xlink:type="simple"/>
            <ows:Post xlink:href="http://eoxserver.org/demo_stable/ows?" xlink:type="simple">
                <ows:Constraint name="PostEncoding">
                    <ows:AllowedValues>
                        <ows:Value>XML</ows:Value>
                    </ows:AllowedValues>
                </ows:Constraint>
            </ows:Post>
        </ows:HTTP>
    </ows:DCP>
</ows:Operation>
```

[65] http://earth.esa.int/object/index.cfm?fobjectid=4320
[66] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=GetCapabilities

- The server will limit the number of CoverageDescription elements in DescribeEOCoverageSet responses:

```
<ows:Constraint name="CountDefault">
    <ows:NoValues/>
    <ows:DefaultValue>100</ows:DefaultValue>
</ows:Constraint>
```

- There is a StitchedMosaic available:

```
<wcs:CoverageSummary>
    <wcs:CoverageId>mosaic_MER_FRS_1P_RGB_reduced</wcs:CoverageId>
    <wcs:CoverageSubtype>RectifiedStitchedMosaic</wcs:CoverageSubtype>
</wcs:CoverageSummary>
```

- There is a DatasetSeries available:

```
<wcseo:DatasetSeriesSummary>
    <ows:WGS84BoundingBox>
        <ows:LowerCorner>-3.43798100 32.26454100</ows:LowerCorner>
        <ows:UpperCorner>27.96859100 46.21844500</ows:UpperCorner>
    </ows:WGS84BoundingBox>
    <wcseo:DatasetSeriesId>MER_FRS_1P_reduced</wcseo:DatasetSeriesId>
    <gml:TimePeriod gml:id="MER_FRS_1P_reduced_timeperiod">
        <gml:beginPosition>2006-08-16T09:09:29</gml:beginPosition>
        <gml:endPosition>2006-08-30T10:13:06</gml:endPosition>
    </gml:TimePeriod>
</wcseo:DatasetSeriesSummary>
```

1.9.2 DescribeCoverage

DescribeCoverage StitchedMosaic[67]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=DescribeCoverage&
    coverageid=mosaic_MER_FRS_1P_RGB_reduced
```

DescribeCoverage Dataset[68]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=DescribeCoverage&
    coverageid=MER_FRS_1PNPDE20060822_092058_000001972050_00308_23408_0077_uint16_reduced_compres
```

1.9.3 DescribeEOCoverageSet

Dataset

DescribeEOCoverageSet Dataset[69]:

```
http://eoxserver.org/demo_stable/ows?
        service=wcs&
        version=2.0.0&
        request=DescribeEOCoverageSet&
        EOId=MER_FRS_1PNPDE20060822_092058_000001972050_00308_23408_0077_uint16_reduced_compresse
```

[67] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=DescribeCoverage&coverageid=mosaic_MER_FRS_1P_RGB_reduced
[68] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=DescribeCoverage&coverageid=MER_FRS_1PNPDE20060822_092058_0000019
[69] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=DescribeEOCoverageSet&EOId=MER_FRS_1PNPDE20060822_092058_000001

StitchedMosaic

DescribeEOCoverageSet StitchedMosaic (4 Datasets returned)[70]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=DescribeEOCoverageSet&
    EOId=mosaic_MER_FRS_1P_RGB_reduced
```

DescribeEOCoverageSet StitchedMosaic, subset in time (3 Datasets returned)[71]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=DescribeEOCoverageSet&
    EOId=mosaic_MER_FRS_1P_RGB_reduced&
    subset=phenomenonTime("2006-08-01","2006-08-22T09:22:00Z")
```

DescribeEOCoverageSet StitchedMosaic, subset in Lat and Long, containment contains (1 Dataset returned)[72]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=DescribeEOCoverageSet&
    EOId=mosaic_MER_FRS_1P_RGB_reduced&
    subset=Lat,http://www.opengis.net/def/crs/EPSG/0/4326(32,47)&
    subset=Long,http://www.opengis.net/def/crs/EPSG/0/4326(11,33)&
    containment=contains
```

DescribeEOCoverageSet StitchedMosaic, returned CoverageDescriptions limited to 2[73]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=DescribeEOCoverageSet&
    EOId=mosaic_MER_FRS_1P_RGB_reduced&
    count=2
```

DatasetSeries

DescribeEOCoverageSet DatasetSeries (5 Datasets returned)[74]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=describeeocoverageset&
    eoid=MER_FRS_1P_reduced
```

DescribeEOCoverageSet DatasetSeries, trim subset in time (4 Datasets returned)[75]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=describeeocoverageset&
```

[70] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=DescribeEOCoverageSet&EOId=mosaic_MER_FRS_1P_RGB_reduced

[71] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=DescribeEOCoverageSet&EOId=mosaic_MER_FRS_1P_RGB_reduced&subset=p
08-01%22,%222006-08-22T09:22:00Z%22)

[72] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=DescribeEOCoverageSet&EOId=mosaic_MER_FRS_1P_RGB_reduced&subset=l

[73] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=DescribeEOCoverageSet&EOId=mosaic_MER_FRS_1P_RGB_reduced&count=2

[74] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=describeeocoverageset&eoid=MER_FRS_1P_reduced

[75] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=describeeocoverageset&eoid=MER_FRS_1P_reduced&subset=phenomenonTime(
08-01%22,%222006-08-22T09:22:00Z%22)

```
    eoid=MER_FRS_1P_reduced&
    subset=phenomenonTime("2006-08-01","2006-08-22T09:22:00Z")
```

DescribeEOCoverageSet DatasetSeries, slice subset in time (2 Dataset returned)[76]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=describeeocoverageset&
    eoid=MER_FRS_1P_reduced&
    subset=phenomenonTime("2006-08-22T09:20:58Z")
```

DescribeEOCoverageSet DatasetSeries, trim subset in time trim, containment contains (2 Dataset returned)[77]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=describeeocoverageset&
    eoid=MER_FRS_1P_reduced&
    subset=phenomenonTime("2006-08-01","2006-08-22T09:22:00Z")&
    containment=contains
```

DescribeEOCoverageSet DatasetSeries, subset in Lat and Long (5 Datasets returned)[78]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=describeeocoverageset&
    eoid=MER_FRS_1P_reduced&
    subset=Lat,http://www.opengis.net/def/crs/EPSG/0/4326(32,47)&
    subset=Long,http://www.opengis.net/def/crs/EPSG/0/4326(11,33)
```

DescribeEOCoverageSet DatasetSeries, subset in Lat and Long, containment contains (2 Dataset returned)[79]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=describeeocoverageset&
    eoid=MER_FRS_1P_reduced&
    subset=Lat,http://www.opengis.net/def/crs/EPSG/0/4326(32,47)&
    subset=Long,http://www.opengis.net/def/crs/EPSG/0/4326(11,33)&
    containment=contains
```

1.9.4 GetCoverage

GetCoverage StitchedMosaic, full (GML incl. contributingFootprint & GeoTIFF)[80]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=GetCoverage&
    coverageid=mosaic_MER_FRS_1P_RGB_reduced&
    format=image/tiff&
    mediatype=multipart/mixed
```

GetCoverage Dataset, full (GML & GeoTIFF)[81]:

[76] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=describeeocoverageset&eoid=MER_FRS_1P_reduced&subset=phenomenonTime(
08-22T09:20:58Z%22)

[77] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=describeeocoverageset&eoid=MER_FRS_1P_reduced&subset=phenomenonTime(
08-01%22,%222006-08-22T09:22:00Z%22)&containment=contains

[78] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=describeeocoverageset&eoid=MER_FRS_1P_reduced&subset=Lat,http://www.op

[79] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=describeeocoverageset&eoid=MER_FRS_1P_reduced&subset=Lat,http://www.op

[80] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=GetCoverage&coverageid=mosaic_MER_FRS_1P_RGB_reduced&format=image

[81] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=GetCoverage&coverageid=MER_FRS_1PNPDE20060822_092058_00000197205

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=GetCoverage&
    coverageid=MER_FRS_1PNPDE20060822_092058_000001972050_00308_23408_0077_uint16_reduced_compress
    format=image/tiff&
    mediatype=multipart/mixed
```

GetCoverage Dataset, subset in pixels[82]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=GetCoverage&
    coverageid=MER_FRS_1PNPDE20060822_092058_000001972050_00308_23408_0077_uint16_reduced_compress
    format=image/tiff&
    mediatype=multipart/mixed&
    subset=x(100,200)&
    subset=y(300,400)
```

GetCoverage Dataset, subset in epsg 4326[83]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=GetCoverage&
    coverageid=MER_FRS_1PNPDE20060822_092058_000001972050_00308_23408_0077_uint16_reduced_compress
    format=image/tiff&
    mediatype=multipart/mixed&
    subset=Lat,http://www.opengis.net/def/crs/EPSG/0/4326(40,41)&
    subset=Long,http://www.opengis.net/def/crs/EPSG/0/4326(17,18)
```

GetCoverage Dataset, full, OutputCRS epsg 3035[84]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=GetCoverage&
    coverageid=MER_FRS_1PNPDE20060822_092058_000001972050_00308_23408_0077_uint16_reduced_compress
    format=image/tiff&
    mediatype=multipart/mixed&
    OutputCRS=http://www.opengis.net/def/crs/EPSG/0/3035
```

GetCoverage Dataset, full, size 200x200[85]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=GetCoverage&
    coverageid=MER_FRS_1PNPDE20060822_092058_000001972050_00308_23408_0077_uint16_reduced_compress
    format=image/tiff&
    mediatype=multipart/mixed&
    size=x(200)&size=y(200)
```

GetCoverage Dataset, full, size 200x400[86]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
```

[82] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=GetCoverage&coverageid=MER_FRS_1PNPDE20060822_092058_00000197205
[83] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=GetCoverage&coverageid=MER_FRS_1PNPDE20060822_092058_00000197205
[84] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=GetCoverage&coverageid=MER_FRS_1PNPDE20060822_092058_00000197205
[85] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=GetCoverage&coverageid=MER_FRS_1PNPDE20060822_092058_00000197205
[86] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=GetCoverage&coverageid=MER_FRS_1PNPDE20060822_092058_00000197205

```
    request=GetCoverage&
    coverageid=MER_FRS_1PNPDE20060822_092058_000001972050_00308_23408_0077_uint16_reduced_compres
    format=image/tiff&
    mediatype=multipart/mixed&
    size=x(200)&size=y(400)
```

GetCoverage Dataset, subset in bands[87]:

```
http://eoxserver.org/demo_stable/ows?
    service=wcs&
    version=2.0.0&
    request=GetCoverage&
    coverageid=MER_FRS_1PNPDE20060822_092058_000001972050_00308_23408_0077_uint16_reduced_compres
    format=image/tiff&
    mediatype=multipart/mixed&
    rangesubset=1,2,3
```

1.10 EO-WCS Request Parameters

Table of Contents

- EO-WCS Request Parameters (page 42)
  - GetCapabilities (page 42)
  - DescribeCoverage (page 43)
  - DescribeEOCoverageSet (page 43)
  - GetCoverage (page 44)

The following tables provide an overview over the available EO-WCS request parameters for each operation supported by EOxServer.

Please see EOxServer's *Demonstration* (page 37) for complete sample requests.

1.10.1 GetCapabilities

Table: "*EO-WCS GetCapabilities Request Parameters* (page 42)" below lists all parameters that are available with Capabilities requests.

[87] http://eoxserver.org/demo_stable/ows?service=wcs&version=2.0.0&request=GetCoverage&coverageid=MER_FRS_1PNPDE20060822_092058_00000197205

Table 1.3: EO-WCS GetCapabilities Request Parameters

Parameter	Description / Subparameter	Allowed value(s) / Example	Mandatory (M) / Optional (O)
→ service	Requested service	WCS	M
→ request	Type of request	GetCapabilities	M
→ version [88]	Version number	2.0.1	O
→ acceptVersions [1]	Prioritized sequence of one or more specification versions accepted by the client, with preferred versions listed first (first supported version will be used) version1[,version2[,...]]	2.0.1, 1.1.2, 1.0.0	O
→ sections	Comma-separated unordered list of zero or more names of zero or more names of sections of service metadata document to be returned in service metadata document. Request only certain sections of Capabilities Document section1[,section2[,...]]	• DatasetSeriesSummary • CoverageSummary • Contents • All • ServiceIdentification • ServiceProvider • OperationsMetadata • Languages	O
→ updateSequence	Date of last issued GetCapabilities request; to receive new document only if it has changed since	"2013-05-08"	O

1.10.2 DescribeCoverage

Table: "*EO-WCS DescribeCoverage Request Parameters* (page 43)" below lists all parameters that are available with DescribeCoverage requests.

Table 1.4: EO-WCS DescribeCoverage Request Parameters

Parameter	Description / Subparameter	Allowed value(s) / Example	Mandatory (M) / Optional (O)
→ service	Requested service	WCS	M
→ request	Type of request	DescribeCoverage	M
→ version [1]	Version number	2.0.1	M
→ coverageId	NCName(s): • valid coverageID of a Dataset • valid coverageID of a StichedMosaic		M

1.10.3 DescribeEOCoverageSet

Table: "*EO-WCS DescribeEOCoverageSet Request Parameters* (page 43)" below lists all parameters that are available with DescribeEOCoverageSet requests.

Table 1.5: EO-WCS DescribeEOCoverageSet Request Parameters

Parameter	Description / Subparameter	Allowed value(s) / Example	Mandatory (M) / Optional (O)
→ service	Requested service	WCS	M
→ request	Type of request	DescribeEOCoverageSet	M
→ version [1]	Version number	2.0.1	M
→ eoId	Valid eoId: • using the coverageId of a Datatset • using the eoId of a DatatsetSeries • using the coverageId of a StitchedMosaic		M
→ subset	Allows to constrain the request in each dimensions and define how these parameters are applied. The spatial constraint is expressed in WGS84, the temporal constraint in ISO 8601. Spatial trimming: Name of an coverage axis (Long or Lat) Temporal trimming: phenomenonTime Plus optional either: • containment = overlaps (default) • containment = contains Any combination thereof (but each value only once per request)	• Lat,http://www.opengis.net/def/crs/EPSG/0/4326(32,47) • Long,http://www.opengis.net/def/crs/EPSG/0/4326(11,33)& • phenomenonTime(“2006-08-01”, “2006-08-22T09:22:00Z”) • Lat,http://www.opengis.net/def/crs/EPSG/0/4326(32,47)& Long,http://www.opengis.net/def/crs/EPSG/0/4326(11,33)& phenomenonTime(“2006-08-01”, “2006-08-22T09:22:00Z”)& containment=contains	O
→ containment	see *subset* parameter	• overlaps (default) • contains	O
→ section	see GetCapabilities	• DatasetSeriesSummary • CoverageSummary • All	O
→ count	Limits the maximum number of DatasetDescriptions returned in the EOCoverageSetDescription.	10	O

1.10.4 GetCoverage

Table: “*EO-WCS GetCoverage Request Parameters* (page 44)” below lists all parameters that are available with GetCoverage requests.

Table 1.6: EO-WCS GetCoverage Request Parameters

Parameter	Description / Subparameter	Allowed value(s) / Example	Mandatory (M) / Optional (O)
→ service	Requested service	WCS	M
→ request	Type of request	GetCoverage	M
→ version [1]	Version number	2.0.1	M
→ coverageId	NCName(s): • valid coverageID of a Dataset • valid coverageID of a StichedMosaic		M
→ format	Requested format of coverage to be returned, currently: • image/tiff • image/jpeg • image/png • image/gif	image/tiff	M
→ mediatype	Coverage delivered directly as image file or enclosed in GML structure • not present or • multipart/mixed	multipart/mixed	O
→ subset	Trimming of coverage dimension (no slicing allowed!) • the label of a coverage axis – plus either: ∗ pixel coordinates ∗ without CRS (→ original projection) ∗ with CRS (→ reprojecting)	• x(400,200) • Lat(12,14) • Long,http://www.opengis.net/def/crs/EPSG/0/4326(17,17.4)	O
→ rangesubset	Subsetting in the range domain (e.g. Band-Subsetting).	• 1,2,3 • Blue,Green,Red	O
→ outputcrs	CRS for the requested output coverage • not present or • CRS	http://www.opengis.net/def/crs/EPSG/0/3035	O
• → size or • → resolution	Mutually exclusive per axis, either: • integer dimension of the requested coverage (per axis) • resolution of one pixel (per axis)	• size=Long(20) • size=x(50) • resolution=long(0.01) • resolution=y(0.3)	O
→ interpolation [89]	Interpolation method to be used • nearest (default)	bilinear	O

1.11 EOxServer Operators' Guide

Table of Contents

- EOxServer Operators' Guide (page 46)
  - Basic Concepts (page 46)
  - Storage Backends (page 47)
    - Local (page 47)
    - FTP Repositories (page 47)
    - Rasdaman Databases (page 47)
  - Coverages (page 47)
    - Range Types (page 48)
    - EO Metadata (page 48)
    - Rectified Datasets (page 48)
    - Referenceable Datasets (page 48)
    - Rectified Stitched Mosaics (page 49)
    - Dataset Series (page 49)
  - Data Preparation and Supported Data Formats (page 49)
    - Raster Data Formats (page 49)
    - Raster Data Preparation (page 49)
    - Metadata Formats (page 50)
    - Metadata Preparation (page 51)
  - Admin Client (page 51)
    - Creating a custom Range Type (page 51)
    - Linking to a Local Path (page 55)
    - Creating a Data Package (page 55)
    - Adding Data Sources (page 55)
    - Creating a Dataset Series (page 55)
  - Command Line Tools (page 55)
    - eoxserver-admin.py create_instance (page 55)
    - eoxs_register_dataset (page 55)
    - eoxs_deregister_dataset (page 55)
    - Updating Datasets (page 59)
    - eoxs_add_dataset_series (page 59)
    - eoxs_synchronize (page 59)
    - eoxs_insert_into_series (page 60)
    - eoxs_remove_from_series (page 61)
    - eoxs_check_id (page 61)
    - Range Type Handling (page 61)
  - Performance (page 62)

1.11.1 Basic Concepts

EOxServer is all about coverages - see the *EOxServer Basics* (page 1) for a short description.

In the language of the OGC Abstract Specification, coverages are mappings from a domain set that is related to some area of the Earth to a range set. So, the data model for coverages contains information about the structure of the domain set and of the range set (the so-called Range Type).

In the *Coverages* (page 47) section below you find more detailed information about what data and metadata is stored by EOxServer.

The actual data EOxServer deals with can be stored in different ways. These storage facilities are discussed below in the section on *Storage Backends* (page 47).

Operators have different possibilities to ingest data into the system. Using the *Admin Client* (page 51), you can edit the contents of the EOxServer database. Especially for batch processing using the *Command Line Tools* (page 55)

may be preferable.

1.11.2 Storage Backends

EOxServer supports different kinds of data stores for coverage data:

- as an image file stored on the local file system
- as an image file stored on a remote FTP server
- as a raster array in a rasdaman[91] database

These different ways of storing data are called Storage Backends. Internally, EOxServer uses the term Location as an abstraction for the different ways access to the data is described. Each storage backend has its own type of Locations that is described in the following subsections.

Local

A path on the local filesystem is the most straightforward way to define the location of a resource. You can use relative paths as well as absolute paths. Please keep in mind that relative paths are interpreted as being relative to the working directory of the process EOxServer runs in. For Apache processes, for instance, this is usually the root directory `/`.

FTP Repositories

EOxServer allows to define locations on a remote FTP server. This is useful if you do not want to transfer a whole large archive to the machine EOxServer runs on. In that case you can define a remote path that consists of information about the FTP server and the path relative to the root directory of the FTP repository.

An FTP Storage record - as it is called in EOxServer - contains the URL of the server and optional port, username and password entries.

Resources stored on an FTP server are transferred only when they are needed. There is however a cache for transferred files on the machine EOxServer runs on.

Rasdaman Databases

The third backend supported at the moment are rasdaman[92] databases. A rasdaman location consists of rasdaman database connection information and the collection of the corresponding resource.

The rasdaman storage records contain hostname, port, database name, user and password entries.

The data is retrieved from the database using the rasdaman GDAL driver (see *Installation* (page 14) for further information).

1.11.3 Coverages

EOxServer coverages fall into three main categories:

- *Rectified Datasets* (page 48)
- *Referenceable Datasets* (page 48)
- *Rectified Stitched Mosaics* (page 49)

In addition there is the *Dataset Series* (page 49) type which corresponds to an inhomogeneous collection of coverages.

[91] http://www.rasdaman.org
[92] http://www.rasdaman.org

Range Types

Every coverage has a range type describing the structure of the data. Each range type has a given data type whereas the following data types are supported:

Data Type Name	Data Type Value
Unknown	0
Byte	1
UInt16	2
Int16	3
UInt32	4
Int32	5
Float32	6
Float64	7
CInt16	8
CInt32	9
CFloat32	10
CFloat64	11

A range type contains of one or more bands. For each band you may specify a name, an identifier and a definition that describes the property measured (e.g. radiation). Furthermore, you can define nil values for each band (i.e. values that indicate that there is no measurement at the given position).

This range type metadata is used in the coverage description metadata that is returned by WCS operations and for configuring WMS layers.

Note that WMS supports only one data type (Byte) and only Grayscale and RGB output. Any other range types will be mapped to these: for single-band coverages, Grayscale output is generated and RGB output using the first three bands for all others. Automatic scaling is applied when mapping from another data type to Byte. That means the minimum-maximum interval for the given subset of the coverage is computed and mapped to the 0-255 interval supported by the Byte data type.

If you want to view other band combinations than the default ones, you can use the EO-WMS features implemented by EOxServer. For each coverage, an additional layers called `<coverage id>_bands` is provided for WMS 1.3. Using this layer and the `DIM_BAND` KVP parameter you can select another combination of bands (either 1 or 3 bands).

EO Metadata

Earth Observation (EO) metadata records are stored for each EO coverage and Dataset Series. They contain the acquisition begin and end time as well as the footprint of the coverage. The footprint is a polygon that describes the outlines of the area covered by the coverage.

Rectified Datasets

Rectified Datasets are EO coverages whose domain set is a rectified grid i.e. which are having a regular spacing in projected or geographic CRS. In practice, this applies to ortho-rectified satellite data. The rectified grid is described by the EPSG SRID of the coordinate reference system, the extent and pixel size of the coverage.

Rectified Datasets can be added to Dataset Series and Rectified Stitched Mosaics.

Referenceable Datasets

Referenceale Datasets are EO coverages whose domain set is a referenceable grid i.e. which are not rectified, but are associated with (one or more) coordinate transformation which relate the image to a projected or geographic CRS. That means that there is some general transformation between the grid cell coordinates and coordinates in an Earth-bound spatial reference system. This applies for satellite data in its original geometry.

At the moment, EOxServer supports only referenceable datasets that contain ground control points (GCPs) in the data files. Simple approximative transformations based on these GCPs are used to generate rectified views on the data for WMS and to calculate subset bounds for WCS GetCoverage requests. Note that these transformations can be very inaccurate in comparison to an actual ortho-rectification of the coverage.

Rectified Stitched Mosaics

Rectified Stitched Mosaics are EO coverages that are composed of a set of homogeneous Rectified Datasets. That means, the datasets must have the same range type and their domain sets must be subsets of the same rectified grid.

When creating a Rectified Stitched Mosaic a homogeneous coverage is generated from the contained Rectified Datasets. Where datasets overlap the most recent one as indicated by the acquisition timestamps in the EO metadata is shown on top hiding the others.

Dataset Series

Any Rectified and Referenceable Datasets can be organized in Dataset Series. Multiple datasets which are spatially and/or temporally overlapping can be organized in a Dataset Series. Furthermore Stitched Mosaics can also be organized in Dataset Series.

1.11.4 Data Preparation and Supported Data Formats

EO Coverages consist of raster data and metadata. The way this data is stored can vary considerably. EOxServer supports a wide range of different data and metadata formats which are described below.

Raster Data Formats

EOxServer uses the GDAL[93] library for raster data handling. So does MapServer[94] whose scripting API (MapScript) is used by EOxServer as well. In principle, any format supported by GDAL[95] can be read by EOxServer and registered in the database.

There is, however, one caveat. Most data formats are composed of bands which contain the data (e.g. ENVISAT N1, GeoTIFF, JPEG 2000). But some data formats (notably netCDF and HDF) have a different substructure: subdatasets. At the moment these data formats are only supported for data output, but not for data input.

For more information on configuration of supported raster file formats read "*Supported Raster File Formats and Their Configuration* (page 103)".

Raster Data Preparation

Usually, raster data does not need to be prepared in a special way to be ingested into EOxServer.

If the raster data file is structured in subdatasets, though, as is the case with netCDF and HDF, you will have to convert it to another format. You can use the `gdal_translate` command for that task:

```
$ gdal_translate -of <Output Format> <Input File Name> <Output File Name>
```

You can display the list of possible output formats with:

```
$ gdalinfo --formats
```

[93] http://www.gdal.org
[94] http://www.mapserver.org
[95] http://www.gdal.org/formats_list.html

For automatic registration of datasets, EOxServer relies on the geospatial metadata stored with the dataset, notably the EPSG ID of the coordinate reference system and the geospatial extent. In some cases the CRS information in the dataset does not contain the EPSG code. If you are using the command line interfaces of EOxServer you can specify an SRID with the `--default-srid` option. As an alternative you can try to add the corresponding information to the dataset, e.g. with:

```
$ gdal_translate -a_srs "+init=EPSG:<SRID>" <Input File Name> <Output File Name>
```

For performance reasons, especially if you are using WMS, you might also consider to add overviews to the raster data files using the `gdaladdo` command (documentation[96]). Note however that this is supported only by a few formats like GeoTIFF and JPEG2000.

Metadata Formats

There are two possible ways to store metadata: the first one is to store it in the data file itself, the second one is to store it in an accompanying metadata file.

Only a subset of the supported raster data formats are capable of storing metadata in the data file. Furthermore there are no standards defining the semantics of the metadata for generic formats like GeoTIFF. For mission specific formats, however, there are thorough specifications in place.

EOxServer supports reading basic metadata from ENVISAT N1 files and files that have a similar metadata structure (e.g. a GeoTIFF file with the same metadata tags).

For other formats metadata files have to be provided. EOxServer supports two XML-based formats:

- OGC Earth Observation Profile for Observations and Measurements (OGC 10-157r2)
- an EOxServer native format

Here is an example for EO O&M:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<eop:EarthObservation gml:id="eop_ASA_WSM_1PNDPA20050331_075939_000000552036_00035_16121_0775" xm
  <om:phenomenonTime>
    <gml:TimePeriod gml:id="phen_time_ASA_WSM_1PNDPA20050331_075939_000000552036_00035_16121_0775
      <gml:beginPosition>2005-03-31T07:59:36Z</gml:beginPosition>
      <gml:endPosition>2005-03-31T08:00:36Z</gml:endPosition>
    </gml:TimePeriod>
  </om:phenomenonTime>
  <om:resultTime>
    <gml:TimeInstant gml:id="res_time_ASA_WSM_1PNDPA20050331_075939_000000552036_00035_16121_0775
      <gml:timePosition>2005-03-31T08:00:36Z</gml:timePosition>
    </gml:TimeInstant>
  </om:resultTime>
  <om:procedure />
  <om:observedProperty />
  <om:featureOfInterest>
    <eop:Footprint gml:id="footprint_ASA_WSM_1PNDPA20050331_075939_000000552036_00035_16121_0775"
      <eop:multiExtentOf>
        <gml:MultiSurface gml:id="multisurface_ASA_WSM_1PNDPA20050331_075939_000000552036_00035_1
          <gml:surfaceMember>
            <gml:Polygon gml:id="polygon_ASA_WSM_1PNDPA20050331_075939_000000552036_00035_16121_0
              <gml:exterior>
                <gml:LinearRing>
                  <gml:posList>-33.03902600 22.30175400 -32.53056000 20.09945700 -31.98492200 17.
                </gml:LinearRing>
              </gml:exterior>
            </gml:Polygon>
          </gml:surfaceMember>
        </gml:MultiSurface>
      </eop:multiExtentOf>
```

[96] http://www.gdal.org/gdaladdo.html

```
    </eop:Footprint>
  </om:featureOfInterest>
  <om:result />
  <eop:metaDataProperty>
    <eop:EarthObservationMetaData>
      <eop:identifier>ASA_WSM_1PNDPA20050331_075939_000000552036_00035_16121_0775</eop:identifier
      <eop:acquisitionType>NOMINAL</eop:acquisitionType>
      <eop:status>ARCHIVED</eop:status>
    </eop:EarthObservationMetaData>
  </eop:metaDataProperty>
</eop:EarthObservation>
```

The native format has the following structure:

```
<Metadata>
    <EOID>some_unique_eoid</EOID>
    <BeginTime>YYYY-MM-DDTHH:MM:SSZ</BeginTime>
    <EndTime>YYYY-MM-DDTHH:MM:SSZ</EndTime>
    <Footprint>
        <Polygon>
            <Exterior>Mandatory - some_pos_list as all-space-delimited Lat Lon pairs (closed polyg
            [
             <Interior>Optional - some_pos_list as all-space-delimited Lat Lon pairs (closed polyg
             ...
            ]
        </Polygon>
    </Footprint>
</Metadata>
```

The automatic registration tools for EOxServer (see below under *Command Line Tools* (page 55)) expect that the metadata file accompanying the data file has the same name with `.xml` as extension.

Metadata Preparation

EOxServer provides a tool to extract metadata from ENVISAT N1 files and convert it to EO O&M format. It can be found under `tools/gen_envisat_md.py`. It accepts an input path to an N1 file and stores the resulting XML file under the same path with the appropriate file name (i.e. replacing the `.N1` extension with `.xml`). Note that EOxServer must be in the Python path and the environment variable `DJANGO_SETTINGS_MODULE` must be set and point to a properly configured EOxServer instance.

1.11.5 Admin Client

The Admin Client is accessible via any standard web browser at the path */admin* under the URL your instance is deployed or simply by following the *admin* link on the start page. *Deployment* (page 23) provides more details.

Use the username and password you provided during the *syncdb* step as described in the *Service Instance Creation and Configuration* (page 20) section.

Creating a custom Range Type

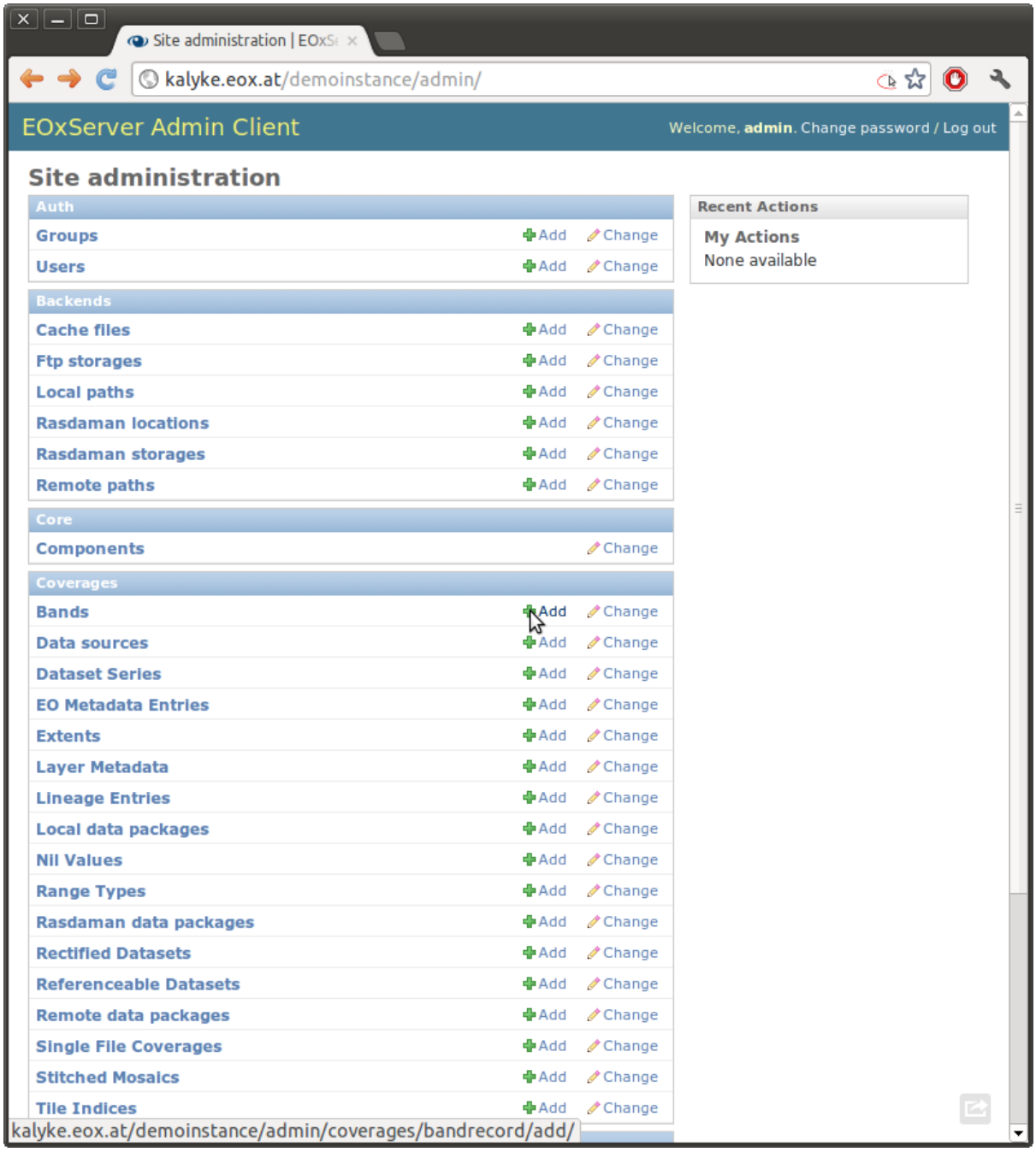
Before registering any data in EOxServer some vital information on the datasets has to be provided. Detailed information regarding the kind of data stored can be defined in the Range Type. A Range Type is a collection of bands which themselves are assigned to a specifig Data Type (see *Range Types* (page 48)).

A simple standard PNG for example holds 4 bands (RGB + Alpha) each of them able to store 8 bit data. Therefore the Range Type would have to be defined with four bands (red, green, blue, alpha) each of them having 'Byte' as Data Type.

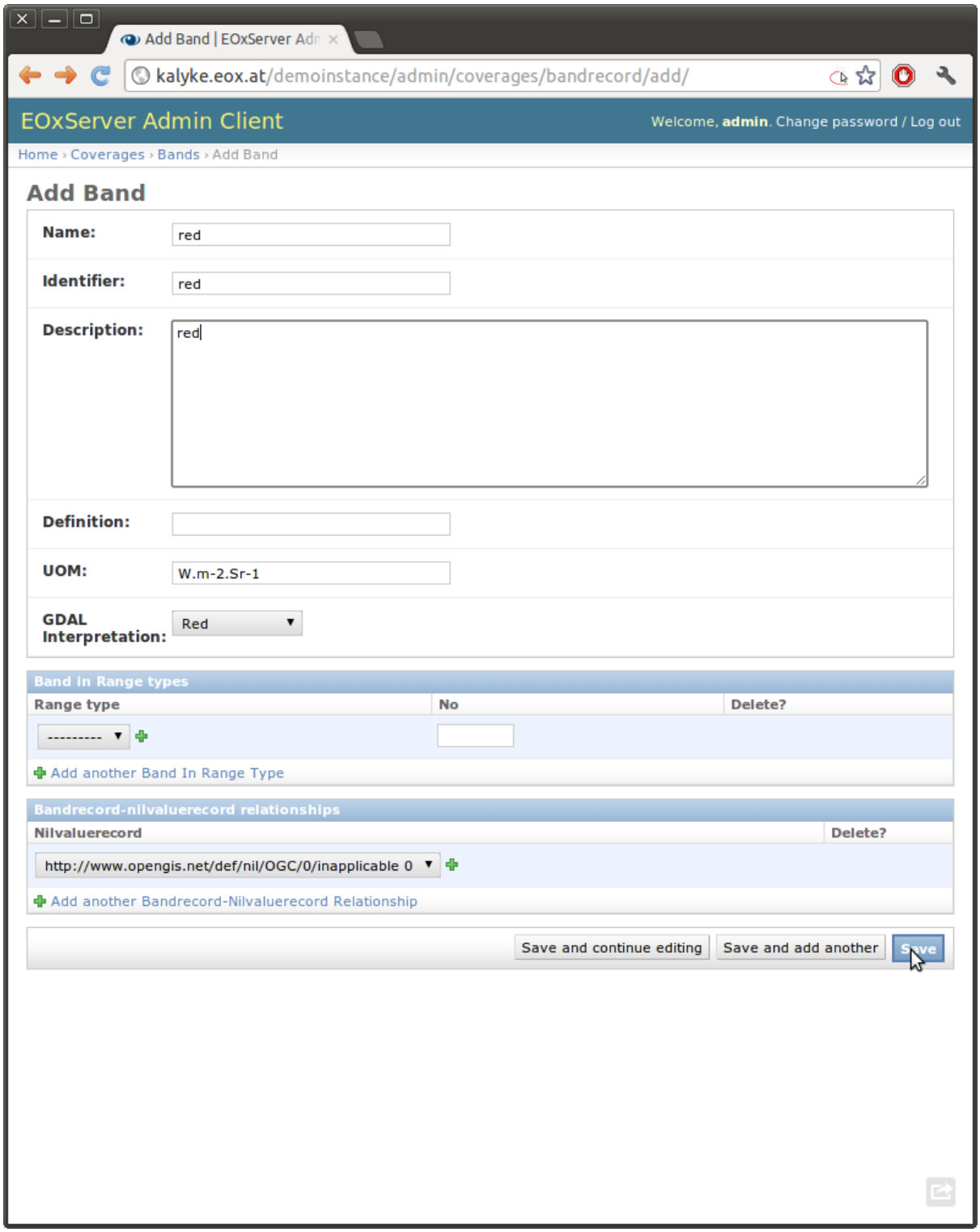
In our example we use the reduced MERIS RGB data provided in the autotest instance. gdalinfo provides us with the most important information:

```
[...]
Band 1 Block=541x5 Type=Byte, ColorInterp=Red
Band 2 Block=541x5 Type=Byte, ColorInterp=Green
Band 3 Block=541x5 Type=Byte, ColorInterp=Blue
```

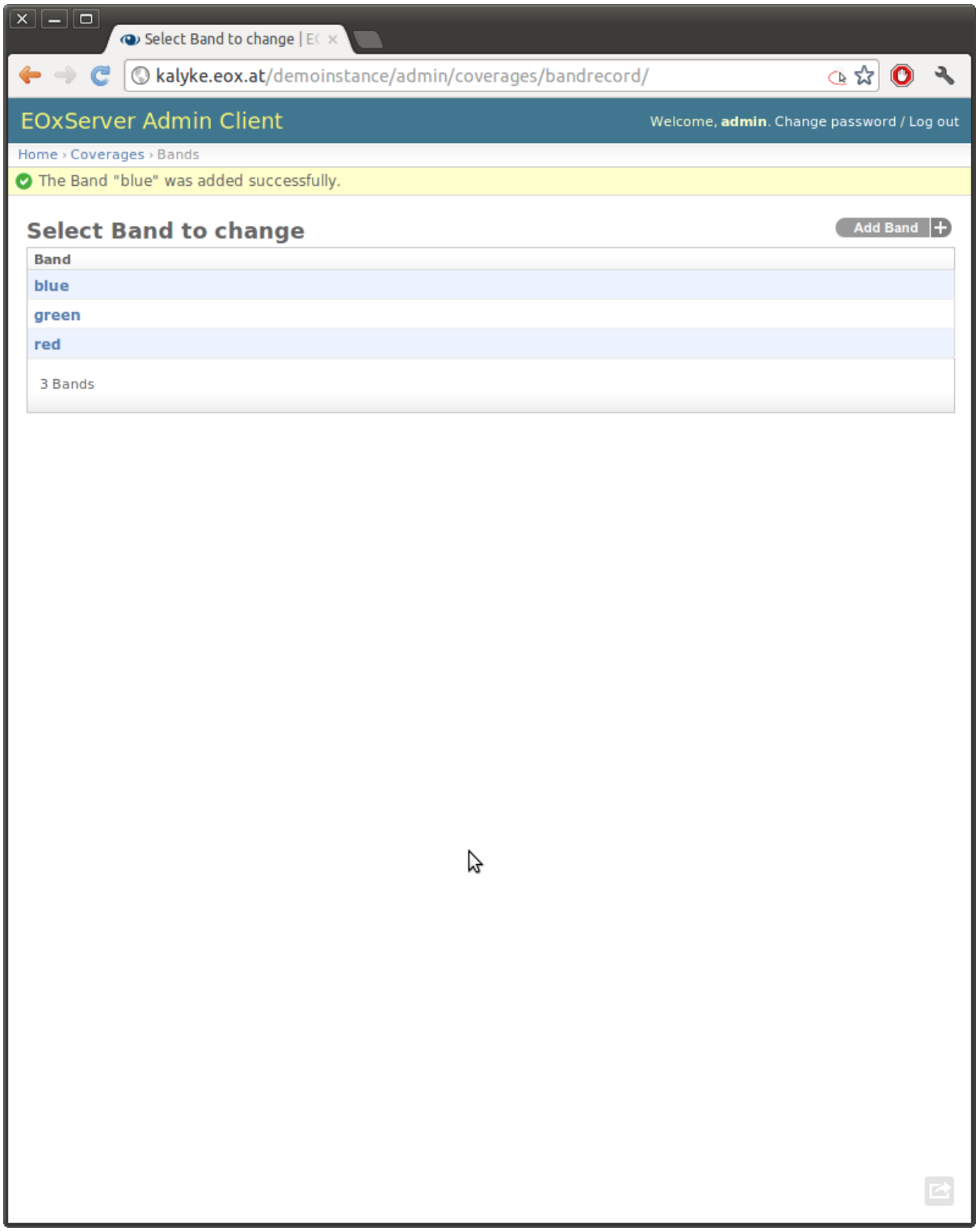
First, we have to define the bands by clicking "add" next to "Bands" in the Admin interface. In "Name", "Identifier" and "Description" you can enter the same content for now. The default "Definition" value for now can be "http://www.opengis.net/def/property/OGC/0/Radiance". "UOM" stands for "unit of measurement" which in our case is radiance defined by the value "W.m-2.Sr-1". For displaying the data correctly it is recommended to assign the respective value in "GDAL Interpretation". NoData values can be defined by adding a "Nilvaluerecord". (see screenshot)



After adding also the green and blue band we can proceed defining the Range Type. After providing the new Range Type with a name you will have to assign a Data Type of all data. In our case we select "Byte". Below we now have to add our three Bands by clicking on the lowermost "+" icon. The important part here is to assign each


Add Band | EOxServer Adr
kalyke.eox.at/demoinstance/admin/coverages/bandrecord/add/
EOxServer Admin Client
Welcome, admin. Change password / Log out
Home › Coverages › Bands › Add Band
Add Band
Name:
red
Identifier:
red
Description:
red
Definition:
UOM:
W.m-2.Sr-1
GDAL Interpretation:
Red
Band in Range types
Range type
No
Delete?

Add another Band In Range Type
Bandrecord-nilvaluerecord relationships
Nilvaluerecord
Delete?
http://www.opengis.net/def/nil/OGC/0/inapplicable 0
Add another Bandrecord-Nilvaluerecord Relationship
Save and continue editing
Save and add another
Save


Select Band to change | E
kalyke.eox.at/demoinstance/admin/coverages/bandrecord/
EOxServer Admin Client
Welcome, admin. Change password / Log out
Home › Coverages › Bands
The Band "blue" was added successfully.
Select Band to change
Add Band
Band
blue
green
red
3 Bands

Band it's respective number ('1' for red and so on). (see screenshot)

Alternatively we could have started with the Range Type and added each band via the "+" icons next to the bands directly.

To list, export, and load range types using the command-line tools see *Range Type Handling* (page 61).

Linking to a Local Path

Click "Add" on "Local paths" and paste the desired local directory where your data is. Make sure the system user under which the web server process is running, typically apache, has read access.

Creating a Data Package

A *Data Package* consists of a GDAL-readable image file and a corresponding XML metadata file using the WCS 2.0 Earth Observation Application Profile (EO-WCS).

Adding Data Sources

After adding a Local Path or location (pointing to a single directory, not a specific file) you can combine this with a search pattern and create a Data Source. A viable search pattern would be something like "*.tif" to add all TIFF files stored in that directory. Please note that in this case, every TIFF needs a XML file with the exact same name holding the EO-Metadata.

Creating a Dataset Series

A Dataset Series can contain any number of EO Coverages i.e. Datasets or Stitched Mosaics. A Dataset Series therefore has its own metadata entry with respect to the metadata of its containing datasets.

1.11.6 Command Line Tools

eoxserver-admin.py create_instance

The first important command line tool is used for *Service Instance Creation and Configuration* (page 20) of EOxServer and is explained in the *Installation* (page 14) section of this user' guide.

eoxs_register_dataset

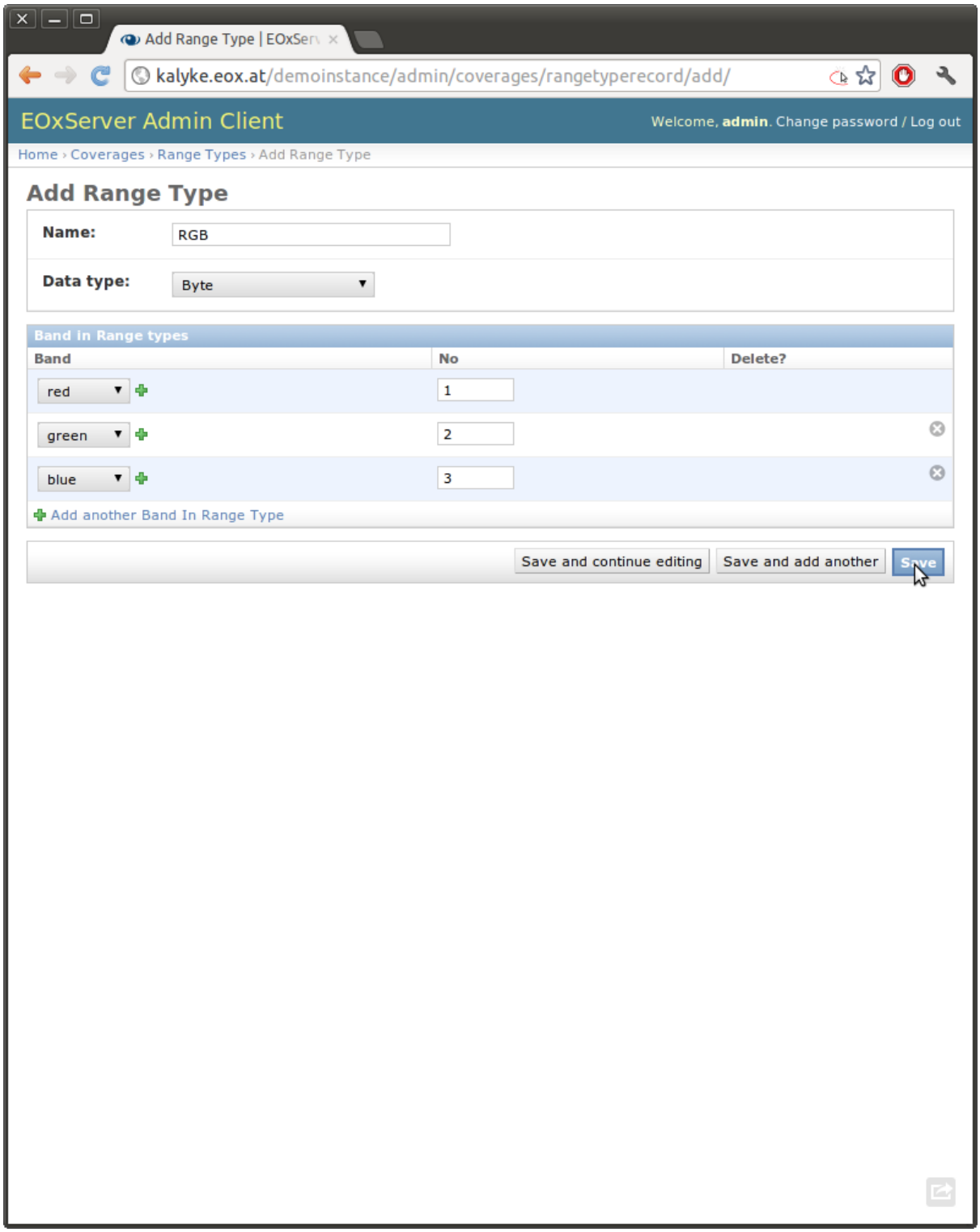
Besides this tool EOxServer adds some custom commands to Django's manage.py script. The `eoxs_register_dataset` command is detailed in the *Data Registration* (page 24) section.

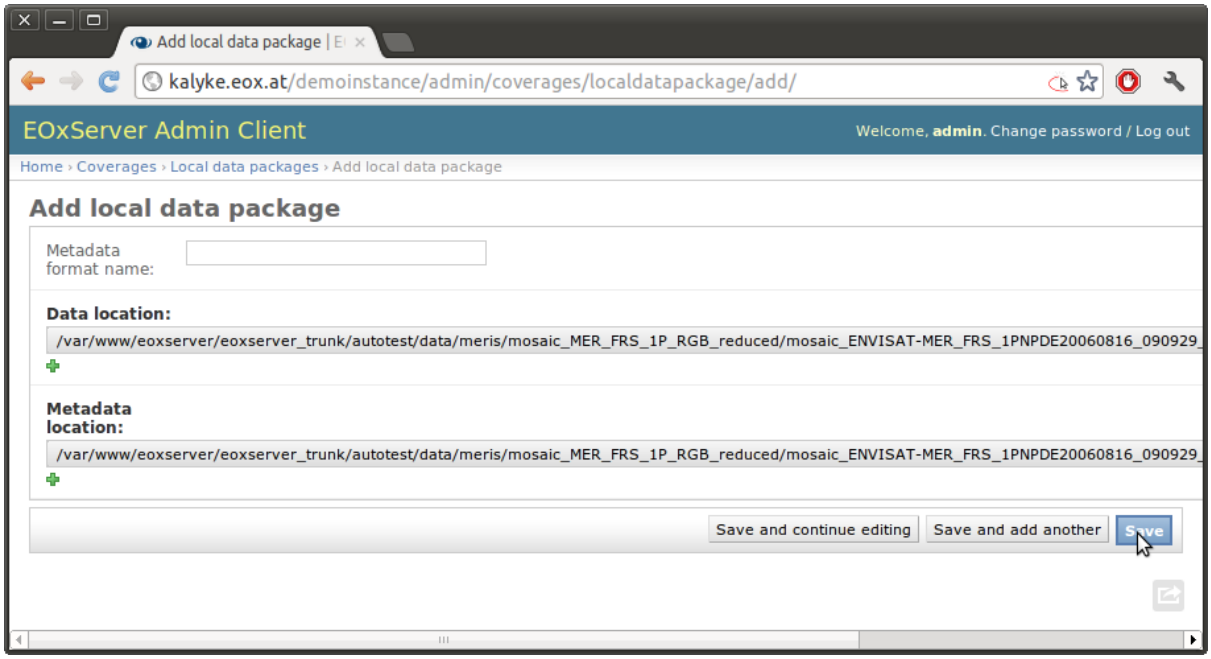
eoxs_deregister_dataset

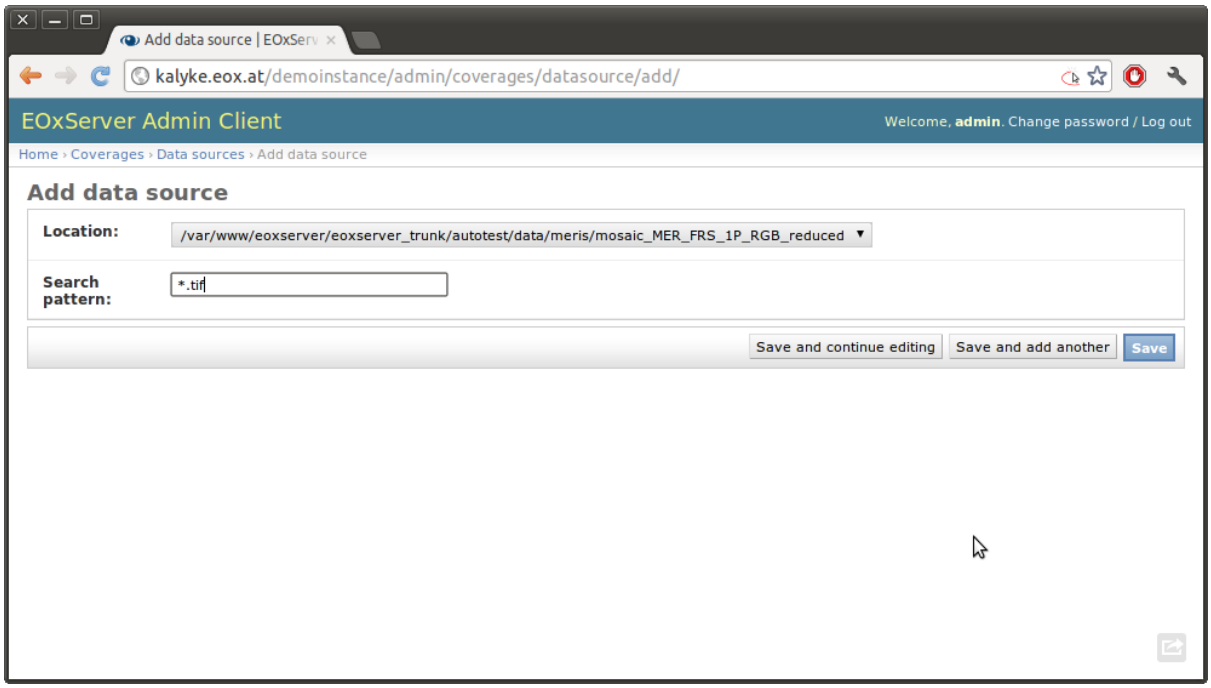
The `eoxs_deregister_dataset` command allows the de-registration of existing datasets (simple coverage types as Rectified and Referenceables datasets only) from an EOxServer instance including proper unlinking from relevant container types. The functionality of this command is complementary to the *eoxs_register_dataset* (page 55) command.

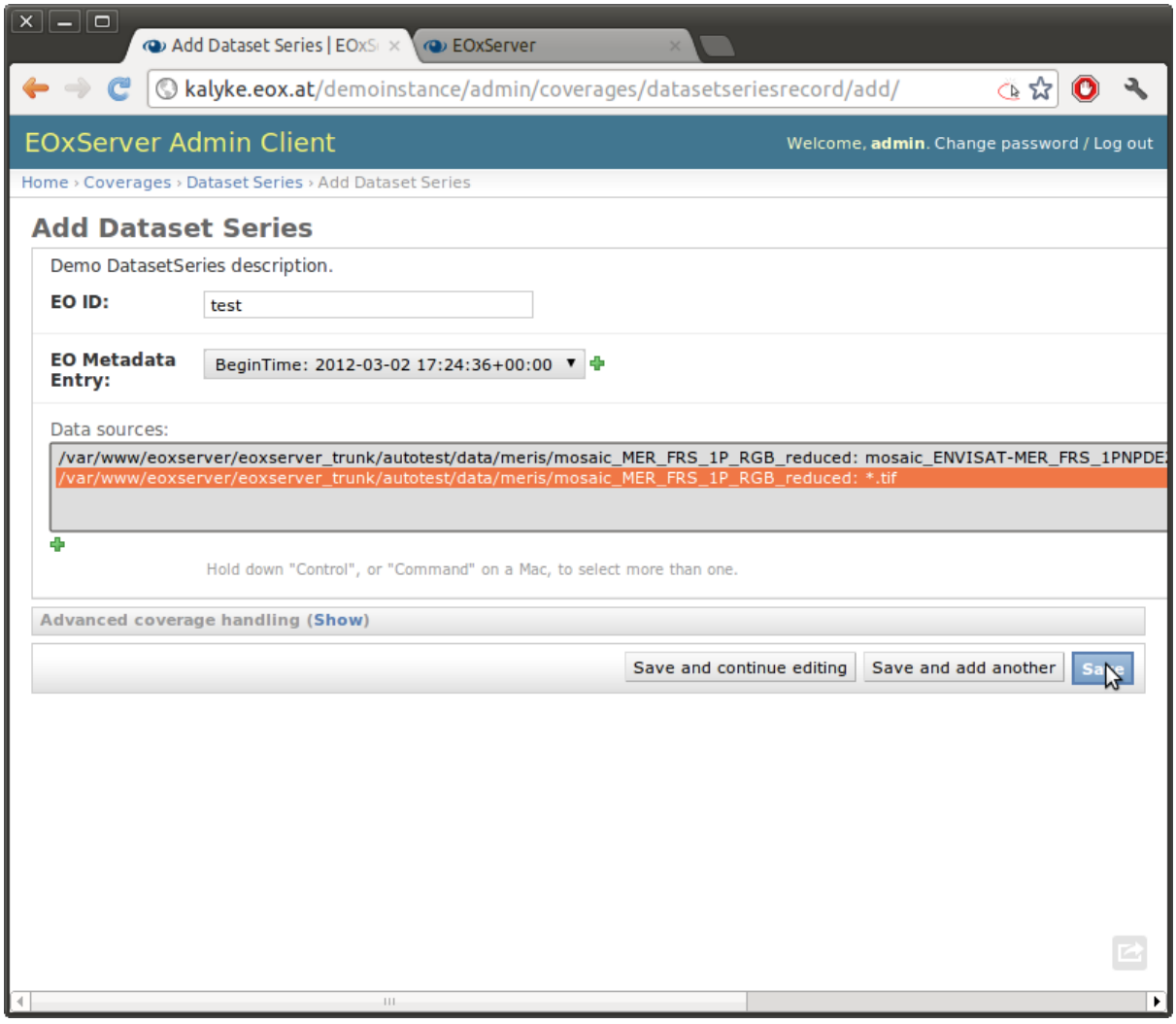
It is worth to mention that the de-registration does not remove physical data stored in the file system or different storage backende. Therefore an extra effort has to be spent to purge the physical data/meta-data files from their storage.

To de-register a dataset (coverage) identified by its (Coverage/EO) identifier the following command shall be invoked:


Add Range Type | EOxSer
kalyke.eox.at/demoinstance/admin/coverages/rangetyperecord/add/
EOxServer Admin Client
Welcome, admin. Change password / Log out
Home › Coverages › Range Types › Add Range Type
Add Range Type
Name:
RGB
Data type:
Byte
Band in Range types
Band
No
Delete?
red
1
green
2
blue
3
Add another Band In Range Type
Save and continue editing
Save and add another
Save


Add local data package | E
kalyke.eox.at/demoinstance/admin/coverages/localdatapackage/add/
EOxServer Admin Client
Welcome, admin. Change password / Log out
Home › Coverages › Local data packages › Add local data package
Add local data package
Metadata format name:
Data location:
/var/www/eoxserver/eoxserver_trunk/autotest/data/meris/mosaic_MER_FRS_1P_RGB_reduced/mosaic_ENVISAT-MER_FRS_1PNPDE20060816_090929_
Metadata location:
/var/www/eoxserver/eoxserver_trunk/autotest/data/meris/mosaic_MER_FRS_1P_RGB_reduced/mosaic_ENVISAT-MER_FRS_1PNPDE20060816_090929_
Save and continue editing
Save and add another
Save


Add data source | EOxServ
kalyke.eox.at/demoinstance/admin/coverages/datasource/add/
EOxServer Admin Client
Welcome, admin. Change password / Log out
Home › Coverages › Data sources › Add data source
Add data source
Location:
/var/www/eoxserver/eoxserver_trunk/autotest/data/meris/mosaic_MER_FRS_1P_RGB_reduced
Search pattern:
*.tif
Save and continue editing
Save and add another
Save


Add Dataset Series | EOxS
EOxServer
kalyke.eox.at/demoinstance/admin/coverages/datasetseriesrecord/add/
EOxServer Admin Client
Welcome, admin. Change password / Log out
Home › Coverages › Dataset Series › Add Dataset Series
Add Dataset Series
Demo DatasetSeries description.
EO ID:
test
EO Metadata Entry:
BeginTime: 2012-03-02 17:24:36+00:00
Data sources:
/var/www/eoxserver/eoxserver_trunk/autotest/data/meris/mosaic_MER_FRS_1P_RGB_reduced: mosaic_ENVISAT-MER_FRS_1PNPDE
/var/www/eoxserver/eoxserver_trunk/autotest/data/meris/mosaic_MER_FRS_1P_RGB_reduced: *.tif
Hold down "Control", or "Command" on a Mac, to select more than one.
Advanced coverage handling (Show)
Save and continue editing
Save and add another

```
python manage.py eoxs_deregister_dataset <CoverageID>
```

The de-registration command allows convenient de-registration of an arbitrary number of datasets at the same time:

```
python manage.py eoxs_deregister_dataset <CoverageID> <CoverageID> ...
```

The `eoxs_deregister_dataset` does not allow the removing of container objects such as Rectified Stitched Mosaics or Dataset Series.

The `eoxs_deregister_dataset` command, by default, does not allow the de-registration of automatic datasets (i.e, datasets registered by the synchronisation process, see *What is synchronization?* (page 60)). Although this restriction can be overridden by the `--force` option, it is not recommended to do so.

Updating Datasets

There is currently no way how to update registered EOxServer datasets from the command line. In case such an action would be needed it is recommended to de-register the existing dataset first (see *eoxs_deregister_dataset* (page 55) command) and register it again with the updated parameters (see *eoxs_register_dataset* (page 55) command). Special attention should be paid to linking of the *updated* dataset to all the container objects during the registration as this information is removed by the de-registration.

eoxs_add_dataset_series

The `eoxs_add_dataset_series` command allows the creation of a dataset series with initial data sources or coverages included. In it's simplest use case, only the `--eo-id` parameter is required, which has to be a valid and not yet taken identifier for the Dataset Series.

When supplied with the `--data-sources` parameter, given data sources will be added once the Dataset Series is created. When using the `--data-sources` it is highly recommended to also use `--patterns`, a list of search patterns which will be used for the data source of the same index. When only one `--pattern` is given, it is used for all data sources.

Range types for datasets can be read from configuration files that are accompanying them. There can be a configuration file for each dataset or one that applies to all datasets contained within a directory corresponding to a data source. Configuration files have the file extension `.conf`. The file name is the same as the one of the dataset (so the dataset `foo.tiff` needs to be accompanied by `foo.conf`) or `__default__.conf` if you want to use the config file for the whole directory. The syntax for the file is as follows:

```
[range_type]
range_type_name=<range type name>
```

Both approaches may be combine and configuration files produced only for some of the datasets in a directory and a default range type defined in `__default__.conf`. EOxServer will first look up the dataset configuration file and fall back to the default only if there is no individual `.conf` file.

Unless the `--no-sync` parameter is given, this also triggers a synchronization as explained in the chaper *What is synchronization?* (page 60).

Already registered datasets can be automatically added to the Dataset Series by using the `--add` option which takes a list of IDs referencing either Rectified Datasets, Referenceable Datasets and Rectified Stitched Mosaics.

The optional `--default-begin-time`, `--default-end-time` and `--default-footprint` parameters can be used to supply some default metadata values. Note: once the Dataset Series is synchronized, these values are overridden.

eoxs_synchronize

This command allows to synchronize an EOxServer instance with the file system.

What is synchronization?

In the context of EOxServer, synchronization is the process of updating the database models for container objects (such as RectifiedStitchedMosaics or DatasetSeries) according to changes in the file system.

Automatic datasets are deleted from the database, when their data files cannot be found in the file system. Similar, new datasets will be created when new files matching the search pattern in the subscripted directories are found.

When datasets are added to or deleted from a container object, the metadata (e.g the footprint of the features of interest or the time extent of the image) of the container is also likely to be adjusted.

Reasons for Synchronization

There are several occasions, where synchronization is necessary:

- A file has been added to a folder associated with a container
- A file from a folder associated with a container has been removed
- EO Metadata has been changed
- A regular check for database consistency

HowTo

Synchronization can be triggered by a custom Django admin command[97], called `eoxs_synchronize`.

To start the synchronization process, navigate to your instances directory and type:

```
python manage.py eoxs_synchronize <IDs>
```

whereas `<IDs>` are the coverage/EO IDs of the containers that shall be synchronized.

Alternatively, with the `-a` or `--all` option, all container objects in the database will be synchronized. This option is useful for a daily cron-job, ensuring the databases consistency with the file system.

```
python manage.py eoxs_synchronize --all
```

The synchronization process may take some time, especially when FTP/Rasdaman storages are used and also depends on the number of synchronized objects.

eoxs_insert_into_series

This command allows to insert (link) existing coverages (datasets) into dataset series.

The same action can be obtained already during the dataset registration by using of the `--dataset-series` option of the *eoxs_register_dataset* (page 55).

To insert a coverage into a dataset series use this command:

```
python manage.py eoxs_insert_into_series <CoverageID> <DatasetSeriesID>
```

For convenience, multiple coverages can be inserted at once:

```
python manage.py eoxs_insert_into_series <CoverageID1> <CoverageID2> ... <DatasetSeriesID>
```

All given IDs but the last are interpreted as coverage IDs and the last as the ID for the dataset series.

The IDs can also be set explicitly via the `--dataset` and `--dataset-series` options, which also allows the insertion of datasets into multiple dataset series:

[97] https://docs.djangoproject.com/en/1.4/ref/django-admin/

```
python manage.py eoxs_insert_into_series --datasets <CoverageID1> <CoverageID2> \
                            --dataset-series <DatasetSeriesID1> <DatasetSeriesID2>
```

eoxs_remove_from_series

This command is complemetary to the *eoxs_insert_into_series* (page 60) as it removes (unlinks) coverages from a dataset series. As these two commands have a very similar semantic, the parameters are the same and have the same meaning.

To remove a single coverage from a dataset series type:

```
python manage.py eoxs_remove_from_series <CoverageID> <DatasetSeriesID>
```

Like *eoxs_insert_into_series* (page 60) also multiple coverages can be excluded at once.

It is worth to mention that the `eoxs_remove_from_series` command does not deregister the unlinked datasets and these still held by the EOxServer. In case the deregistration of datasets is desired the *eoxs_deregister_dataset* (page 55) command does so together with unlinking of the datasets from all datasets.

eoxs_check_id

The `eoxs_check_id` commands allows checking about status of the queried coverage/EO identifier. The command returns the status via its return code (0 - `True` or 1 - `False`).

By default the command checks whether an identifier can be used (is available) as a new Coverage/EO ID:

```
python manage.py eoxs_check_id <ID> && echo True || echo False
```

The default behaviour is equivalent to `--is-available` option:

```
python manage.py eoxs_check_id --is-available <ID> && echo True || echo False
```

The *available* coverage/EO ID is neither *used* by an existing objects nor *reserved* for use by a future object.

In order to check whether a coverage/EO ID is used by an existing object apply the `--is-used` option:

```
python manage.py eoxs_check_id --is-used <ID> && echo True || echo False
```

In order to check whether a coverage/EO ID is registered for future use apply the `--is-reserved` option:

```
python manage.py eoxs_check_id --is-reserved <ID> && echo True || echo False
```

Range Type Handling

The `eoxs_list_rangetypes` command, by default, lists the names of all registered range types:

```
python manage.py eoxs_list_rangetypes
```

In case of more range types details required verbose listing may be requested by `--details` option. When one or more range type names are specified the output will be limited to the specified range-types only:

```
python manage.py eoxs_list_rangetypes --details [<range-type-name> ...]
```

The same command can be also used to export rangetype in JSON format (`--json` option). Following example prints the selected RGB range type in JSON format:

```
python manage.py eoxs_list_rangetypes --json RGB
```

The output may be directly savaved to file by using the `-o` option. Following example saves all the registered range-types to a file named `rangetypes.json`:

```
python manage.py eoxs_list_rangetypes --json -o rangetypes.json
```

The rangetypes saved in JSON format can be loaded (e.g., by another *EOxServer* instance) by using of the `eoxs_load_rangetypes` command. By default, this command reads the JSON data from the standard input. To force the command to read the input from a file use `-i`

```
python manage.py eoxs_load_rangetypes -i rangetypes.json
```

1.11.7 Performance

The performance of different EOxServer tasks and services depends heavily on the hardware infrastructure and the data to be handled. Tests were made for two typical operator use cases:

- registering a dataset
- generating a mosaic

The tests for **registering datasets** were performed on a quad-core machine with 4 GB of RAM and with a SQLite/SpatiaLite database. The test datasets were 58 IKONOS multispectral (4-band 16-bit), 58 IKONOS panchromatic (1-band 16-bit) and 58 IKONOS pansharpened (3-band 8-bit) scenes in GeoTIFF format with file sizes ranging between 60 MB and 1.7 GB. The file size did not have any discernible impact on the time it took to register. The average registration took about 61 ms, meaning that registering nearly 1000 datasets per minute is possible.

The tests for the **generation of mosaics** were performed on a virtual machine with one CPU core allocated and 4 GB of RAM. Yet again, the input data were IKONOS scenes in GeoTIFF format.

Datasets	Data Type	Files	Input File Size	Tiles Generated	Time	GB per minute
IKONOS multispectral	4-band 16-bit	68	8.9 GB	8.819	10 m	0.89 GB
IKONOS panchromatic	1-band 16-bit	68	35.1 GB	126.750	1:05 h	0.54 GB
IKONOS pansharpened	3-band 8-bit	68	52.7 GB	126.750	1:46 h	0.49 GB

As the results show the file size of the input files has a certain impact on performance, but the effect seems to level off.

Regarding the performance of the services there are many influence factors:

- the hardware configuration of the machine
- the network connection bandwith
- the database configuration (SQLite or PostGIS)
- the format and size of the raster data files
- the processing steps necessary to fulfill the request (e.g. resampling, reprojection)
- the coverage type (processing referenceable grid coverages is considerably more expensive than processing rectified grid coverages)
- the setup of IDM components (if any)

For hints on improving performance of the services see *Hardware Requirements* (page 15), *Data Preparation and Supported Data Formats* (page 49) and *Improving Performance with MapCache* (page 63).

1.12 The Webclient Interface

Table of Contents

- The Webclient Interface (page 62)
  - Enable the Webclient Interface (page 63)
    - Available configuration options (page 63)
    - Improving Performance with MapCache (page 63)
  - Using the webclient interface (page 64)

The webclient interface is an application running in the browser and provides a preview of all Datasets in a specified Dataset Series. It uses an OpenLayers[98] display to show a WMS view of the datasets within a map context. The background map tiles are provided by OSGeo[99].

It can further be used to provide a download mechanism for registered datasets.

1.12.1 Enable the Webclient Interface

To enable the webclient interface, several adjustments have to be made to the instances *settings.py* and *urls.py*.

First off, the *eoxserver.webclient* has to be inserted in the *INSTALLED_APPS* option of your *settings.py*. As the interface also requires several static files like style-sheets and script files, the option *STATIC_URL* has to be set to a path the webserver is able to serve, for example */static/*. The static media files are located under *path/to/eoxserver/webclient/static*.

To finally enable the webclient, a proper URL scheme has to be set up in *urls.py*. The following lines would enable the index and the webclient view on the URL *www.yourdomain.com/client*.

```
urlpatterns = patterns('',
    ...
    (r'^client/$', 'eoxserver.webclient.views.index'),
    (r'^client/(.*)', 'eoxserver.webclient.views.webclient'),
    ...
)
```

Available configuration options

Optionally some configuration settings can be set in the "eoxserver.conf" config file. These settings have to be put into the "webclient" section:

```
preview_service=...
outline_service=...
preview_url=...
outline_url=...
```

The *preview_...* settings defined the settings for the preview layer, showing an actual RGB representation of the registered datasets, whereas the *outline_...* settings are used for displaying the footprint of all registered datasets.

The *..._service* parameter is used to define the service type used to retrieve the image tiles displayed on the map. Currently, "wms" and "wmts" are supported and "wms" is the default.

The *..._url* parameter defines the URL of the service providing the image tiles. This configuration defaults to the configuration given for the "http_service_url" setting in the "services.owscommon" section.

Improving Performance with MapCache

WMS offers a very flexible way to view the data; on the other hand performance is often a problem, especially when dealing with very large data files and different projections. In order to boost performance, you can use

[98] http://openlayers.org/
[99] http://www.osgeo.org/

caching techniques. There are different software packages that provide caching for WMS services; in this context we present MapCache[100], an open source tool that is part of the MapServer project.

MapCache supports various tile-based interfaces including the OGC Web Map Tile Service[101] (WMTS). We suggest to use WMTS for caching purposes, as it is a genuine OGC standard whereas the alternatives (WMS-C, TMS) are mere suggestions witout binding character.

The MapCache sub-package provides an Apache2 HTTP Server module. In order to install it you must download the latest trunk version of MapServer and change to the `mapcache` subdirectory. There you can build and install the software in the common way:

```
$ ./configure
$ make
$ sudo make install
```

For comprehensive installation instructions and alternative setups see the MapCache Installation and Configuration[102] documentation.

Once you have installed the module you can deploy a MapCache instance. Therefore you have to add something like the following to your Apache2 configuration:

```
<IfModule mapcache_module>
   <Directory /path/to/directory>
      Order Allow,Deny
      Allow from all
   </Directory>
   MapCacheAlias /mapcache "/path/to/directory/mapcache.xml"
</IfModule>
```

The XML file the `MapCacheAlias` directive points to contains the configuration of the cache. It specifies the services to be provided, the data sources, the provided layers, how they are cut into tiles and many other things. For a complete reference please refer to the MapCache Configuration File Docs[103].

Specifically for EOxServer, the data source URL has to be set to the EOxServer OGC Web Services URL, usually something like `http://www.example.com/eoxserver_instance/ows`.

As the web client expects input data in the WGS84 coordinate reference system (EPSG:4326), your MapCache instance must support this CRS. You have to define a grid using this CRS or use the predefined `WGS84` grid. Note that the web client expects that the map scale increases with the zoom level index. Level 0 is the minimum scale showing the whole covered area (e.g. the whole world for the predefined `WGS84` grid).

If you want to use WMTS with the EOxServer web client you have to define a tile set for each Recitfied Stitched Mosaic and Dataset Series on your site. The tile set name must be the same as the CoverageID for Rectified Stitched Mosaics and the EOID for Dataset Series.

Note that usually a tile will be rendered and written to the cache only when it is requested, but you can pre-seed the cache using the `mapcache_seed` command. Once you have built MapCache, you can find this tool in the `mapcache/src` subdirectory of your MapServer directory. For a reference, see the MapCache Seeder Docs[104].

Once you have set up a WMTS instance, you can set the EOxServer configuration parameters `preview_service` to `wmts` and `preview_url` to the URL your MapCache instance is running under (see also *Available configuration options* (page 63)).

1.12.2 Using the webclient interface

The webclient interface can be accessed via the given URL in *urls.py* as described in the instructions above, whereas the URL *www.yourdomain.com/client* would open an index view, displaying links to the webclient for every dataset series registered in the system. To view the webclient for a specific dataset series, use this URL: *www.yourdomain.com/client/<EOID>* where *<EOID>* is the EO-ID of the dataset series you want to inspect.

[100] http://www.mapserver.org/trunk/mapcache/index.html
[101] http://www.opengeospatial.org/standards/wmts
[102] http://www.mapserver.org/trunk/mapcache/install.html
[103] http://http://www.mapserver.org/trunk/mapcache/config.html
[104] http://www.mapserver.org/trunk/mapcache/seed.html

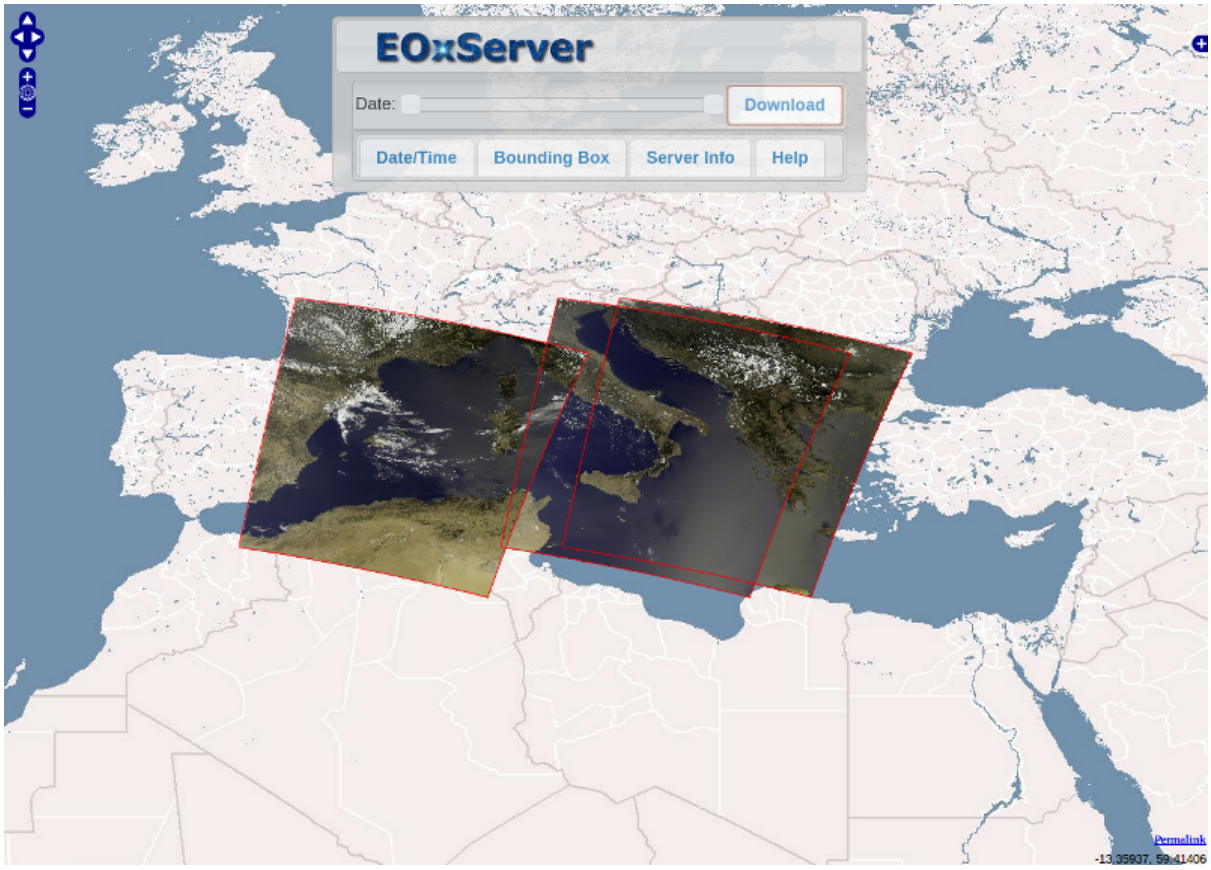


Figure 1.11: *The webclient showing the contents of the autotest instance.*

The map can be panned with via mouse dragging or the map-moving buttons in the upper left of the screen. Alternatively, the arrow keys can be used. The zoomlevel can be adjusted with the mouse scrolling wheel or the zoom-level buttons located directly below the pan control buttons.

A click on the small "+" sign on the upper right of the screen reveals the layer switcher control, where the preview and outline layers of the dataset series can be switched on or off. By default, the preview layer is switched off and only the outlines layer is visible.

In the upper center the EOxServer panel can be seen. It is used to select temporal and spatial subsets for the dataset series. It can be placed anywhere on the screen by dragging it like a window.

The slider in the middle is used to select the spatial subset for datasets. The left slider handle determines the minimum date boundary and the right one the maximum date boundary for datasets to be displayed.

While moving, the value of the minimum and maximum date can be viewed in the first tab, "Date/Time". There, it can also be adjusted manually, either as a text input or via a date-picker widget. For extra fine-grained queries, the minimum and maximum time values can be adjusted.

Once the date/time has changed from either the slider or the input fields, the map is updated with the new parameters. The results varies, depending on the background map viewing service used, as WMTS services simply ignore the time parameter. If WMS services are configured, only datasets should be visible that are within the given date/time slice. Please refer to *Available configuration options* (page 63) for detailed information.

Hidden under the second tab are controls for configuring the bounding box. The bounding box can either be entered manually with the input fields or drawn on the map once the "Draw BBOX" function is activated. The bouning box marker and the input values are tied together, a change on one affects the other.

Unlike the date/time selection, the bounding box has no affect on the preview or the outlines visible. It is only used for the offering of coverages at the final Download of data.

The third tab, "Service Info" displays the visible meta-data about your WCS service as configured by your instance

and shown via GetCapabilities. This meta-data includes information about the service itself (type, keywords, abstract etc.) and its provider.

The "Download" dialog is shown after the "Download" button in the EOxServer panel is clicked. It displays a list of all datasets matching the give spatial and temporal subsets. If no datasets with the given parameters were found, an error message is shown.

Each coverage can be (de)selected using the checkbox. Only checked datasets will be downloaded when the "Start Download" button is clicked.

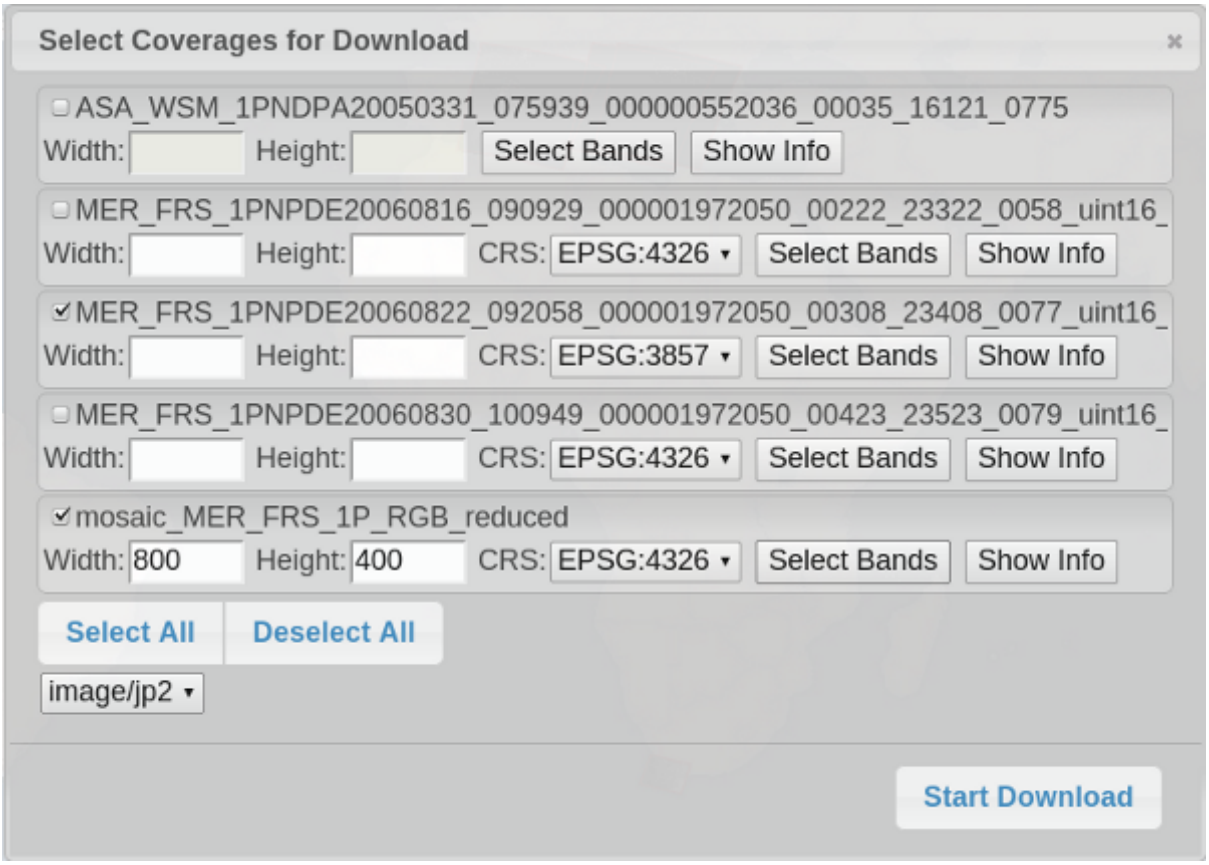


Figure 1.12: *The download selection view.*

The meaning of the size input fields depends on the actual type of the dataset. Rectified datasets can be scaled to the given size after all subsets are applied. Referencable datasets cannot be scaled, and so the size input fields only hint the overall (not subsetted) size of the raster data.

When the "Select Bands" button is clicked, a dialog opens which allows the selecting and ordering of requested bands (range-subset). At least one band has to be selected. The ordering of the bands can be changed with dragging/dropping the bands on the desired index.

Each coverage can be further inspected with the Coverage Info View which shows once the button "Show Info" for a coverage is clicked. In this view, addictional meta-data of the coverage is displayed and the coverage bands can be further selected and ordered.

Due to limitations of the nature of this preview, only one or three bands can be viewed at a time. The selection is done with the small checkboxes associated with every band. The order of the bands can be manipulated by dragging/dropping the bands on the desired index.

Once the "Start Download" Button is clicked, all selected coverages with the given spatial and temporal subsets and all given parameters are downloaded. The actual behavior depends on the used browser, commonly a save file dialog is displayed.

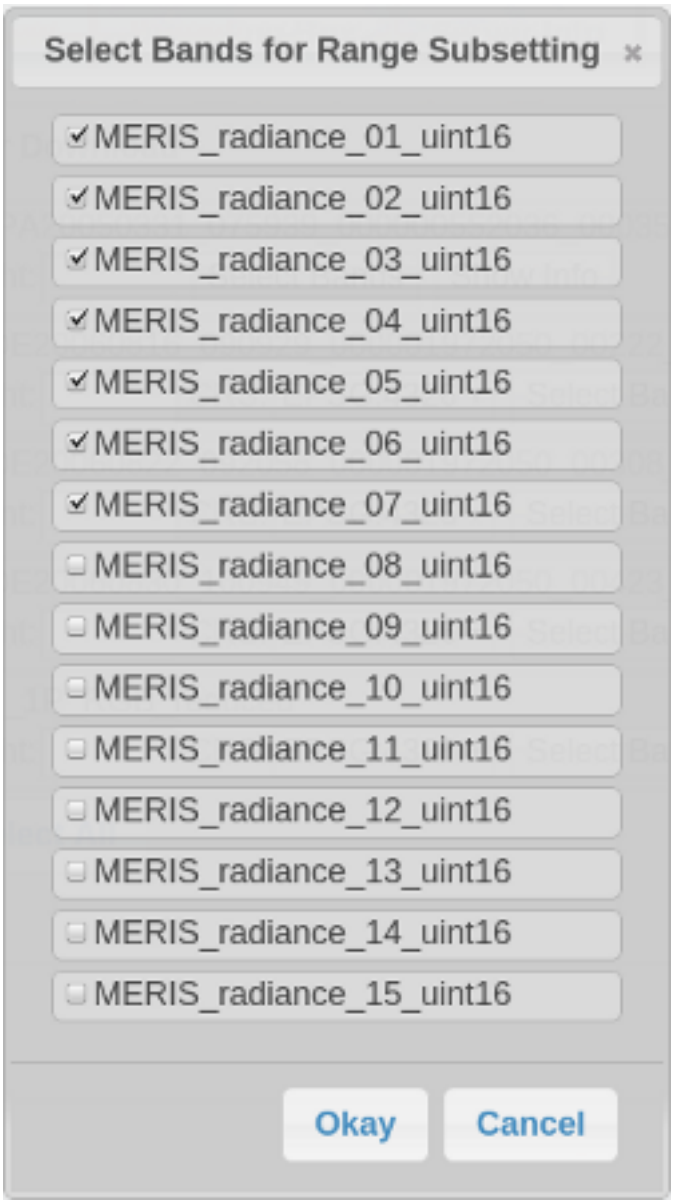


Figure 1.13: *A selection of bands for a soon-to-be downloaded coverage.*

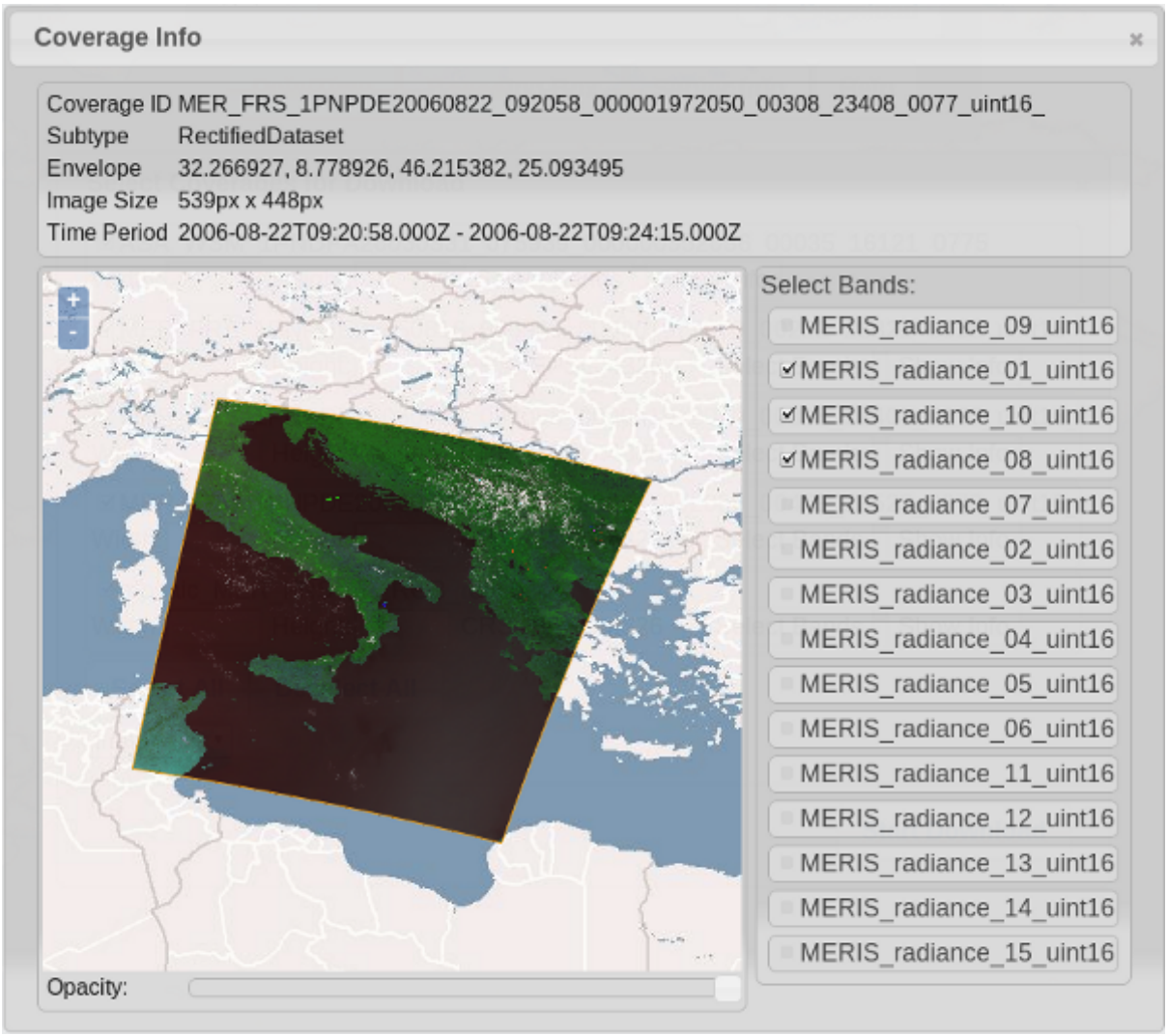


Figure 1.14: *The Coverage Info View displaying details of the coverage and selected bands.*

1.13 Identity Management System

Table of Contents

- Identity Management System (page 69)
  - Installation and Configuration (page 71)
    - Prerequisites (page 71)
    - LDAP Directory (page 71)
    - Authorisation Service (page 72)
      - XACML Policies for the Authorisation Service (page 72)
    - General Configuration for CHARON services (page 79)
  - HTTP and SOAP Specific Components (page 80)

The Identity Management System (IDMS) provides access control capabilities for security relevant data. The current IDMS supports EOxServer with a native security component for HTTP KVP and POST/XML protocol binding as well as external components for SOAP binding. The system is based on other free and open software projects, namely the Charon Project[105], the Shibboleth framwork[106] and the HMA Authentication Service[107]. In the context of EOxServer, the SOAP support in the IDMS can be used to provide authentication and authorisation capabilities for the *SOAP Proxy* (page 92).

The IDMS uses two different schemes for authentication: The native EOxServer component relies on Shibboleth for Authentication, the SOAP components use the Charon framework.

The approach chosen for the SOAP part of the IDMS follows the OGC best practice document User Management Interfaces for Earth Observation Services[108] for the authentication concept. The authentication part is following the ideas of the XACML data flow pattern[109]: The IDMS authorisation part consists of a Policy Decision Point (PDP, here represented through the Charon Policy Management And Authentication Service) and the Policy Enforcement Point (PEP, represented through the Charon PEP Service). The following figure gives an overview of the IDMS SOAP part:

The HMA Authentication Service, or Security Token Service (STS), and the Charon PEP components were both modified in order to be compatible. This is a result of the ESA project Open-standard Online Observation Service[110] (O3S). The STS now also supports SAML 2.0 security tokens, which the PEP components can interpret and validate. The IDMS supports trust relationships between identity providers and enforcement components on the basis of certificate stores.

The HTTP or native EOxServer part of the IDMS uses exactly the same scheme for authorisation as the SOAP part, but uses the Shibboleth federated identity management system for authentication.

Two requirements must be met to use the IDMS in this case:

- A Shibboleth Identity Provider (IdP) must be available for authentication
- A Shibboleth Service Provider (SP) must be installed and configured in an Apache HTTP Server[111] to protect the EOxServer resource.

A user has to authenticate at an IdP in order to perform requests to an EOxServer with access control enabled. The IdP issues a SAML token which will be validated by the SP.

Is the user valid, the SP adds the user attributes received from the IdP to the HTTP Header of the original service requests and conveys it to the protected EOxServer instance. The whole process ensures, that only authenticated users can access the data and services provided by EOxServer. The attributes from Shibboleth are used by the EOxServer security components to make a XACMLAuthzDecisionQuery to the Charon Authorisation Service.

[105] http://www.enviromatics.net/charon/
[106] http://shibboleth.internet2.edu/
[107] http://wiki.services.eoportal.org/tiki-index.php?page=HMA+Authentication+Service
[108] http://portal.opengeospatial.org/files/?artifact_id=40677
[109] http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf
[110] http://wiki.services.eoportal.org/tiki-index.php?page=O3S
[111] http://httpd.apache.org/

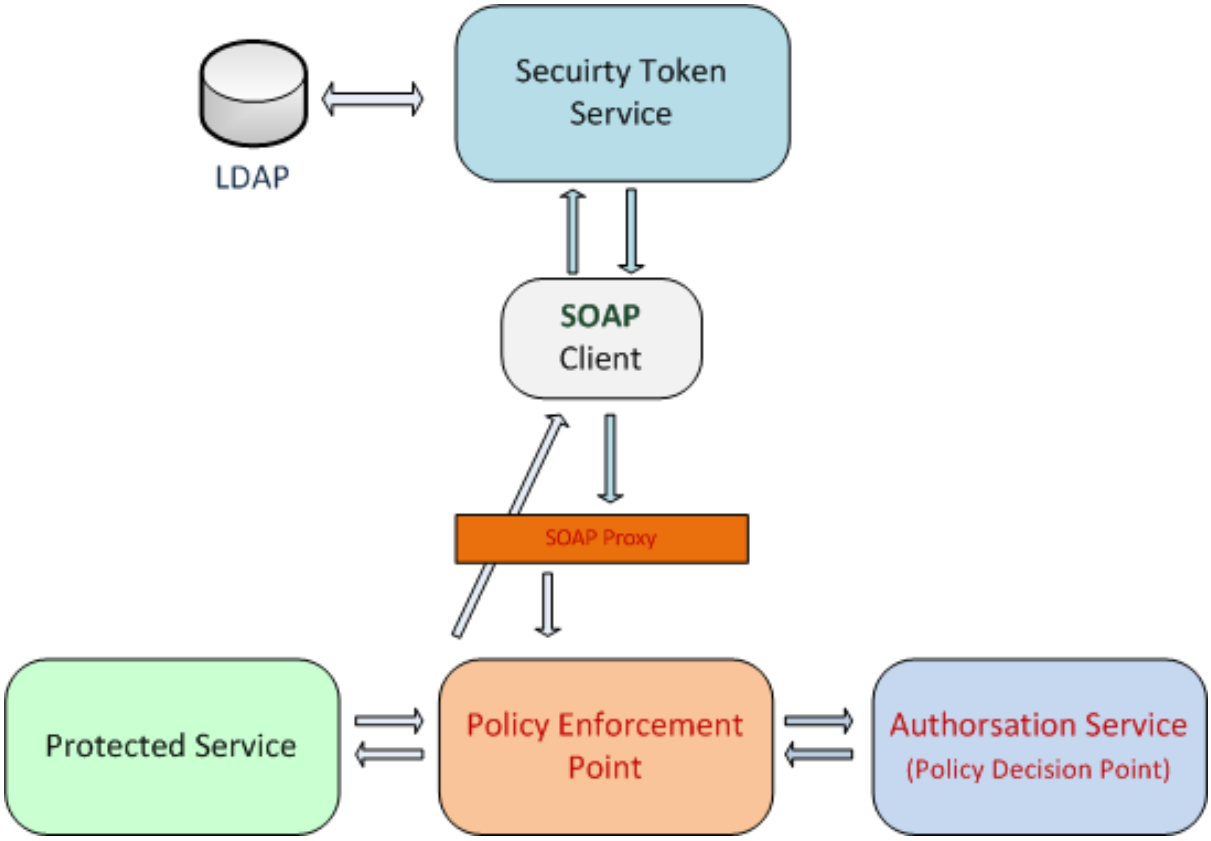


Figure 1.15: *IDMS SOAP Access Control Overview*

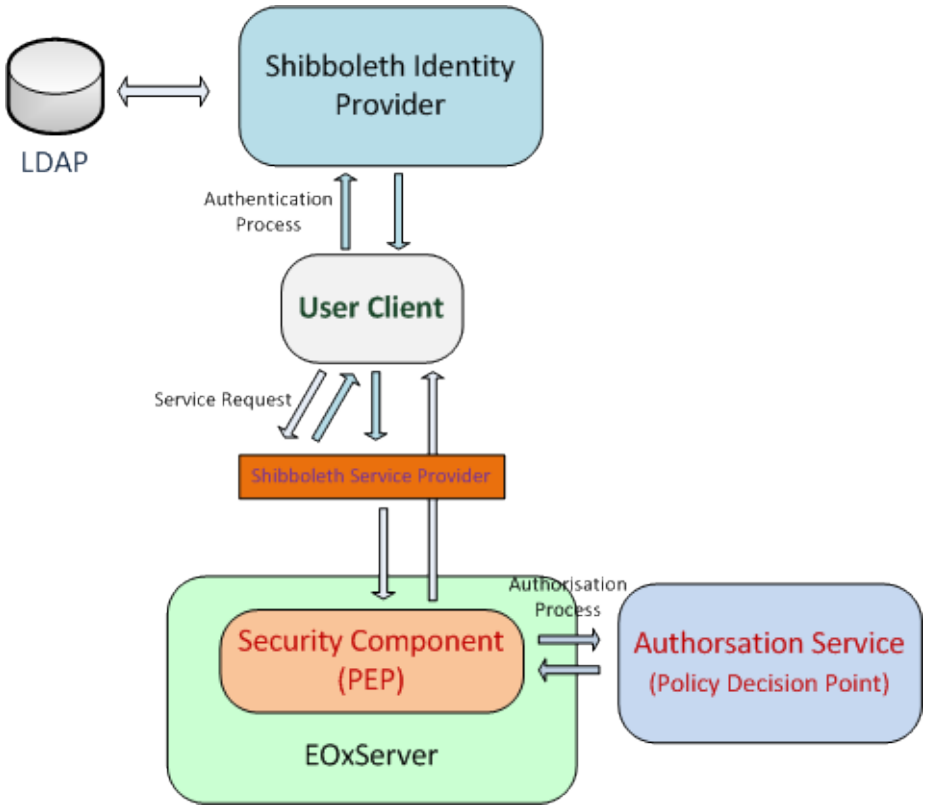


Figure 1.16: *IDMS EOxServer Access Control Overview*

1.13.1 Installation and Configuration

The following services are needed both for the HTTP and the SOAP security part:

- Charon *Authorisation Service* (page 72).
- *LDAP Directory* (page 71).

Prerequisites

Download locations for the IDMS components:

- Shibboleth: http://shibboleth.internet2.edu/downloads.html
- CHARON Authorisation Service: http://www.enviromatics.net/charon/ or http://packages.eox.at/idm/
- Security Token Service: http://packages.eox.at/idm/
- PEP Service: http://packages.eox.at/idm/
- EOxServer: http://eoxserver.org/wiki/Download

The following software is needed to run the IDMS:

- A *LDAP Directory* (page 71).
- Java[112] JDK 6 or higher
- Apache Tomcat[113] 6 or higher
- Apache Axis2[114] 1.4.1 or higher
- MySQL[115] 5
- Apache HTTP Server[116] 2.x

The following software is needed to build the IDMS components:

- Java[117] SDK 6 or higher
- Apache Ant[118] 1.6.2 or higher
- Apache Maven[119] 2 or higher

LDAP Directory

The IDMS uses a LDAP directory to store user data (attributes, passwords, etc). You can use any directory implementation, supporting the Lightweight Directory Access Protocol (v3).

Known working implementations are:

- Apache Directory Service[120]
- OpenLDAP[121]

A good graphical client for LDAP directories is the Apache Directory Studio[122].

[112] http://www.oracle.com/technetwork/java/index.html
[113] http://tomcat.apache.org/
[114] http://axis.apache.org/axis2/java/core/
[115] http://dev.mysql.com/downloads/
[116] http://httpd.apache.org/
[117] http://www.oracle.com/technetwork/java/index.html
[118] http://ant.apache.org/
[119] http://maven.apache.org/
[120] http://directory.apache.org/
[121] http://openldap.org
[122] http://directory.apache.org/studio/

Authorisation Service

Before installing the Authorsation Service, refer to the *General Configuration for CHARON services* (page 79).

The Authorisation Service is responsible for the authorisation of service requests. It makes use of XACML[123], a XML based language for access policies. The Authorisation Service is part of the CHAORN[124] project.

The Authorisation Service relies on a MySQL database to store all XACML policies. So in order to install the Authorisation Service, you first need to prepare a MySQL database:

- Install the MySQL database on your system.
- Change the *root* password. You can use the command line for this:

```
mysqladmin -u root password 'root' -p
```

- Run the SQL script bundle with the Authorisation Service in order to create the policy database:

```
mysql -u root -h localhost -p < PolicyAuthorService.sql
```

The Service needs the following additional dependencies in the `${AXIS2_HOME}\lib` folder:

- `mysql-connector-java-5.1.6.jar`
- `spring-2.5.1.jar`

The next step is deploying the Authorisation Service, therefore extract the ZIP archive into the directory of your `${AXIS2_HOME}`.

Now you have to configure the service. All configuration files are in the `${AXIS2_HOME}/WEB-INF/classes` folder and its sub-folders.

- Open the `PolicyAuthorService.properties` and change the `axisURL` parameter to the URL URL where you are actually deploying your service.
- You can change the database connection in the `config/GeoPDP.xml` configuration file if necessary.

To add new XACML policies to the Authorisation Service, refer to the *XACML Policies for the Authorisation Service* (page 72).

XACML Policies for the Authorisation Service

As mentioned before, the Charon Authorisation Service uses a MySQL database to store all XACML policies. The policies are stored in the database `policy_author` and the table `policy`. To add new policies, use an SQL client

```
INSERT INTO policy(policy) VALUES (' your xacml policy')
```

An XACML policy usually consists of a policy wide target and and several specific rules. The three main identifiers are subjects, targets and actions. Subjects (or users) can be identified through the "asserted user attributes" which are provided by the Shibboleth framework. The EOxServer security components also provide an attribute `REMOTE_ADDR` for subjects, which contains the IP address of the user. The resource is mainly identified through the attribute `urn:oasis:names:tc:xacml:1.0:resource:resource-id`, which is the service address of the secured service in case of an secured SOAP service and the host name or a ID set in the configuration in case of the EOxServer. The EOxServer also provides the atributes serverName (the host name) and serviceType (type of the service, i.e. wcs or wms). The action identifies the operation performed on the service, i.e. `getcapabilities` or `getcoverage`. In the following there are two example policies for the EOxServer WMS and WCS. Please note the comments inline.

A XACML policy to permit a user "wms_user" full accesss to the EOxServer WMS:

[123] http://www.oasis-open.org/committees/xacml/#XACML20
[124] http://www.enviromatics.net/charon/index.html

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy
    xsi:schemalocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os http://docs.oasis-open.org/
    PolicyId="wms_user_policy"
    RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides"
    xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ns="http://www.enviromatics.net/WS/PolicyManagementAndAuthorisationService/types /2.0">

    <Target>
    <Subjects>
        <Subject>
        <!-- Here we specify the user who has access to the service. Default identifier is the ui
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">wms_user</Attribute
            <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" Attribu
        </SubjectMatch>
        </Subject>
    </Subjects>
    <Resources>
        <Resource>
        <!-- The attribute urn:oasis:names:tc:xacml:1.0:resource:resource-id specifies the protec
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">eoxserver.example.
            <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" Attri
        </ResourceMatch>

        <!-- The attribute serviceType specifies the protected service (wms or wcs) -->
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">wms</AttributeValue
            <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" Attri
        </ResourceMatch>
        </Resource>
    </Resources>
    </Target>


    <!--
    In the following rules we allow the specified user to perform selected operations
    on the service.
    -->

    <!--
    GetCapabilities
    -->


    <Rule RuleId="PermitGetCapabilitiesCC" Effect="Permit">
    <Target>
        <Actions>
        <Action>
            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">GetCapabilities</At
            <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
            </ActionMatch>
        </Action>
        </Actions>
    </Target>
    </Rule>

    <Rule RuleId="PermitGetCapabilitiesSC" Effect="Permit">
    <Target>
        <Actions>
```

```
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">getcapabilities</At
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<!--
GetMap
-->

<Rule RuleId="GetMapCC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">GetMap</AttributeVa
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<Rule RuleId="GetMapSC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">getmap</AttributeVa
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<!--
GetFeatureInfo
-->

<Rule RuleId="GetFeatureInfoCC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">GetFeatureInfo</Att
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<Rule RuleId="GetFeatureInfoSC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">getfeatureinfo</Att
```

```
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<!--
DescribeLayer
-->

<Rule RuleId="DescribeLayerCC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">DescribeLayer</Att
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<Rule RuleId="DescribeLayerSC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">describelayer</Att
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<!--
GetLegendGraphic
-->

<Rule RuleId="GetLegendGraphicCC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">GetLegendGraphic</
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<Rule RuleId="GetLegendGraphicSC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">getlegendgraphic</
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
```

```
        </Actions>
    </Target>
    </Rule>

    <!--
    GetStyles
    -->

    <Rule RuleId="GetStylesCC" Effect="Permit">
    <Target>
        <Actions>
        <Action>
            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">GetStyles</Attribut
            <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
            </ActionMatch>
        </Action>
        </Actions>
    </Target>
    </Rule>

    <Rule RuleId="GetStylesSC" Effect="Permit">
    <Target>
        <Actions>
        <Action>
            <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">getstyles</Attribut
            <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
            </ActionMatch>
        </Action>
        </Actions>
    </Target>
    </Rule>

</Policy>
```

A XACML policy to permit a user "wcs_user" full accesss to the EOxServer WCS:

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy
    xsi:schemalocation="urn:oasis:names:tc:xacml:2.0:policy:schema:os http://docs.oasis-open.org/
    PolicyId="wcs_user_policy"
    RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-combining-algorithm:permit-overrides"
    xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ns="http://www.enviromatics.net/WS/PolicyManagementAndAuthorisationService/types /2.0">

    <Target>
    <Subjects>
        <Subject>
        <!-- Here we specify the user who has access to the service. Default identifier is the ui
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">wcs_user</Attribute
            <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" Attribu
        </SubjectMatch>
        </Subject>
    </Subjects>
    <Resources>
        <Resource>
        <!-- The attribute urn:oasis:names:tc:xacml:1.0:resource:resource-id specifies the protec
        <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">eoxserver.example.
            <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" Attri
```

```
    </ResourceMatch>

    <!-- The attribute serviceType specifies the protected service (wms or wcs) -->
    <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">wcs</AttributeValue
        <ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string" Attri
    </ResourceMatch>
    </Resource>
</Resources>
</Target>



<!--
GetCapabilities
-->

<Rule RuleId="PermitGetCapabilitiesCC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">GetCapabilities</At
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<Rule RuleId="PermitGetCapabilitiesSC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">getcapabilities</At
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<!--
DescribeCoverage
-->

<Rule RuleId="DescribeCoverageCC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">DescribeCoverage</A
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<Rule RuleId="DescribeCoverageSC" Effect="Permit">
<Target>
    <Actions>
```

```
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">describecoverage</A
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<!--
GetCoverage
-->

<Rule RuleId="DescribeCoverageCC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">GetCoverage</Attril
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<Rule RuleId="GetCoverageSC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">getcoverage</Attril
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<!--
DescribeEOCoverageSet
-->

<Rule RuleId="DescribeEOCoverageSetCC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">DescribeEOCoverage
        <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
        </ActionMatch>
    </Action>
    </Actions>
</Target>
</Rule>

<Rule RuleId="DescribeEOCoverageSetSC" Effect="Permit">
<Target>
    <Actions>
    <Action>
        <ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
        <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">describeeocoverage
```

```
            <ActionAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:action:action" Da
            </ActionMatch>
        </Action>
        </Actions>
    </Target>
    </Rule>

</Policy>
```

General Configuration for CHARON services

- The Charon services need the `acs-xbeans-1.0.jar` dependency in the `\lib` folder of your Axis2 installation (presumably the `webapps/axis2` of your Apache Tomcat installation).
- Furthermore, you have to activate the EIGSecurityHandler in the **Global Modules** section of your axis2 configuration (`${AXIS2_HOME}/WEB-INF/conf/axis2.xml`).
- You may configure the logging for the services in the Log4J configuration file (`${AXIS2_HOME}/WEB-INF/classes/log4j.properties`).

Both, the Security Token Service and the PEP service make use of Java Keystores: The IDMS uses Keystores to store a) the certificate used by the Security Token Service for signing SAML tokens and b) the public keys of those authenticating authorities trusted by the Policy Enforcement Point. The `keytool` of the Java distribution can be used to create and manipulate Java Keystores:

- The following command creates a new Keystore with the password :secret: and a suitable key pair with the alias :authenticate: for the Security Token Service:

  ```
  keytool -genkey -alias authenticate -keyalg RSA -keystore
  keystore.jks -storepass secret -validity 360
  ```

- The following command exports the public certificate from a key pair :authenticate: to the file `authn.crt`:

  ```
  keytool -export -alias authenticate -file authn.crt -keystore
  keystore.jks
  ```

- The following command imports a certificate to a Keystore:

  ```
  keytool -import -alias trusted_sts -file authn.crt -keystore
  keystore.jks
  ```

You can use the Apache HTTP Server as a proxy, it will enable your services running in Tomcat to be accessible over the Apache server. This can be useful when your services have to be accessible over the HTTP standard port *80*:

- First you have to enable `mod_proxy_ajp` and `mod_proxy`.
- Create a virtual host in your `httpd.conf`:

  ```
  <VirtualHost *:80>
     ServerName server.example.com

     <Proxy *>
       AddDefaultCharset Off
       Order deny,allow
       Allow from all
     </Proxy>

     ProxyPass /services/AuthenticationService ajp://localhost:8009/axis2/services/Authenti
     ProxyPassReverse /services/AuthenticationService ajp://localhost:8009/axis2/services/A

  </VirtualHost>
  ```

- The `ProxyPass` and `ProxyPassReverse` directives have to point to your services. Please note that the Tomcat server hosting your services must have the AJP interface enabled.

1.13.2 HTTP and SOAP Specific Components

For the installation and configuration please refer to the HTTP or SOAP specific documentation:

HTTP Components

Table of Contents

- HTTP Components (page 80)
  - Shibboleth Identity Provider (page 80)
  - Shibboleth Service Provider (page 85)
  - Configure Shibboleth SP and IdP (page 88)
  - Configure the EOxServer Security Components (page 89)
    - General Configuration Options (page 89)
    - Adding new Subject attributes to the EOxServer Security Components (page 89)

The following services are needed for the HTTP security part:

- Charon Authorisation Service
- Shibboleth Service Provider
- Shibboleth Identity Provider
- EOxServer

To install and configure the HTTP secuirty components, you have to follow these steps:

1. Install the Charon *Authorisation Service* (page 72).
2. Install the *Shibboleth Identity Provider* (page 80).
3. Install the *Shibboleth Service Provider* (page 85).
4. Follow the documentation of section *Configure Shibboleth SP and IdP* (page 88).
5. Follow the documentation of section *Configure the EOxServer Security Components* (page 89).

Shibboleth Identity Provider

The Shibboleth IdP is implemented as an Java Servlet, thus it needs an installed Servlet container. The Shibboleth project offers an installation manual for the Shibboleth IdP on their website[125]. This documentation will provide help for the basic configuration to get the authentication process working with your EOxServer instance and also the installation process for the use with Tomcat and Apache HTTPD. Before you begin with your installation, set up your Tomcat servlet container and install and configure an LDAP service.

Important URLs for your Shibboleth IDP:

- Status message: `https://${IDPHOST}/idp/profile/Status`
- Information page: `https://${IDPHOST}/idp/status`
- Metadata: `https://${IDPHOST}/idp/profile/Metadata/SAML`

Warning: IdP resource paths are case sensitive!

- Download[126] the IdP and unzip the archive.
- Run either ./install.sh (on Linxu/Unix systems) or install.bat (on Windows systems).
- Follow the on-screen instructions of the script.

Your `${IDP_HOME}` directory contains the following directories:

[125] https://wiki.shibboleth.net/confluence/display/SHIB2/IdPInstall
[126] http://shibboleth.internet2.edu/downloads.html

- `bin`: This directory contains various tools useful in running, testing, or deploying the IdP
- `conf`: This directory contains all the configuration files for the IdP
- `credentials`: This is were the IdP's signing and encryption credential, called idp.key and idp.crt, is stored
- `lib`: This directory contains various code libraries used by the tools in bin/
- `logs`: This directory contains the log files for the IdP . **Don't forget to make this writeable for your Tomcat server!**
- `metadata`: This is the directory in which the IdP will store its metadata, in a file called idp-metadata.xml. It is recommend you store any other retrieved metadata here as well.
- `war`: This contains the web application archive (war) file that you will deploy into the servlet container

The next step is to deploy the IdP into your Tomcat:

- Increase the memory reserved for Tomcat. Recommended values are `-Xmx512m -XX:MaxPermSize=128m`.
- Add the libraries endorsed by the Shibboleth project to your endorsed Tomcat directories: `-Djava.endorsed.dirs=${IDP_HOME}/lib/endorsed/`
- Create a new XML document `idp.xml` in `${TOMCAT_HOME}/conf/Catalina/ localhost/`.
- Insert the following content:

```
<Context docBase="${IDP_HOME}/war/idp.war"
         privileged="true"
         antiResourceLocking="false"
         antiJARLocking="false"
         unpackWAR="false"
         swallowOutput="true" />
```

- Dont't forget to replace `${IDP_HOME}` with the appropriate path.

To use the Apache HTTP server as an proxy for your IdP, you have to generate a certificate and a key file for SSL/TLS first.

- Generate a private key:

```
openssl genrsa -des3 -out server.key 1024
```

- Generate a CSR (Certificate Signing Request):

```
openssl req -new -key server.key -out server.csr
```

- Make a copy from the the original server key:

```
cp server.key copy_of_server.key
```

- Remove the Passphrase from your Key:

```
openssl rsa -in copy_of_server.key -out server.key
```

- Generating a Self-Signed Certificate:

```
openssl x509 -req -days 365 -in server.csr -signkey server.key
-out server.crt
```

The next step is to configure your Apache HTTP Server:

- First you have to enable `mod_proxy_ajp`, `mod_proxy` and `mod_ssl`.
- Create a new configuration file for your SSL hosts (for example `ssl_hosts.conf`).
- Add a new virtual host in your new hosts file. Please note the comments in the virtual host configuration.

```
<VirtualHost _default_:443>

    # Set appropriate document root here
    DocumentRoot "/var/www/"

    # Set your designated IDP host here
    ServerName ${IDP_HOST}

    # Set your designated logging directory here
    ErrorLog logs/ssl_error_log
    TransferLog logs/ssl_access_log
    LogLevel warn

    SSLEngine on

    SSLProtocol all -SSLv2

     # Important: mod_ssl should not verify the provided certificates
    SSLVerifyClient optional_no_ca

    SSLCipherSuite ALL:!ADH:!EXPORT:!SSLv2:RC4+RSA:+HIGH:+MEDIUM:+LOW

    # Set the correct paths to your certificate and key here
    SSLCertificateFile    ${IDP_HOST_CERTIFICATE}
    SSLCertificateKeyFile ${IDP_HOST_CERTIFICATE_KEY}

    <Files ~ "\.(cgi|shtml|phtml|php3?)$">
        SSLOptions +StdEnvVars
    </Files>
    <Directory "/var/www/cgi-bin">
        SSLOptions +StdEnvVars
    </Directory>

    # AJP Proxy to your IDP servlet
    ProxyPass /idp/ ajp://localhost:8009/idp/
    ProxyPassReverse /idp ajp://localhost:8009/idp

    SetEnvIf User-Agent ".*MSIE.*" nokeepalive ssl-unclean-shutdown downgrade-1.0 force-r

    CustomLog logs/ssl_request_log "%t %h %{SSL_PROTOCOL}x %{SSL_CIPHER}x \"%r\" %b"

</VirtualHost>
```

- Restart your HTTP server.

The next step is to configure our IdP Service with an LDAP service. Please keep in mind that this documentation can only give a small insight into all configuration possibilities of Shibboleth.

Open the `handler.xml`

- Add a new LoginHandler

```
<LoginHandler xsi:type="UsernamePassword"
              jaasConfigurationLocation="file://${IDP_HOME}/conf/login.config">
              <AuthenticationMethod>urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtec
</LoginHandler>
```

- Remove (or comment out) the LoginHandler element of type RemoteUser.

Open the `login.config` and comment out or delete the other entries that might exist. Add your own LDAP configuration:

```
ShibUserPassAuth {
  edu.vt.middleware.ldap.jaas.LdapLoginModule required
     host="${LDAP_HOST}"
```

```
    port="${LDAP_PORT}"
    serviceUser="${LDAP_ADMIN}"
    serviceCredential="${LDAP_ADMIN_PASSWORD}"
    base="${LDAP_USER_BASE}"
    ssl="false"
    userField="uid"
    subtreeSearch="true";
};
```

Enable your LDAP directory as attribute provider:

- Open the `attribute-resolver.xml`.
- Add your LDAP:

```
<resolver:DataConnector id="localLDAP" xsi:type="LDAPDirectory"
          xmlns="urn:mace:shibboleth:2.0:resolver:dc" ldapURL="ldap://${LDAP_HOST}:${LDAP
          baseDN="${LDAP_USER_BASE}" principal="${LDAP_ADMIN}"
          principalCredential="${LDAP_ADMIN_PASSWORD}">
<FilterTemplate>
    <![CDATA[
          (uid=$requestContext.principalName)
    ]]>
</FilterTemplate>
</resolver:DataConnector>
```

- Configure the IdP to retrieve the attributes by adding new attribute definitions:

```
<resolver:AttributeDefinition id="transientId" xsi:type="ad:TransientId">
    <resolver:AttributeEncoder xsi:type="enc:SAML1StringNameIdentifier"
        nameFormat="urn:mace:shibboleth:1.0:nameIdentifier"/>
    <resolver:AttributeEncoder xsi:type="enc:SAML2StringNameID"
        nameFormat="urn:oasis:names:tc:SAML:2.0:nameid-format:transient"/>
</resolver:AttributeDefinition>

<resolver:AttributeDefinition id="displayName" xsi:type="Simple"
    xmlns="urn:mace:shibboleth:2.0:resolver:ad" sourceAttributeID="displayName">
    <resolver:Dependency ref="localLDAP"/>
    <resolver:AttributeEncoder xsi:type="SAML1String"
        xmlns="urn:mace:shibboleth:2.0:attribute:encoder"
        name="urn:mace:dir:attribute-def:displayName"/>
    <resolver:AttributeEncoder xsi:type="SAML2String"
        xmlns="urn:mace:shibboleth:2.0:attribute:encoder"
        name="urn:oid:2.16.840.1.113730.3.1.241" friendlyName="displayName"/>
</resolver:AttributeDefinition>

<resolver:AttributeDefinition id="givenName" xsi:type="Simple"
    xmlns="urn:mace:shibboleth:2.0:resolver:ad" sourceAttributeID="givenName">
    <resolver:Dependency ref="localLDAP"/>
    <resolver:AttributeEncoder xsi:type="SAML1String"
        xmlns="urn:mace:shibboleth:2.0:attribute:encoder"
        name="urn:mace:dir:attribute-def:givenName"/>
    <resolver:AttributeEncoder xsi:type="SAML2String"
        xmlns="urn:mace:shibboleth:2.0:attribute:encoder" name="urn:oid:2.5.4.42"
        friendlyName="givenName"/>
</resolver:AttributeDefinition>

<resolver:AttributeDefinition id="description" xsi:type="Simple"
    xmlns="urn:mace:shibboleth:2.0:resolver:ad" sourceAttributeID="description">
    <resolver:Dependency ref="localLDAP"/>
    <resolver:AttributeEncoder xsi:type="SAML1String"
        xmlns="urn:mace:shibboleth:2.0:attribute:encoder"
        name="urn:mace:dir:attribute-def:description"/>
    <resolver:AttributeEncoder xsi:type="SAML2String"
```

```
            xmlns="urn:mace:shibboleth:2.0:attribute:encoder" name="urn:oid:2.5.4.13"
            friendlyName="description"/>
    </resolver:AttributeDefinition>

    <resolver:AttributeDefinition id="cn" xsi:type="Simple"
        xmlns="urn:mace:shibboleth:2.0:resolver:ad" sourceAttributeID="cn">
        <resolver:Dependency ref="localLDAP"/>
        <resolver:AttributeEncoder xsi:type="SAML1String"
            xmlns="urn:mace:shibboleth:2.0:attribute:encoder" name="urn:mace:dir:attribute-de
        <resolver:AttributeEncoder xsi:type="SAML2String"
            xmlns="urn:mace:shibboleth:2.0:attribute:encoder" name="urn:oid:2.5.4.3"
            friendlyName="cn"/>
    </resolver:AttributeDefinition>

    <resolver:AttributeDefinition id="sn" xsi:type="Simple"
        xmlns="urn:mace:shibboleth:2.0:resolver:ad" sourceAttributeID="sn">
        <resolver:Dependency ref="localLDAP"/>
        <resolver:AttributeEncoder xsi:type="SAML1String"
            xmlns="urn:mace:shibboleth:2.0:attribute:encoder" name="urn:mace:dir:attribute-de
        <resolver:AttributeEncoder xsi:type="SAML2String"
            xmlns="urn:mace:shibboleth:2.0:attribute:encoder" name="urn:oid:2.5.4.4"
            friendlyName="sn"/>
    </resolver:AttributeDefinition>

    <resolver:AttributeDefinition id="uid" xsi:type="Simple"
        xmlns="urn:mace:shibboleth:2.0:resolver:ad" sourceAttributeID="uid">
        <resolver:Dependency ref="localLDAP"/>
        <resolver:AttributeEncoder xsi:type="SAML1String"
            xmlns="urn:mace:shibboleth:2.0:attribute:encoder" name="urn:mace:dir:attribute-de
        <resolver:AttributeEncoder xsi:type="SAML2String"
            xmlns="urn:mace:shibboleth:2.0:attribute:encoder" name="urn:oid:2.5.4.45"
            friendlyName="uid"/>
    </resolver:AttributeDefinition>
```

Add the new attributes to your `attribute-filter.xml` by adding a new AttributeFilterPolicy:

```
<afp:AttributeFilterPolicy id="attribFilter">
    <afp:PolicyRequirementRule xsi:type="basic:ANY"/>

    <afp:AttributeRule attributeID="givenName">
        <afp:PermitValueRule xsi:type="basic:ANY"/>
    </afp:AttributeRule>

    <afp:AttributeRule attributeID="displayName">
        <afp:PermitValueRule xsi:type="basic:ANY"/>
    </afp:AttributeRule>

    <afp:AttributeRule attributeID="description">
        <afp:PermitValueRule xsi:type="basic:ANY"/>
    </afp:AttributeRule>

    <afp:AttributeRule attributeID="cn">
        <afp:PermitValueRule xsi:type="basic:ANY"/>
    </afp:AttributeRule>

    <afp:AttributeRule attributeID="sn">
        <afp:PermitValueRule xsi:type="basic:ANY"/>
    </afp:AttributeRule>

    <afp:AttributeRule attributeID="uid">
        <afp:PermitValueRule xsi:type="basic:ANY"/>
    </afp:AttributeRule>
```

```
</afp:AttributeFilterPolicy>
```

Now you have to check if the generated metadata is correct. To do this, open the `idp-metadata.xml` file. Known issues are:

- Incorrect ports: For example port 8443 at the AttributeService Bindings instead of no specific port.
- Wrong X509Certificate for Attribute Resolver. Use your previously generated SSL/TLS `${IDP_HOST_CERTIFICATE}` instead.

After this, restart your Shibboleth IdP.

Shibboleth Service Provider

The installation procedure for the Shibboleth SP is different for all supported Operating Systems. The project describes the different installation methods in an own installation manual[127]. This documentation will provide help for the basic configuration to get the authentication process working with your EOxServer instance.

Important URLs for your Shibboleth SP:

- Status page: `https://${SPHOST}/Shibboleth.sso/Status`
- Metadata: `https://${SPHOST}/Shibboleth.sso/Metadata`
- Session summary: `https://${SPHOST}/Shibboleth.sso/Session`
- Local logout: `https://${SPHOST}/Shibboleth.sso/Logout`

Warning: SP resource paths are case sensitive!

STEP 1

The Shibboleth SP has two relevant configuration files. We begin with the `attribute-map.xml` file, where we configure the mapping of the attributes received from the IdP to the secured service (in our case the EOxServer):

```
<Attributes xmlns="urn:mace:shibboleth:2.0:attribute-map" xmlns:xsi="http://www.w3.org/2001/XMLSc

    <!-- First some useful eduPerson attributes that many sites might use. -->

    <Attribute name="urn:mace:dir:attribute-def:eduPersonPrincipalName" id="eppn">
        <AttributeDecoder xsi:type="ScopedAttributeDecoder"/>
    </Attribute>
    <Attribute name="urn:oid:1.3.6.1.4.1.5923.1.1.1.6" id="eppn">
        <AttributeDecoder xsi:type="ScopedAttributeDecoder"/>
    </Attribute>

    <Attribute name="urn:mace:dir:attribute-def:eduPersonScopedAffiliation" id="affiliation">
        <AttributeDecoder xsi:type="ScopedAttributeDecoder" caseSensitive="false"/>
    </Attribute>
    <Attribute name="urn:oid:1.3.6.1.4.1.5923.1.1.1.9" id="affiliation">
        <AttributeDecoder xsi:type="ScopedAttributeDecoder" caseSensitive="false"/>
    </Attribute>

    <Attribute name="urn:mace:dir:attribute-def:eduPersonAffiliation" id="unscoped-affiliation">
        <AttributeDecoder xsi:type="StringAttributeDecoder" caseSensitive="false"/>
    </Attribute>
    <Attribute name="urn:oid:1.3.6.1.4.1.5923.1.1.1.1" id="unscoped-affiliation">
        <AttributeDecoder xsi:type="StringAttributeDecoder" caseSensitive="false"/>
    </Attribute>

    <Attribute name="urn:mace:dir:attribute-def:eduPersonEntitlement" id="entitlement"/>
    <Attribute name="urn:oid:1.3.6.1.4.1.5923.1.1.1.7" id="entitlement"/>
```

[127] https://wiki.shibboleth.net/confluence/display/SHIB2/Installation

```
<!-- A persistent id attribute that supports personalized anonymous access. -->

<!-- First, the deprecated/incorrect version, decoded as a scoped string: -->
<Attribute name="urn:mace:dir:attribute-def:eduPersonTargetedID" id="targeted-id">
    <AttributeDecoder xsi:type="ScopedAttributeDecoder"/>
    <!-- <AttributeDecoder xsi:type="NameIDFromScopedAttributeDecoder" formatter="$NameQualif
</Attribute>

<!-- Second, an alternate decoder that will decode the incorrect form into the newer form. --
<!--
<Attribute name="urn:mace:dir:attribute-def:eduPersonTargetedID" id="persistent-id">
    <AttributeDecoder xsi:type="NameIDFromScopedAttributeDecoder" formatter="$NameQualifier!$
</Attribute>
-->

<!-- Third, the new version (note the OID-style name): -->
<Attribute name="urn:oid:1.3.6.1.4.1.5923.1.1.1.10" id="persistent-id">
    <AttributeDecoder xsi:type="NameIDAttributeDecoder" formatter="$NameQualifier!$SPNameQuali
</Attribute>

<!-- Fourth, the SAML 2.0 NameID Format: -->
<Attribute name="urn:oasis:names:tc:SAML:2.0:nameid-format:persistent" id="persistent-id">
    <AttributeDecoder xsi:type="NameIDAttributeDecoder" formatter="$NameQualifier!$SPNameQuali
</Attribute>

<!--Examples of LDAP-based attributes, uncomment to use these... -->
<Attribute name="urn:mace:dir:attribute-def:cn" id="cn"/>
<Attribute name="urn:mace:dir:attribute-def:sn" id="sn"/>
<Attribute name="urn:mace:dir:attribute-def:givenName" id="givenName"/>
<Attribute name="urn:mace:dir:attribute-def:mail" id="mail"/>
<Attribute name="urn:mace:dir:attribute-def:telephoneNumber" id="telephoneNumber"/>
<Attribute name="urn:mace:dir:attribute-def:title" id="title"/>
<Attribute name="urn:mace:dir:attribute-def:initials" id="initials"/>
<Attribute name="urn:mace:dir:attribute-def:description" id="description"/>
<Attribute name="urn:mace:dir:attribute-def:carLicense" id="carLicense"/>
<Attribute name="urn:mace:dir:attribute-def:departmentNumber" id="departmentNumber"/>
<Attribute name="urn:mace:dir:attribute-def:displayName" id="displayName"/>
<Attribute name="urn:mace:dir:attribute-def:employeeNumber" id="employeeNumber"/>
<Attribute name="urn:mace:dir:attribute-def:employeeType" id="employeeType"/>
<Attribute name="urn:mace:dir:attribute-def:preferredLanguage" id="preferredLanguage"/>
<Attribute name="urn:mace:dir:attribute-def:manager" id="manager"/>
<Attribute name="urn:mace:dir:attribute-def:seeAlso" id="seeAlso"/>
<Attribute name="urn:mace:dir:attribute-def:facsimileTelephoneNumber" id="facsimileTelephoneN
<Attribute name="urn:mace:dir:attribute-def:street" id="street"/>
<Attribute name="urn:mace:dir:attribute-def:postOfficeBox" id="postOfficeBox"/>
<Attribute name="urn:mace:dir:attribute-def:postalCode" id="postalCode"/>
<Attribute name="urn:mace:dir:attribute-def:st" id="st"/>
<Attribute name="urn:mace:dir:attribute-def:l" id="l"/>
<Attribute name="urn:mace:dir:attribute-def:o" id="o"/>
<Attribute name="urn:mace:dir:attribute-def:ou" id="ou"/>
<Attribute name="urn:mace:dir:attribute-def:businessCategory" id="businessCategory"/>
<Attribute name="urn:mace:dir:attribute-def:physicalDeliveryOfficeName" id="physicalDeliveryO

<Attribute name="urn:oid:2.5.4.3" id="cn"/>
<Attribute name="urn:oid:2.5.4.4" id="sn"/>
<Attribute name="urn:oid:2.5.4.42" id="givenName"/>
<Attribute name="urn:oid:0.9.2342.19200300.100.1.3" id="mail"/>
<Attribute name="urn:oid:2.5.4.20" id="telephoneNumber"/>
<Attribute name="urn:oid:2.5.4.12" id="title"/>
<Attribute name="urn:oid:2.5.4.43" id="initials"/>
<Attribute name="urn:oid:2.5.4.13" id="description"/>
<Attribute name="urn:oid:2.16.840.1.113730.3.1.1" id="carLicense"/>
<Attribute name="urn:oid:2.16.840.1.113730.3.1.2" id="departmentNumber"/>
```

```
    <Attribute name="urn:oid:2.16.840.1.113730.3.1.3" id="employeeNumber"/>
    <Attribute name="urn:oid:2.16.840.1.113730.3.1.4" id="employeeType"/>
    <Attribute name="urn:oid:2.16.840.1.113730.3.1.39" id="preferredLanguage"/>
    <Attribute name="urn:oid:2.16.840.1.113730.3.1.241" id="displayName"/>
    <Attribute name="urn:oid:0.9.2342.19200300.100.1.10" id="manager"/>
    <Attribute name="urn:oid:2.5.4.34" id="seeAlso"/>
    <Attribute name="urn:oid:2.5.4.23" id="facsimileTelephoneNumber"/>
    <Attribute name="urn:oid:2.5.4.9" id="street"/>
    <Attribute name="urn:oid:2.5.4.18" id="postOfficeBox"/>
    <Attribute name="urn:oid:2.5.4.17" id="postalCode"/>
    <Attribute name="urn:oid:2.5.4.8" id="st"/>
    <Attribute name="urn:oid:2.5.4.7" id="l"/>
    <Attribute name="urn:oid:2.5.4.10" id="o"/>
    <Attribute name="urn:oid:2.5.4.11" id="ou"/>
    <Attribute name="urn:oid:2.5.4.15" id="businessCategory"/>
    <Attribute name="urn:oid:2.5.4.19" id="physicalDeliveryOfficeName"/>

    <Attribute name="urn:oid:2.5.4.45" id="uid"/>
</Attributes>
```

The next step is to edit the `shibboleth2.xml` file: Locate the element `ApplicationDefaults` and set the value of the attribute `entityID` to `${SP_HOST}\Shibboleth`.

STEP 2

The next step is to configure your Apache HTTP Server. To do this, you have to generate a certificate and a key file for your SSL/TLS Shibboleth SP Host first (see Shibboleth IdP section). Then add a virtual host to your Apache HTTP Server:

```
 <VirtualHost _default_:443>

    # Include the apache22.conf from Shibboleth
    include ${SP_HOME}/apache22.config

    # Set appropriate document root here
    DocumentRoot "/var/www/"

    # Set your designated IDP host here
    ServerName ${IDP_HOST}

    # Set your designated logging directory here
    ErrorLog logs/ssl_error_log
    TransferLog logs/ssl_access_log
    LogLevel warn

    SSLEngine on

    SSLProtocol all -SSLv2

     # Important: mod_ssl should not verify the provided certificates
    SSLVerifyClient optional_no_ca

    SSLCipherSuite ALL:!ADH:!EXPORT:!SSLv2:RC4+RSA:+HIGH:+MEDIUM:+LOW

    # Set the correct paths to your certificate and key here
    SSLCertificateFile    ${SP_HOST_CERTIFICATE}
    SSLCertificateKeyFile ${SP_HOST_CERTIFICATE_KEY}

    <Files ~ "\.(cgi|shtml|phtml|php3?)$">
        SSLOptions +StdEnvVars
    </Files>
    <Directory "/var/www/cgi-bin">
        SSLOptions +StdEnvVars
    </Directory>
```

```
    SetEnvIf User-Agent ".*MSIE.*" nokeepalive ssl-unclean-shutdown downgrade-1.0 force-response-

    CustomLog logs/ssl_request_log "%t %h %{SSL_PROTOCOL}x %{SSL_CIPHER}x \"%r\" %b"

</VirtualHost>
```

STEP 3

Open `shibboleth2.xml` and change the `entityID` in the element `ApplicationDefaults` to your `${SP_HOST}`. Restart your SP and try to access your SP Metadata `https://${SPHOST}/Shibboleth.sso/Metadata`

Configure Shibboleth SP and IdP

- Download SP Metadata and store it locally as `${SP_METADATA_FILE}`.
- Open the `relying-party.xml` of the Shibboleth IdP and change the Metadata Provider entry to

```
<!-- MetadataProvider the combining other MetadataProviders -->
<metadata:MetadataProvider id="ShibbolethMetadata" xsi:type="metadata:ChainingMetadataPro


    <metadata:MetadataProvider id="IdPMD" xsi:type="metadata:ResourceBackedMetadataProvic
        <!-- This is usually set correctly by the IdP installation script -->
        <metadata:MetadataResource xsi:type="resource:FilesystemResource"
            file="${IDP_METADATA_FILE}"/>
    </metadata:MetadataProvider>

     <!-- This is the new MetadataProvider for your SP metadata -->
    <MetadataProvider id="URLMD" xsi:type="FilesystemMetadataProvider"
        xmlns="urn:mace:shibboleth:2.0:metadata"
        metadataFile="${SP_METADATA_FILE}">


        <MetadataFilter xsi:type="ChainingFilter" xmlns="urn:mace:shibboleth:2.0:metadata
            <MetadataFilter xsi:type="EntityRoleWhiteList"
                xmlns="urn:mace:shibboleth:2.0:metadata">
                <RetainedRole>samlmd:SPSSODescriptor</RetainedRole>
            </MetadataFilter>
        </MetadataFilter>


    </MetadataProvider>

</metadata:MetadataProvider>
```

- Add the `${SP_HOST_CERTIFICATE}` to your Java Keystore:

```
keytool -import -file ${SP_HOST_CERTIFICATE} -alias ${SP_HOST}
-keystore ${JAVA_JRE_HOME}\lib\security\cacerts
```

- Open `shibboleth2.xml` of your Shibboleth SP add a new SessionInitiator to the `Sessions` element:

```
<!-- Default example directs to a specific IdP's SSO service (favoring SAML 2 over Shib 1
<SessionInitiator type="Chaining" Location="/Login"
            isDefault="true" id="Intranet" relayState="cookie"
            entityID="https://{IDP_HOST}/idp/shibboleth">
            <SessionInitiator type="SAML2" acsIndex="1"
              template="bindingTemplate.html"/>
            <SessionInitiator type="Shib1" acsIndex="5"/>
</SessionInitiator>
```

- Then add a new MetadataProvider:

```
<!-- Chains together all your metadata sources. -->
<MetadataProvider type="Chaining">
            <MetadataProvider type="XML"
                        uri="https://{IDP_HOST}/idp/profile/Metadata/SAML"
                        backingFilePath="federation-metadata.xml"
                        reloadInterval="7200">
            </MetadataProvider>
</MetadataProvider>
```

  Alternatively you can reference the metadata from your local IdP:

```
<!-- Chains together all your metadata sources. -->
<MetadataProvider type="Chaining">
            <MetadataProvider type="XML"
                        path="${IDP_HOME}/metadata/idp-metadata.xml"
            </MetadataProvider>
</MetadataProvider>
```

- Restart your IdP, the SP and the Apache HTTPD

Configure the EOxServer Security Components

This section describes the configuration of the EOxServer security components.

General Configuration Options The configuration of the EOxServer security components is done in the `eoxserver.conf` configuration file of your EOxServer instance. All security related configuration is done in the section `[services.auth.base]`:

- `pdp_type`: Determines the Policy Decision Point type; defaults to `none` which deactivates authorisation. Currently, only the type `charonpdp` is implemented.
- `authz_service`: The URL of the Authorisation Service.
- `attribute_mapping`: The file path to a dictionary with a mapping from identity attributes received from the Shibboleth IdP to a XACMLAuthzDecisionQuery. If the key is set to `default`, a standard dictionary is used.
- `serviceID`: Identifier for the EOxServer instance to an external Authorisation Service. Is used as resource ID in an XACMLAuthzDecisionQuery. If the key is set to `default`, the host name will be used.
- `allowLocal`: If set to `True`, the security components will alloways allow access to requests from the local machine. *Use with care!*

Adding new Subject attributes to the EOxServer Security Components In order to register new Subject attributes from your LDAP to the IDMS, you have to configure the Shibboleth IdP, the Shibboleth SP, and the EOxServer. Let's assume we want to add the new attribute *foo*.

Shibboleth IdP

Add a new AttributeResolver to your `attribute-resolver.xml` configuration file:

```
<resolver:AttributeDefinition id="foo" xsi:type="Simple"
    xmlns="urn:mace:shibboleth:2.0:resolver:ad" sourceAttributeID="description">
    <resolver:Dependency ref="localLDAP"/>
    <resolver:AttributeEncoder xsi:type="SAML1String"
        xmlns="urn:mace:shibboleth:2.0:attribute:encoder"
        name="urn:mace:dir:attribute-def:description"/>
    <resolver:AttributeEncoder xsi:type="SAML2String"
        xmlns="urn:mace:shibboleth:2.0:attribute:encoder" name="foo"
        friendlyName="foo"/>
</resolver:AttributeDefinition>
```

Add or extend a AttributeFilterPolicy in your `attribute-filter.xml` configuration file:

```
<afp:AttributeFilterPolicy id="fooFilter">
    <afp:PolicyRequirementRule xsi:type="basic:ANY"/>

    <afp:AttributeRule attributeID="foo">
        <afp:PermitValueRule xsi:type="basic:ANY"/>
    </afp:AttributeRule>

</afp:AttributeFilterPolicy>
```

Shibboleth SP

Add the new attribute to the `attribute-map.xml`

```
<Attribute name="foo" id="foo"/>
```

EOxServer

- Make a copy of the default attribute dictionary (`{$EOXSERVER_CODE_DIRECTORY}/conf/defaultAttributeDic`
- Add the attribute:

  ```
  foo=foo
  ```

- Register the new dictionary in the EOxServer configuration.

SOAP Components

Table of Contents

- SOAP Components (page 90)
  - Security Token Service (page 90)
  - Policy Enforcement Point Service (page 91)
  - SOAP Security Proxy (page 92)
    - Generating the Proxy (page 92)
    - Installing the Proxy (page 92)

The following services are needed for the SOAP security part: The following services are needed for the SOAP security part:

- Security Token Service
- Charon Authorisation Service
- Policy Enforcement Point Service
- SOAP Security Proxy

To install and configure the HTTP secuirty components, you have to follow these steps:

1. Install the Charon *Authorisation Service* (page 72).
2. Install the *Security Token Service* (page 90).
3. Install the *Policy Enforcement Point Service* (page 91).
4. Install the *SOAP Security Proxy* (page 92).

Security Token Service

The Security Token Service (STS) is responsible for the authentication of users and is documented and specified in the OASIS WS-Trust[128] specification. The authentication assertion produced by the STS is formulated in the

[128] http://docs.oasis-open.org/ws-sx/ws-trust/200512/ws-trust-1.3-os.html

Security Assertion Markup Language[129]. A client trying to access a service secured by the IDMS has to embed this assertion in every service request.

The STS implementation used by the IDMS is the HMA Authentication Service[130]. Please refer to the documentation included in the `\docs` folder of the HMA Authentication Service package how to compile the service. This document will only deal on how to install the service. To deploy the service successfully, you first have to install and configure an LDAP service. Then proceed with the following steps:

- Put the `authentication_v2.1.aar` folder in the `${AXIS2_HOME}/WEB-INF/services/` folder. The `authentication_v2.1.aar` folder contains all configuration files for the STS.
- The main configuration of the service takes place in the `authentication-service.properties`.
- Using the `saml-ldap-attributes-mapping.properties`, you can map your LDAP attributes to SAML attributes if necessary.
- You may configure the logging behaviour in the Log4J configuration file in `authentication-service-log4j.properties`.

Following properties can be set in the `authentication-service.properties` configuration file:

`LDAPURL` URL to the LDAP service.

`LDAPSearchContext` Search context for users.

`LDAPPrincipal` The *"user name"* used by the STS to access the LDAP service.

`LDAPCredentials` The password used in combination with `LDAPPrincipal`

`KEYSTORE_LOCATION` Path to the Keystore file containing the certificate used for signing the SAML tokens.

`KEYSTORE_PASSWORD` The keystore password.

`AUTHENTICATION_CERTIFICATE_ALIAS` Alias of the keystore entry wich is used for signing the SAML tokens.

`AUTHENTICATION_CERTIFICATE_PASSWORD` Password corresponding to the `AUTHENTICATION_CERTIFICATE_ALIAS`

`CLIENT_CERTIFICATE_ALIASES` Comma serperated list with keystore aliases of trusted clients.

`SAML_TOKEN_EXPIRY_PERIOD` Defines how long a SAML token is valid.

`SAML_ASSERTION_ISSUER` SAML Token issure.

`SAML_ASSERTION_ID_PREFIX` SAML Token prefix.

`SAML_ASSERTION_NODE_NAMESPACE` Namespace for attribute assertions.

`ENCRYTION_ENABLE` Enables or disables encryption of SAML tokens.

`INCLUDE_CERTIFICATE` Enables or disables inclusion of SAML tokens.

`LOG4J_CONFIG_LOCATION` Path to the Log4J configuration file.

Policy Enforcement Point Service

Before installing the Policy Enforcement Point Service, refer to the *General Configuration for CHARON services* (page 79).

The Policy Enforcement Point enforces the authorisation decisions made by the Authorisation Service.

The next step is deploying the PEP Service, therefore extract the ZIP archive into the directory of your `${AXIS2_HOME}`.

[129] http://www.oasis-open.org/committees/download.php/3406/oasis-sstc-saml-core-1.1.pdf
[130] http://wiki.services.eoportal.org/tiki-index.php?page=HMA+Authentication+Service

Now you have to configure the service. The configuration files are in the `${AXIS2_HOME}/WEB-INF/classes` folder. Open the `PEPConfiguration.xml` to configure the service. The configuration file already contains documentation of the single elements.

SOAP Security Proxy

Before installing the SOAP Security Proxy, refer to the *General Configuration for CHARON services* (page 79). If you want to secure a Web Coverage Service, you can use the provided WCS Security Proxy. In this case, jump to *Installing the Proxy* (page 92).

Generating the Proxy The SOAP Proxy is used as a proxy for a secured service. This means a user client does not communicate directly with a secured service, instead it sends all requests to the proxy service.

First, you have to generate the proxy service. In order to do this, open a shell and navigate to the `${ProxyCodeGen_HOME}/bin` directory. Run the script to generate the proxy service:

- Linux, Unices:

```
./ProxyGen.sh -wsdl path/to/wsdl
```

- Windows:

```
.\ProxyGen.bat -wsdl path\to\wsdl
```

The parameter `-wsdl` points to a file with the WSDL of the secured service.

After a successful service generation, the folder `${ProxyCodeGen_HOME}/tmp/ dist` contains the new proxy service.

Installing the Proxy Take the service zip and deploy it by unpacking its content to the `${AXIS2_HOME}` folder. For MTOM support, please make sure that the parameter `enableMTOM` in the file `${AXIS2_HOME}/axis2.xml` is enabled.

Edit the `ProxyConfiguration_${SERVICE_NAME}.xml` to configure the service. The configuration file already contains documentation of the single elements.

1.14 SOAP Proxy

Table of Contents

- SOAP Proxy (page 92)
  - SOAP Access to WCS (page 92)
  - Installation (page 93)
    - Quick installation guide for EOxServer on CentOS (page 93)
    - Old installation guide without rpms (page 94)

1.14.1 SOAP Access to WCS

SOAP access to services provided by EOxServer is possible if the functionality is installed by the service provider. The protocol is SOAP 1.2 over HTTP.

EOxServer responds to the following WCS-EO requests via its SOAP service interface:

- DescribeCoverage
- DescribeEOCoverageSet
- GetCapabilities

- GetCoverage

To access the EOxServer by means of SOAP requests, you need to obtain the access ULR from the service provider. For machine readable configuration the SOAP service exposes the WSDL configuration file: given a service address of 'http://example.org/eo_wcs' the corresponding WSDL file may be downloaded at the URL 'http://example.org/eo_wcs?wsdl'.

1.14.2 Installation

A quick-intall quide is provided below. For a full installation guide see the INSTALL file in the source tree.

Quick installation guide for EOxServer on CentOS

0. Prerequisites:

- *EOxServer* (page 14) installed and configured, including MapServer and Apache HTTP Server
- Add the yum repository as described in the *Installation on CentOS* (page 17) available at http://packages.eox.at (recommended) or directoy obtain the RPM packages from http://yum.packages.eox.at/el/6/testing/x86_64/.

1. Basic install:

The following standard installation sets up soap_proxy for an installed eoxserver service accessible at http://127.0.0.1/eoxserver/ows

Caution: if upgrading an existing installation of soap_proxy, please be sure to make a backup of the directory */usr/share/axis2c_eo/services/soapProxy*. The eo_soap_proxy-1.0.1-1 package does not correctly preserve this directory duing upgrading.

Via the repository:

```
sudo yum install axis2c_eo eo_soap_proxy
sudo /etc/init.d/httpd restart
```

or the packages:

```
sudo rpm -i axis2c_eo-1.6.0-3.x86_64.rpm
sudo rpm -i eo_soap_proxy-1.0.1-1.x86_64.rpm
sudo /etc/init.d/httpd restart
```

2. Test:

To test open a webbrowser to the page:

http://<your_server>/sp_eowcs?wsdl

You should see the wsdl.

Further testing may be done via soapui. See the file `soap_proxy/test/README.txt` in the source tree.

3. Add another service:

To add another service to the basic installation, perform the following steps as root:

By way of example let us say our new soap_proxy service shall be available at http://example.org/sp_foo, and the corresponding backend eoxserver is accessible at http://127.0.0.1/eoxs_foo

First, in the directory `/usr/local/share/axis2c/services` recursively copy the subdirectory `soapProxy` to `soapFoo`:

```
cp -r soapProxy soapFoo
cd soapFoo
```

In `soapFoo` rename `libsoapProxy.so` and `soapProxy.wsdl`:

```
mv libsoapProxy.so libsoapFoo.so
mv soapProxy.wsdl soapFoo.wsdl
```

Note that if selinux is enabled you may need adjust the object type of `libsoapFoo.so`.

edit `soapFoo.wsdl` - at the bottom of the file chage `soap:address location` to the new endpoint:

```
<soap:address location="http://example.org/sp_foo"/>
```

edit `services.xml` - change ServiceClass, BackendURL, and SOAPOperationsURL:

```
<parameter name="ServiceClass" locked="xsd:false">soapFoo</parameter>
<parameter name="BackendURL">http://127.0.0.1/eoxs_foo/ows</parameter>
<parameter name="SOAPOperationsURL">http://example.org/sp_foo</parameter>
```

Optionally, you may consider updating the `<description>`.

Edit the file `/etc/httpd/conf.d/030_axis2c.conf`: In the block `<IfModule mod_proxy.c>`, add 'ProxyPass' and 'ProxyPassReverse' lines corresponding to your new service:

```
ProxyPass         /sp_foo  http://127.0.0.1/sp_axis/services/soapFoo
ProxyPassReverse  /sp_foo  http://127.0.0.1/sp_axis/services/soapFoo
```

Old installation guide without rpms

0. Prerequisites:

The following is required before you can proceed with installing soap_proxy:

- `mapserver` installed & configured.
- Apache `httpd` server(`httpd2` on some systems) installed and running
- `eoxserver` is optional

1. Old Non-rpm installation

This is suitable for general installation e.g. if you are not using eoxerver but wish to use mapserver direcly.

Warning: some of the configuration details are out of date, but the changes are not structural.

Also see the INSTALL file in the source tree.

Download from http://ws.apache.org/axis2/c/download.cgi

Make a directory for the code:

```
cd someplace
mkdir axis2c
setenv AXIS2C_HOME /path/to/someplace/axis2c
```

Follow the instructions in 'doc' to compile, and use something like the following configure line to get `mod_axis2` configured for compiling at the same time:

```
./configure --with-apache2="/usr/include/apache2" \
  --with-apr="/usr/include/apr-1" --prefix=${AXIS2C_HOME}
```

Execute the standard sequence:

```
make
make install
```

Copy `lib/libmod_axis2.so.0.6.0` to `<apache2 modules directory>` as `mod_axis2.so`.

Edit the file `${AXIS2C_HOME}/axis2.xml` and ensure that the parameter enableMTOM has the value `true`.

Check that the following directory exits, if not create it: `${AXIS2C_HOME}/services`

2. Deploy axis2 via your webserver

Configure `mod_axis2` in the apache server config file. On Suse Linux one might edit the file `/etc/apache2/default-server.conf`.

Set up a proxy:

```
<IfModule mod_proxy.c>
  ProxyRequests Off
  ProxyPass         /sp_wcs   http://127.0.0.1/o3s_axis/services/soapProxy
  ProxyPassReverse  /sp_wcs   http://127.0.0.1/o3s_axis/services/soapProxy
  ...
  <Proxy *>
    Order deny,allow
    Deny from all
    ...
  </Proxy>
</IfModule>
```

and deploy axis2:

```
LoadModule axis2_module  /usr/lib64/apache2/mod_axis2.so
Axis2RepoPath /path/to/AXIS2C_HOME
Axis2LogFile /tmp/ax2logs
Axis2MaxLogFileSize 204800
Axis2LogLevel info
<Location /o3s_axis>
    SetHandler axis2_module
</Location>
```

3. Verify the deployment of axis2

Resart the webserver (`httpd2`) and open the following page:

```
http://127.0.0.1/o3s_axis/services
```

You should get a page that displays the text “Deployed Services” and is otherwise blank.

4. Configure and Compile Soap Proxy.

Change your working directory to the service directory in the soap_proxy source code:

```
cd <...>/soap_proxy/service
```

In `soapProxy.wsdl` set `<soap:address location=.../>`. Copy TEMLATE_services.xml to `services.xml`. In `services.xml` set `BackendURL` to the address of eoxserver.

Now change to the src directory:

```
cd src
```

In your environment or in the `Makefile` set `AXIS2C_HOME` appropriately, and execute:

```
make inst
```

Restart you httpd server and check that http://127.0.0.1/o3s_axis/services shows the soapProxy service offering the four EO-WCS operations.

Further testing may be done via soapui. See the file `soap_proxy/test/README.txt` in the source tree.

1.15 EOxServer Presentations

Table of Contents

- EOxServer Presentations (page 96)
  - FOSS4G 2011, Denver (page 96)
  - AGIT 2011, Salzburg (page 96)
  - HMA-AWG February 2012, ESA ESRIN (page 96)
  - FOSSGIS 2012, Dessau (page 97)
  - Linuxwochen Wien 2012 (page 97)
  - FOSS4G-CEE 2012, Prague (page 97)
  - HMA-AWG June 2012, ESA ESRIN (page 97)
  - Sentinel-3 OLCI/SLSTR and MERIS/(A)ATSR workshop 2012, ESA ESRIN (page 97)
  - SOMAP 2012, Vienna (page 98)

This sections holds some links to presentations related to EOxServer.

1.15.1 FOSS4G 2011, Denver

WCS in MapServer 6.0[131]

`Download the presentation`

The FOSS4G[132] is a global conference focused on Free and Open Source Software for Geospatial, organized by OSGeo[133].

1.15.2 AGIT 2011, Salzburg

Introducing WCS 2.0, EO-WCS, and Open Source implementations (MapServer, rasdaman, and EOxServer) enabling the Online Data Access to Heterogeneous Multidimensional Satellite Data[134]

`Download the presentation`

The Angewandte Geoinformatik (AGIT)[135] is a conference for applied geo-informatics held annually in Salzburg, Austria. Since 5 years it includes the OSGeo Day where the presentation was given.

1.15.3 HMA-AWG February 2012, ESA ESRIN

WCS Standardization & Reference Implementation[136]

`Download the presentation`

[131] http://2011.foss4g.org/sessions/enhanced-support-ogcs-web-coverage-service-wcs-mapserver-60
[132] http://2011.foss4g.org/
[133] http://osgeo.org
[134] http://www.agit.at/index.php?option=com_content&task=view&id=132&Itemid=72
[135] http://agit.at
[136] https://wiki.services.eoportal.org/tiki-index.php?page=HMA%20AWG%20Meeting%231%202012%2015%20February%202012

The Heterogeneous Missions Access Architecture Working Group[137] has been defined by the European Space Agency together with other relevant EO data owners (national agencies, European institutions and industry) for the management of the evolution of the interoperability interface standards defined within the HMA project and in follow on activities.

1.15.4 FOSSGIS 2012, Dessau

EOxServer, GDAL, MapServer - Zugang zu großen Archiven von Erdbeobachtungsdaten[138]

`Download the presentation`

Freie und Open Source Software für Geoinformationssysteme (FOSSGIS)[139] is the German speaking annual OSGeo conference

1.15.5 Linuxwochen Wien 2012

EOxServer & Mapserver - Open Source Lösungen für Erdbeobachtungsdaten[140]

`Download the presentation`

Linuxwochen[141] is Austria's biggest event series dedicated to Open Source and Free Software.

1.15.6 FOSS4G-CEE 2012, Prague

EOxServer: A Solution for Online Access to Large Collections of Earth Observation Data[142]

`Download the presentation`

FOSS4G-CEE[143] & Geoinformatics 2012 is the first local conference focused on Free and Open Source Software for Geospatial in Central and Eastern Europe. This year, it is organized together with the traditional Geoinformatics FCE CTU conference in Prague.

1.15.7 HMA-AWG June 2012, ESA ESRIN

Web Coverage Service 2.0 MapServer Implementation[144]

`Download the presentation`

Description: See *HMA-AWG February 2012, ESA ESRIN* (page 96) above

1.15.8 Sentinel-3 OLCI/SLSTR and MERIS/(A)ATSR workshop 2012, ESA ESRIN

EOxServer - An Open Source Solution for Standardized Online Access to Earth Observation Data[145]

`Download the poster`

The Sentinel-3 OLCI/SLSTR and MERIS/(A)ATSR workshop[146] is organized by the European Space Agency, together with Eumetsat, and hosted in ESA-ESRIN, Frascati, Italy. The workshop is open to ESA Principle Investigators and co-investigators, scientists and students using MERIS/(A)ATSR data, future follow-on Sentinel-3

[137] https://wiki.services.eoportal.org/tiki-index.php?page=HMA+AWG
[138] http://www.fossgis.de/konferenz/2012/programm/events/379.de.html
[139] http://www.fossgis.de/konferenz.html
[140] http://linuxwochen.at/index.php?option=com_content&view=article&id=331&Itemid=83
[141] http://linuxwochen.at/
[142] http://foss4g-cee.org/program/presentations/eoxserver-a-solution-for-online-access-to-large-collections-of-earth-observation-data/
[143] http://foss4g-cee.org/
[144] https://wiki.services.eoportal.org/tiki-index.php?page=HMA%20AWG%20Meeting%20no2%202012%208%20June%202012
[145] http://congrexprojects.com/sen3symposium/poster-sessions
[146] http://www.sen3symposium.org/

OLCI/SLSTR data users, representatives from GMES services, national, European and international space agencies and value adding industries.

1.15.9 SOMAP 2012, Vienna

EOxServer - Accessing Large Archives of Earth Observation Data Online[147] (photo[148])

`Download the presentation`

The Symposium on Service-Oriented Mapping[149] aims to be a multidisciplinary event, spanning from computer science to geobusiness. The aim is to bring together various stakeholders in the area of Service-Oriented mapping (data producers, mapping agencies and companies, infrastructure providers, software developers, cartographers, artists, ...) in order to discuss the influence of this new production environment (the networked spatial infrastructure and its service-oriented distribution) on the map production and the perspectives of the new paradigm for research and development in cartography.

1.16 Configuration Options

In this section, all valid configuration options and their interpretations are listed.

1.16.1 [core.system]

`instance_id`

Mandatory. The ID (name) of your instance. This is used on several locations throughout EOxServer and is inserted into a number of service responses.

`logging_filename`

Mandatory. The value of this option shall be a valid path to an existing file where all logs made by EOxServer will be saved. The process running EOxServer needs write permissions to that file.

`logging_format`

The format used to write each log entry to the log file. Since EOxServer uses the standard library logging[150] for all logging purposes, the value of this parameter has to adhere to the logging format rules[151] of the module.

`logging_level`

This parameter determines which logging levels are to be inserted into the logfile. The possible values are (from lowest priority to highest): *DEBUG*, *INFO*, *WARNING*, *ERROR* and *CRITICAL* whereas *DEBUG* is the default. Only messages with at least this level are actually written to the logfile.

1.16.2 [core.interfaces]

`runtime_validation_level`

The runtime validation level. Tells the core whether to include type checks at runtime. Possible values are 'trust', 'warn', 'fail'. Defaults to 'trust'.

[147] http://somap.cartography.at/?SOMAP_2012:Program:November_23rd_2012
[148] http://somap.cartography.at/plugins/gallery/includes/image.php?pic=L2hvbWUvLnNpdGVzLzEyL3NpdGUyNDMvd2ViL3NvbWFwMjAxMi9nYWxsZXJ5L
[149] http://somap.cartography.at/?SOMAP_2012
[150] http://docs.python.org/library/logging.html
[151] http://docs.python.org/library/logging.html#logrecord-attributes

1.16.3 [core.ipc]

In this section, options for controlling inter process communication will be added, once it is implemented.

1.16.4 [core.registry]

`module_dirs`

This parameter is currently not used.

`modules`

Mandatory. A comma-separated list of modules that contain implementations of EOxServer interfaces. Use module identifiers as with normal Python import statements[152].

`system_modules`

This parameter is currently not used.

1.16.5 [processing.gdal.reftools]

`vrt_tmp_dir`

A path to a directory for temporary files created during the orthorectification of referencial coverages. This configuration option defaults to the systems standard[153].

1.16.6 [backends.cache]

In future, options in this section will influence the behavior of caching of FTP and rasdaman data.

1.16.7 [resources.coverages.coverage_id]

`reservation_time`

Determines the time a coverage ID is reserved when inserting a coverage into the system. Needs to be in the following form: <days>:<hours>:<minutes>:<seconds> and defaults to *0:0:30:0*.

1.16.8 [services.owscommon]

`http_service_url`

Mandatory. This parameter is the actual domain and path URL to the OWS services served with the EOxServer instance. This parameter is used in various contexts and is also included in several OWS service responses.

1.16.9 [services.ows.wms]

`supported_formats=<MIME type>[,<MIME type>[,<MIME type> ... ]]`

A comma-separated list of MIME-types defining the raster file format supported by the WMS `getMap()` operation. The MIME-types used for this option must be defined in the *Format Registry* (see "*Supported Raster File Formats and Their Configuration* (page 103)").

[152] http://docs.python.org/reference/simple_stmts.html#the-import-statement
[153] http://docs.python.org/library/tempfile.html#tempfile.mkstemp

```
supported_crs= <EPSG-code>[,<EPSG-code>[,<EPSG-code> ... ]]
```

List of common CRSes supported by the WMS `getMap()` operation (see also "*Supported CRSs and Their Configuration* (page 102)").

1.16.10 [services.ows.wcs]

```
supported_formats=<MIME type>[,<MIME type>[,<MIME type> ... ]]
```

A comma-separated list of MIME-types defining the raster file format supported by the WCS `getCoverage()` operation. The MIME-types used for this option must be defined in the *Format Registry* (see "*Supported Raster File Formats and Their Configuration* (page 103)").

```
supported_crs= <EPSG-code>[,<EPSG-code>[,<EPSG-code> ... ]]
```

List of common CRSes supported by the WCS `getMap()` operation. (see also "*Supported CRSs and Their Configuration* (page 102)").

1.16.11 [services.ows.wcs20]

```
paging_count_default
```

The maximum number of *wcs:coverageDescription* elements returned in a WCS 2.0 *EOCoverageSetDescription*. This also limits the *count parameter* (page 43). Defaults to 10.

```
default_native_format=<MIME-type>
```

The default *native format* cases when the source format cannot be used (read-only GDAL driver) and there is no explicit source-to-native format mapping. This option must be always set to a valid format (GeoTIFF by default). The MIME-type used for this option must be defined in the *Format Registry* (see "*Supported Raster File Formats and Their Configuration* (page 103)").

```
source_to_native_format_map=[<src.MIME-type,native-MIME-type>[,<src.MIME-type,native-MIME-type> .
```

The explicit source to native format mapping. As the name suggests, it defines mapping of the (zero, one, or more) source formats to a non-defaults native formats. The source formats are not restricted to the read-only ones. This option accepts comma-separated list of MIME-type pairs. The MIME-types used for this option must be defined in the *Format Registry* (see "*Supported Raster File Formats and Their Configuration* (page 103)").

1.16.12 [services.ows.wcst11]

```
allow_multiple_actions
```

This flag enables/disables mutiple actions per WCSt request. Defaults to *False*.

NOTE: It is safer to keep this feature disabled. In case of a failure of one of the multiple actions, an OWS exception is returned without any notification which of the actions were actually performed, and which have not been performed at all. Therefore, we recomend to use only one action per request.

```
allowed_actions
```

Comma-separated list of allowed actions. Each item is one of *Add*, *Delete*, *UpdateAll*, *UpdateMetadata* and *UpdateDataPart*. By default no action is allowed and each needs to be explicitly activated. Currently, only the *Add* and *Delete* actions are implemented by the EOxServer.

```
path_wcst_temp
```

Mandatory. A path to an existing directory for temporary data storage during the WCS-T request processing. This should be a directory which is not used in any other context, since it might be cleared under certain circumstances.

`path_wcst_perm`

Mandatory. A path to a directory for permanent storage of transacted data. This is the final location where transacted datasets will be stored. It is also a place where the *Delete* action (when enabled) is allowed to remove the stored data.

1.16.13 [services.auth.base]

For detailed information about authorization refer to the documentation of the *Identity Management System* (page 69).

`pdb_type`

Determine the Policy Decision Point type; defaults to 'none' which deactives authorization.

`authz_service`

URL of the Authorization Service.

`attribute_mapping`

Path to an attribute dictionary for user attributes.

`serviceID`

Sets a custom service identifier

`allowLocal`

Allows full local access to the EOxServer. Use with care!

1.16.14 [webclient]

The following configuration options affect the behavior of the *Webclient interface* (page 62).

`preview_service`
`outline_service`

The service type for the outline and the preview layer in the webclient map. One of *wms* (default) or *wmts*.

`preview_url`
`outline_url`

The URL of the preview and outline service. Defaults to the vaule of the *services.owscommon.http_service_url* configuration option.

1.16.15 [testing]

These configuration options are used within the context of the *Autotest instance* (page 118).

`binary_raster_comparison_enabled`

Enable/disable the binary comparison of rasters in test runs. If disabled these tests will be skipped. By default this feature is activated but might be turned off in order to prevent test failures originating on platform differences.

`rasdaman_enabled`

Enable/disable rasdaman test cases. If disabled these tests will be skipped. Defaults to *false*.

1.17 Supported CRSs and Their Configuration

Table of Contents

- Supported CRSs and Their Configuration (page 102)
  - Coordinate Reference Systems (page 102)
  - Web Map Service (page 102)
  - Web Coverage Service (page 102)

This section describes configuration of Coordinate Reference Systems for both WMS and WCS services.

1.17.1 Coordinate Reference Systems

The Coordinate Reference System (CRS) denotes the projection of coordinates to an actual position on Earth. EOxServer allows the configuration of supported CRSes for WMS and WCS services. The CRSes used by EOxServer are specified exclusively by means of EPSG numerical codes[154].

1.17.2 Web Map Service

EOxServer allows the specification of the overall list of CRSes supported by all published map layers (listed at the top layer of the WMS `Capabilities` XML document). In case of no common CRS the list can be empty. In addition to the list of common CRSes each individual layer has its *native* CRS which need not to be necessarily listed among the common CRSes. The meaning of the *native* CRS changes based on the EO dataset:

- Rectified Datasets - the actual CRS of the source geo-rectified raster data,
- Rectified Stitched Mosaic - the actual CRS of the source geo-rectified raster data,
- Referenceable Dataset - the CRS of the geo-location grid tie-points.
- Time Series - always set to WGS 84 (may be subject to change in future).

This *native* CRS is also used as the CRS in which the geographic extent (bounding-box) is published.

The list of WMS common CRSes is specified as a comma separated list of EPSG codes in the EOxServer's configuration (`<instance path>/conf/eoxserver.conf`) in section `serices.ows.wms`:

```
[services.ows.wms]
supported_crs= <EPSG-code>[,<EPSG-code>[,<EPSG-code> ... ]]
```

1.17.3 Web Coverage Service

EOxServer allows the specification of a list of CRCes to be used by the WCS. These CRSes can be used to select subsets of the desired coverage or, in case of rectified datasets (including rectified stitched mosaics) to specify the CRS of the output image data. The latter case is not applicabe to referenceable datasets as these are always returned in the original image geometry.

The list of WCS supported CRSes is specified as a comma-separated list of EPSG codes in the EOxServer configuration (`<instance path>/conf/eoxserver.conf`) in section `serices.ows.wcs`:

```
[services.ows.wcs]
supported_crs= <EPSG-code>[,<EPSG-code>[,<EPSG-code> ... ]]
```

[154] http://www.epsg-registry.org

1.18 Supported Raster File Formats and Their Configuration

Table of Contents

- Supported Raster File Formats and Their Configuration (page 103)
  - Format Registry (page 103)
  - Format Configuration (page 103)
  - Web Coverage Service - Format Configuration (page 104)
  - Web Coverage Service - Native Format Configuration (page 104)
  - Web Map Service - Format Configuration (page 104)
  - References (page 105)

In this section, the EOxServer's handling of raster file formats and OWS service specific format configuration is described.

1.18.1 Format Registry

The format registry is the list of raster file formats recognised by EOxServer. It holds definitions of both input and output formats. Each format record defines the MIME-type (unique, primary key), library, driver, and the default file extension.

Currently, EOxServer handles the raster data exclusively by means of the GDAL[155] library. Thus, in principle, any raster file format supported by the GDAL[156] library is supported by EOxServer. In particular, any raster file format readable by the GDAL library (provided that the file structure can be decomposed to one single-type, single- or multi-band image) can be used as the input and, vice versa, any raster file format writeable by the GDAL library can used as the output produced by WCS and WMS services.

Any raster file format intended to be used by EOxServer must be defined in the format registry. The format registry then provides unique mappings from MIME-type to the (GDAL) format driver.

1.18.2 Format Configuration

The format registry configuration is split in two parts (files):

- per-installation (mandatory) format configuration (set up automatically during the EOxServer installation) defining the default baseline set of formats (`<instal.path>/eoxserver/conf/default_formats.conf`).
- per-instance (optional) format configuration allowing customization of the format registry (`<instance path>/conf/formats.conf`).

In case of conflicting format definitions, the per-instance configuration takes precedence. Both formats' configuration files share the same text file format.

The formats' configuration is a simple text file containing a simple list of format definitions. One format definition (record) per line. Each record is then a comma separated list of the following text fields:

```
<MIME-type>, <driver>, <file extension>
```

The mime type is used as the primary key and thus any repeated MIME-type will rewrite the previous format definition(s) using this MIME-type. The driver field should be in format `GDAL/<GDAL driver name>`. To list available drivers provided by your GDAL installation use the following command:

```
gdalinfo --formats
```

[155] http://www.gdal.org
[156] http://www.gdal.org/formats_list.html

The GDAL prefix is used as place-holder to allow future use of additional library back-ends. The file extension shall be written including the separating dot .. Any leading or trailing white-characters as well as empty lines are ignored. The # character is used as line-comment and any content between this character and the end of the line is ignored.

An example format definition:

```
image/tiff,GDAL/GTiff,.tif # GeoTIFF raster file format
```

Since the list of supported drivers may vary for different installations of the back-end (GDAL) library, the library drivers are checked by EOxServer ignoring any format definitions requiring non-supported library drivers. Any invalid format record is reported to the EOxServer log. Further, EOxServer checks automatically which of the library drivers are 'read-only', i.e., which cannot be used to produce output images, and restricts these to be used for data input only.

1.18.3 Web Coverage Service - Format Configuration

The list of the file formats supported by the *Web Coverage Service* (WCS) is specified in the EOxServer configuration (<instance path>/conf/eoxserver.conf) in the section serices.ows.wcs:

```
[services.ows.wcs]
supported_formats=<MIME type>[,<MIME type>[,<MIME type> ... ]]
```

The supported WCS formats are specified as a comma-separated list of MIME-types. The listed MIME-types must be defined in the format registry otherwise they will be ignored. Read-only file formats will also be ignored.

The supported formats are announced through the WCS Capabilities and CoverageDescription (the output may vary based on the WCS version used). The use of in invalid MIME-types (not listed among the supported formats) in getCoverge() requests will lead to errors (OWS Exceptions).

1.18.4 Web Coverage Service - Native Format Configuration

The *native format* (as defined by WCS 2.0.1 [OGC 09-110r4][157]) is the default raster file format returned by the getCoverage() operation in case of a missing explicit format specification. By default, EOxServer sets the *native format* to the format of the stored source data (source format), however, in cases when the source format cannot be used ('read-only' source format) and/or another default format is desired, EOxServer allows the configuration of WCS *native formats* (<instance path>/conf/eoxserver.conf, section services.ows.wcs20):

```
[services.ows.wcs20]
default_native_format=<MIME-type>
source_to_native_format_map=[<src.MIME-type,native-MIME-type>[,<src.MIME-type,native-MIME-type> .
```

The default *native format* option is used in cases when the source format cannot be used (read-only) and no source to native format mapping is present. This option must always be set to a valid format (GeoTIFF by default). The source to native format mapping, as the name suggests, maps the (zero, one, or more) source format(s) to non-default native formats. The source formats are not restricted to the read-only ones. This option accepts a comma-separated list of MIME-type pairs.

1.18.5 Web Map Service - Format Configuration

The list of the file formats supported by the *Web Map Service's* (WMS) getMap() operation is specified in the EOxServer configuration (<instance path>/conf/eoxserver.conf) in section serices.ows.wms:

```
[services.ows.wms]
supported_formats=<MIME type>[,<MIME type>[,<MIME type> ... ]]
```

[157] http://www.opengeospatial.org/standards/wcs

The supported WMS formats are specified as a comma-separated list of MIME-types. The listed MIME-types must be defined in the format registry otherwise they will be ignored. The read-only file formats will be ignored.

The supported formats are announced through the WMS `Capabilities` (the output may vary based on the WMS version used).

1.18.6 References

[OGC 09-110r4] http://www.opengeospatial.org/standards/wcs

1.19 Asynchronous Task Processing

Table of Contents

- Asynchronous Task Processing (page 105)
  - Introduction (page 105)
  - Tasks (page 105)
    - Introduction (page 105)
    - Life-cycle (page 106)
  - ATP Installation and Configuration (page 106)
  - ATP Operation (page 107)
  - ATP Demo Application (page 108)
  - Performance considerations (page 108)
  - Further reading (page 108)

1.19.1 Introduction

The *Asynchronous Task Processing* (ATP) subsystem, as the name suggests, extends the *EOxServer* functionality by the ability to process tasks asynchronously, i.e., in background independently of the default *EOxServer's* synchronous client request processing.

Although the ATP subsystem is primarily designed to support asynchronous request processing of OGC Web Services such as the Web Coverage Service transaction extension (WCS-T) and/or the Web Processing Service (WPS), it is not limited to these and other parts of *EOxServer* may use it as well.

The ATP subsystem employs the model of a single shared task queue and one or more *Asynchronous Task Processing Daemons* (APTD) executing the pending tasks pulled from the task queue. A single ATPD is not restricted to a single processed task at time and it can internally process multiple tasks concurrently, e.g., by employing a pool of parallel worker threads assigned to multiple CPU cores.

The ATP subsystem is implemented as Django application using a DB model as the task queue. Although the underlying DB storage may be seen as suboptimal in terms of performance and latency it assures tolerance of the subsystem to possible failures or maintenance shut-downs of both *EOxServer* and/or APTDs.

1.19.2 Tasks

Introduction

For the correct operation of the ATP subsystem it is essential to understand the concept of a *task* and its life-cycle.

A *task* is an atomic and isolated action (amount of work) to be performed by *EOxServer*. When created, each task has a handler subroutine (python code to be executed) and a set of task specific input parameters to be processed by the handler subroutine. When finished, the tasks produce outputs.

The tasks may be created by different applications (*EOxServer's* apps and services). The tasks sharing the same handler subroutine and generic parameters belong to the same task *type*.

The ATP is expected to be shared by multiple applications. APTDs pull the tasks from the shared queue in First-In-First-Out fashion (regardless of the task type) and execute the given handler subroutines. Significant benefit of this shared nature of the APT subsystem is the control over the processing resources (pool of workers) and isolation of the execution details from the application (isolated from details such as the number of ATPD and working threads).

Life-cycle

The life-cycle of an asynchronous task, i.e., its possible states and state transitions are displayed in Fig.3.

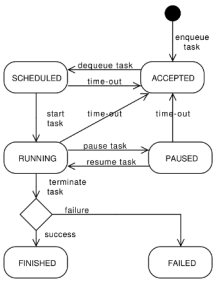


Figure 1.17: *Fig.1: ATP Task State Diagram*

Any existing task can be in one of the following states:

- ACCEPTED - a new enqueued task waiting to be pulled by an ATPD (initial state)
- SCHEDULED - a task pulled (dequeued) by an ATPD but not yet started
- RUNNING - a task being processed by an ATPD
- PAUSED - a task which has been put on hold and which is waiting to be resumed
- FINISHED - a task which has been finished successfully (terminal state)
- FAILED - a task which has been finished by a failure (terminal state)

When a task is created and enqueued for processing (ACCEPTED) it is stored in the DB task queue waiting for an ATPD to pull the task out. In this state, it is safely stored and protected against failures and shut-downs of both of the producer (ATPD can access the DB) and of the ATPD (producer can access the DB).

When a task is in one of the intermediate states (SCHEDULED, RUNNING, or PAUSED) it is being processed by an ATPD and it is vulnerable to possible failures. In these states, any unexpected crash of the ATPD could leave a task in an intermediate state forever. Therefore each task type has assigned a security time-out after which the task is considered to be abandoned and shall be re-enqueued for new processing (ACCEPTED). A task, however, can be re-enqueued for limited times (3 times by default). After the number of restarts has been exceeded the task will be rejected (FAILED). This mechanism ensures that no task would be abandoned unfinished after an occasional ATPD crash but also that a defective task would get stacked in the time-out loop.

When a task is in one of the terminal states (FINISHED or FAILED) it is safely stored in the DB. By default a terminated task will be stored forever, however, it is possible to specify an task type specific time-out after which the terminated tasks will be removed automatically.

1.19.3 ATP Installation and Configuration

There are no specific steps to install and configure the ATP subsystem except the basic *EOxServer* installation and configuration. The ATP is tightly coupled with *EOxServer* and works right out of box.

To track the status of the executed tasks and view the stored outputs auxiliary ATP HTML views can be enabled by adding following lines to the URL patterns ('url.py' configuration file) of the actual *EOxServer* instance:

```
urlpatterns = patterns('',

    ...

    (r'^process/status$', procViews.status ),
    (r'^process/status/(?P<requestType>[^/]{,64})/(?P<requestID>[^/]{,64})$', procViews.status ),
    (r'^process/task$', procViews.task ),
    (r'^process/response/(?P<requestType>[^/]{,64})/(?P<requestID>[^/]{,64})', procViews.response

    ...
)
```

1.19.4 ATP Operation

The ATP operation requires at least one ATPD to be running. Currently, there is only one ATPD implemented in EOxServer. This ATPD uses multiple sub-processes to process the tasks concurrently. By default, the numbers of sub-processes equals the number of available CPU cores. This ATPD can be executed as follows:

```
$ export PYTHONPATH=<EOxServer install.path>:<EOxServer instance path>
$ export DJANGO_SETTINGS_MODULE=autotest.settings
$ <EOxServer install.path>/tools/asyncProcServer.py

[0x504DD5AE614D562C] INFO: Default number of working threads: 4
[0x504DD5AE614D562C] INFO: 'autotest.settings' ... is set as the Django settings module
SpatiaLite version ..: 2.4.0    Supported Extensions:
    - 'VirtualShape'    [direct Shapefile access]
    - 'VirtualDbf'      [direct Dbf access]
    - 'VirtualText'     [direct CSV/TXT access]
    - 'VirtualNetwork'  [Dijkstra shortest path]
    - 'RTree'       [Spatial Index - R*Tree]
    - 'MbrCache'        [Spatial Index - MBR cache]
    - 'VirtualFDO'      [FDO-OGR interoperability]
    - 'SpatiaLite'      [Spatial SQL - OGC]
PROJ.4 Rel. 4.7.1, 23 September 2009
GEOS version 3.2.2-CAPI-1.6.2
[0x504DD5AE614D562C] INFO: ATPD Asynchronous Task Processing Daemon has just been started!
[0x504DD5AE614D562C] INFO: ATPD: id=0x504DD5AE614D562C (5786516041174439468)
[0x504DD5AE614D562C] INFO: ATPD: hostname=localhost
[0x504DD5AE614D562C] INFO: ATPD: pid=3295
```

The `PYTHONPATH` and `DJANGO_SETTINGS_MODULE` values can be passed as command line arguments by the '-p' and '-s' options, respectively. The default number of worker sub-processes can be overridden by the '-n' option:

```
$ <EOxServer install.path>/tools/asyncProcServer.py -n 6 -s "autotest.settings" -p "<EOxServer ins

[0xADDB15DB482ED425] INFO: Default number of working threads: 4
[0xADDB15DB482ED425] INFO: Setting number of working threads to: 6
[0xADDB15DB482ED425] INFO: 'autotest.settings' ... is set as the Django settings module
SpatiaLite version ..: 2.4.0    Supported Extensions:
    - 'VirtualShape'    [direct Shapefile access]
    - 'VirtualDbf'      [direct Dbf access]
    - 'VirtualText'     [direct CSV/TXT access]
    - 'VirtualNetwork'  [Dijkstra shortest path]
    - 'RTree'       [Spatial Index - R*Tree]
    - 'MbrCache'        [Spatial Index - MBR cache]
    - 'VirtualFDO'      [FDO-OGR interoperability]
    - 'SpatiaLite'      [Spatial SQL - OGC]
PROJ.4 Rel. 4.7.1, 23 September 2009
GEOS version 3.2.2-CAPI-1.6.2
[0xADDB15DB482ED425] INFO: ATPD Asynchronous Task Processing Daemon has just been started!
```

```
[0xADDB15DB482ED425] INFO: ATPD: id=0xADDB15DB482ED425 (-5919113253695335387)
[0xADDB15DB482ED425] INFO: ATPD: hostname=holly3
[0xADDB15DB482ED425] INFO: ATPD: pid=3345
```

The server can be gracefully terminated by using 'Ctrl-C' or the TERM signal.

1.19.5 ATP Demo Application

There is a demo application showing the running of the ATPD and the ATP as such available in the default EOxServer installation. This demo application can be executed as follows:

```
$ export PYTHONPATH=<EOxServer install.path>:<EOxServer instance path>
$ export DJANGO_SETTINGS_MODULE=autotest.settings
$ <EOxServer install.path>/atp_test.py
SpatiaLite version ..: 2.4.0    Supported Extensions:
    - 'VirtualShape'    [direct Shapefile access]
    - 'VirtualDbf'      [direct Dbf access]
    - 'VirtualText'     [direct CSV/TXT access]
    - 'VirtualNetwork'  [Dijkstra shortest path]
    - 'RTree'       [Spatial Index - R*Tree]
    - 'MbrCache'        [Spatial Index - MBR cache]
    - 'VirtualFDO'      [FDO-OGR interoperability]
    - 'SpatiaLite'      [Spatial SQL - OGC]
PROJ.4 Rel. 4.7.1, 23 September 2009
GEOS version 3.2.2-CAPI-1.6.2
ENQUEUE: test_5710ffb4189c4345aebde828d2bbc640 000000
ENQUEUE: test_47e161ec633b4105a1d174759f4a933d 000001
ENQUEUE: test_e53cf3ae654a447191e1308d805d8777 000002
ENQUEUE: test_fb71659cb9274383a8820e0110c86e15 000003
ENQUEUE: test_0e6e5edcdf8244d9b25a932cbd8c6112 000004
ENQUEUE: test_be5fa7af84444c47aba731c8e816f99b 000005
ENQUEUE: test_aae3faa14b5e4f48b8cabae7a0b01a3b 000006
ENQUEUE: test_6be7ea23f0984efbb09181503aa1a974 000007
```

1.19.6 Performance considerations

The ATP is designed for resource demanding longer running tasks (10 seconds and more) which in case of a synchronous operation could clog the system or lead to connection time-outs. On contrary, *light* tasks (less than 1 sec.) should preferably be executed synchronously.

1.19.7 Further reading

The database model used in the ATP subsystem is described in the *Task Tracker Data Model* (page 118) section. The developers' guide, helping with the creation of ATP based applications, can be found in the *Asynchronous Task Processing - Developers Guide* (page 128) section. The complete API reference can be found in *Module eoxserver.resources.processes.tracker* (page 185).

1.20 Web Coverage Service - Transaction Extension

Table of Contents

- Web Coverage Service - Transaction Extension (page 108)
  - Introduction (page 109)
  - Implementation Details (page 109)
    - Configuration (page 109)
    - Adding New Coverages (page 109)
    - Deleting Existing Coverages (page 110)
    - Asynchronous Operation (page 111)
  - References (page 111)

1.20.1 Introduction

This section describes the *Web Coverage Service - Transaction* (WCS-T) extension as implemented in *EOxServer*. The WCS-T interface is specified by the *Open Geospatial Consortium* (OGC) Web Coverage Service - Transaction operation extension (WCS-T) [OGC 07-068r4][158] standard which describes the invocation of the service in detail. The WCS-T functionality is closely related to the data model of the WCS 2.0 *Earth Observation Application Profile* (EO-WCS) employed by *EOxServer* and allows the specification of EO-WCS metadata for newly inserted EO datasets.

1.20.2 Implementation Details

EOxServer provides to option to insert (*Add* action) and delete (*Delete*) coverages (datasets in EO-WCS jargon) via the WCS-T service.

Configuration

For details on the WCS-T configuration see *[services.ows.wcst11]* (page 100).

Adding New Coverages

Currently, it is possible to insert only *Rectified* and *Referenceable* datasets. It is beyond the capabilities of the WCS-T service to assign datasets to container coverage types such as the *Rectified Stitched Mosaic* or *Dataset Series*. Neither is it possible to create plain (non-EO-WCS) coverages.

The input image data must be in valid GeoTIFF file format. No other file format is currently supported. The input is passed to the WCS-T service as a reference (URL, e.g., a *GetCoverage* KVP encoded request). It is not possible to embed the input image data in the WCS-T request.

The creation of a new EO-WCS dataset requires the specification of EO metadata. These metadata can be either passed by the user (recommended way) as a reference using the `ows:medatata` XML element, or generated automatically by the WCS-T service guessing some of the parameters from the GeoTIFF annotation.

The user provided EO-WCS metadata can be either in form of an EO-O&M XML document or arbitrary XML document with embedded EO-O&M XML fragment (such as the *DescribeCoverage* response of a WCS service).

The following is an example of a valid request to add a coverage:

```
<?xml version="1.0" encoding="UTF-8"?>
<wcst:Transaction service="WCS" version="1.1"
  xmlns:wcst="http://www.opengis.net/wcs/1.1/wcst"
  xmlns:ows="http://www.opengis.net/ows/1.1"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengis.net/wcs/1.1/wcst http://schemas.opengis.net/wcst/1.1/wcs
    <wcst:InputCoverages>
```

[158] http://portal.opengeospatial.org/files/?artifact_id=28506

```
        <wcst:Coverage>
            <!-- optional coverage identifier -->
            <ows:Identifier>CoverageId</ows:Identifier>
            <!-- reference to image data -->
            <ows:Reference
                xlink:href="http://foo.eox.at/ows?service=WCS&amp;version=2.0.0&amp;request=getCo
                xlink:role="urn:ogc:def:role:WCS:1.1:Pixels"/>
            <!-- optional reference to EO metadata -->
            <ows:Metadata
                xlink:href="http://foo.eox.at/ows?service=WCS&amp;version=2.0.0&amp;request=descr
                xlink:role="http://www.opengis.net/eop/2.0/EarthObservation"/>
            <wcst:Action codeSpace="http://schemas.opengis.net/wcs/1.1.0/actions.xml">Add</wcst:A
        </wcst:Coverage>
    </wcst:InputCoverages>
</wcst:Transaction>
```

The coverage identifier specified by the `ows:Identifier` element is optional. When not specified or not usable (most likely because it is already in use by another coverage) a new, unique identifier is generated automatically. Thus the WCS-T service is not bound to the user provided identifier and the actual identifier shall always be read from the transaction response:

```
<?xml version="1.0" encoding="utf-8"?>
<TransactionResponse
  xmlns="http://www.opengis.net/wcs/1.1/wcst"
  xmlns:ows="http://www.opengis.net/ows/1.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengis.net/wcs/1.1/wcst http://schemas.opengis.net/wcst/1.1/wcs
    <RequestId>wcstReq_btjiFfo4aOvT1BQL-ki5</RequestId>
    <ows:Identifier>wcstCov_LoEYNGm3d10ZhUUGdlmm</ows:Identifier>
</TransactionResponse>
```

Unless there is a need for a specific coverage identifier we recommend to leave the identifier selection to be performed by the WCS-T service and omit the `ows:Identifier` element in case of WCS-T coverage inserts.

Deleting Existing Coverages

The coverages inserted via the WCS-T *Add* action can be removed by means of the WCS-T *Delete* action. For security reasons, only the coverages inserted via WCS-T can be actually removed via WCS-T. The only parameter required in the removal request is the coverage (dataset) identifier (`wcst:InputCoverages` XML element):

```
<?xml version="1.0" encoding="UTF-8"?>
<wcst:Transaction service="WCS" version="1.1"
  xmlns:wcst="http://www.opengis.net/wcs/1.1/wcst"
  xmlns:ows="http://www.opengis.net/ows/1.1"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengis.net/wcs/1.1/wcst http://schemas.opengis.net/wcst/1.1/wcs
    <wcst:InputCoverages>
        <wcst:Coverage>
            <!-- required coverage identifier -->
            <ows:Identifier>wcstCov_LoEYNGm3d10ZhUUGdlmm</ows:Identifier>
            <wcst:Action codeSpace="http://schemas.opengis.net/wcs/1.1.0/actions.xml">Delete</wcst
        </wcst:Coverage>
    </wcst:InputCoverages>
</wcst:Transaction>
```

Asynchronous Operation

EOxServer supports asynchronous WCS-T requests as specified by the [OGC 07-068r4][159] standard. Asynchronous request processing can be invoked by any WCS-T request including the `wcst:ResponseHandler` element. This element shall contain an URL of the remote response handler where the response shall be sent once the asynchronous processing is finished:

```
<?xml version="1.0" encoding="UTF-8"?>
<wcst:Transaction service="WCS" version="1.1"
  xmlns:wcst="http://www.opengis.net/wcs/1.1/wcst"
  xmlns:ows="http://www.opengis.net/ows/1.1"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengis.net/wcs/1.1/wcst http://schemas.opengis.net/wcst/1.1/wcst
    <wcst:InputCoverages>
        ...
    </wcst:InputCoverages>
    <wcst:RequestId>RequestId</wcst:RequestId>
    <!-- XML element enabling the asynchronous WCS-T processing -->
    <wcst:ResponseHandler>http://foo.eox.at/WCSTResponseHandler</wcst:ResponseHandler>
</wcst:Transaction>
```

Currently, the WCS-T implementation supports HTTP and FTP URL schemas for the response handler. In the first case the response is delivered using HTTP/POST. In the latter case, the response is uploaded to a remote FTP server. In case of FTP, the user may specify a full file-name of the delivered file or target directory. If the FTP target is a directory the file-name of the stored response is generated from the request ID returned by the acknowledgement response:

```
<?xml version="1.0" encoding="utf-8"?>
<Acknowledgement
  xmlns="http://www.opengis.net/wcs/1.1/wcst"
  xmlns:ows="http://www.opengis.net/ows/1.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengis.net/wcs/1.1/wcst http://schemas.opengis.net/wcst/1.1/wcst
    <TimeStamp>2012-04-13T16:00:07Z</TimeStamp>
    <RequestId>wcstReq_6syhsJbO2TtYwVxFHOur</RequestId>
</Acknowledgement>
```

It is worth to mention that request identifiers can be specified in WCS-T requests, however this identifier provides only a hint to the WCS-T server and the server may change it to another value. Thus it is recommended to rely on the request identifier written in the WCS-T response and better omit the optional `wcst:RequestId` XML element in the WCS-T request.

It is possible to specify user/password for the response handler for both HTTP and FTP using the typical URL structure:

```
<schema>://[<username>@<password>]<host>/<path>
```

No other authentication is currently supported.

The asynchronous WCS-T operation requires the ATP (Asynchronous Task Processing) subsystem and, in particular, an ATPD (ATP Daemon) running. For more info on the ATP subsystem see the *Asynchronous Task Processing* (page 105) section.

1.20.3 References

[OGC 07-068r4] http://portal.opengeospatial.org/files/?artifact_id=28506

[159] http://portal.opengeospatial.org/files/?artifact_id=28506

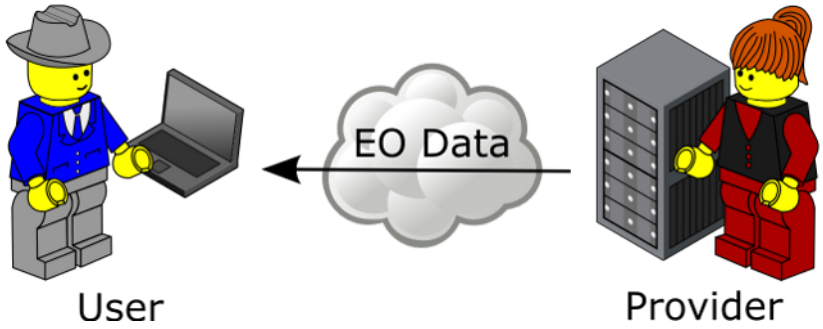
CHAPTER

TWO

EOXSERVER DEVELOPERS' GUIDE

The Developers' Guide is intended for people who want to use EOxServer as a development framework for geospatial services, or do have to extend EOxServer's functionality to implement specific data and metadata formats for instance.

Users of the EOxServer software stack please refer to the *EOxServer Users' Guide* (page 1). Users range from administrators installing and configuring the software stack and operators registering the available *EO Data* on the *Provider* side to end users consuming the registered *EO Data* on the *User* side.



2.1 Basics

Table of Contents

- Basics (page 113)
  - Architectural Layout (page 113)
    - Django (page 114)
    - Database (page 114)
    - MapServer (page 114)
    - GDAL/OGR (page 114)
  - Software Architecture (page 114)

The basic design of EOxServer has been proposed in *RFC 1: An Extensible Software Architecture for EOxServer* (page 248) and *RFC 2: Extension Mechanism for EOxServer* (page 270). Both are worth reading, although some of the concepts mentioned there have not (yet) been fully implemented.

This is a short description of the basic elements of the EOxServer software architecture.

2.1.1 Architectural Layout

EOxServer is Python software that builds on a handful of external packages. Most of the description in the following sections is related to the structure of the Python code, but in this section we present the building blocks

used for EOxServer.

For further information on the dependencies please refer to the *Installation* (page 14) document in the *EOxServer Users' Guide* (page 1).

Django

EOxServer is designed as a Django app. It reuses the object-relational mapping Django provides as an abstraction layer for database access. Therefore, it is not bound to a specific database application, but can be run with different backends.

Database

Metadata and part of the EOxServer configuration is stored in a database. A handful of geospatially enabled database systems is supported, though we recommend either PostGIS or SpatiaLite.

MapServer

One of the most important components is MapServer[1] which EOxServer uses through its Python bindings to handle certain OGC Web Service requests.

GDAL/OGR

In some cases EOxServer uses the GDAL/OGR[2] library for access to geospatial data directly (rather than through MapServer).

2.1.2 Software Architecture

The basic software architecture of EOxServer's Python code is layed out in *RFC 2: Extension Mechanism for EOxServer* (page 270). The main intention of the design is to keep EOxServer modular and extensible.

In order to reach that goal, EOxServer relies on a central registry of classes that implement certain behaviour. The registry allows to find appropriate implementations (e.g. for certain OGC Web Service operations) according to a set of parameters.

- Registry
- Factories
- Wrappers
- Resources
- Records

2.2 Core

2.3 Data Model

The core resources in EOxServer are coverages, more precisely GridCoverages. The EOxServer data model adopts and strongly relates to the data model from EO-WCS (OGC 10-140) as shown below in Figure: "*EO-WCS Data Model from OGC 10-140* (page 115)".

[1] http://www.mapserver.org
[2] http://www.gdal.org

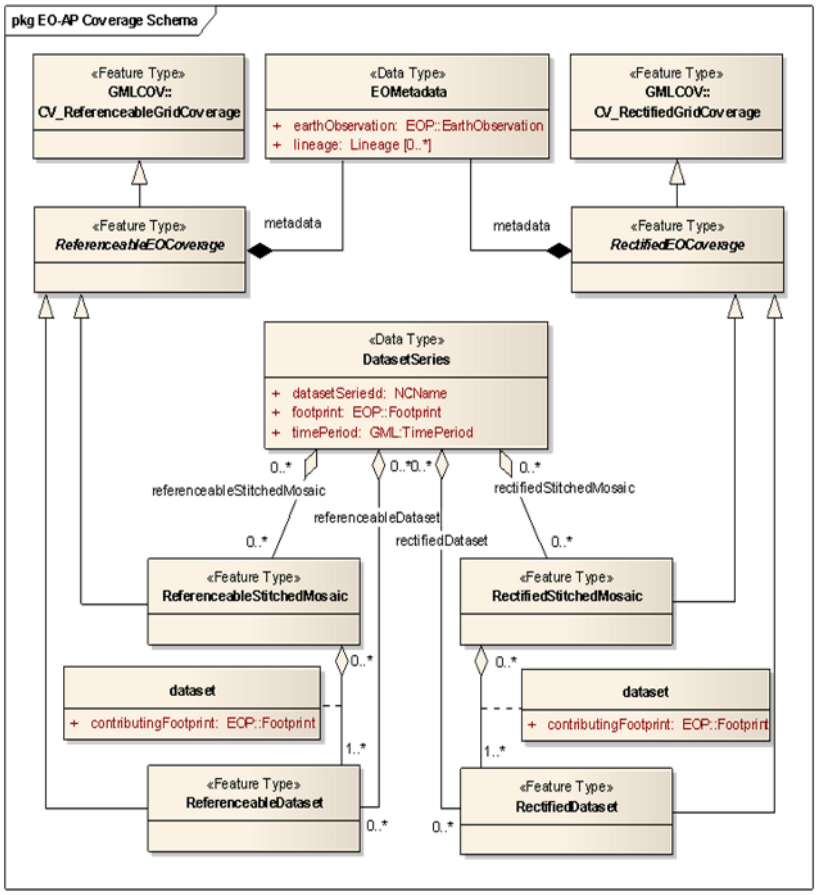


Figure 2.1: *EO-WCS Data Model from OGC 10-140*

2.3.1 Core

Figure: "*EOxServer Data Model for the Core* (page 116)" below shows the data model of EOxServer's core.

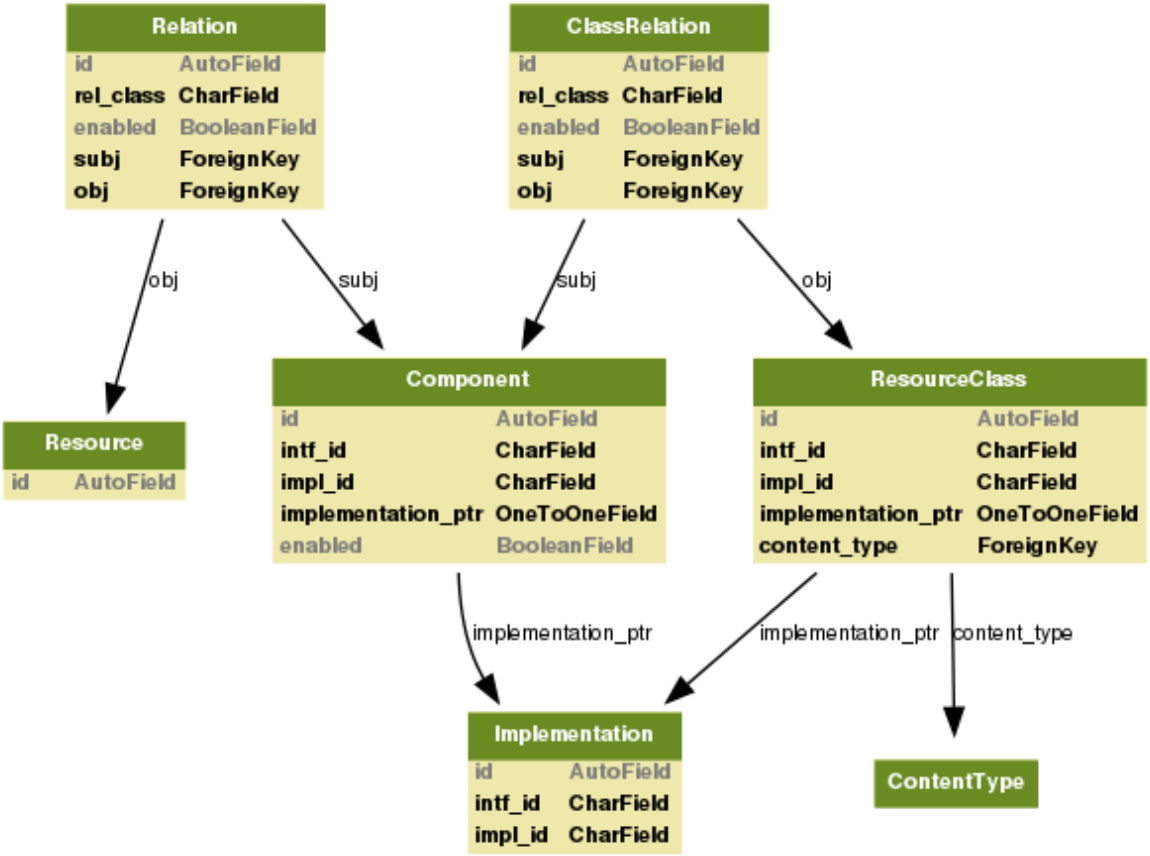


Figure 2.2: *EOxServer Data Model for the Core*

2.3.2 Data Integration Layer

Figure: "*EOxServer Data Model for Coverage Resources* (page 116)" below shows the data model of the coverage resources. Note the correlation with the EO-WCS data model as shown above.

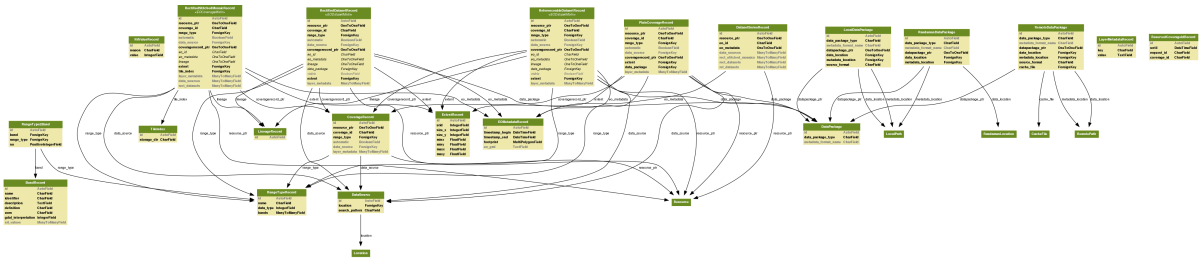


Figure 2.3: *EOxServer Data Model for Coverage Resources*

2.3.3 Data Access Layer

Figure: "*EOxServer Data Model for Back-ends* (page 117)" below shows the data model of the back-ends layer.

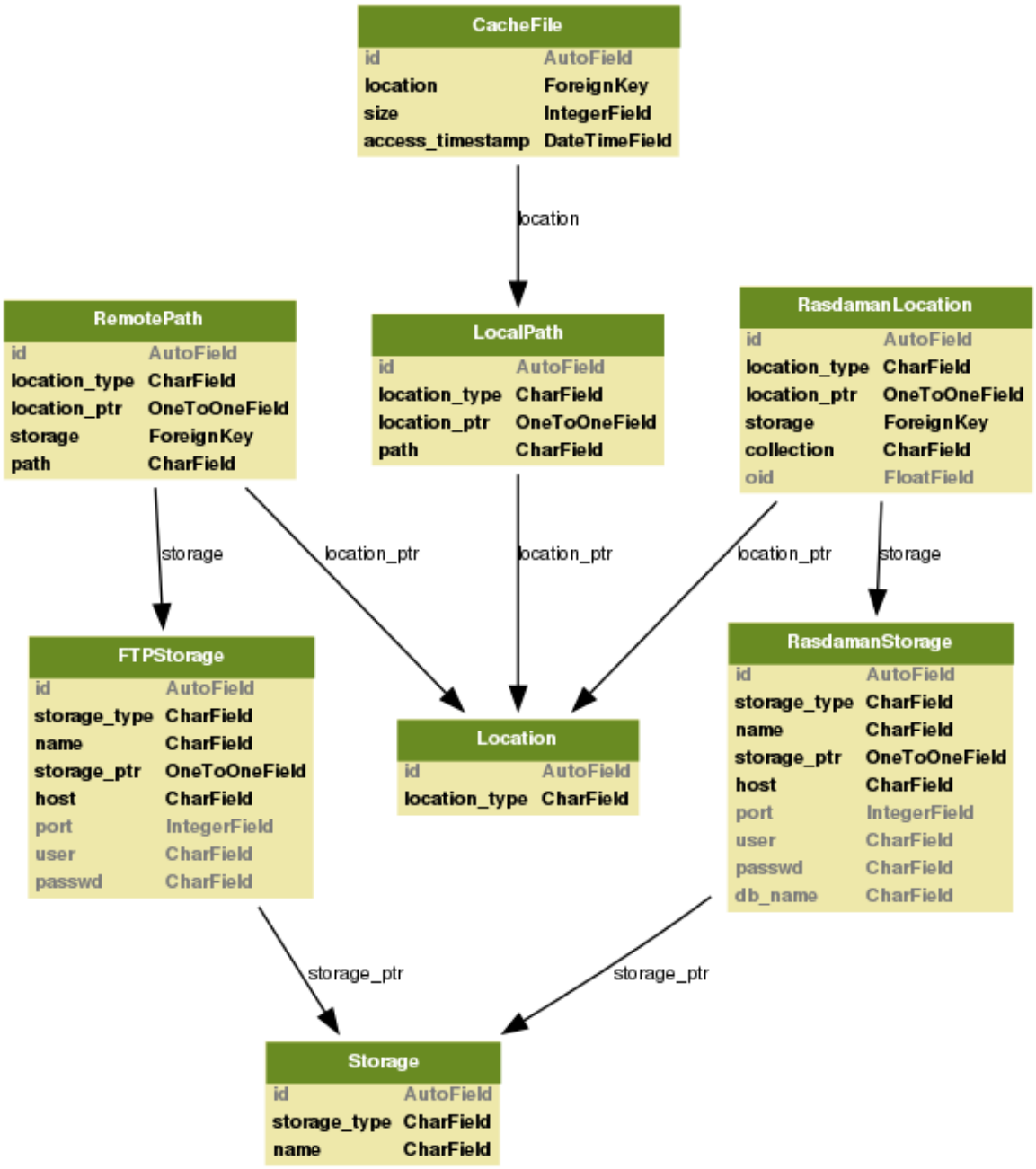


Figure 2.4: *EOxServer Data Model for Back-ends*

2.3.4 Task Tracker Data Model

Asynchrounous Task Processing (ATP) uses its own DB model displayed in Figure: "*EOxServer Data Model of ATP Task Tracker* (page 118)" to implement the task queueu, store the task inputs and outputs and track the tasks' status. (For more detail on ATP subsystem see "*Asynchronous Task Processing* (page 105)").

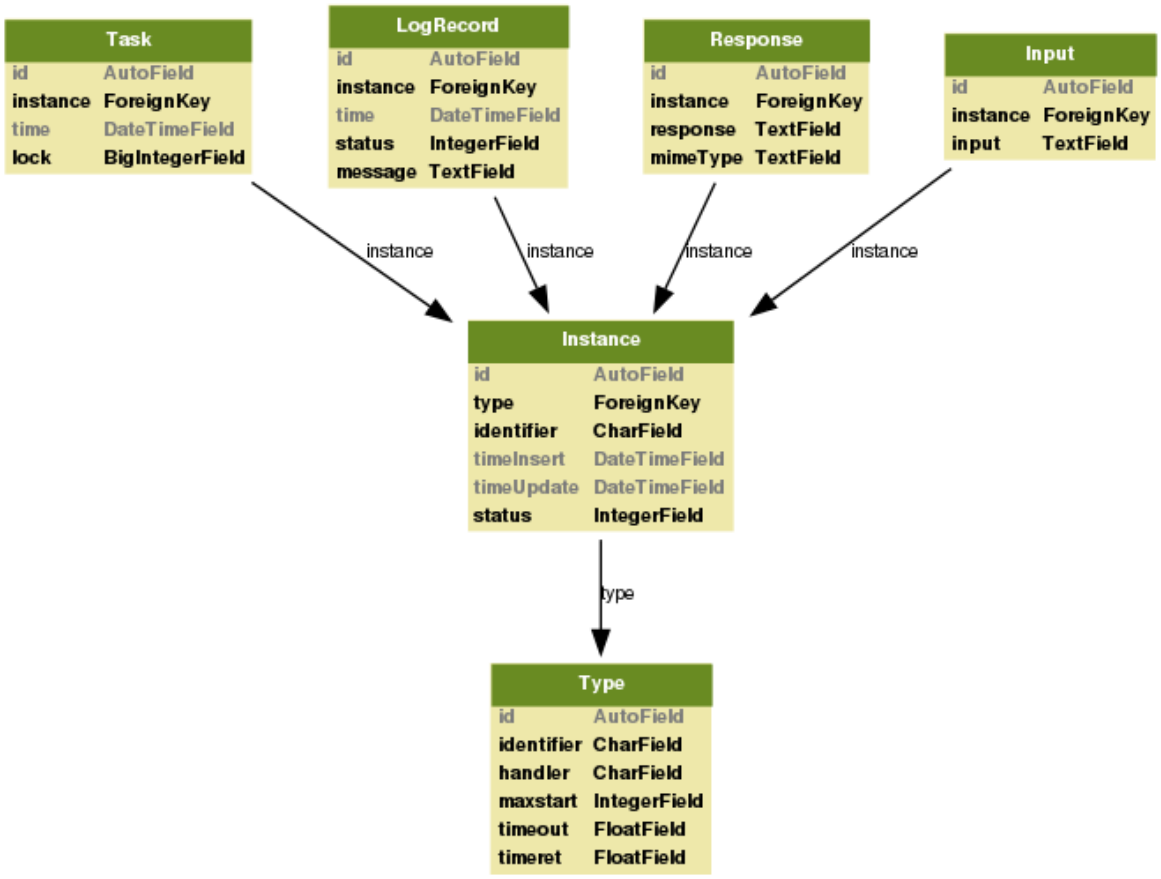


Figure 2.5: *EOxServer Data Model of ATP Task Tracker*

2.4 Plugins

2.5 Services

2.6 Data Formats

2.7 Metadata Formats

2.8 The *autotest* instance

Table of Contents

- The *autotest* instance (page 118)
  - Installation (page 119)
  - Running tests (page 120)
    - Running single tests (page 120)
    - XML Validation (page 120)
  - Running the *autotest* instance (page 120)
    - Loading test data (page 121)
    - Running the development web server (page 121)
  - Selenium (page 121)

The *autotest* package provides test data and expected results to test the services app of EOxServer. The package content need to be copied in a minimal EOxServer instance from where the tests can be run.

2.8.1 Installation

In order to run the tests, a new EOxServer instance has to be created with the `eoxserver-admin.py` script which creates a basic directory and file structure for a minimal EOxServer instance:

```
eoxserver-admin.py create_instance autotest
```

Note: Any valid instance name may be used instead of *autotest*. Just make sure to adjust the following commands.

Use the `--init_spatialite` to initialize a SQLite database needed for *running* (page 120) the *autotest* tests against SQLite:

```
eoxserver-admin.py create_instance autotest --init_spatialite
```

Now the EOxServer instance can be filled with its content, downloaded for example from the EOxServer project download page[3] and unpacked into the previously created instance:

```
wget http://eoxserver.org/export/head/downloads/EOxServer_autotest-<version>.tar.gz
tar xvfz EOxServer_autotest-<version>.tar.gz
cp -R EOxServer_autotest-<version>/* autotest
```

Note: The version needs to be adjusted to the version of EOxServer under test. If testing the current development branch or the latest of any stable branch please download the *autotest* data directly from the repository.

In order to successfully run all tests two configuration directives in `conf/eoxserver.conf` need to be adjusted:

```
[services.auth.base]
pdp_type=dummypdp

[services.ows.wcst11]
allowed_actions=Add,Delete
```

Currently there are two configuration directives in `conf/eoxserver.conf` in the `testing` directive which allow to skip certain test cases known to be problematic on some systems. Please refer to the *corresponding section in the configuration options documentation* (page 101).

Note: The reference platform used during the continuous integration builds is based on Ubuntu 12.04 64bit. If the testing is performed on a different platform please disable the binary comparison of rasters (see above).

---

[3] http://eoxserver.org/wiki/Download

Two configuration directives in `settings.py`, which is automatically generated by the `eoxserver-admin.py` script, are particular important for running the tests:

- `FIXTURE_DIRS` has to include the directory holding the autotest fixtures which is usually `data/fixtures`
- `TEST_RUNNER` should be set to `eoxserver.testing.core.EOxServerTestRunner` in order to be able to run test cases by regular expression search (see `eoxserver.testing.core` (page 242))

The autotest instance is now installed and ready for some testing!

2.8.2 Running tests

Most of the tests in EOxServer use the Django test framework[4], which itself is built upon Python's unittest framework[5].

To run tests against a component of EOxServer simply run:

```
cd autotest/autotest
python ../manage.py test <component>
```

where <*component*> is one of *services*, *core*, *backends*, *coverages* and *processes*. If all components shall be tested in one pass, just omit the <*component*> parameter. Detailed information about running Django tests can be found in the according chapter of the Django documentation[6].

Running single tests

Single tests or groups of tests can be run by appending the test name or beginning of the test name to the component:

```
python manage.py test services.WCS20GetCapabilities
```

XML Validation

In order to speed up the tests and also to pass certain tests it is highly recommended to make usage of locally stored schemas via XML Catalog:

```
wget http://eoxserver.org/export/head/downloads/EOxServer_schemas-<version>.tar.gz
tar xvfz EOxServer_schemas-<version>.tar.gz
export XML_CATALOG_FILES="`pwd`"/EOxServer-<version>/schemas/catalog.xml"
```

This allows the underlying libxml2 to vastly improve the performance of parsing schemas and the validation of XML trees against them. Also, several schemas contain small errors which makes it impossible to correctly use them in a real validation scenario. The self contained schemas package provides only useable schemas.

2.8.3 Running the *autotest* instance

First the configuration of the instance has to be finalized. After the successful *Database Setup* (page 22) it needs to be initialized:

```
cd autotest
python manage.py syncdb
```

Either a Django superuser needs to be defined while running the command or the `auth_data.json` loaded as described in the next section.

[4] https://docs.djangoproject.com/en/1.4/topics/testing/
[5] http://docs.python.org/library/unittest.html
[6] https://docs.djangoproject.com/en/1.4/topics/testing/#running-tests

Loading test data

Test data is provided as fixtures plus image files. To register all available test data simply run:

```
cd autotest
python manage.py loaddata auth_data.json initial_rangetypes.json \
                          testing_base.json testing_coverages.json \
                          testing_asar_base.json testing_asar.json \
                          testing_reprojected_coverages.json
```

The following fixtures are provided:

- initial_data.json - Base data to enable components. Loaded with syncdb.
- auth_data.json - An administration account.
- initial_rangetypes.json - Range types for RGB and gray-scale coverages.
- testing_base.json - Range type for the 15 band uint16 test data.
- testing_coverages.json - Metadata for the MERIS test data.
- testing_asar_base.json - Range type for the ASAR test data.
- testing_asar.json - Metadata for the ASAR test data.
- testing_reprojected_coverages.json - Metadata for the reprojected MERIS test data.
- testing_rasdaman_coverages.json - Use this fixtures in addition when rasdaman is installed and configured.
- testing_backends.json - This fixtures are used for testing the backend layer only and shouldn't be loaded in the test instance.

Running the development web server

Django provides a lightweight development web server[7] which can be used to run the *autotest* instance:

```
cd autotest
python manage.py runserver
```

The *autotest* instance is now available via a standard web browser at http://localhost:8000/

The *Admin Client* (page 51) is available at http://localhost:8000/admin or via the *Admin Client* link from the start page. Note that if the `auth_data.json` has been loaded there is a superuser login available with username and password "admin".

Sample service requests are described in the *Demonstration* (page 37) section.

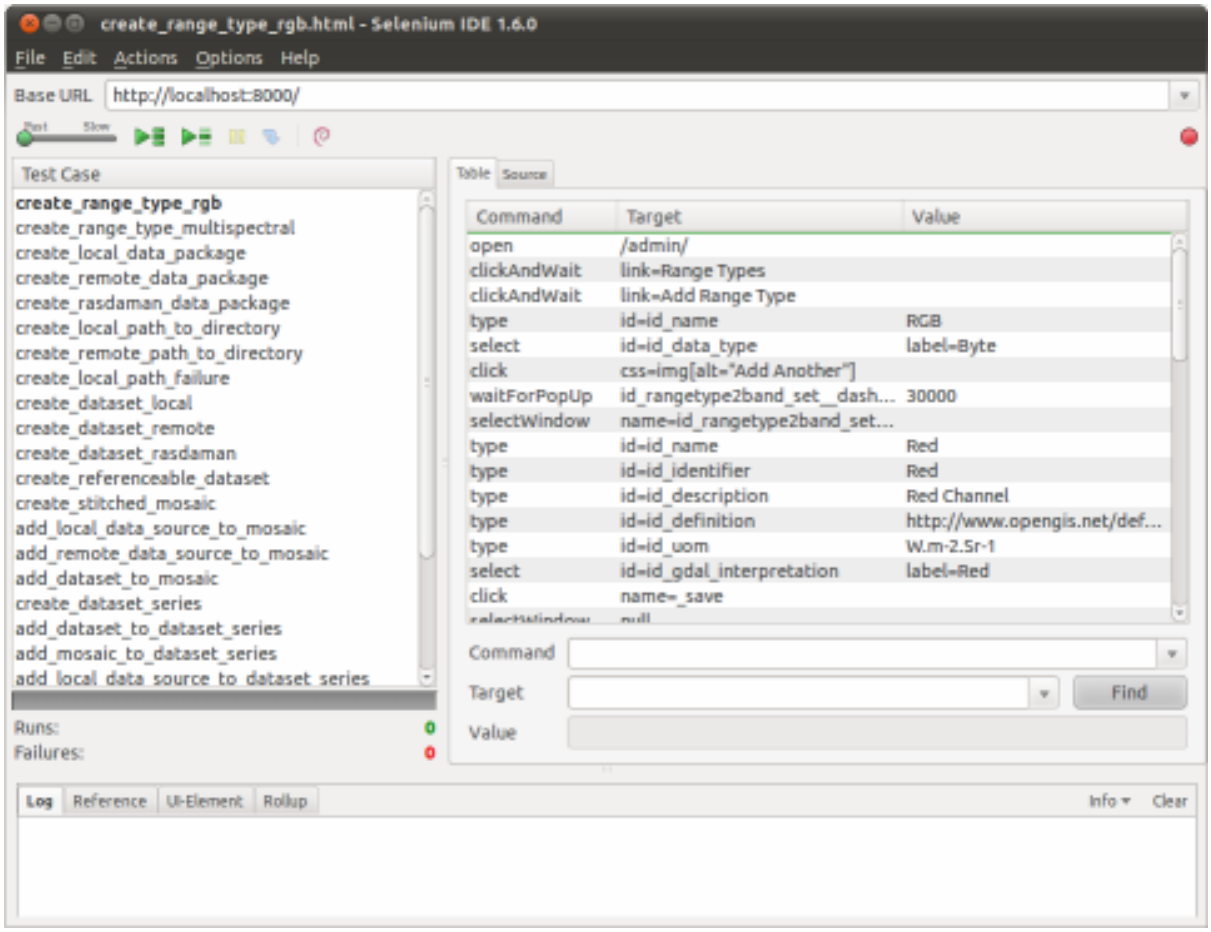
2.8.4 Selenium

The Selenium testing framework[8] is a powerful tool to create and run GUI test cases for any browser and HTML based application. It uses low-level mechanisms, such as simulating simple user input, to automate the browser and to test the application.

Currently the only browser supported is Firefox[9] using the Selenium IDE[10] plugin. It is basically a tool to record and play test cases and it also supports exporting the test scripts to several scripting languages as Java, Ruby, Python and *Selenese*, a basic HTML encoding.

Before the test cases can be run, ensure that the databases *backends* and *coverages* are empty and the EOxServer is run by either its developement server or within a webserver environment. To clear the databases in question type:

[7] https://docs.djangoproject.com/en/1.4/ref/django-admin/#runserver-port-or-address-port
[8] http://seleniumhq.org/
[9] http://www.mozilla.org/en-US/firefox/new/
[10] http://seleniumhq.org/projects/ide/




```
python manage.py reset coverages backends
```

and confirm the deletion. But be aware that this deletes all data previously entered in the database.

The *autotest* instance provides two test suites, one for the *Admin interface* (page 51) and one for the *Webclient interface* (page 62). To open a testsuite with Selenium IDE navigate to *File->Open Test Suite...* and open the suite of your choice.

To start the test run click on the *Play entire test suite* button. Alternatively, you can choose a single test case by double clicking it and then press the *Play current test case button*. Note: especially in the admin test suite several test cases have dependencies on other test cases to be run first, so many test cases will fail when its dependencies are not fullfilled. The best option is to play the entire test suite as a whole and view the results afterwards.

Note that the test speed should be decreased in order to allow enough time to fill the pages and thus pass the tests.

Don't forget to adjust the base URL when the *autotest* instance is not run locally.

Note that when testing the admin interface, before the tests can be rerun, the database has to be emptied, as explained in the example above.

2.9 SOAP Proxy

Table of Contents

- SOAP Proxy (page 122)
  - Architecture (page 123)
    - Supported Interfaces (page 123)
    - Overview (page 123)
  - Implementation (page 124)

2.9.1 Architecture

Soap_proxy is an adapter proxy which accepts POST request in XML ecoded in SOAP 1.2 messages, and passes these on to EOxServer. The proxy may also be configured to pass the messages as POST requests to a suitable mapserver executable instead of an EOxServer, for example for testing purposes.

Supported Interfaces

Soap_proxy uses SOAP 1.2 over HTTP.

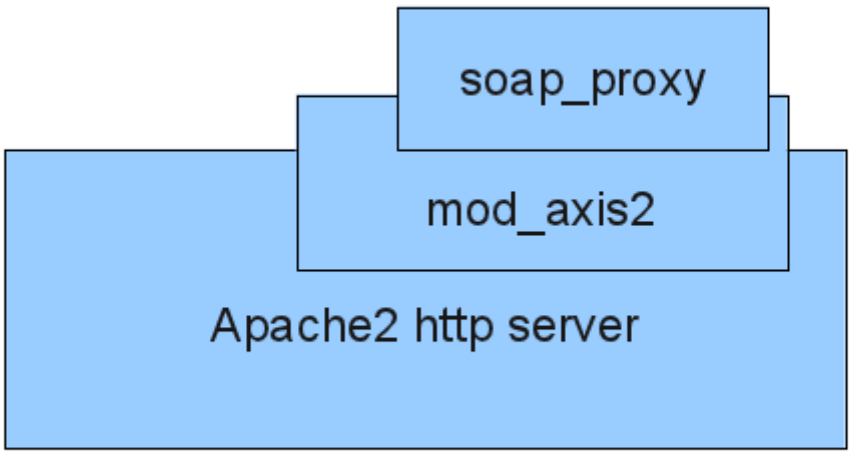
EOxServer responds to the following WCS-EO requests through SOAP service interface:

- DescribeCoverage
- DescribeEOCoverageSet
- GetCapabilities
- GetCoverage

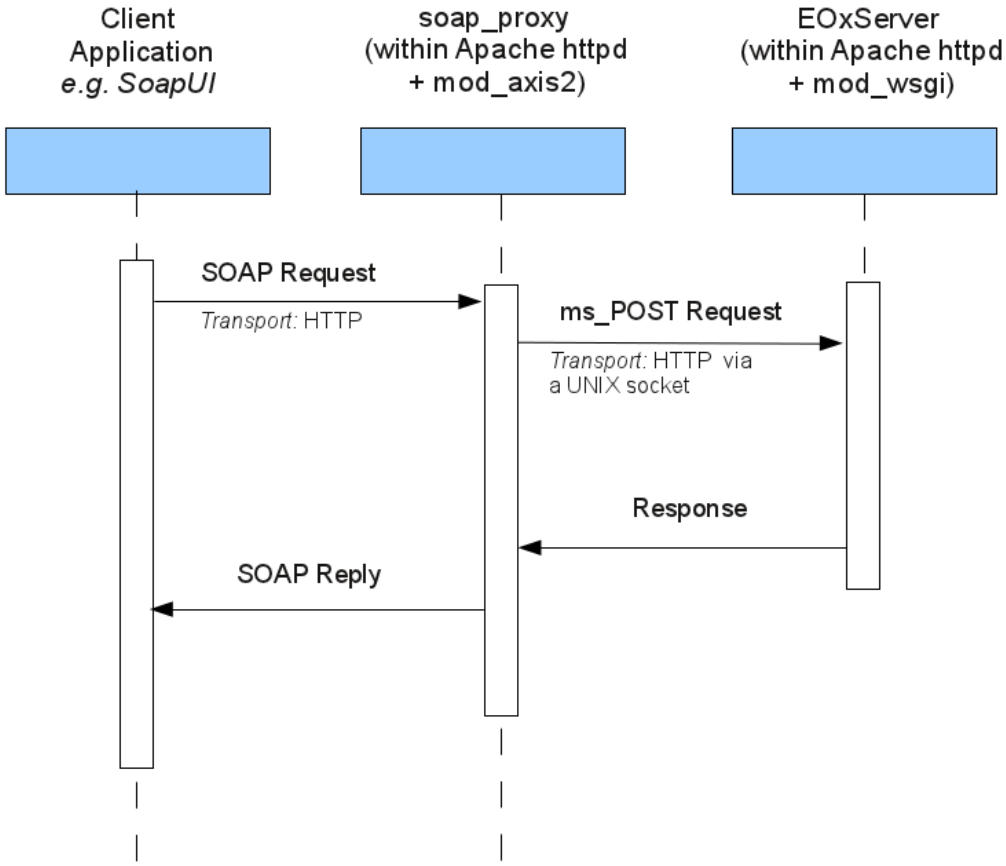
Overview

Soap_proxy uses the axis2/C framework. An important feature of axis2/C is that it correctly handles SOAP 1.2 MTOM Attachments.

The overall deployment context is shown in the figure below. Soap_proxy is implemented as an axis2/c service, running within the apache2 httpd server as a mod_axis2 module.



The next figure shows a sequence diagram for a typical request-response message exchange from a client through the soap_proxy to an instance of EOxServer.



2.9.2 Implementation

The implementation is provided in the src directory. The file sp_svc.c is the entry point where the Axis2/c framework calls the soap_proxy implentation code via rpSvc_invoke(), which calls rp_dispatch_op() to do most of the work.

2.10 Handling Coverages

2.10.1 Creating coverages

The best (and suggested) way to create a coverage is to use a coverage manager. For each type of coverage there is the according coverage manager. As usual in, EOxServer the correct coverage manager can be retrieved by the systems registry, using the interface ID of the manager and the type of the coverages to identify and find the correct manager:

```
mgr = System.getRegistry().findAndBind(
    intf_id="resources.coverages.interfaces.Manager",
    params={
        "resources.coverages.interfaces.res_type": "eo.rect_dataset"
    }
)
```

The managers `create` method can now be used to create a new record of the requested coverage. Since the possible arguments vary for each coverage type and use case, please refer to the actual `implementation documentation` of the manager for the complete list of possible parameters.

The following example creates a rectified dataset as simple as passing a local path to a data file and a meta-data-file and the name of the range type, which unfortunately cannot be identified otherwise at the time being.

```
mgr.create(
    "SomeCoverageID",
    local_path="path/to/data.tif",
    md_local_path="path/to/metadata.xml",
    range_type_name="RGB"
)
```

Coverage ID Uniqueness

The `CoverageIdManager` (page 205) helps during creation of new, and querying existing Coverage IDs:

```
from eoxserver.resources.coverages.covmgrs import CoverageIDManager
idmgr = CoverageIDManager()
```

The Coverage ID must be unique for all types of coverages, such as, *Rectified* or *Referenceable* data-sets. This aspect is especially important for graceful handling of Coverage IDs' conflicts in case of concurrent inserts of new coverages. Once a new Coverage ID is approved by the EOxServer in course of the processing of an insert request, any other insert request must not be allowed to use the same Coverage ID. Therefore the `CoverageIdManager` (page 205) allows Coverage ID *reservation* to grant the Coverage ID exclusivity during of the actual coverage insert. The reservation is performed by the `reserve()` method:

```
from datetime import datetime, timedelta
idmgr.reserve("SomeCoverageID", until=datetime.now() + timedelta(days=1))
```

If the Coverage ID cannot be reserved (most likely, because it is used by an existing coverage, or reserved by another insert request) an exception is raised, as described in the method's documentation.

The reservation is released automatically after expiration of the given time-out (the optional `until` parameter). The default time-out value can be configured via EOxServer configuration file (section `resources.coverages.coverage_id`, field `reservation_time`, default value `0:0:30:0`, i.e., 30 min.).

The reservation can be revoked by the `release()` method:

```
idmgr.release("SomeCoverageID")
```

Although it is not necessary to release a booked Coverage ID, we encourage to do so when possible.

Whether a Coverage ID is neither in use nor reserved can be checked by the `available()` method:

```
if idmgr.available(someID):
    # there is neither coverage nor cov.ID reservation
    ...
```

2.10.2 Finding Coverages

There are several techniques to search for coverages in the system, depending on what information is desired and/or provided. In a case, when the Coverage ID is known, it is possible to use `check()` method of `CoverageIdManager` (page 205) to check whether this ID is used by an existing coverage:

```
if idmgr.check(someID):
    # there is an coverage with this ID
```

Once we know there is an existing coverage we can query type of the coverage by the `getCoverageType()` method in order to select the proper handling of the coverage type:

```
ctype = idmgr.getCoverageType(someID):

if   ctype == "PlainCoverage" :
    ...
elif ctype == "RectifiedDataset" :
    ...
elif ctype == "ReferenceableDataset" :
    ...
elif ctype == "RectifiedStitchedMosaic" :
    ...
else :
    # invalid coverage ID
    ...
```

Alternatively, a factory can used to get the correct wrapper of a coverage, namely the EOCoverageFactory (page 228). The simplest case is to find a coverage according to its Coverage ID:

```
from eoxserver.core.system import System

coverage_wrapper = System.getRegistry().getFromFactory(
    "resources.coverages.wrappers.EOCoverageFactory",
    {"obj_id": coverage_id}
)
```

This command returns the proper coverage wrapper according to the coverages type, or None, if no such coverage exists.

For more sophisticated searches, filter expressions have to be used. In case of coverage filters, the CoverageExpressionFactory (page 196) creates the required expressions. In the following example, we create a filter expression to get all coverages whose footprint intersects with the area defined by the BoundedArea (page 194):

```
from eoxserver.resources.coverages.filters import BoundedArea

filter_exprs = []
filter_exprs.append(System.getRegistry().getFromFactory(
    "resources.coverages.filters.CoverageExpressionFactory",
    {
        "op_name": "footprint_intersects_area",
        "operands": (BoundedArea(srid, minx, miny, maxx, maxy),)
    }
))
```

With our filter expressions, we are now able to get the list of coverages complying to our filters with the `find` method of the EOCoverageFactory (page 228) which returns a list of all objects intersecting with our region.:

```
factory = System.getRegistry().bind(
    "resources.coverages.wrappers.EOCoverageFactory"
)
coverages = factory.find(filter_exprs=filter_exprs)
```

2.10.3 Updating Coverages

Updating a coverage is either done by the wrappers or, on a more higher level, with the coverage manager.

Updating with the wrappers is limited to several methods on the specific wrapper itself (e.g.: the addCoverage() (page 224) method of the RectifiedStitchedMosaicWrapper (page 224)) or the setAttrValue() (page 156) method. The latter one is directly coupled to the wrappers `FIELDS` lookup dictionary which expands to field lookup on the according model.

The following example demonstrates either use:

```
rect_stitched_mosaic_wrapper = System.getRegistry().getFromFactory(
    "resources.coverages.wrappers.EOCoverageFactory",
    {"obj_id": mosaic_coverage_id}
)

rect_stitched_mosaic_wrapper.addCoverage(
    System.getRegistry().getFromFactory(
        "resources.coverages.wrappers.EOCoverageFactory",
        {"obj_id": coverage_id}
    )
)

rect_stitched_mosaic_wrapper.setAttrValue("size_x", 1000)
rect_stitched_mosaic_wrapper.setAttrValue("size_y", 1000)
```

To know what attributes are allowed in the *setAttrValue*, either look up the class variable `FIELDS` or call the `getAttrNames()` method of the wrapper .

Another way to update existing coverages is to use the correct coverage manager. Its `update()` method can be supplied three (optional) dictionary arguments:

- `link`: adds a reference to another object in the database. This is used, e.g., for `container_ids`, `coverages` or `data_sources`.
- `unlink`: removes a reference to another object. It has the same arguments as the `link` dictionary
- `set`: Sets an integral value or a collection of values in the database object. Here are also keys from the `FIELDS` accepted.

The usable arguments depend on the actually used coverage manager type, but are almost the same as the arguments available for the `create` method.

The following example demonstrates the use of the coverage managers `update` method with a rectified stitched mosaic:

```
mgr = System.getRegistry().findAndBind(
    intf_id="resources.coverages.interfaces.Manager",
    params={
        "resources.coverages.interfaces.res_type": "eo.rect_stitched_mosaic"
    }
)

mgr.update(
    obj_id=mosaic_coverage_id,
    link={
        "coverage_ids": ["RectifiedDatasetCoverageID"]
    },
    unlink={
        "container_ids": ["DatasetSeriesEOID"]
    }
    set={
        "size_x": 1000,
        "size_y": 1000,
        "eo_metadata": EOMetadata(
            "NewEOID",
            timestamp_begin,
            timestamp_end,
            GEOSGeometry(some_footprint)
        )
    }
)
```

2.11 Asynchronous Task Processing - Developers Guide

Table of Contents

- Asynchronous Task Processing - Developers Guide (page 128)
  - Introduction (page 128)
  - Simple ATP Application (page 128)
    - Step 1 - Handler Subroutine (page 129)
    - Step 2 - New Task Type Registration (page 129)
    - Step 3 - Creating New Task (page 129)
    - Step 4 - Polling the task status (page 129)
    - Step 5 - Getting the logged task history (page 130)
    - Step 6 - Getting the task results (page 130)
    - Step 7 - Removing the task (page 130)
  - Executing ATP Task (page 130)
    - Pulling a task from queue (page 130)
    - Task Execution (page 131)
    - DB Cleanup (page 131)

2.11.1 Introduction

This guide is intended to help with the creation of applications using the *Asynchronous Task Processing* subsystem of EOxServer.

The first part is guiding creation of the simple task producer, i.e., an application needing the asynchronous processing capabilities.

The second part helps with creation of a task consumer, i.e., the part of code pulling tasks from the work queue and executing them. The task consumer is part of Asynchronous Task Processing Daemon.

An overview of the ATP capabilities is presented in "*Asynchronous Task Processing* (page 105)". The database model used in by the ATP subsystem is described in "*Task Tracker Data Model* (page 118)". The complete API reference can be found in "*Module eoxserver.resources.processes.tracker* (page 185)".

2.11.2 Simple ATP Application

Here in this section we will prepare step-by-step a simple demo application making use of the ATP subsystem. The complete application is available at location:

```
<EOxServer instal.dir.>/tools/atp_demo.py
```

The prerequisite of starting the application is that the correct path to the *EOxServer* installation and instance is set together with the correct *Django* `settings` module.

Initially the application must import the right python objects from the `tracker()` (page 185) module:

```
from eoxserver.resources.processes.tracker import \
    registerTaskType, enqueueTask, QueueFull, \
    getTaskStatusByIdentifier, getTaskResponse, deleteTaskByIdentifier
```

By this command we imported following objects: i) task type registration function, ii) the task creation (enqueue) subroutine, iii) an exception class risen in case of full task queue unable to accept (most likely temporarily) new tasks, iv) task's status polling subroutine, v) the response getter function and finally vi) the subroutine deleting an existing task. These are the ATP Python objects needed by our little demo application.

Step 1 - Handler Subroutine

Let's start with preparation of an example of subroutine to be executed - handler subroutine. The example handler below sums sequence of numbers and stores the result:

```
def handler( taskStatus , input ) :
    """ example ATP handler subroutine """
    sum = 0
    # sum the values
    for val in input :
        try :
            sum += float( val )
        except ValueError:
            # stop in case on ivalid input
            taskStatus.setFailure("Input must be a sequence of numbers!")
            return
    # store the response and terminate
    taskStatus.storeResponse( str(sum) )
```

Any handler subroutine (see also `dummyHandler()` (page 186)) receives two parameters: i) an instance of the `TaskStatus` (page 186) class and an input parameter. The input parameter is set during the task creation and can be any Python object serialisable by the `pickle` module.

Step 2 - New Task Type Registration

Once we have prepared the handler subroutine we can register the task type to be performed by this subroutine:

```
registerTaskType( "SequenceSum" , "tools.atp_demo.handler" , 60 , 600 , 3 )
```

The `registerTaskType()` (page 185) subroutine registers a new task type named "SequenceSum". Any task instance of this task type will be processed by the `handler` subroutine. The handler subroutine is specified as importable module path. Any task instance not processed by an ATPD within 60 seconds (measured from the moment the ATPD pulls a task from the queue) is considered to be abandoned and it is automatically re-enqueued for new processing. The number of the re-enqueue attempts is limited to 3. Once a task instance is finished it will be stored for min. 10 minutes (600 seconds) before it gets removed.

Step 3 - Creating New Task

Once the task handler has been registered as a new task type we can create a task's instance:

```
while True :
    try:
        enqueueTask( "SequenceSum" , "Task001" , (1,2,3,4,5) )
        break
    except QueueFull : # retry if queue full
        print "QueueFull!"
        time.sleep( 5 )
```

The `enqueueTask()` (page 186) creates a new task instance "Task001" of task type "SequenceSum". The tuple `(1,2,3,4,5)` is the input to the handler subroutine. In case of full task queue new task cannot be accepted and the `QueueFull()` (page 189) is risen. Since we want the task to be enqueued a simple re-try loop must be employed.

Step 4 - Polling the task status

After task has been created enqueued for processing its status can be polled:

```
while True :
    status = getTaskStatusByIdentifier( "SequenceSum" , "Task001" )
    print time.asctime() , "Status: " , status[1]
```

```
    if status[1] in ( "FINISHED" , "FAILED" ) : break
    time.sleep( 5 )
```

The task status is polled until the final status (`FINISHED` or `FAILED`) is reached. The task must be identified by unique pair of task type and task instance identifiers.

NOTE: The task instance is guaranteed to be unique for given task type identifier, i.e., there might be two task with the same instance identifier but different type identifier.

Step 5 - Getting the logged task history

The history of the task processing is logged and the log messages can be extracted by `getTaskLog()` (page 188) function:

```
print "Processing history:"
for rec in getTaskLog( "SequenceSum" , "Task001" ) :
    print "-" , rec[0] , "Status: " , rec[1][1] , "\t" , rec[2]
```

This function returns list of log records sorted by time (older first).

Step 6 - Getting the task results

Once the task has been finished the task response can be retrieved:

```
if status[1] == "FINISHED" :
    print "Result: " , getTaskResponse( "SequenceSum" , "Task001" )
```

Step 7 - Removing the task

Finally, the result task is not needed any more and can be removed from DB:

```
deleteTaskByIdentifier( "SequenceSum" , "Task001" )
```

2.11.3 Executing ATP Task

In this section we will briefly describe all the steps necessary to pull and execute task instance from the queue. As working example we encourage you the source Python code of the ATPD located at:

```
<EOxServer instal.dir.>/tools/asyncProcServer.py
```

The invocation of the ATP server is described in "*Asynchronous Task Processing* (page 105)".

Initially the application must import the python objects from the `tracker` (page 185) module:

```
from eoxserver.resources.processes.tracker import *
```

For convenience we have made available whole content of the module.

Pulling a task from queue

The ATPD is expected to pull task from the queue repeatedly. For simplicity we avoid the loop definition and we will rather focus on the loop body. Following command pulls a list of tasks from queue:

```
try:
    # get a pending task from the queue
    taskIds = dequeueTask( SERVER_ID )
except QueueEmpty : # no task to be processed
    # wait some ammount of time
```

```
    time.sleep( QUEUE_EMPTY_QUERY_DELAY )
    continue
```

This command tries to pull exactly one task at time from the DB queue but the applied mechanism of pulling does not guaranties that none or more than one task would be return. Thus the dequeuing function returns a list of tasks and the implementation must take this fact into account. Further, the dequeue function requires unique ATPD identifier (`SERVER_ID`).

The `dequeueTask()` (page 187) function changes automatically the status from `ENQUEUED` to `SCHEDULED` and log the state transition. The optional logging message can be provided.

Task Execution

In case we have picked one of the pulled tasks and stored it to `taskId` variable we can proceed with the task execution:

```
# create instance of TaskStatus class
pStatus = TaskStatus( taskId )
try:
    # get task parameters and change status to STARTED
    requestType , requestID , requestHandler , inputs = startTask( taskId )
    # load the handler
    module , _ , funct = requestHandler.rpartition(".")
    handler = getattr( __import__(module,fromlist=[funct]) , funct )
    # execute handler
    handler( pStatus , inputs )
    # if no terminating status has been set do it right now
    stopTaskSuccessIfNotFinished( taskId )
except Exception as e :
    pStatus.setFailure( unicode(e) )
```

In order to execute the task couple of actions must be performed. First an instance of the `TaskStatus` (page 186) class must be created.

The parameters of the task (task type identifier, task instance identifier, request handler and task inputs) must be retrieved by the `dequeueTask()` (page 187) function. The function also changes the status of the task from `SCHEDULED` to `RUNNING` and logs the state transition automatically.

The handler "dot-path" must be split to module and function name and loaded dynamically by the `__import__()` function.

Once imported the handler function is executed passing the TaskStatus and inputs as the arguments.

The handler function is allowed but not required to set the successful terminal state of the processing (`FINISHED`) and if not set it is done by the `stopTaskSuccessIfNotFinished()` (page 187) function.

Obviously, the implementation must catch any possible Python exception and record the failure (`try-except` block).

DB Cleanup

In addition to the normal operation each ATPD implementation is responsible for maintenance of the ATPD subsystem in a consistent state. Namely, i) the ATPD must repeatedly check for the abandoned "zombie" tasks and restart them by calling `reenqueueZombieTasks()` (page 188) function and ii) the ATPD must remove DB records of the finished "retired" tasks by calling `deleteRetiredTasks()` (page 188) function.

2.12 Modules

Table of Contents

- Modules (page 131)
  - EOxServer Core (page 132)
  - Utils (page 157)
  - Service Layer (page 168)
  - Processing Layer (page 185)
  - Data Integration Layer (page 189)
  - Data Access Layer (page 234)
  - Testing (page 242)

2.12.1 EOxServer Core

Module eoxserver.core.config

This module provides an implementation of a system configuration that relies on different configuration files. It is used by `eoxserver.core.system` (page 157) to store the current system configuration.

class `eoxserver.core.config.`**Config**

The `Config` (page 132) class represents a system configuration. Internally, it relies on two configuration files:

- the default configuration file (`eoxserver/conf/default.conf`)
- the instance configuration file (`conf/eoxserver.conf` in the instance directory)

Configuration values are read from these files.

getConcurringConfigValues (*section*, *key*)

Returns a dictionary od concurring configuration parameter values. It may have two entries

- `default`: the default configuration parameter value
- `instance`: the instance configuration value

If there is no configuration parameter value defined in the respective configuration file, the entry is omitted.

The `section` and `key` arguments denote the parameter to be looked up.

getConfigValue (*section*, *key*)

Returns a configuration parameter value. The `section` and `key` arguments denote the parameter to be looked up. The value is searched for first in the instance configuration file; if it is not found there the value is read from the default configuration file.

getDefaultConfigValue (*section*, *key*)

Returns a configuration parameter default value (read from the default configuration file). The `section` and `key` arguments denote the parameter to be looked up.

getEOxSPath ()

Returns the path to the EOxServer installation (not to the instance).

getInstanceConfigValue (*section*, *key*)

Returns a configuration parameter value as defined in the instance configuration file, or `None` if it is not found there. The `section` and `key` arguments denote the parameter to be looked up.

class `eoxserver.core.config.`**ConfigFile** (*config_filename*)

This is a wrapper for a configuration file. It is based on the Python builtin `ConfigParser`[11] module.

get (*section*, *key*)

Return the configuration parameter value, or `None` if it is not defined.

[11] http://docs.python.org/2.7/library/configparser.html#ConfigParser

The `section` argument denotes the section of the configuration file where to look for the parameter named `key`. See the `ConfigParser`[12] module documentation for details on the config file syntax.

Module eoxserver.core.exceptions

This module contains exception classes used throughout EOxServer.

exception `eoxserver.core.exceptions.`**`BindingMethodError`**(*msg*)
This exception shall be raised by the registry if it cannot bind to implementations of a given interface because the binding method does not allow it.

exception `eoxserver.core.exceptions.`**`ConfigError`**(*msg*)
This exception shall be raised if the system configuration is invalid.

exception `eoxserver.core.exceptions.`**`DecoderException`**(*msg*)
This is the base class for exceptions raised by decoders as defined in `eoxserver.core.util.decoders` (page 158).

exception `eoxserver.core.exceptions.`**`EOxSException`**(*msg*)
Base class for EOxServer exceptions. Expects the error message as its single constructor argument.

exception `eoxserver.core.exceptions.`**`FactoryQueryAmbiguous`**(*msg*)
This exception shall be raised when ... TODO

exception `eoxserver.core.exceptions.`**`IDInUse`**(*msg*)
This exception shall be raised if a requested unique ID is already in use.

exception `eoxserver.core.exceptions.`**`ImplementationAmbiguous`**(*msg*)
This exception shall be raised by the registry if the input data matches more than one implementation.

exception `eoxserver.core.exceptions.`**`ImplementationDisabled`**(*msg*)
This exception shall be raised by the registry if the requested implementation is disabled.

exception `eoxserver.core.exceptions.`**`ImplementationNotFound`**(*msg*)
This exception shall be raised by the registry if an implementation ID is not found.

exception `eoxserver.core.exceptions.`**`InternalError`**(*msg*)
`InternalError` (page 133) shall be raised by EOxServer modules whenever they detect a fault that stems from errors in the EOxServer implementation. It shall NOT be used for error conditions that are caused by incorrect or invalid user or service input or that originate from the individual system configuration.

In a web service environment, an `InternalError` (page 133) should lead to the server responding with a HTTP Status of `500 INTERNAL SERVER ERROR`.

exception `eoxserver.core.exceptions.`**`InvalidExpressionError`**(*msg*)
This exception shall be raised if a filter expression statement is invalid, e.g. because of incorrect operands.

exception `eoxserver.core.exceptions.`**`InvalidParameterException`**(*msg*)
This exception shall be raised if a parameter is found to be invalid.

exception `eoxserver.core.exceptions.`**`IpcException`**(*msg*)
This exception shall be raised in case of communication faults in the IPC system.

exception `eoxserver.core.exceptions.`**`KVPDecoderException`**(*msg*)
This is the base class for exceptions raised by the KVP decoder.

exception `eoxserver.core.exceptions.`**`KVPKeyNotFound`**(*msg*)
This exception shall be raised if the KVP decoder does not encounter a given key. It inherits from `KVPDecoderException` (page 133) and `MissingParameterException` (page 134).

exception `eoxserver.core.exceptions.`**`KVPKeyOccurrenceError`**(*msg*)
This exception shall be raised if the number of occurrences of a given KVP key does not lay within the occurrence range defined by the applicable decoding schema. It inherits from `KVPDecoderException` (page 133) and `InvalidParameterException` (page 133).

[12] http://docs.python.org/2.7/library/configparser.html#ConfigParser

exception eoxserver.core.exceptions.**KVPTypeError**(*msg*)
This exception shall be raised if the requested KVP value is of another type than defined in the decoding schema. It inherits from KVPDecoderException (page 133) and InvalidParameterException (page 133).

exception eoxserver.core.exceptions.**MissingParameterException**(*msg*)
This exception shall be raised if an expected parameter is not found.

exception eoxserver.core.exceptions.**TypeMismatch**(*msg*)
This exception shall be raised by interfaces in case they detect that an implementation method has been called within an argument of the wrong type.

exception eoxserver.core.exceptions.**UniquenessViolation**(*msg*)
This excetion shall be raised if a database record cannot be created due to uniqueness constraints.

exception eoxserver.core.exceptions.**UnknownAttribute**(*msg*)
This exception shall be raised if an unknown or invalid attribute is requested from a resource.

exception eoxserver.core.exceptions.**UnknownParameterFormatException**(*msg*)
This exception shall be raised if a parameter is not in the format expected by the implementation.

exception eoxserver.core.exceptions.**XMLDecoderException**(*msg*)
This is the base class for exceptions raised by the XML decoder.

exception eoxserver.core.exceptions.**XMLEncoderException**(*msg*)
This exception shall be raised if the XML encoder finds an error in an encoding schema.

exception eoxserver.core.exceptions.**XMLNodeNotFound**(*msg*)
This exception shall be raised if the XML decoder does not encounter a given XML node. It inherits from XMLDecoderException (page 134) and MissingParameterException (page 134).

exception eoxserver.core.exceptions.**XMLNodeOccurrenceError**(*msg*)
This exception shall be raised if the number of occurrences of a given XML node does not lay within the occurrence range defined by the applicable decoding schema. It inherits from XMLDecoderException (page 134) and InvalidParameterException (page 133).

exception eoxserver.core.exceptions.**XMLTypeError**(*msg*)
This exception shall be raised if the requested XML node value is of another type than defined in the decoding schema. It inherits from XMLDecoderException (page 134) and InvalidParameterException (page 133).

Module eoxserver.core.filters

This module defines interfaces for filter expressions and filters. These can be used to refine searches for resources.

class eoxserver.core.filters.**FilterExpressionInterface**
Filter expressions can be used to constrain searches for resources using a factory. They provide a uniform way to define these constraints without respect to the concrete resource implementation.

Internally, filter expressions are translated to filters (i.e. implementations of FilterInterface (page 135)) that can be applied to a resource.

The binding method for filter expressions is factory, i.e. implementations are accessible through a factory that implements FactoryInterface (page 152). Developers have to write their own factory implementations for each category of expressions.

getOpName()
This method shall return the operator name. The name can depend on the instance data but does not have to. Depending on the factory implementation, the name may or may not vary with the instance data.

getOpSymbol()
This method shall return the operator symbol if applicable or None otherwise. Depending on the factory implementation, the symbol may or may not vary with the instance data

getNumOperands()
This method shall return the number of operands required by the operator. Depending on the factory implementation the number may or may not vary with the instance data.

getOperands()
This method shall return a tuple of operands the instance was initialized with.

initialize(***kwargs*)
This method shall initialize the expression; it takes keyword arguments; each implementation has to define the arguments it accepts individually.

Interface ID core.filters.FilterExpression

class eoxserver.core.filters.**FilterInterface**
Filter expressions are translated to filters that can be applied to a given QuerySet. This is the interface for this operation.

Binding to implementations of this interface is possible using key-value-pair matching.

Interface ID core.filters.Filter

Kvp keys

- core.filters.res_class_id: the implementation ID of the resource class
- core.filters.expr_class_id: the implementation ID of the filter expression class

applyToQuerySet(*expr*, *qs*)
This method shall apply a given filter expression expr to a given Django QuerySet qs and return the resulting QuerySet.

resourceMatches(*expr*, *res*)
This method shall return True if the resource wrapped by res matches the filter expression expr, False otherwise.

class eoxserver.core.filters.**SimpleExpression**
An implementation of FilterExpressionInterface (page 134) intended to serve as a base class for simple expressions.

NUM_OPS = 1
The expected number of operands; has to be overridden by concrete implementations

OP_NAME = ''
The operator name of the simple expression; has to be overridden by concrete implementations

OP_SYMBOL = None
The operator symbol of the simple expression; None by default; has to be overridden by concrete implementations

getNumOperands()
Returns the expected number of operands.

getOpName()
Returns the operator name.

getOpSymbol()
Returns the operator symbol if applicable, None by default.

getOperands()
Returns the operands of the simple expression instance.

initialize(***kwargs*)
Initialize the simple expression instance. This method accepts one optional keyword argument, namely operands which is expected to be a tuple of operands.

Raises InternalError (page 133) in case the number of operands does not match.

Note: Further validation steps may be added by concrete implementations.

class eoxserver.core.filters.**SimpleExpressionFactory**

This is the base class for a simple expression factory.

find(***kwargs*)

Returns a list of filter expressions. The method accepts a single, optional keyword argument `op_list` which is expected to be a list of dictionaries of the form:

```
{
    "op_name": <operator_name>,
    "operands": <operand_tuple>
}
```

The dictionaries will be passed as keyword arguments to `get()` (page 136)

get(***kwargs*)

Returns a filter expression. The method accepts two keyword arguments:

- `op_name` (mandatory): the operator name for the expression
- `operands` (optional): a tuple of operands for the expression; the number and type of expected operands is defined by each filter expression class individually

The method raises `InternalError` (page 133) if the `op_name` parameter is missing or unknown.

Module eoxserver.core.interfaces

This module contains the core logic for interface declaration and validation.

Introduction

Interfaces play a key role in the extension mechanism of EOxServer which is described in *RFC 1: An Extensible Software Architecture for EOxServer* (page 248) and *RFC 2: Extension Mechanism for EOxServer* (page 270). Extensibility is one of the main features of the EOxServer architecture. Based on its generic core, the different EOxServer layers shall be able to dynamically integrate additional behaviour by defining interfaces that can be implemented by different modules and plugins.

The module `eoxserver.core.registry` (page 145) implements the actual extension mechanism based on the capabilities of this module.

`eoxserver.core.interfaces` (page 136) is completely independent from the EOxServer extension mechanism on the other hand. Actually, due its generic nature, it is completely independent from the EOxServer project itself and can be used in any other situation where interface declaration and validation might be of interest.

How does it work? An interface is an ordinary Python class deriving from the `Interface` (page 139) class or one of its descendants. Interfaces contain method declarations. As there is no way to declare method signatures without implementing the method in Python an alternative solution has been chosen: declarations are made using class variables that contain instances of the `Method` (page 139) class provided by this module.

Interfaces inherit declarations from their parents. They even support multiple inheritance. This allows to extend and combine existing interfaces in a straightforward way comparable to the way interfaces are declared in Java for instance.

The `Method` (page 139) constructor accepts an arbitrary number of input argument declarations as well as an optional output declaration. Similar to method declarations, argument declarations are made using instances of special classes derived from the `Arg` (page 140) base class.

Implementations can be derived from new-style classes which we will call *implementing class* in this document. Each interface has an `implement()` (page 139) method that accepts an implementing class and returns an implementation (which is a Python class deriving from the implementing class). An exception will be raised when

calling `implement()` (page 139) with an implementing class that does not validate, e.g. because methods do not match the declarations made in the interface.

As a development tool, `eoxserver.core.interfaces` (page 136) supports runtime validation of interfaces. This allows to check for consistency of the argument types sent by a calling object to an implementation instance with the argument type declaration in the interface.

Interface Declaration

As mentioned in the introduction, interfaces are ordinary Python classes deriving from `Interface` (page 139) or one of its descendants. Method declarations are made using the `Method` (page 139) class.

The `Method` (page 139) constructor accepts an arbitrary number of argument declarations as positional arguments as well as an optional output declaration stated with the `returns` keyword argument. Argument and output declarations are made using instances of the `Arg` (page 140) class and its descendants.

All argument types take a name as input. For an implementation to validate, this must be a valid Python argument name (except for output declarations). Furthermore, all argument types accept a `default` keyword argument that defines a default value for the argument and marks it as optional.

Let's see an example:

```
from eoxserver.core.interfaces import *

class SomeInterface(Interface):

    f = Method(
        IntArg("x"),
        returns = IntArg("@return")
    )

class AnotherInterface(Interface):

    g = Method(
        FloatArg("x", default=0.0),
        returns = FloatArg("@return")
    )

class SomeDerivedInterface(SomeInterface, AnotherInterface):

    pass
```

In this short code snippet, we declare three interfaces. Implementations of `SomeInterface` shall have a method `f()` that takes an integer `x` as an argument and returns an integer value. Implementations of `AnotherInterface` shall have a method `g()` that takes a float `x` as an argument and returns a float value. `SomeDerivedInterface` inherits from both, so implementations of that interface must exhibit `f()` and `g()` methods that work in the way described above.

Interfaces can have an interface configuration, i.e. a class variable called `INTERFACE_CONF` which contains a dictionary of configuration values. So far, only `runtime_validation_level` is supported, see *Validation of Implementations* (page 138).

Implementations

Implementations will be constructed from implementing classes using the `implement()` (page 139) method of the interface. This method will validate the implementating class and return an implementation class that inherits from the input class.

The implementation exhibits exactly the behaviour of the implementing class. Internally, the implementation may differ considerably from the implementing class, especially if you use runtime validation capabilities, see under *Descriptors* (page 142) below.

Note that you can define any number of additional public or private methods in an implementing class which will be present in the implementation as well. You cannot omit any method or argument declared in the interface, though, as the implementing class would not validate then.

Now for an example of implementations of the interface defined above in section *Interface Declaration* (page 137):

```
class SomeImplementingClass(object):

    def f(self, x):
        return int(x)

class AnotherImplementingClass(object):

    def g(self, x=0.0):
        return float(x)

class AThirdImplementingClass(SomeImplementingClass):

    def g(self, x=0.0):
        return 2.0 * float(x)

SomeImplementation = SomeInterface.implement(SomeImplementingClass)
AnotherImplementation = AnotherInterface.implement(AnotherImplementingClass)
AThirdImplementation = SomeDerivedInterface.implement(AThirdImplementingClass)
```

As you can see, `SomeImplementingClass` implements `SomeInterface`. The required method `f()` is present and has the correctly named input parameters and even enforces that the output has the correct type, though this can only be validated using runtime validation (not when creating the implementation).

In `AnotherImplementingClass` you see an example for default value declaration.

`AThirdImplementingClass` is interesting in two ways. First, it derives from `SomeImplementingClass` inheriting its `f()` method. This way you can build hierarchies of implementing classes similar to the way you can build hierarchies of interfaces. Second, you see that the implementation hierarchies may deviate from the interface hierarchies; instead of inheriting the `g()` method from `AnotherImplementingClass` an alternative version of this method is implemented that again matches the interface declaration.

If you have an implementation and want to know which interface it implements you can use the magic `__ifclass__` attribute:

```
>>> AThirdImplementingClass.__ifclass__.__name__
'SomeDerivedInterface'
```

Implementing classes can define an implementation configuration, i.e. class variable called `IMPL_CONF` that contains a dictionary of configuration settings. So far, only `runtime_validation_level` is supported, see *Validation of Implementations* (page 138).

Validation of Implementations

The validation of implementations is performed in two ways:

- at class creation time
- at instance method invocation time ("runtime")

Validation at class creation time checks:

- if all methods declared by the interface are implemented
- if the method arguments of the interface and implementation match in the sense that
  - all declared arguments are present
  - the names and the order of the arguments in the implementation match the interface declaration
  - the optional default value declarations match

Class creation time validation is performed unconditionally.

Instance method invocation time ("runtime") validation is optional. It can be triggered by the `runtime_validation_level` setting. There are three possible values for this option:

- `trust`: no runtime validation
- `warn`: argument types are checked against interface declaration; in case of mismatch a warning is written to the log file
- `fail`: argument types are checked against interface declaration; in case of mismatch an exception is raised

The `runtime_validation_level` option can be set

- globally (in the configuration file, see *Config Reader* (page 141))
- per interface (in the `INTERFACE_CONF` dictionary)
- per implementation (in the `IMPL_CONF` dictionary)

where stricter settings override weaker ones.

Note: The `warn` and `fail` levels are intended for use throughout the development process. In a production setting `trust` should be used.

Reference

This documentation concentrates on the public methods of the involved classes. Actually, there is only one public method you will need to invoke and that is `Interface.implement()` (page 139); all others are public only to the extent that they are invoked by other objects defined in this module.

The implementation of `eoxserver.core.interfaces` (page 136) involves some deep and beautiful Python magic. We skip most of these details here, only in the *Descriptors* (page 142) sections you will find a reference to some of it.

Interfaces

class `eoxserver.core.interfaces.`**`Interface`**

This is the base class for all interface declarations. Derive from it or one of its subclasses to create your own interface declaration.

The `Interface` (page 139) class has only class variables (the method declarations) and class methods.

classmethod **`implement`**(*InterfaceCls*, *ImplementationCls*)

This method takes an implementing class as input, validates it, and returns the implementation.

In the validation step, `Method.validateImplementation()` (page 140) is called for each method declared in the interface. `InternalError` (page 133) is raised if a method is not found or if the method signature does not match the declaration.

If validation has passed, the implementation is getting prepared. The implementation inherits from the implementing class. The `__ifclass__` magic attribute is added to the class dictionary. If runtime validation has been enabled, the methods of the implementing class defined in the interface are replaced by descriptors (instances of `WarningDescriptor` (page 142) or `FailingDescriptor` (page 142)).

Finally, the implementation class is generated and returned.

Methods

class `eoxserver.core.interfaces.`**`Method`**(**args*, ***kwargs*)

The `Method` (page 139) is used for method declarations in interfaces. Its constructor accepts an arbitrary number of positional arguments representing input arguments to the method to be defined, and one optional keyword argument `returns` which represents the methods return value, if any.

All arguments must be instances of `Arg` (page 140) or one of its subclasses.

The methods of the `Method` (page 139) class are intended for internal use by the `Interface` (page 139) validation algorithms only.

validateArgs(*args*)
Validate the input arguments. That is, check if they are in the right order and no argument is defined more than once. Raises `InternalError` (page 133) if the arguments do not validate.

Used internally by the constructor during instance creation.

validateImplementation(*impl_method*)
This method is at implementation class creation time to check if the implementing class method conforms to the method declaration. It expects the corresponding method as its single input argument `impl_method`. It makes extensive use of Python's great introspection capabilities.

Raises `InternalError` (page 133) in case the implementation does not validate.

validateReturnType(*method_name*, *ret_value*)
This method is called for runtime argument type validation. It expects the method name `method_name` and the return value `ret_value` as input and checks the return value against the return value declaration, if any.

Raises `TypeMismatch` (page 134) if validation fails.

validateType(*method_name*, **args*, ***kwargs*)
This method is called for runtime argument type validation. It gets the input of the implementing method and checks it against the argument declarations.

Raises `TypeMismatch` (page 134) if validation fails.

Arguments

class `eoxserver.core.interfaces.`**Arg**(*name*, ***kwargs*)
This is the common base class for arguments of any kind; it can be used in interface declarations as well to represent an argument of arbitrary type.

The constructor requires a `name` argument which denotes the argument name. The validation will check at class creation time if the method of an implementing class defines an argument of the given name, so you should always use valid Python variable names here (you can use arbitrary strings for return value declarations though).

Furthermore, the constructor accepts a `default` keyword argument which defines a default value for the declared argument. The validation will check at class creation time if this default value is present in the implementing class and fail if it is not.

Its methods are intended for internal use in runtime validation.

getExpectedType()
Returns the expected type name; used in error messages only. This method is overridden by `Arg` (page 140) subclasses in order to customize error reporting. The base class implementation returns `""`.

isOptional()
Returns `True` if the argument is optional, meaning that a default value has been defined for it, `False` otherwise.

isValid(*arg_value*)
Returns `True` if `arg_value` is an acceptable value for the argument, `False` otherwise. Acceptable values are either the default value if it has been defined or values of the expected type.

isValidType(*arg_value*)
Returns `True` if the argument value `arg_value` has a valid type, `False` otherwise. This method is overridden by `Arg` (page 140) subclasses in order to check for individual types. The base class implementation always returns `True` meaning that all types of argument values are accepted.

class eoxserver.core.interfaces.**StrArg**(*name*, ***kwargs*)
Represents an argument of type `str`.

class eoxserver.core.interfaces.**UnicodeArg**(*name*, ***kwargs*)
Represents an argument of type `unicode`.

class eoxserver.core.interfaces.**StringArg**(*name*, ***kwargs*)
Represents an argument of types `str` or `unicode`.

class eoxserver.core.interfaces.**BoolArg**(*name*, ***kwargs*)
Represents an argument of type `bool`.

class eoxserver.core.interfaces.**IntArg**(*name*, ***kwargs*)
Represents an argument of type `int`.

class eoxserver.core.interfaces.**LongArg**(*name*, ***kwargs*)
Represents an argument of type `long`.

class eoxserver.core.interfaces.**FloatArg**(*name*, ***kwargs*)
Represents an argument of type `float`.

class eoxserver.core.interfaces.**RealArg**(*name*, ***kwargs*)
Represents a real number argument, i.e. an argument of types `int`, `long` or `float`.

class eoxserver.core.interfaces.**ComplexArg**(*name*, ***kwargs*)
Represents a complex number argument of type `complex`.

class eoxserver.core.interfaces.**IterableArg**(*name*, ***kwargs*)
Represents an iterable argument.

class eoxserver.core.interfaces.**SubscriptableArg**(*name*, ***kwargs*)
Represents a subscriptable argument.

class eoxserver.core.interfaces.**ListArg**(*name*, ***kwargs*)
Represents an argument of type `list`.

class eoxserver.core.interfaces.**DictArg**(*name*, ***kwargs*)
Represents an argument of type `dict`[13].

class eoxserver.core.interfaces.**ObjectArg**(*name*, ***kwargs*)
Represents an new-style class argument. The range of accepted objects can be restricted by providing the `arg_class` keyword argument to the constructor. Runtime validation will then check if the argument value is an instance of `arg_class` (or one of its subclasses) and fail otherwise.

class eoxserver.core.interfaces.**PosArgs**(*name*, ***kwargs*)
Represents arbitrary positional arguments as supported by Python with the `method(self, *args)` syntax. The range of accepted objects can be restricted by providing the `arg_class` keyword argument to the constructor. Runtime validation will then check if the argument value is an instance of `arg_class` (or one of its subclasses) and fail otherwise.

Note that a `PosArgs` (page 141) argument declaration can only be followed by a `KwArgs` (page 141) declaration, otherwise validation will fail.

class eoxserver.core.interfaces.**KwArgs**(*name*, ***kwargs*)
Represents arbitrary keyword arguments as supported by Python with the `method(self, **kwargs)` syntax. Note that this must always be the last input argument declaration in a method, otherwise validation will fail.

Config Reader

class eoxserver.core.interfaces.**IntfConfigReader**(*config*)
This is the configuration reader for `eoxserver.core.interfaces` (page 136).

Its constructor expects a `Config` instance `config` as input.

[13] http://docs.python.org/2.7/library/stdtypes.html#dict

getRuntimeValidationLevel()
Returns the global runtime validation level setting or None if it is not defined.

validate()
Validates the configuration. Raises ConfigError (page 133) if the runtime_validation_level configuration setting in the core.interfaces section contains an invalid value.

Descriptors Descriptors are used to customize method access in Python. They are some of the more advanced Python language features; if you want to know more about them, please refer to the Python Language Reference[14].

class eoxserver.core.interfaces.**ValidationDescriptor**(*method*, *func*)
This is the common base class for WarningDescriptor (page 142) and FailingDescriptor (page 142). The constructor expects the method declaration method and the implementing function func as input.

The __get__() method returns a callable wrapper around the instance it is called with, the method declaration and the function that implements the method. It is that object that gets finally invoked when runtime validation is enabled.

class eoxserver.core.interfaces.**ValidationWrapper**(*method*, *func*, *instance*)
This is the common base class for WarningWrapper (page 142) and FailingWrapper (page 142). Its constructor expects the method declaration, the implementing function and the instance as input.

class eoxserver.core.interfaces.**WarningDescriptor**(*method*, *func*)

class eoxserver.core.interfaces.**WarningWrapper**(*method*, *func*, *instance*)
This wrapper is callable. Its __call__() method expects arbitrary positional and keyword arguments, validates them against the method declaration using Method.validateType() (page 140), calls the implementing function with these arguments and returns whatever it returns, calling Method.validateReturnType() (page 140).

If the validation methods raise a TypeMismatch (page 134) exception the exception text is logged as a warning, but the normal process of execution goes on.

class eoxserver.core.interfaces.**FailingDescriptor**(*method*, *func*)

class eoxserver.core.interfaces.**FailingWrapper**(*method*, *func*, *instance*)
This wrapper is callable. Its __call__() method expects arbitrary positional and keyword arguments, validates them against the method declaration using Method.validateType() (page 140), calls the implementing function with these arguments and returns whatever it returns, calling Method.validateReturnType() (page 140).

If the validation methods raise a TypeMismatch (page 134) exception it will not be caught and thus cause the program to fail.

Module eoxserver.core.readers

This module defines an interface for configuration readers.

class eoxserver.core.readers.**ConfigReaderInterface**
This interface is intended to provide a way to validate and access the system configuration to modules which rely on configuration parameters. It defines only one mandatory validate() (page 142) method, but developers are free to add methods or attributes that give easy access to the underlying configuration values.

validate(*config*)
This method shall validate the given system configuration config (a Config (page 132) instance). It shall raise a ConfigError (page 133) exception in case the configuration with respect to the sections and parameters concerned by the implementation is invalid. It has no return value.

[14] http://docs.python.org/reference/datamodel.html#invoking-descriptors

The validate() (page 142) method is called automatically at system startup or configuration reset. If it fails system startup or reset will not succeed. So please be careful to raise ConfigError (page 133) only in situations

- when the components that need the parameter(s) are enabled
- when the configuration will always lead to an error

Otherwise, configuration errors of one optional module might break the whole system.

Module eoxserver.core.records

This module provides interfaces as well as a simple implementation for record wrappers. The design objective for this module was to provide a more lightweight alternative to resource wrappers based on eoxserver.core.resources (page 153).

Record wrappers shall couple data stored in the database with additional application logic. They are lazy in the sense that data assigned to the wrapper is not written to the database immediately. They are also immutable, i.e. once they have been initialized their data cannot be changed any more. Last but not least, they are able to cope with non-abstract model inheritance.

class eoxserver.core.records.**RecordWrapper**

This is a common base class for RecordWrapperInterface (page 144) implementations.

Concrete implementations may derive from it overriding the respective methods.

delete(*commit=True*)

Delete the model record wrapped by the instance from the database and perform any additional logic related to the deletion. Do nothing if there is no model record defined. See ResourceWrapperInterface.delete() for a description of the commit parameter.

getRecord(*fetch_existing=False*)

Get the model record wrapped by the instance (i.e. an instance of a subclass of django.db.models.Model[15]). This method calls sync() (page 143) to fetch or create a record if none has been defined. The fetch_existing argument is parsed to sync() (page 143).

getType()

This method shall return the type of record the wrapper represents. Raises InternalError (page 133) by default.

setAttrs(***kwargs*)

Assign the attributes given as keyword arguments to the instance. This method raises InternalError (page 133) if attributes do not validate.

setRecord(*record*)

Assign the model record record to the instance. This method raises InternalError (page 133) if the record does not validate.

sync(*fetch_existing=False*)

class eoxserver.core.records.**RecordWrapperFactory**

This factory gives access to record wrappers.

create(***kwargs*)

Create a data package wrapper instance of a given type. This method expects a type keyword argument that indicates the data package type of the wrapper to be created. All other keyword arguments are passed on to the setAttrs() (page 143) method of the respective wrapper class.

InternalError (page 133) is raised if the type keyword argument is missing or does not contain a valid type name. InternalError (page 133) exceptions raised by RecordWrapper.setAttrs() (page 143) are passed on as well.

delete(***kwargs*)

Delete model records in bulk. Not yet implemented.

[15] https://docs.djangoproject.com/en/1.4/ref/models/instances/#django.db.models.Model

find(***kwargs*)
Find database model records and return the corresponding record wrappers. Not yet implemented.

get(***kwargs*)
Get a record wrapper for a database model record. This method accepts either one of the following two keyword arguments:

- `pk`: primary key of a record
- `record`: a model record

`InternalError` (page 133) is raised if none of these is given. The data package wrapper returned will be of the right type for the given model record.

getOrCreate(***kwargs*)
Get a wrapper for an existing record with the given attributes or create a new one. This calls `create()` (page 143) and `RecordWrapper.sync()` (page 143). The returned wrapper will always contain a database record (it is not lazy).

`InternalError` (page 133) is raised if there are mandatory attribute keyword arguments missing and `UniquenessViolation` (page 134) if the record could not be created due to unqiueness constraints.

update(***kwargs*)
Update model records in bulk. Not yet implemented.

class `eoxserver.core.records.`**RecordWrapperFactoryInterface**
This is the interface for factories returning record wrappers, i.e. implementations of `RecordWrapperInterface` (page 144). It inherits from `FactoryInterface` (page 152).

create(***kwargs*)
Create a record wrapper with the given attributes. The keyword arguments accepted by this method shall correlate to the attribute keyword arguments accepted by the underlying `ResourceWrapperInterface` implementations.

For factories that generate different types of record wrappers a mandatory `type` keyword argument shall be required that shall correlate to the return value of the `ResourceWrapperInterface.getType()` method of the desired record wrapper type.

getOrCreate(***kwargs*)
Get or create a record wrapper with the given attributes. That is, if a database record matching all the given attributes exists, return a wrapper with this record, otherwise create a new record. The keyword arguments accepted by this method shall correlate to the attribute keyword arguments accepted by the underlying `ResourceWrapperInterface` implementations.

For factories that generate different types of record wrappers a mandatory `type` keyword argument shall be required that shall correlate to the return value of the `ResourceWrapperInterface.getType()` method of the desired record wrapper type.

update(***kwargs*)
Update model records in bulk. Return the record wrappers for the updated records.

delete(***kwargs*)
Delete model records in bulk and apply any additional logic defined by the specific record wrapper implementations; see `RecordWrapperInterface.delete()` (page 145).

class `eoxserver.core.records.`**RecordWrapperInterface**
This class defines an interface for simple lazy record wrappers which are used throughout EOxServer to couple data and metadata stored in the configuration database with additional application logic.

Implementations of this interface shall wrap a model record and couple it with additional attributes and dynamic behaviour. The wrapper shall be lazy, i.e. any changes to the model record or to the attributes will not affect the database until the programmer explicitly calls `save()`, `getRecord()` (page 145).

getType()
This method shall return the type of record the wrapper represents. This method is needed especially

by factories which return multiple classes of `RecordWrapperInterface` (page 144) implementations that wrap models inheriting from a common base model.

setRecord(*record*)
This method shall initialize the record wrapper with an existing model record. It shall raise `InternalError` (page 133) in case the record is of an incompatible type

setAttrs(***kwargs*)
This method shall initialize the record wrapper with implementation dependent attributes. It shall raise `InternalError` (page 133) in case there are mandatory attributes missing.

sync(*fetch_existing=False*)
Synchronize with the database, i.e. fetch or create a record with the instance attributes.

The method shall respect uniqueness constraints on the underlying model, i.e. return an existing record matching the instance attributes if possible and create a new one only if all constraints are satisfied. In case neither is possible `UniquenessViolation` (page 134) shall be raised.

If the optional `fetch_existing` argument is set to `True`, try to get an existing record with the same attributes from the database even if no uniqueness constraints apply. If there is none, create a new one.

getRecord(*fetch_existing=False*)
Return the record wrapped by the implementation. If none has been defined yet, fetch or create one with the instance attributes.

The method shall respect uniqueness constraints on the underlying model, i.e. return an existing record matching the instance attributes if possible and create a new one only if all constraints are satisfied. In case neither is possible `UniquenessViolation` (page 134) shall be raised.

If the optional `fetch_existing` argument is set to `True`, try to get an existing record with the same attributes from the database even if no uniqueness constraints apply. If there is none, create a new one.

delete(*commit=True*)
Delete the model record and perform any related logic.

This method accepts an optional boolean parameter `commit` which defaults to `True`. If it is set to `False` do not actually delete the record, but do perform the additional logic. This is useful for bulk deletion by factories; it should be used with great care as it might leave the system in an inconsistent state if the database record is not removed afterwards.

Module eoxserver.core.registry

This module contains the implementation of the registry as well as associated interface declarations. The registry is the core component of EOxServer that links different parts of the system together. The registry allows for components to bind to implementations of registered interfaces. It supports modularity, extensibility and flexibility of EOxServer.

Introduction

The registry has been introduced as a core component of EOxServer with *RFC 2: Extension Mechanism for EOxServer* (page 270). The registry is the central entry point for:

- automated detection of registered interfaces and implementations
- dynamic binding to the implementations
- configuration of components and relations between them

The concept of the registry builds on interfaces and their implementations as defined in `eoxserver.core.interfaces` (page 136).

This module defines a specialized class of interfaces called RegisteredInterface (page 152). This subclass of Interface (page 139) defines additional metadata needed by the registry and adds some more logic to the implementation process. In order to include implementations in the registry the interface has to be derived from RegisteredInterface (page 152).

Implementations of registered interfaces can be detected automatically by the registry and are then ingested into it. The information stored in the registry consists of:

- registered interfaces
- implementations of registered interfaces
- status of implementations (components)
- binding method and parameters

Implementations can be switched on and off. The registry will only return instances of implementations that are enabled. This feature can be used to fine-tune the behaviour of the system.

There are several binding methods defined that determine how to get instances of implementations of registered interfaces. Binding can be parametrized, so that the appropriate implemenation is chosen based on some parameters conveyed with the request. As the parameters can be defined at runtime and as new implementations with other binding parameters can be added in a flexible way, we speek of dynamic binding.

Registered Interfaces and Implementations

The registry is a repository for interfaces and implementations. Only interfaces derived from RegisteredInterface (page 152) and their respective implementations will be included in the registry.

RegisteredInterface (page 152) adds some features and requirements to the Interface (page 139) base class. First of all, it expects a REGISTRY_CONF dictionary class variable for the interface declaration. The following keys are accepted or required:

- name: The name of the interface (mandatory)
- intf_id: The unique ID of the interface; by convention this should include the dotted module name (mandatory)
- binding_method: The name of the binding method (optional, defaults to direct)

Depending on the binding method additional parameters may be required. See the *Dynamic Binding* (page 147) section.

The implement() method of RegisteredInterface (page 152) validates the REGISTRY_CONF and adds "magic" methods and attributes to the interface class that can be queried at runtime in order to retrieve information about the interface.

Detection and Registration

At startup the registry is initialized by the System class in eoxserver.core.system (page 157). It calls the registry's load() (page 151) method which automatically detects registered interfaces and their implementations in certain modules. Configuration settings define which modules will be scanned.

The settings for the registry can be found in the [core.registry] section of default.conf and eoxserver.conf. The following settings are recognized:

- system_modules (in default.conf only): system modules that will always be scanned for registered interfaces and their implementations
- module_dirs: a comma separated list of directories; every Python module in these directories will be scanned
- modules: a comma separated list of modules that shall be scanned

The settings in `eoxserver.conf` can be customized by the user in order to leave out certain parts of the EOxServer distribution or to load additional extensions (plugins).

When loading modules, the registry looks for classes that implement certain magic functions which are tagged onto them by the `RegisteredInterface.implement()` method. These implementation classes are registered together with the interfaces they implement.

In the registration process, the implementation and interface classes are stored in indexes where they can be looked up in the finding and binding process.

Dynamic Binding

The registry provides four binding methods:

- direct binding
- KVP binding
- test binding
- factory binding

Direct binding means that the implementation to bind to is directly referenced by the caller using its implementation ID:

```
from eoxserver.core.system import System

impl = System.getRegistry().bind(
    "somemodule.SomeImplementation"
)
```

Direct binding is available for every implementation. You can also set the `binding_method` in the `REGISTRY_CONF` of an interface to `direct`, meaning that its implementations are reachable only by this method. This is used e.g. for component managers and factories.

The easiest method for parametrized dynamic binding is key-value-pair matching, or KVP binding. It is used if an interface defines `kvp` as its `binding_method`. The interface must then define in its `REGISTRY_CONF` one or more `registry_keys`, the implementations in turn must define `registry_values` for these keys. When looking up a matching implementation, the parameters given with the request are matched against these key-value-pairs. Finally, the registry returns an instance of the matching implementation:

```
from eoxserver.core.system import System

def dispatch(service_name, req):

    service = System.getRegistry().findAndBind(
        intf_id = "services.interfaces.ServiceHandler",
        params = {
            "services.interfaces.service": service_name.lower()
        }
    )

    response = service.handle(req)

    return response
```

This binding method is used e.g. for binding to service, version and operation handlers for OGC Web Services based on the parameters sent with the request.

A more flexible way to determine which implementation to bind to is the test binding method (`"binding_method": "testing"`). In this case, the interface must be derived from `TestingInterface` (page 152). The implementation must provide a `test()` method which will be invoked by the registry in order to determine if it is suitable for a given set of parameters. This can be used e.g. to determine which format handler to use for a given dataset:

```
from eoxserver.core.system import System

format = System.getRegistry().findAndBind(
    intf_id = "resources.coverages.formats.FormatInterface",
    params = {
        "filename": filename
    }
)

...
```

The fourth binding method is factory binding (`"binding_method": "factory"`). In this case the registry invokes a factory that returns an instance of the desired implementation. Factories must be implementations of a descendant of `FactoryInterface` (page 152). Implementations and factories are linked together only at runtime, based on the metadata collected during the detection phase. This binding method is used e.g. for binding to instances of a resource wrapper:

```
from eoxserver.core.system import System

resource = System.getRegistry().getFromFactory(
    factory_id = "resources.coverages.wrappers.SomeResourceFactory",
    obj_id = "some_resource_id"
)
```

In order to access other functions of the factory you can bind to it directly. For retrieving all resources that are accessible through a factory you would use code like this:

```
from eoxserver.core.system import System

resource_factory = System.getRegistry().bind(
    "resources.coverages.wrappers.SomeResourceFactory"
)

resources = resource_factory.find()
```

Components and Resources

The registry has its own data model which distinguishes between components (the active parts of the system, like OWS handlers etc.) and resources (the data components deal with, like coverages etc.).

At the moment, the registry itself does not detect if a given implementation is a resource class or a component, but this will change in future versions of the software.

Components have a status, i.e. they can be enabled or disabled. That status is a configuration parameter stored in the database. At system startup the registry will synchronize the status of the implementations it detects with the status in the database. If a given implementation is not found in the database, a new `Implementation` record will be generated, with its status set to disabled.

If some component is trying to get a disabled component from the registry an `ImplementationDisabled` (page 133) exception will be raised.

RFC 2: Extension Mechanism for EOxServer (page 270) proposes a far more sophisticated system for dealing with resources and components. This will be implemented step by step in future versions.

Reference

Registry

class `eoxserver.core.registry.`**`Registry`**(*config*)

The `Registry` (page 148) class implements the functionalities for detecting, registering, finding and binding to implementations of registered interfaces. It is instantiated by `eoxserver.core.system.System` (page 157) during the startup process.

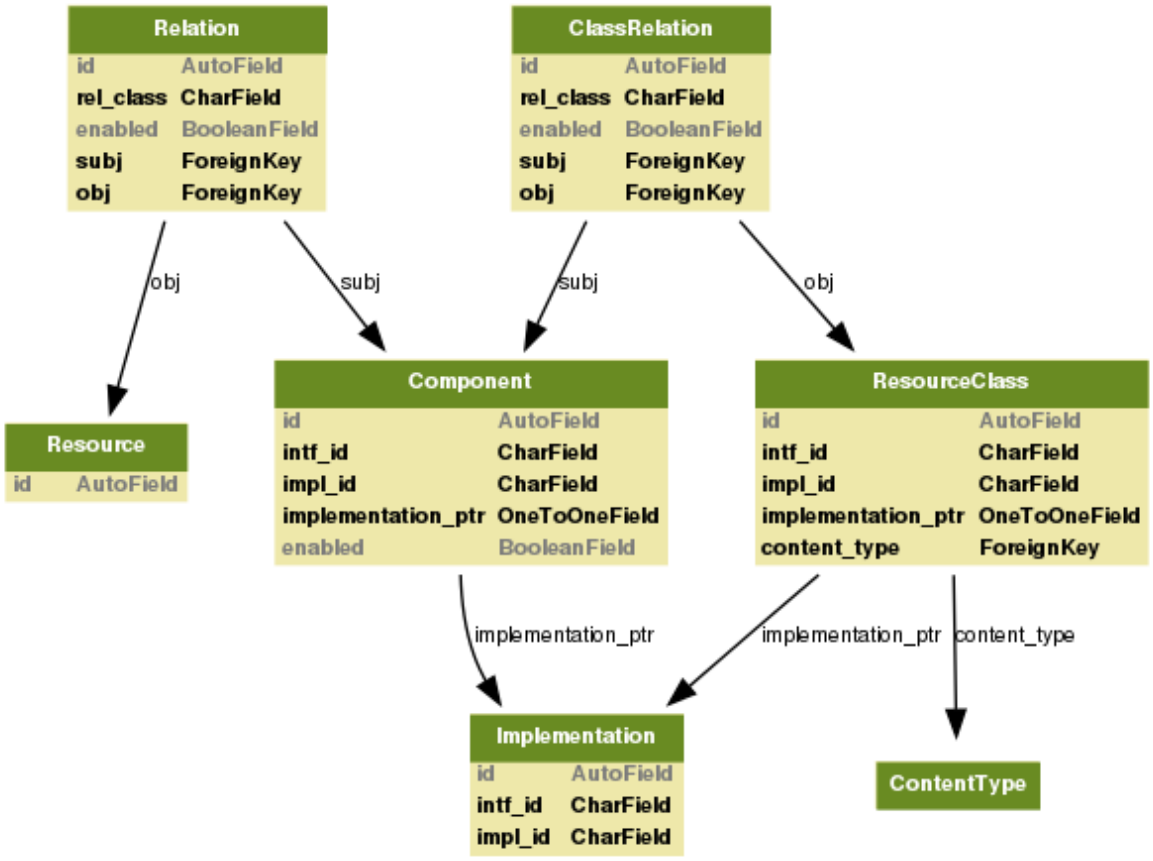


Figure 2.6: *Database Model for the Registry*

The constructor expects a Config (page 132) instance as input. The values will be validate and read using a RegistryConfigReader (page 152) instance.

bind(*impl_id*)
Bind to the implementation with ID impl_id. This method returns a new instance of the requested implementation if it is enabled.

If the implementation is disabled ImplementationDisabled (page 133) will be raised. If the ID impl_id is not known to the registry ImplementationNotFound will be raised.

clone()
Returns an exact copy of the registry.

disableImplementation(*impl_id*)
Changed the implementation status to disable for implementation ID impl_id. Note that this change is not automatically stored to the database (you have to call save() (page 151) to do that).

Raises InternalError (page 133) if the implementation ID is unknown.

enableImplementation(*impl_id*)
Changes the implementation status to enabled for implementation ID impl_id. Note that this change is not automatically stored to the database (you have to call save() (page 151) to do that).

Raises InternalError (page 133) if the implementation ID is unknown.

findAndBind(*intf_id*, *params*)
This method finds implementations based of a registered interface with ID intf_id using the parameter dictionary params and returns an instance of the matching implementation. This works only for the kvp and testing binding methods, in other cases BindingMethodError (page 133) will be raised.

If the binding method of the interface is kvp the params dictionary must map the registry keys defined in the interface declaration to values. The KVP combination will be compared with the values given in the respective implementations. If a matching implementation is found an instance will be returned, otherwise ImplementationNotFound (page 133) is raised. If the class found is disabled ImplementationDisabled (page 133) is raised.

If the binding method of the interface is testing the params dictionary will be passed to the test() (page 152) method of the respective implementations. If no implementation matches ImplementationNotFound (page 133) will be raised. If more than one are found ImplementationAmbiguous (page 133) will be raised.

findImplementations(*intf_id*, *params=None*, *include_disabled=False*)
This method returns a list of implementations of a given interface. It requires the interface ID as a parameter.

Furthermore, a parameter dictionary can be passed to the method. The results will then be filtered according to these parameters. The dictionary does not have to contain all the parameters defined by the interface; in case some parameters are omitted, the result list may contain several different implementations.

Third is an optional include_disabled parameter with defaults to False. If True, disabled implementations will be reported as well.

An InternalError (page 133) is raised if parameters are passed to the method that are not defined in the interface declaration or not recognized by the interface test() method.

getFactoryImplementations(*factory*)
Returns a list of implementations for a given factory.

Raises InternalError (page 133) if the factory is not found in the registry.

getFromFactory(*factory_id*, *params*)
Get an implementation instance from the factory with ID factory_id using the parameter dictionary params. This is a shortcut which binds to the factory and calls its get() (page 152) method then.

InternalError (page 133) will be raised if required arguments are missing in the params dictionary. ImplementationDisabled (page 133) will be raised if either the factory or the appropriate implementation are disabled. ImplementationNotFound (page 133) will be raised if either the factory or the appropriate implementation are unknown to the registry.

getImplementationIds(*intf_id*, *params=None*, *include_disabled=False*)
This method returns a list of implementation IDs for a given interface. It requires the interface ID as a parameter.

Furthermore, a parameter dictionary can be passed to the method. The results will then be filtered according to these parameters. The dictionary does not have to contain all the parameters defined by the interface; in case some parameters are omitted, the result list may contain several different implementations.

Third is an optional include_disabled parameter with defaults to False. If True, disabled implementations will be reported as well.

An InternalError (page 133) is raised if parameters are passed to the method that are not defined in the interface declaration or not recognized by the interface test() method.

getImplementationStatus(*impl_id*)
Returns the implementation status (True for enabled, False for disabled) for the given implementation ID impl_id.

Raises InternalError (page 133) if the implementation ID is unknown.

getRegistryValues(*intf_id*, *registry_key*, *filter=None*, *include_disabled=False*)
This method returns a list of registry values of implementations of an interface with interface_id and a registry key registry_key defined in the interface declaration.

With the filter argument you can impose certain restrictions on the implementations (registry values) to be returned. It is expected to contain a dictionary of registry keys and values that the implementation must expose to be included.

Using the include_disabled argument you can determine whether.

This method raises InternalError (page 133) if the interface ID is unknown.

load()
This method loads the registry, i.e. it scans the modules specified in the configuration for interfaces and implementations. It is invoked by the ~.System class upon initialization.

You should *never* invoke this method directly. Always use init() (page 157) to initialize and getRegistry() to access the registry.

There are three configuration settings taken into account.

First, the system_modules setting in default.conf. These modules are always loaded and cannot be left aside in individual instances.

Second, the module_dirs setting in the local configuration of the instance (eoxserver.conf) is taken into account. This expected to be a comma-separated list of directories. These directories and all the directory trees underneath them are searched for Python modules.

Third, the modules setting in eoxserver.conf. This is expecte to be a comma-separated list of module names which shall be loaded.

All modules specified or detected by scanning directories will be loaded and searched for interfaces descending from RegisteredInterface (page 152) as well as their implementations. These will be automatically included in the registry and accessible using the different binding methods provided.

As a last step, the registry is synchronized with the database. This means that the implementation looks up the entries for the different implementations in the database and determines whether they are enabled or not. If it finds an implementation which has not yet been registered it will be saved to the database but disabled by default.

save()
This saves the registry configuration to the database. This means the status of the enabled / disabled flag for each implementation will be saved overriding any previous settings stored.

validate()
This method is intended to validate the component configuration.

It looks up all implementations of ComponentManagerInterface (page 152) and calls their respective validate() methods.

At the moment, no component managers are implemented, so this method does not have any effects.

Interfaces

class eoxserver.core.registry.**RegisteredInterface**
This class is the base class for all interfaces to be registered in the registry. All interfaces whose implementations shall be registered must be derived from RegisteredInterface (page 152).

All interfaces derived from RegisteredInterface (page 152) must contain a REGISTRY_CONF dictionary. See the introduction for details.

class eoxserver.core.registry.**TestingInterface**
This class is a descendant of RegisteredInterface (page 152) that adds a single method. It is used for binding by test, which enables binding decisions that cannot easily be implemented by key-value-pair comparisons.

test(*params*)
This method is invoked by the registry when determining which implementation to bind to. Based on the parameter dictionary params the method shall decide whether the implementation is applicable and return True. If it is not applicable the method shall return False.

class eoxserver.core.registry.**FactoryInterface**
This is the basic interface for factories. It is a descendant of RegisteredInterface (page 152).

get(***kwargs*)
This method shall return an instance of an implementation that matches the parameters given as keyword arguments. The set of arguments understood depends on the individual factory and can be found in the respective documentation.

The method shall raise an exception if no matching implementation or instance thereof can be found, or if the choice is ambiguous.

find(***kwargs*)
This method shall return a list of implementation instances that matches the parameters given as keyword arguments. The set of arguments understood depends on the individual factory and can be found in the respective documentation.

class eoxserver.core.registry.**ComponentManagerInterface**
This interface is not in use at the moment. It was intended to provide an API for controlling the status of a larger set of implementations and their dependencies, though the concept has never been elaborated.

Config Reader

class eoxserver.core.registry.**RegistryConfigReader**(*config*)
This class provides some functions for reading configuration settings used by the Registry (page 148).

getModuleDirectories()
This method returns a list of directory paths where to look for modules to load (see also Registry.load() (page 151)). The values are read from the module_dirs setting in the [core.registry] section of the instance specific eoxserver.conf configuration file.

The format of the setting is expected to be a comma-separated list of paths.

getModules()
This method returns a list of dotted names of modules to be loaded (see also Registry.load()

(page 151)). The values are read from the `modules` setting in the `[core.registry]` section of the instance specific `eoxserver.conf` configuration file.

The format of the setting is expected to be a comma-separated list of dotted module names.

getSystemModules()
This method returns a list of dotted names of system modules. The values are read from `system_modules` setting in the `[core.registry]` section of the `default.conf` configuration file.

The format of the setting is expected to be a comma-separated list of the module names.

validate()
Validates the configuration; a no-op at the moment.

Module eoxserver.core.resources

class `eoxserver.core.resources.`**ResourceFactory**
This is the base class for implementations of `ResourceFactoryInterface` (page 154).

create(***kwargs*)
This method creates a resource according to the given parameters and returns it to the caller. It accepts one mandatory and two optional parameters:

- `subj_id`: the id of the calling component (optional)
- `impl_id`: the implementation ID of the resource to be created (mandatory)
- `params`: a dictionary of parameters to initialize the resource with; the format of this dictionary is specific to the resource class

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

delete(***kwargs*)
This method deletes a selection of resources. It accepts the following parameters:

- `subj_id`: the id of the calling component
- `obj_id`: the resource ID of the resource
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

The `obj_id` argument and the `impl_ids` and `filter_exprs` arguments on the other hand are mutually exclusive. `InternalError` is raised if these conditions are not met.

exists(***kwargs*)
Returns `True` if there are resources matching the given criteria, or `False` otherwise.

- `subj_id`: the id of the calling component
- `obj_id`: the id of the requested resource
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources

Note that `filter_exprs` will not be taken into account when `obj_id` is given.

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

find(***kwargs*)
Returns a list of resource instances matching the given search criteria. This method accepts three optional arguments:

- `subj_id`: the id of the calling component

- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

get (***kwargs*)
Returns the resource instance wrapping the resource model defined by the input parameters. This method accepts three optional keyword arguments:

- `subj_id`: the id of the calling component
- `obj_id`: the resource ID of the resource
- `filter_exprs`: a list of filter expressions that define the resource

Note that `obj_id` and `filter_exprs` are mutually exclusive, but exactly one of them must be provided. The `subj_id` argument will be used to check for relations to the resource (not yet implemented).

getAttrValues (***kwargs*)
This method returns the values of a given attribute for a selection of resources.

- `subj_id`: the id of the calling component
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources
- `attr_name`: the attribute name (mandatory)

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

Raises `InternalError` (page 133) if the `attr_name` argument is missing, or `UnknownAttribute` (page 134) if the attribute name is not known to a resource.

getIds (***kwargs*)
This method returns the IDs of a selection of resources. It accepts the following parameters:

- `subj_id`: the id of the calling component
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

update (***kwargs*)
This method runs updates on a selection of resources and returns the updated resources. It accepts the following parameters:

- `subj_id`: the id of the calling component
- `obj_id`: the resource ID of the resource
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources
- `attrs`: a dictionary of attribute names and values; the attribute names are specific to the resource classes
- `params`: a dictionary of parameters to update the resource with; the format of this dictionary is specific to the resource classes

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

The `obj_id` argument and the `impl_ids` and `filter_exprs` arguments on the other hand are mutually exclusive. The `attrs` and `params` arguments are mutually exclusive as well, exactly one of them has to be specified. `InternalError` is raised if these conditions are not met.

class eoxserver.core.resources.**ResourceFactoryInterface**

This is the interface for resource factories. It extends FactoryInterface (page 152) considerably by adding functionality to create, update and delete resources.

create(***kwargs*)

This method shall create a resource according to the given parameters and returns it to the caller. The range of applicable parameters is defined by each implementation.

update(***kwargs*)

This method shall update a resource or a set of resources according to the given parameters and return the updated resources to the caller. The range of applicable parameters is defined by each implementation.

delete(***kwargs*)

This method shall delete a resource or a set of resources according to the given parameters. The range of applicable parameters is defined by each implementation.

getIds(***kwargs*)

This method shall return a list of resource IDs (i.e. the contents of the resource model's id_field field, see ResourceInterface (page 155)) for the resources given by the

getAttrValues(***kwargs*)

This method shall return the values of a given attribute for a selection of resources.

exists(***kwargs*)

This method shall return True if there are resources matching the given search criteria, False otherwise.

class eoxserver.core.resources.**ResourceInterface**

This is the interface for resource wrappers. Resource wrappers add application logic to database models based on eoxserver.core.models.Resource.

ResourceInterface (page 155) expects two additional, mandatory parameters in REGISTRY_CONF:

- model_class: the model class for the resources wrapped by the implementation
- id_field: the name of the id field of the implementation

In order to enforce the relation model defined in eoxserver.core.models ResourceInterface (page 155) implementations shall have two different states: if they are mutable, any operations modifying the underlying model are allowed; if they are immutable only non-modifying operations are enabled.

setModel(*model*)

This method shall be used to set the resource model, which is expected to be an instance of Resource or one of its subclasses. An InternalError (page 133) shall be raised if the model class does not match the one defined in the model_class registry setting.

getModel()

This method shall return the resource model. In case the resource is not mutable InternalError (page 133) shall be raised.

createModel(*params*)

This method shall create a database model with the data given in parameter dictionary params. The keys the method understands may vary from implementation to implementation; they are the same as for updateModel() (page 155). In case the resource is not mutable InternalError (page 133) shall be raised.

updateModel(*params*)

This method shall update the database model with the data given in parameter dictionary params. The keys the method understands may vary from implementation to implementation; they are the same as for createModel() (page 155). Note that the new data is not saved immediately but only when saveModel() (page 155) is called. In case the resource is not mutable InternalError (page 133) shall be raised.

saveModel()
This method shall save the model to the database. In case the resource is not mutable `InternalError` (page 133) shall be raised.

deleteModel()
This method shall delete the model from the database. In case the resource is not mutable `InternalError` (page 133) shall be raised.

setMutable(*mutable*)
This method shall set the mutability status of the resource. The optional boolean argument `mutable` defaults to `True`. The implementation shall make sure the mutability status cannot be overridden once it has been set. In case of an attempt to set it a second time `InternalError` (page 133) shall be raised.

getId()
This method shall return the ID of the resource, i.e. the content of the resource's `id_field` field.

getAttrNames()
This method shall return a list of attribute names for the resource. For each attribute name, a call to `getAttrField()` (page 156) reveals the corresponding model field and a call to `getAttrValue()` (page 156) returns the attribute value for the resource.

getAttrField(*attr_name*)
This method shall return a the field name for a given attribute name.

getAttrValue(*attr_name*)
This method shall return the attribute value for a given attribute name.

setAttrValue(*attr_name*, *value*)
This method shall set the attribute with the given name to the given value. In case the resource is not mutable `InternalError` (page 133) shall be raised.

class `eoxserver.core.resources.`**ResourceWrapper**
This is the base class for resource wrapper implementations.

createModel(*params*)
This method shall be used to create models for the concrete coverage type.

deleteModel()
Delete the coverage model.

getAttrField(*attr_name*)
Returns the field name for the attribute named `attr_name`. An `UnknownAttribute` (page 134) exception is raised if there is no attribute with the given name.

getAttrNames()
Returns a list of names of the accessible attributes of the resource.

getAttrValue(*attr_name*)
Returns the value of the attribute named `attr_name`. An `UnknownAttribute` exception is raised in case there is no attribute with the given name.

getId()
This method shall return the model ID, i.e. the content of its `id_field` field. Child classes may override it in order to implement more efficient data access.

getModel()
Returns the model wrapped by this implementation.

saveModel()
Save the coverage model to the database.

setAttrValue(*attr_name*, *value*)
Sets the value of the attribute named `attr_name` to `value`. An `InternalError` (page 133) is raised if the resource is not mutable.

setModel(*model*)
Use this function to set the coverage model that shall be wrapped.

setMutable(*mutable=True*)
This method sets the mutability status of the resource. It accepts one optional boolean argument `mutable` which defaults to `True`. The mutability status can be set only once for each resource, attempts to change it will cause an `InternalError` (page 133) to be raised.

Module eoxserver.core.startup

This module defines an interface for startup handlers that are called during system startup or reset.

class `eoxserver.core.startup.`**StartupHandlerInterface**
This is an interface for startup handlers. These handlers are called automatically in the startup sequence; see the `eoxserver.core.system` (page 157) module documentation. It is intended to be implemented by modules or components that need additional global system setup operations.

startup(*config*, *registry*)
This method is called in the startup sequence after the configuration has been validated and the registry has been set up. Those are passed as `config` and `registry` parameters respectively.

It may perform any additional logic needed for the setup of the components concerned by the implementation.

reset(*config*, *registry*)
This method is called in the reset sequence after the new configuration has been validated and the registry has been set up. Those are passed as `config` and `registry` parameters respectively.

It may perform any additional logic needed by the components concerned by the implementation for switching to the new configuration.

Module eoxserver.core.system

class `eoxserver.core.system.`**System**
TODO

classmethod init()
TODO

2.12.2 Utils

Module eoxserver.core.util.bbox

This module contains definition of the auxiliary 2D bounding box class.

class `eoxserver.core.util.bbox.`**BBox**(*sx=None*, *sy=None*, *ox=0*, *oy=0*, *ux=0*, *uy=0*)
Simple 2D bounding box primitive.

Possible initializations:

- size and zerro offset `BBox(sx,sy)`
- size and offset (lower corner) `BBox(sx,sy,ox,oy)`
- lower and upper corners `BBox(None,None,ox,oy,ux,uy)`

BBox class supports following operators:

& - area intersection (maximum common area) | - area expansion (minimum area contaning both boxes) + - offset translation (adding vector to current value) - - offset translation (substracting vector from current value)

__and__ (*other*)
operator - intersection

__or__ (*other*)
operator - expansion

__add__ (*(ox, oy)*)
operator - offset translation

__sub__ (*(ox, oy)*)
operator - offset translation

as_tuple()
Get bbounding box as (sx.sy,ox,oy) tuple

cup
upper corner tuple (RO)

ext
extent/box area (RO)

off
offset tuple (RO)

ox
x offset/lower corner (RO)

oy
y offset/lower corner (RO)

size
size tuple (RO)

sx
x size (RO)

sy
y size (RO)

ux
x upper corner (RO)

uy
y upper corner (RO)

Module eoxserver.core.util.decoders

This module contains interfaces and partial implementations for parameter decoding. These classes are primarily intended for OWS request parsing, therefore there are currently two concrete implementations:

- `KVPDecoder` (page 162) which provides KVP parameter parsing and
- `XMLDecoder` (page 165) for XML input parsing

The module documentation starts with a bit on type definitions in the schemas used by `KVPDecoder` (page 162) and `XMLDecoder` (page 165).

Schemas

EOxServer parameter decoders use schemas to define how to obtain the values for given parameter names or keys; these schemas are defined as Python dictionaries which follow certain rules. The basic structure of a schema is as follows:

```
PARAM_SCHEMA = {
    "<parameter_name>": {
        "<location_parameter>": "<location>",
        "<type_parameter>": "<type_definition>",
        ["<optional_parameter>": <optional_value>, ...]
    }
}
```

where

- `parameter_name` denotes the name that can be used to retrieve the parameter value in calls to `getValue()` (page 160) or `getValueStrict()` (page 161),
- `location_parameter` denotes the name defined by the concrete decoder implementation for the location parameter, e.g. `kvp_key` for KVP Decoders,
- `location` denotes the location string which must be provided by the developer in a format defined by the decoder implementation, e.g. an XPath expression for XML Decoders,
- `type_parameter` denotes the name defined by the concrete decoder implementation for the type parameter, e.g. `kvp_type` for KVP decoders
- `type_definition` denotes a type definition string formed according to the rules below to be provided by the developer,
- optional parameters may be defined by the decoder implementations.

Type definition strings have a common format. They consist of a basetype definition followed by an optional occurrence definition. The most straightforward way is to simply define a parameter which is expected to occur exactly once:

```
"<base_type_name>"
```

The type of the return value is determined by the `base_type_name` you choose. Now that was easy.

For validation purposes you might want to add an occurrence definition. This means you specify minimum and/or maximum expected occurrence for the parameters. The call to `getValue()` (page 160) and "~.Decoder.getValueStrict' will fail if the actual occurrence of the parameter is outside the bounds defined in the schema. A list will be returned in case the defined maximum occurrence exceeds 1.

The occurrences are declared in square brackets following the `base_type_name`:

```
"<base_type_name>[<min_occ>:<max_occ>]"
```

The parameters `min_occ` and `max_occ` must be castable to integers and will be translated to the expected minimum and maximum occurrences respectively. This is the strictest form of occurrence definitions, but there are shortcuts.

Omitting `min_occ` is allowed; minimum occurrence is then set to 0. Omitting `max_occ` is allowed, meaning the occurrence is unbounded. The occurrence definition may contain only a single occurrence value, meaning the parameter is expected exactly `occ` times:

```
"<base_type_name>[<occ>]"
```

And finally empty brackets mean arbitrary occurrence:

```
"<base_type_name>[]"
```

Important: Occurrence definitions always refer to the occurrence of the parameter itself, and never to its content. For `intlist` and similar base types they do not refer to the length of the parameter list! So `intlist[2]` does not denote a list of two integer values, but a list containing two lists of integer values each with arbitrary length.

Interfaces

class eoxserver.core.util.decoders.**DecoderInterface**
This is the common interface for request parameter decoding.

setParams(*params*)
This method shall set the parameters object to be parsed by the decoder implementation

setSchema(*schema*)
This method shall set the schema used by the decoder instance for parsing the parameters object. The input is expected to be a dictionary that represents the schema.

getValueStrict(*expr*)
This method shall return a parameter value according to the expression `expr`.

`MissingParameterException` (page 134) or one of its descendants shall be raised if the requested value could not be found in the parameters.

`InvalidParameterException` (page 133) or one of its descendants shall be raised if:

- the requested value could not be converted to the expected type
- the occurrence bounds given by the schema are violated
- some other validation of the content defined in the schema fails

`InternalError` (page 133) shall be raised in case the expression `expr` is invalid.

getValue(*expr*, *default=None*)
This method shall return a parameter value according to the expression `expr`.

It shall the optional `default` value or `None` if the requested value could not be found in the parameters.

`InvalidParameterException` (page 133) or one of its descendants shall be raised if:

- the requested value cannot be converted to the expected type
- the occurrence bounds given by the schema are violated
- some other validation of the content defined in the schema fails

getParams()
This method shall return the parameters object the decoder is parsing.

getParamType()
This method shall return the a significative code for the type of parameters the decoder is operating on.

Implementations

class eoxserver.core.util.decoders.**Decoder**(*params=None*, *schema=None*)
This is a partial implementation of the DecoderInterface which defines the basic structure of parameter decoders. It provides canonical implementations of `getValueStrict()` (page 161), `getValue()` (page 160) and the constructor as well as private methods for schema parsing support.

getParamType()
This method is required by the interface definition. It shall return a significative code for the type of parameters the decoder is operating on. It must be overridden by concrete implementations. It raises `InternalError` (page 133) by default.

getParams()
This method is required by the interface definition. It shall return the parameters object the decoder implementation is parsing. It must be overridden by concrete implementations. It raises `InternalError` (page 133) by default.

getValue(*expr*, *default=None*)
This method is required by the interface definition. It invokes `getValueStrict()` (page 161), but returns `None` or an optional default value in case the parameter is not found.

getValueStrict(*expr*)
This method is required by the interface definition. It returns the parsing result for the given expression `expr`. Any occurring exceptions will be passed on to the caller.

The method invokes either `_getValueSchema()` if a schema has been defined or `_getValueDefault()` otherwise.

setParams(*params*)
This method is required by the interface definition. It shall set the parameters object to be parsed by the decoder. It may also validate and prepare parsing of the input. It must be overridden by the concrete implementations; `InternalError` (page 133) is raised by default.

setSchema(*schema*)
This method is required by the interface definition. It shall set the schema used for parsing. It can be overidden by the implementations in order to validate and parse he schema.

Module eoxserver.core.util.filetools

This module contains utility functions for file operations.

`eoxserver.core.util.filetools.`**findFiles**(*dir*, *pattern*)
This function mimicks the behaviour of the `find` shell command. It expects a directory path `dir` and a file name pattern `pattern` which may contain wildcards as accepted by the `fnmatch.fnmatch()`[16] function. It returns a list of paths to matching files in `dir` and its subdirectories.

If `dir` does not exist or does not point to a directory or if no matching files are found an empty list is returned.

Directories and files whose name starts with "." are omitted.

`eoxserver.core.util.filetools.`**pathToModuleName**(*path*)
This function takes a module path `path` as argument and returns the corresponding dotted name of the module.

class `eoxserver.core.util.filetools.`**TmpFile**(*suffix*, *prefix=''*)
temporary file object - `with - as` statement friendly

__str__()
Converts class to name of the temporary file.

__enter__()
Begin of `with - as` block - returns name of the temporary file.

__exit__(*type*, *value*, *traceback*)
End of `with - as` block - discards the temporary file.

Module eoxserver.core.util.geotools

Module eoxserver.core.util.kvptools

This module contains the parameter decoder implementation for key-value-pair encoded URL parameters. See also *Module eoxserver.core.util.decoders* (page 158) for general information on parameter decoders.

[16] http://docs.python.org/2.7/library/fnmatch.html#fnmatch.fnmatch

Decoding Schemas

KVP decoding schemas can be defined following the general rules for schemas defined in the *Module eoxserver.core.util.decoders* (page 158). The KVP decoder expects `kvp_key` for the location parameter and `kvp_type` for the type definition parameter. That means, KVP decoding schemas generally have the form:

```
PARAM_SCHEMA = {
    "<parameter_name>": {
        "kvp_key": "<kvp_key>",
        "kvp_type": "<type_definition>"
    },
    ...
}
```

where

- `kvp_key` designates the KVP key to be looked for,
- `type_definition` is a valid type definition as defined in *Module eoxserver.core.util.decoders* (page 158).

The valid base type names for KVP decoders are:

- `string`: the string value of the parameter is returned as is,
- `int`: the value will be typecasted to an integer; an exception is raised if the cast fails
- `float`: the value will be typecasted to a float; an exception is raised if the cast fails
- `stringlist`: the value is expected to be a comma separated list; a list of strings will be returned
- `intlist`: the value is expected to be a comma separated list of integers; a list of integers will be returned; if typecasting fails, an exception is raised
- `floatlist`: the value is expected to be a comma separated list of floats; a list of floats will be returned; if typecasting fails, an exception is raised.

Minimum and maximum occurrences can be defined as described for the *Module eoxserver.core.util.decoders* (page 158) and will be validated.

Classes

class `eoxserver.core.util.kvptools.`**`KVPDecoder`**

This class provides a parameter decoder for key-value-pair parameters.

`KVPDecoder`(*params=None*, *schema=None*)

The constructor accepts two optional arguments:

- `params` is expected to be either an URL-encoded string or a `django.http.QueryDict`[17] instance containing request parameter information
- `schema` is expected to be a schema as described under *Decoding Schemas* (page 162) above.

`setParams`(*params*)

This method accepts one mandatory parameter `params` which is expected to be either an URL-encoded string or a `django.http.QueryDict`[18] instance containing request parameter information.

The input `params` is converted to a canonical format internally. This information can be retrieved using `getParams()` (page 163).

`setSchema`(*schema*)

This method accepts a KVP decoding schema as described under *Decoding Schemas* (page 162) above and sets the internal schema to this value. Note that the schema is validated only when `getValue()`

[17] https://docs.djangoproject.com/en/1.4/ref/request-response/#django.http.QueryDict
[18] https://docs.djangoproject.com/en/1.4/ref/request-response/#django.http.QueryDict

(page 163) or `getValueStrict()` (page 163) are called. Invalid schemas will cause exceptions then.

getValue(*expr*, *default=None*)
This method accepts an expression *expr* and a default value *default* as input.

If no schema has been defined, `expr` will be interpreted as being the key of a key-value-pair. The string value of the last occurrence of the key will be returned; if the value is missing `default` is returned.

If a schema has been defined, `expr` will be looked up in the schema, and the according value will be returned. If it is not found, `default` will be returned.

This method raises a `KVPKeyOccurrenceError` (page 133) if the minimum or maximum occurrence bounds for the given KVP key are violated. In case the raw value of the KVP could not be casted to the expected type `KVPTypeError` (page 133) is raised. In case `expr` is not defined in the schema or an error in the schema definition is detected, `InternalError` (page 133) is raised.

getValueStrict(*expr*)
This method accepts an expression `expr` as input.

If no schema has been defined, `expr` will be interpreted as being the key of a key-value-pair. The string value of the last occurrence of the key will be returned; if the value is missing `KVPKeyNotFound` (page 133) will be raised.

If a schema has been defined, `expr` will be looked up in the schema, and the according value will be returned. If it is not found, `KVPKeyNotFound` (page 133) will be raised.

This method raises a `KVPKeyOccurrenceError` (page 133) if the minimum or maximum occurrence bounds for the given KVP key are violated. In case the raw value of the KVP could not be casted to the expected type `KVPTypeError` (page 133) is raised. In case `expr` is not defined in the schema or an error in the schema definition is detected, `InternalError` (page 133) is raised.

getParams()
Returns a dictionary of params. The keys of the dictionary correspond to the KVP keys provided, the values are lists of KVP values (this is to account for multiple definitions for the same KVP key).

getParamType()
Returns `"kvp"`.

Module eoxserver.core.util.multiparttools

This module contains implementation of MIME multipart packing and unpacking utilities.

The main benefit of the utilities over other methods of mutipart handling is that the functions of this module do not manipulate the input data buffers and especially avoid any unnecessary data copying.

eoxserver.core.util.multiparttools.**mpPack**(*parts*, *boundary*)
Low-level memory-friendly MIME multipart packing.

Note: The data payload is passed untouched and no transport encoding of the payload is performed.

Inputs:

- **parts - list of part-tuples, each tuple shall have two elements** the header list and (string) payload. The header itsels should be a sequence of key-value pairs (tuples).
- boundary - boundary string

Ouput:

- list of strings (which can be directly passsed as a Django response content)

eoxserver.core.util.multiparttools.**mpUnpack**(*cbuffer*, *boundary*, *capitalize=False*)
Low-level memory-friendly MIME multipart unpacking.

Note: The payload of the multipart package data is neither modified nor copied. No decoding of the transport encoded payload is performed.

Note: The subroutine does not unpack any nested mutipart content.

Inputs:

- `cbuffer` - character buffer (string) containing the the header list and (string) payload. The header itsels should be a sequence of key-value pairs (tuples).
- `boundary` - boundary string
- `capitalize` - by default the header keys are converted to lower-case (e.g., 'content-type'). To capitalize the names (e.g., 'Content-Type') set this option to true.

Output:

- list of parts - each part is a tuple of the header dictionary, payload `cbuffer` offset and payload size.

Utilities

`eoxserver.core.util.multiparttools.`**`getMimeType`**(*content_type*)
Extract MIME-type from Content-Type string and convert it to lower-case.

`eoxserver.core.util.multiparttools.`**`getMultipartBoundary`**(*content_type*)
Extract boundary string from mutipart Content-Type string.

`eoxserver.core.util.multiparttools.`**`capitalize`**(*header_name*)
Capitalize header field name. Eg., 'content-type' is capilalized to 'Content-Type'.

Module eoxserver.core.util.timetools

`eoxserver.core.util.timetools.`**`getDateTime`**(*s*)

`eoxserver.core.util.timetools.`**`isotime`**(*dt*)

class `eoxserver.core.util.timetools.`**`UTCOffsetTimeZoneInfo`**

Module eoxserver.core.util.xmltools

This module contains utils for XML encoding, decoding and printing.

XML Decoding Schemas

XML decoding schemas can be defined following the general rules for schemas defined in the *Module eoxserver.core.util.decoders* (page 158). The XML decoder expects `xml_location` for the location parameter and `xml_type` for the type definition parameter. That means, XML decoding schemas generally have the form:

```
PARAM_SCHEMA = {
    "<parameter_name>": {
        "xml_location": "<xpath_expr>",
        "xml_type": "<type_definition>"
    },
    ...
}
```

where

- `xpath_expr` is an XPath expression which designates the element or attribute to be evaluated,
- `type_definition` is a valid type definition as defined in *Module eoxserver.core.util.decoders* (page 158).

EOxServer only supports a small subset of XPath expressions, see the class documentation of `XPath` (page 166) below. Relative XPath expressions are interpreted to be refer to the document root element. The valid base type names for XML decoders are:

- `string`: the string value of the parameter is returned as is,
- `int`: the value will be typecasted to an integer; an exception is raised if the cast fails
- `float`: the value will be typecasted to a float; an exception is raised if the cast fails
- `intlist`: the value is expected to be a list of integers separated by whitespace; a list of integers will be returned; if typecasting fails, an exception is raised
- `floatlist`: the value is expected to be a list of floats separated by whitespace; a list of floats will be returned; if typecasting fails, an exception is raised.
- `tagName`: return the tag name of the designated element
- `localName`: return the local name of the designated element
- `element`: return the designated DOM Element
- `attr`: return the designated DOM Attribute mode
- `dict`: return a dictionary of values; the values will be retrieved according to a subschema given by the entry `xml_dict_elements`; the subschema follows the same rules as any decoding schema with the exception that relative XPath expressions are rooted at the element designated by `xml_location` instead of the document root element

Minimum and maximum occurrences can be defined as described for the *Module eoxserver.core.util.decoders* (page 158) and will be validated.

XML Decoder

class `eoxserver.core.util.xmltools.`**`XMLDecoder`**

This class provides XML Decoding facilities.

`XMLDecoder`(*params=None*, *schema=None*)

The constructor accepts two optional arguments:

- `params` is expected to be either a string containing well-formed XML;
- `schema` is expected to be a schema as described under *XML Decoding Schemas* (page 164) above.

`setParams`(*params*)

This method accepts one mandatory parameter `params` which is expected to be a string containing well-formed XML.

The input `params` is parsed into a DOM structure using Python's `xml.dom.minidom`[19] module.

`setSchema`(*schema*)

This method accepts an XML decoding schema as described under *XML Decoding Schemas* (page 164) above and sets the internal schema to this value.

Internally, the schema is parsed into a node structure. `InternalError` (page 133) is raised in case the schema does not validate.

`getValue`(*expr*, *default=None*)

This method accepts an expression *expr* and a default value *default* as input.

If no schema has been defined, `expr` will be interpreted as an XPath expression. The string value of the element text is returned; if the value is missing `default` is returned.

If a schema has been defined, `expr` will be looked up in the schema, and the according value will be returned. If it is not found, `default` will be returned.

[19] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom

This method raises a XMLNodeOccurrenceError (page 134) if the minimum or maximum occurrence bounds for the given XML element are violated. In case the text value of the XML element or attribute could not be casted to the expected type XMLTypeError (page 134) is raised. In case expr is not defined in the schema InternalError (page 133) is raised.

getValueStrict(*expr*)
This method accepts an expression expr as input.

If no schema has been defined, expr will be interpreted as an XPath expression. The string value of the element text is returned; if the value is missing XMLNodeNotFound (page 134) will be raised.

If a schema has been defined, expr will be looked up in the schema, and the according value will be returned. If it is not found, XMLNodeNotFound (page 134) will be raised.

This method raises a XMLNodeOccurrenceError (page 134) if the minimum or maximum occurrence bounds for the given XML element are violated. In case the text value of the XML element or attribute could not be casted to the expected type XMLTypeError (page 134) is raised. In case expr is not defined in the schema InternalError (page 133) is raised.

getParams()
Returns the input XML.

getParamType()
Returns "xml".

XML Decoding Utilities

class eoxserver.core.util.xmltools.**XPath**(*init_data*)
This class represents an XPath expression. It provides methods for decoding an encoding XPath expressions as well as looking up the specified nodes in an XML structure.

The constructor accepts either an XPath expression or a list of locators as generated by XPathExprToList() (page 166) as input.

Note that this implementation supports only a small subset of XPath expressions, namely parts of the abbreviated notation.

```
xpath_expr          ::=   "/" | ["/"] locator_list
locator_list        ::=   (element_locator "/")* node_locator
node_locator        ::=   element_locator | attribute_locator
element_locator     ::=   locator | "*"
attribute_locator   ::=   "@" locator | "@*"
locator             ::=   prefix ":"  localname | ["{" namespaceuri "}"] localnam
prefix              ::=   NCName
localname           ::=   NCName
namespaceuri        ::=   URI | "*"
```

classmethod XPathExprToList(*xpath_expr*)
This method converts an XPath expression to a list of locators.

append(*other*)
Append another XPath expression and return the result.

getNodeType()
Returns "element" if the XPath expression points to an element, or "attribute" if it points to an attribute.

getNodes(*context_element*)
Return all the nodes designated by the XPath expression.

isAbsolute()
Returns True if the XPath expression is absolute, False otherwise.

classmethod **listToXPathExpr**(*xpath*)
This method converts a list of locators to an XPath expression.

classmethod **reverse**(*node*)
Generate an absolute XPath expression for the given node from the DOM.

XML Encoder

class eoxserver.core.util.xmltools.**XMLEncoder**(*schemas=None*)
This is the base class for XML encoders. It is intended to be subclassed by concrete encoder implementations which can use its utility methods to compose XML documents.

_initializeNamespaces()
This method must be overridden by descendants in order to initialize the namespace dictionary of the object. The dictionary keys are interpreted as namespace prefixes whereas the values contain the namespace URIs.

The return value is the namespace dictionary.

_makeElement(*prefix*, *tag_name*, *content*)
This method creates elements. It expects three arguments as input:

- the namespace prefix of the element; this can be the empty string for the default namespace or unqualified names.
- the tag name of the element
- the content of the element

If the content is

- a DOM Element; it will be appended to the newly created element's child nodes;
- a list of node definitions; these nodes will be created and then appended to the newly created element
- some other argument, it will be converted to a string and be appended to the element as text value.

Node definition lists contain tuples that describe the elements or attributes to be created and/or to be appended to the parent element. 3-tuples of `(prefix, tag_name, content)` will be interpreted in the same way as the input parameters.

If the `prefix` or `tag_name` parameters start with a `@` an attribute will be created and appended to the parent element. If the `prefix` or `tag_name` parameters contain `@@` a text node will be created.

The `content` parameter can contain a node definition list as well.

Alternatively, 1-tuples containing a DOM Element can be specified. The DOM Element will be appended to the respective parent element.

Functions

eoxserver.core.util.xmltools.**DOMElementToXML**(*element*, *nsmap=None*)
This function takes a DOM element as input and returns an XML document with the input element as root encoded as ISO-8859-1 string. This function is namespace aware.

The optional `nsmap` parameter may contain a dictionary of XML prefixes and namespace URIs; these namespace definitions will be appended to the document root elements list of xmlns attributes. In case it is missing, the namespaces used throughout the document are automatically determined and the corresponding xmlns attributes will be created.

eoxserver.core.util.xmltools.**DOMtoXML**(*xmldom*, *nsmap=None*)
Takes a DOM document as input and returns the corresponding XML (no pretty printing) encoded as ISO-8859-1 string. This function is namespace aware.

The optional `nsmap` parameter may contain a dictionary of XML prefixes and namespace URIs; these namespace definitions will be appended to the document root elements list of xmlns attributes. In case it is missing, the namespaces used throughout the document are automatically determined and the corresponding xmlns attributes will be created.

2.12.3 Service Layer

Module eoxserver.services.base

class `eoxserver.services.base.`**BaseExceptionHandler**(*schemas=None*)
This is the basic handler for exceptions. It allows to generate exception reports.

handleException(*req*, *exception*)
This method can be invoked in order to handle an exception and produce an exception report. It starts by logging the error to the default log. Then the appropriate XML encoder is fetched (the `_getEncoder()` method has to be overridden by the subclass). Finally, the exception report itself is encoded and the appropriate HTTP status code determined. The method returns a `Response` (page 174) object containing this information.

class `eoxserver.services.base.`**BaseRequestHandler**
Base class for all EOxServer Handler Implementations.

There are two private methods that have to be overridden by child classes.

_handleException(*req*, *exception*)
Abstract method which must be overridden by child classes to provide specific exception reports. If the exception report cannot be generated in this specific handler, the exception should be re-raised. This is also the default behaviour.

The method expects an `OWSRequest` (page 174) object `req` and the exception that has been raised as input. It should return a `Response` (page 174) object containing the exception report.

_processRequest(*req*)
Abstract method which must be overridden to provide the specific request handling logic. Should not be invoked from external code, use the `handle()` (page 168) method instead. It expects an `OWSRequest` (page 174) object `req` as input and should return a `Response` (page 174) object containing the response to the request. The default method does not do anything.

handle(*req*)
Basic request handling method which should be invoked from external code. This method invokes the `_processRequest()` (page 168) method and returns the resulting `Response` (page 174) object unless an exception is raised. In the latter case `_handleException()` (page 168) is called and the appropriate response is returned.

Module eoxserver.services.connectors

Connectors are used to configure the data sources for MapServer requests. Because of the different nature of the data sources (files, tile indices, rasdaman databases) they have to be set up differently. Connectors allow to do this transparently.

class `eoxserver.services.connectors.`**FileConnector**
The `FileConnector` (page 168) class is the most common connector. It configures a file as data source for the MapServer request.

configure(*layer*, *eo_object*, *filter_exprs=None*)
This method takes three arguments: `layer` is a MapServer `layerObj` instance, `eo_object` a EO-WCS object (either a `RectifiedDatasetWrapper` (page 219) or `RectifiedStitchedMosaicWrapper` (page 224) instance) and the optional `filter_exprs` argument is currently not used.

The method configures the MapServer layer by setting its `data` property to the path. It invokes the `prepareData()` method of the `DataPackageWrapper` (page 190) instance related to the object.

The method also sets the projection on the layer.

class `eoxserver.services.connectors.`**`RasdamanArrayConnector`**

The `RasdamanArrayConnector` (page 169) class is intended for `RectifiedDatasetWrapper` (page 219) instances that store their data in rasdaman arrays.

`configure`(*layer*, *eo_object*, *filter_exprs=None*)

This method takes three arguments: `layer` is a MapServer `layerObj` instance, `eo_object` a `RectifiedDatasetWrapper` (page 219) instance and the optional `filter_exprs` argument is currently not used.

The method sets the `data` property of the MapServer layer to the connection string to the rasdaman database array, see the GDAL rasdaman format[20] page for details.

Furthermore, the projection settings on the layer are configured according to the metadata in the EOxServer database. As the rasdaman arrays have pixel coordinates only, the parameters for conversion from pixel coordinates to the spatial reference system have to be set explicitly.

class `eoxserver.services.connectors.`**`TiledPackageConnector`**

The `TiledPackageConnector` (page 169) class is intended for `RectifiedStitchedMosaicWrapper` (page 224) instances that store their data in tile indices.

`configure`(*layer*, *eo_object*, *filter_exprs=None*)

This method takes three arguments: `layer` is a MapServer `layerObj` instance, `eo_object` a `RectifiedStitchedMosaicWrapper` (page 224) instance and the optional `filter_exprs` argument is currently not used.

The method sets the `tileindex` property of the MapServer layer to point to the shape file where the paths of the tiles are stored.

The method also sets the projection on the layer.

Module eoxserver.services.exceptions

exception `eoxserver.services.exceptions.`**`InvalidAxisLabelException`**(*msg*)

This exception indicates that an invalid axis name was chosen in a WCS 2.0 subsetting parameter.

exception `eoxserver.services.exceptions.`**`InvalidRequestException`**(*msg*, *error_code*, *locator*)

This exception indicates that the request was invalid and an exception report shall be returned to the client.

The constructor takes three arguments, namely `msg`, the error message, `error_code`, the error code, and `locator`, which is needed in OWS exception reports for indicating which part of the request produced the error.

How exactly the exception reports are constructed is not defined by the exception, but by exception handlers.

exception `eoxserver.services.exceptions.`**`InvalidSubsettingException`**(*msg*)

This exception indicates an invalid WCS 2.0 subsetting parameter was submitted.

exception `eoxserver.services.exceptions.`**`VersionNegotiationException`**(*msg*)

This exception indicates that version negotiation fails. Such errors can happen with OWS 2.0 compliant "new-style" version negotation.

Module eoxserver.services.interfaces

This module defines interfaces for service request handlers.

[20] http://http://www.gdal.org/frmt_rasdaman.html

EOxServer follows a cascaded approach for handling OWS requests: First, a service handler takes in all requests for a specific service, e.g. WMS or WCS. Second, the request gets passed on to the appropriate version handler. Last, the actual operation handler for that request is invoked.

This cascaded approach shall ensure that features that relate to every operation of a service or service version (most importantly exception handling) can be implemented centrally.

class `eoxserver.services.interfaces.`**`ExceptionEncoderInterface`**
This interface is intended for encoding OWS exception reports.

`encodeInvalidRequestException`(*exception*)
This method shall return an exception report for an `InvalidRequestException`.

`encodeVersionNegotiationException`(*exception*)
This method shall return an exception report for a `VersionNegotiationException`.

`encodeException`(*exception*)
This method shall return an exception report for any kind of exception.

class `eoxserver.services.interfaces.`**`ExceptionHandlerInterface`**
This interface is intended for exception handlers. These handlers shall be invoked when an exception is raised during the processing of an OWS request.

`handleException`(*req*, *exception*)
This method shall handle an exception. It expects the original `OWSRequest` (page 174) object `req` as well as the exception object as input. The expected output is a `Response` (page 174) object which shall contain an exception report and whose content will be sent to the client.

In case the exception handler does not recognize a given type of exception or cannot produce an appropriate exception report, the exception shall be re-raised.

class `eoxserver.services.interfaces.`**`OperationHandlerInterface`**
This interface inherits from `RequestHandlerInterface` (page 170). It adds no methods, but the registry keys `services.interfaces.service`, `services.interface.version` and `services.interfaces.operation` which allow to bind to an implementation given the name of the service, the version descriptor and the operation name.

class `eoxserver.services.interfaces.`**`RequestHandlerInterface`**
This is the basic interface for OWS request handling. It is the parent class of the other handler interfaces. The binding method is KVP. The interface does not define any keys though, which is done by the child classes.

`handle`(*req*)
This method shall be called for handling the request. It expects an `OWSRequest` (page 174) object as input `req` and shall return a `Response` (page 174) object.

class `eoxserver.services.interfaces.`**`ServiceHandlerInterface`**
This interface inherits from `RequestHandlerInterface` (page 170). It adds no methods, but a registry key `services.interfaces.service` which allows to bind to an implementation given the name of the service.

class `eoxserver.services.interfaces.`**`VersionHandlerInterface`**
This interface inherits from `RequestHandlerInterface` (page 170). It adds no methods, but the registry keys `services.interfaces.service` and `services.interface.version` which allow to bind to an implementation given the name of the service and the version descriptor.

Module eoxserver.services.mapserver

This module contains the abstract base classes for request handling.

class `eoxserver.services.mapserver.`**`MapServerDataConnectorInterface`**
This interface is intended for objects that configure the input data for a MapServer layer. The basic rationale for this is that there are at least three different types of data sources that need differentt treatment:

- data stored in single plain files

- tiled data with references in a tile index (a shape file)
- rasdaman arrays

Others might be added in the course of development of EOxServer.

congigure(*layer*, *eo_object*)
This method takes a `mapscript.layerObj` object and an `eo_object` as input and configures the MapServer layer according to the type of data package used by the `eo_object` (RectifiedDataset, ReferenceableDataset or RectifiedStitchedMosaic).

class `eoxserver.services.mapserver.`**MapServerLayerGeneratorInterface**
This interface is not in use.

class `eoxserver.services.mapserver.`**MapServerOperationHandler**
MapServerOperationHandler serves as parent class for all operations involving calls to MapServer. It is not an abstract class, but implements the most basic functionality, i.e. simply passing on a request to MapServer as it comes in.

This class implements a workflow for request handling that involves calls to MapServer using its Python binding (mapscript). Requests are processed in six steps:

- retrieve coverage information (`createCoverages()` method)
- configure a MapServer `OWSRequest` object with parameters from the request (`configureRequest()` (page 171) method)
- configure a MapServer `mapObj` object with parameters from the request and the config (`configureMapObj()` method)
- add layers to the MapServer `mapObj` (`addLayers()` method)
- dispatch the request, i.e. let MapServer actually do its work; return the result (`dispatch()` (page 171) method)
- postprocess the MapServer response (`postprocess()` method)

Child classes may override the configureRequest, configureMap, postprocess and postprocessFault methods in order to customize functionality. If possible, the handle and dispatch methods should not be overridden.

configureRequest()
This method configures the `ms_req.ows_req` object (an instance of `mapscript.OWSRequest`) with the parameters passed in with the user request. This method can be overridden in order to change the treatment of parameters.

dispatch()
This method actually executes the MapServer request by calling `ms_req.map.OWSDispatch()`. This method should not be overridden by child classes.

class `eoxserver.services.mapserver.`**MapServerRequest**(*req*)
This class inherits from `OWSRequest` (page 174).

The constructor expects a single argument `req` which is expected to contain an `OWSRequest` (page 174) instance. The parameters and decoder will be taken from that instance.

`MapServerRequest` (page 171) objects add two properties: first a `map` property which contains a `mapscript.mapObj`, and second an `ows_req` property which contains a `mapscript.OWSRequest` object. These properties are not configured at the beginning.

class `eoxserver.services.mapserver.`**MapServerResponse**(*ms_response*, *ms_content_type*, *ms_status*, *headers={}*, *status=None*)
This class inherits from `Response` (page 174). It adds methods to handle with response data obtained from MapServer, including methods for multipart messages.

The constructor takes several arguments. In `ms_response`, the response buffer as returned by MapServer is expected. The `ms_content_type` argument shall be set to the MIME type of the response content. `ms_status` shall contain the MapServer status as returned by the call to

mapscript.mapObj.OWSDispatch(). headers and status are optional and have the same meaning as in Response (page 174).

getContentType()
Returns the content type of the response.

getProcessedResponse(*response_xml*, *headers_xml=None*, *boundary='wcs'*, *subtype='mixed'*)
This method returns a Response (page 174) object that contains the coverage data generated by the original MapServer call and the XML data contained in the response_xml argument.

The headers_xml parameter may contain a dictionary of headers to be tagged on the XML part of the multipart response. The boundary argument shall contain the boundary string used for delimiting the different parts of the message and defaults to wcs. The subtype argument relates to the second part of the MIME type statement and defaults to mixed for a complete MIME type of multipart/mixed.

getStatus()
Returns the HTTP status code of the response.

splitResponse()
This method splits a multipart response into its different parts.

The XML part is stored in the ms_response_xml property of the object. The coverage data is stored in the ms_response_data property of the object. The headers of the parts are stored in the ms_response_xml_headers and ms_response_data_headers properties respectively.

eoxserver.services.mapserver.**gdalconst_to_imagemode**(*const*)
This function translates a GDAL data type constant as defined in the gdalconst module to a MapScript image mode constant.

eoxserver.services.mapserver.**gdalconst_to_imagemode_string**(*const*)
This function translates a GDAL data type constant as defined in the gdalconst module to a string as used in the MapServer map file to denote an image mode.

Module eoxserver.services.ogc

This module contains old style exception handlers that use the OGC namespace for exception reports (prior to OWS Common).

class eoxserver.services.ogc.**OGCExceptionEncoder**(*schemas=None*)
Encoder class for OGC namespace exception reports.

class eoxserver.services.ogc.**OGCExceptionHandler**(*schemas=None*)
Handler class for the OGC namespace.

Module eoxserver.services.owscommon

class eoxserver.services.owscommon.**OWSCommon11ExceptionEncoder**(*schemas=None*, *version=None*)
Encoder for OWS Common 1.1 compliant exception reports. Implements ExceptionEncoderInterface (page 170).

class eoxserver.services.owscommon.**OWSCommon11ExceptionHandler**(*schemas*, *version*)
This exception handler is intended for OWS Common 1.1 based exception reports. Said standard defines a framework for exception reports that can be extended by individual OWS standards with additional error codes, for instance.

This class inherits from BaseExceptionHandler (page 168).

class eoxserver.services.owscommon.**OWSCommonConfigReader**
This class implements the ConfigReaderInterface (page 142). It provides convenience functions for reading OWS related settings from the instance configuration.

getHTTPServiceURL()
Returns the value of the *http_service_url'* in the `services.owscommon` section. This is used for reporting the correct service address in the OWS capabilities.

validate(*config*)
Raises `ConfigError` (page 133) if the mandatory `http_service_url` setting is missing in the `services.owscommon` section of the instance configuration.

class `eoxserver.services.owscommon.`**OWSCommonExceptionEncoder**(*schemas=None*)
Encoder for OWS Common 2.0 compliant exception reports. Implements `ExceptionEncoderInterface` (page 170).

class `eoxserver.services.owscommon.`**OWSCommonExceptionHandler**(**args*)
This exception handler is intended for OWS Common 2.0 based exception reports. Said standard defines a framework for exception reports that can be extended by individual OWS standards with additional error codes, for instance.

This class inherits from `BaseExceptionHandler` (page 168).

setHTTPStatusCodes(*additional_http_status_codes*)
In OWS Common 2.0 the HTTP status codes for exception reports can differ depending on the error code. There are several exceptions listed in the standard itself, but more can be added by OWS standards relying on OWS Common 2.0.

This method allows to configure the exception handler with a dictionary of additional codes. The dictionary keys shall contain the OWS error codes and the values the corresponding HTTP status codes as integers.

class `eoxserver.services.owscommon.`**OWSCommonHandler**
This class is the entry point for all incoming OWS requests.

It tries to determine the service the request and directed to and invokes the appropriate service handler. An `InvalidRequestException` (page 169) is raised if the service is unknown.

Due to a quirk in WMS where the service parameter is not mandatory, the WMS service handler is called in the absence of an explicit service parameter.

class `eoxserver.services.owscommon.`**OWSCommonServiceHandler**
This is the base class for OWS service handlers. It inherits from `BaseRequestHandler` (page 168).

This handler parses the OWS request parameters for a service version. The version parameter is mandatory for all OGC Web Services and operations except for the respective "GetCapabilities" calls. So, if the request is found to be "GetCapabilities" the version negotiation routines are started in order to determine the actual OWS version handler to be called. Otherwise the version parameter is read from the request or an `InvalidRequestException` (page 169) is raised if it is absent or relates to an unknown or disabled version of the service.

Version negotiation is implemented along the lines of OWS Common 2.0. This means, the handler checks for the presence of an AcceptVersions parameter. If it is present new-style version negotiation is triggered and old-style version negotiation otherwise.

New-style version negotiation will take the first version defined in the AcceptVersion parameter that is implemented and raise an exception if none of the versions is known. The version parameter is always ignored.

Old-style version negotiation will look for the version parameter and chose the version indicated if it is implemented. If the version parameter is lacking the highest implemented version of the service will be selected. If the version parameter is present but refers to a version that is not implemented, the highest version lower than that is selected. If that fails, too, the lowest implemented version will be selected.

Note that OWS Common 2.0 refers to old-style version negotiation as deprecated and includes it only for backwards compatibility. But for EOxServer which exhibits OWS versions relying on OWS Common as well as versions prior to it, the fallback to old-style version negotiation is always required. Binding to older versions would otherwise not be possible.

class eoxserver.services.owscommon.**OWSCommonVersionHandler**
This is the base class for OWS version handlers. It inherits from BaseRequestHandler (page 168).

Based on the value of the request parameter, the appropriate operation handler is chosen and invoked. An InvalidRequestException (page 169) is raised if the operation name is unknown or disabled.

This class implements exception handling behaviour which is common across the operations of each OWS version but not among different versions of the same service.

Module eoxserver.services.requests

This module defines basic classes for OWS requests and responses to OWS requests.

class eoxserver.services.requests.**OWSRequest**(*http_req*, *params=''*, *param_type='kvp'*, *decoder=None*)
This class is used to encapsulate information about an OWS request.

The constructor expects one required parameter, a Django HttpRequest[21] object http_req.

The params argument shall contain the parameters sent with the request. For GET requests, this can either contain a Django QueryDict[22] object or the query string itself. For POST requests, the argument shall contain the message body as a string.

The param_type argument shall be set to kvp for GET requests and xml for POST requests.

Optionally, a decoder (either a KVPDecoder (page 162) or XMLDecoder (page 165) instance initialized with the parameters) can already be conveyed to the request. If it is not present, the appropriate decoder type will be chosen and initialized based on the values of params and param_type.

getHeader(*header_name*)
Returns the value of the HTTP header header_name, or None if not found.

getParamType()
Returns kvp or xml.

getParamValue(*key*, *default=None*)
Returns the value of a parameter named key. The name relates to the schema set for the decoder. You can provide a default value which will be returned if the parameter is not present.

getParamValueStrict(*key*)
Returns the value of a parameter named key. The name relates to the schema set for the decoder. A DecoderException (page 133) will be raised if the parameter is not present.

getParams()
Returns the parameters. This method calls the KVPDecoder (page 162) or XMLDecoder (page 165) method of the same name. In case of KVP data, this means that a dictionary with the parameter values will be returned instead of the query string, even if the OWSRequest (page 174) object was initially configured with the query string.

getVersion()
Returns the version for the OGC Web Service. This method is used for version negotiation, in which case the appropriate version cannot simply be read from the request parameters.

setSchema(*schema*)
Set the decoding schema for the parameter decoder (see eoxserver.core.util.decoders (page 158))

setVersion(*version*)
Sets the version for the OGC Web Service. This method is used for version negotiation, in which case the appropriate version cannot simply be read from the request parameters.

class eoxserver.services.requests.**Response**(*content=''*, *content_type='text/xml'*, *headers={}*, *status=None*)
This class encapsulates the data needed for an HTTP response to an OWS request.

[21] https://docs.djangoproject.com/en/1.4/ref/request-response/#django.http.HttpRequest
[22] https://docs.djangoproject.com/en/1.4/ref/request-response/#django.http.QueryDict

The `content` argument contains the content of the response message. The `content_type` argument is set to the MIME type of the response content. The `headers` argument is expected to be a dictionary of additional HTTP headers to be sent with the response. The `status` parameter is used to set the HTTP status of the response.

Module eoxserver.services.views

This model contains Django views for the EOxServer software. Its main function is ows() which handles all incoming OWS requests

`eoxserver.services.views.`**`ows`**(*request*)

This function handles all incoming OWS requests.

It prepares the system by a call to `eoxserver.core.system.System.init()` (page 157) and generates an `OWSRequest` (page 174) object containing the request parameters and passes the handling on to an instance of `OWSCommonHandler` (page 173).

If security handling is enabled, the Policy Decision Point (PDP) is called first in order to determine if the request is authorized. Otherwise the response of the PDP is sent back to the client. See also `eoxserver.services.auth.base` (page 184).

Module eoxserver.services.ows.wcs.wcs20.desccov

This module contains the handler for WCS 2.0 / EO-WCS DescribeCoverage requests.

class `eoxserver.services.ows.wcs.wcs20.desccov.`**`WCS20DescribeCoverageHandler`**

This handler generates responses to WCS 2.0 / EO-WCS DescribeCoverage requests. It inherits directly from `BaseRequestHandler` (page 168) and does NOT reuse MapServer.

The workflow implemented by the handler starts with the `createCoverages()` (page 175) method and generates the coverage descriptions using the `WCS20EOAPEncoder` (page 181) method `encodeCoverageDescriptions()`.

`createCoverages`(*req*)

This method retrieves the coverage metadata for the coverages denoted by the coverageid parameter of the request. It raises an `InvalidRequestException` (page 169) if the coverageid parameter is missing or if it contains an unknown coverage ID.

Module eoxserver.services.ows.wcs.wcs20.desceo

This method provides a handler for EO-WCS DescribeEOCoverageSet operations.

class `eoxserver.services.ows.wcs.wcs20.desceo.`**`WCS20DescribeEOCoverageSetHandler`**

This handler generates responses to EO-WCS DescribeEOCoverageSet requests. It derives directly from `BaseRequestHandler` (page 168) and does not reuse MapServer (as MapServer does not support EO-WCS).

The implented workflow begins with a call to `createWCSEOObjects()` (page 175) and then goes on to encode the EO coverage and Dataset Series metadata.

The handler is aware of the count and sections parameters of DescribeEOCoverageSet which allow to limit the number of coverage and Dataset Series descriptions returned and the sections (CoverageDescriptions, DatasetSeriesDescriptions, All) included in the requests.

An `InvalidRequestException` (page 169) will be raised if incorrect parameters are encountered or the mandatory eoid parameter is missing.

`createWCSEOObjects`(*req*)

This method returns a tuple `(dataset_series_set, coverages)` of two lists containing Dataset Series or EO Coverage objects respectively. It parses the request parameters in `req` in order to determine the subset of EO-WCS objects to be included.

The method makes use of `getFilterExpressions()` in order to parse subset expressions sent with the request and to obtain filter expressions that restrict the subset of EO-WCS objects to be included.

The method will raise `InvalidRequestException` (page 169) if parameters are missing, subset expressions are invalid or if the eoid parameter contains unknown names.

Module eoxserver.services.ows.wcs.wcs20.getcap

This module provides handlers for WCS 2.0 / EO-WCS GetCapabilities requests.

class `eoxserver.services.ows.wcs.wcs20.getcap.`**WCS20GetCapabilitiesHandler**

This is the handler for WCS 2.0 / EO-WCS GetCapabilities requests. It inherits from `WCSCommonHandler` (page 178).

As for all handlers, the entry point is the `handle()` method. The handler then performs a workflow that is described in the `WCSCommonHandler` (page 178) documentation.

This handler follows this workflow with adaptations to the `createCoverages()` (page 176), `configureMapObj()` (page 176) and `postprocess()` (page 176) methods. The latter one modifies the GetCapabilities response obtained by MapServer to contain EO-WCS specific extensions.

configureMapObj()

This method extends the `configureMapObj` method to include informations on the available output formats as well as the supported CRSes.

createCoverages()

This method adds all Rectified Datasets and Rectified Stitched Mosaics to the `coverages` property of the handler. For each of these coverages, a layer will be added to the MapScript `mapObj`.

getMapServerLayer(*coverage*)

This method returns a MapScript `layerObj` for the input `coverage`. It extends the `getMapServerLayer` function by configuring the input data using the appropriate connectors (see `eoxserver.services.connectors` (page 168)).

postprocess(*resp*)

This method transforms the standard WCS 2.0 response `resp` obtained from MapServer into an EO-WCS compliant GetCapabilities response and returns the corresponding `Response` (page 174) object.

Specifically,

- the xsi:schemaLocation attribute of the document root is set to the EO-WCS schema URL
- the extensions supported by EOxServer are added to the wcs:ServiceMetadata element
- the supported EO-WCS profiles are added to the wcs:ServiceIdentification element
- the metadata for the DescribeEOCoverageSet operation is added to the ows:OperationsMetadata element
- the wcs:Contents section is replaced by an EO-WCS compliant structure

The wcs:Contents section is configured with the coverage summaries of all visible Rectified and Referenceable Datasets, of all Rectified Stitched Mosaics and the summaries of all Dataset Series. Note that the handler is aware of the OWS Common sections parameter which allows to deselect all or parts of the wcs:Contents section and acts accordingly.

Should MapServer return an exception report in `resp`, it is passed on unchanged except for the xsi:schemaLocation attribute.

Module eoxserver.services.ows.wcs.wcs20.getcov

This module contains handlers for WCS 2.0 / EO-WCS GetCoverage requests.

class eoxserver.services.ows.wcs.wcs20.getcov.**WCS20CorrigendumGetCoverageHandler**
This handler takes care of all WCS 2.0.1 / EO-WCS GetCoverage requests. It inherits from WCS20GetCoverageHandler (page 177).

class eoxserver.services.ows.wcs.wcs20.getcov.**WCS20GetCoverageHandler**
This handler takes care of all WCS 2.0 / EO-WCS GetCoverage requests. It inherits from WCSCommonHandler (page 178).

The main processing step is to determine the coverage concerned by the request and delegate the request handling to the handlers for Referenceable Datasets or other (rectified) coverages according to the coverage type.

An InvalidRequestException (page 169) is raised if the coverage ID parameter is missing in the request or the coverage ID is unknown.

class eoxserver.services.ows.wcs.wcs20.getcov.**WCS20GetRectifiedCoverageHandler**
This is the handler for GetCoverage requests for Rectified Datasets and Rectified Stitched Mosaics. It inherits from WCSCommonHandler (page 178).

It follows the workflow of the base class and modifies the createCoverages() (page 177), configureMapObj() (page 177), getMapServerLayer() (page 177) and postprocess() (page 177) methods.

configureMapObj()
This method extends the base method (configureMapObj() (page 178)). The format configurations are added to the MapScript mapObj.

createCoverages()
This method retrieves the coverage object denoted by the request and stores it in the coverages property of the handler. The method also checks if the subset expressions (if present) match with the coverage extent.

An InvalidRequestException (page 169) is raised if the coverageid parameter is missing or the coverage ID is unknown or the subset expressions do not match with the coverage extent.

getMapServerLayer(*coverage*)
This method returns a MapServer layerObj for the corresponding coverage. It extends the base class method getMapServerLayer. The method configures the input data for the layer using the appropriate connector for the coverage (see eoxserver.services.connectors (page 168)). Furthermore, it sets WCS 2.0 specific metadata on the layer.

postprocess(*resp*)
This method overrides the no-op method of the base class. It adds EO-WCS specific metadata to the multipart messages that include an XML coverage description part. It expects a MapServerResponse (page 171) object resp as input and returns it either unchanged or a new Response (page 174) object containing the modified multipart message.

MapServer returns a WCS 2.0 coverage description, but this does not include EO-WCS specific parts like the coverage subtype (Rectified Dataset or Rectified Stitched Mosaic) and EO-WCS metadata. Therefore the description is replaced with the corresponding EO-WCS complient XML.

class eoxserver.services.ows.wcs.wcs20.getcov.**WCS20GetReferenceableCoverageHandler**
This class handles WCS 2.0 / EO-WCS GetCoverage requests for Referenceable datasets. It inherits from BaseRequestHandler (page 168). It is instantiated by WCS20GetCoverageHandler (page 177).

handle(*req*, *coverage*)
This method handles the GetCoverage request for Referenceable Datasets. It takes two parameters: the OWSRequest (page 174) object req and the ReferenceableDatasetWrapper (page 221) object coverage.

The method makes ample use of the functions in eoxserver.processing.gdal.reftools in order to transform the pixel coordinates to the underlying CRS.

It starts by decoding the (optional) subset parameters using the methods of WCS20SubsetDecoder. There are two possible meanings of the subset coordinates: absent a CRS definition, they are assumed

to be expressed in pixel coordinates (imageCRS); otherwise they are treated as coordinates in the respective CRS.

In the latter case, the subset is transformed to pixel coordinates using `eoxserver.processing.gdal.reftools.rect_from_subset()`. This results in a pixel subset that contains the whole area of the subset taking into account the GCP information. See the function docs for details.

The next step is to determine the format of the response data. This is done based on the format parameter and the format configurations (see also `eoxserver.resources.coverages.formats` (page 196)). The format MIME type has to be known to the server and it has to be supported by GDAL, otherwise an `InternalError` (page 133) is raised.

For technical reasons, though, the initial dataset is not created with the output format driver, but as a virtual dataset in the memory. Only later the dataset is copied using the `CreateCopy()` method of the GDAL driver.

The method tags several metadata items on the output, most importantly the GCPs. Note that all GCPs of the coverage are tagged on the output dataset even if only a subset has been requested. This because all of them may have influence on the computation of the coordinate transformation in the subset even if they lie outside.

Finally, the response is composed. According to the mediatype parameter, either a multipart message containing the coverage description of the output coverage and the output coverage data or just the data is returned.

Module eoxserver.services.ows.wcs.common

This module contains handlers and functions commonly used by the different WCS version implementations.

class `eoxserver.services.ows.wcs.common.`**`WCSCommonHandler`**

This class provides the common operation handler for handling WCS operation requests using MapServer. It inherits from `MapServerOperationHandler` (page 171).

The class implements a handling chain:

- first, the request parameters are validated using `validateParams()` (page 179)
- then, the coverage(s) the request relates to are retrieved using `createCoverages()` (page 178)
- then, the `mapscript.OWSRequest` and `mapscript.mapObj` instances are configured using `configureRequest()` and `configureMapObj()` (page 178)
- then the layers are added using `addLayers()` (page 178)
- then the request is carried out by MapServer using `dispatch()`
- finally, postprocessing steps on the response retrieved from MapServer can be performed using `postprocess()` (page 178)

`addLayers``()`

This method adds layers to the `mapscript.mapObj` stored by the handler. By default it inserts a layer for every coverage. The layers are retrieved by calls to `getMapServerLayer()` (page 178).

`configureMapObj``()`

This method configures the `map` property of the handler (an instance of `mapscript.mapObj`) with parameters from the config. This method can be overridden in order to implement more sophisticated behaviour.

`createCoverages``()`

This method creates coverages, i.e. it adds coverage objects to the `coverages` list of the handler. It has to be overridden by child classes.

`getMapServerLayer``(`*coverage*`)`

This method creates and returns a `mapscript.layerObj` instance and configures it according to the metadata stored in the `coverage` object.

postprocess(*resp*)
This method postprocesses a MapServerResponse (page 171) object resp. By default the response is returned unchanged. The method can be overridden by child classes.

validateParams()
This method is intended to validate the parameters. It has to be overridden by child classes.

eoxserver.services.ows.wcs.common.**getMSOutputFormatsAll**(*coverage=None*)
Setup all the supported MapServer output formats. When the coverage parameter is provided than the range type is used to setup format's image mode.

eoxserver.services.ows.wcs.common.**getMSWCS10FormatMD**()
get the space separated list of supported formats to be passed to MapScript setMedata("wcs_formats",...) method

eoxserver.services.ows.wcs.common.**getMSWCSFormatMD**()
get the space separated list of supported formats to be passed to MapScript setMedata("wcs_formats",...) method

eoxserver.services.ows.wcs.common.**getMSWCSSRSMD**()
get the space separated list of CRS EPSG codes to be passed to MapScript setMedata("wcs_srs",...) method

eoxserver.services.ows.wcs.common.**parse_format_param**(*format_param*)
This utility function is used to parse a MIME type expression format_param into its parts. It returns a tuple (mime_type, format_options) which contains the mime type as a string as well as a list of format options. The input is expected as a MIME type like string of the form:

```
<type>/<subtype>[;<format_option_key>=<format_option_value>[;...]]
```

This is used for an EOxServer specific extension of the WCS format parameter which allows to tag additional format creation options such as compression and others to format expressions, e.g.:

```
image/tiff;compression=LZW
```

Module eoxserver.services.ows.wcs.encoders

This module contains XML encoders for WCS metadata based on GML, EO O&M, GMLCOV, WCS 2.0 and EO-WCS.

class eoxserver.services.ows.wcs.encoders.**CoverageGML10Encoder**(*schemas=None*)
This encoder provides methods for obtaining GMLCOV 1.0 compliant XML encodings of coverage descriptions.

encodeBoundedBy(*(minx, miny, maxx, maxy), srid=4326*)
This method returns a xml.dom.minidom[23] element representing the gml:boundedBy element. It expects the extent as a 4-tuple (minx, miny, maxx, maxy). The srid parameter is optional and represents the EPSG ID of the spatial reference system as an integer; default is 4326.

encodeDomainSet(*coverage*)
This method returns a xml.dom.minidom[24] element containing the GMLCOV represenation of the domain set for rectified or referenceable coverages. The coverage argument is expected to implement EOCoverageInterface (page 200).

The domain set can be represented by either a referenceable or a rectified grid; encodeReferenceableGrid() (page 180) or encodeRectifiedGrid() (page 180) are called accordingly.

encodeNilValue(*nil_value*)
This method returns the SWE Common encoding of a nil value as an xml.dom.minidom[25] element; the input parameter shall be of type NilValue (page 218).

[23] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[24] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[25] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom

encodeRangeType(*coverage*)
This method returns the range type XML encoding based on GMLCOV and SWE Common as an xml.dom.minidom[26] element. The coverage parameter shall implement EOCoverageInterface (page 200).

encodeRangeTypeField(*range_type*, *band*)
This method returns the the encoding of a SWE Common field as an xml.dom.minidom[27] element. This XML structure represents a band in terms of typical EO data. The range_type parameter shall be a RangeType (page 216) object, the band parameter a Band (page 217) object.

encodeRectifiedGrid(*size*, *(minx*, *miny*, *maxx*, *maxy)*, *srid*, *id*)
This method returns a xml.dom.minidom[28] element containing the GMLCOV representation of a rectified grid. It expects four parameters as input: size shall be a 2-tuple of width and height of the subset; the extent shall be represented by a 4-tuple (minx, miny, maxx, maxy); the srid shall contain the EPSG ID of the spatial reference system; finally, the id string is used to generate gml:id attributes on certain elements that require it.

encodeReferenceableGrid(*size*, *srid*, *id*)
This method returns a xml.dom.minidom[29] element containig the GMLCOV representation of a referenceable grid. It expects three parameters: size is a 2-tuple of width and height of the grid, the srid is the EPSG ID of the spatial reference system and the id string is used to generate gml:id attributes on elements that require it.

Note that the return value is a gml:ReferenceableGrid element that actually does not exist in the GML standard.

The reason is that EOxServer geo-references datasets using ground control points (GCPs) provided with the dataset. With the current GML implementations of gml:AbstractReferenceableGrid it is not possible to specify only the GCPs in the description of the grid. You'd have to calculate and encode the coordinates of every grid point instead. This would blow up the XML descriptions of typical satellite scenes to several 100 MB - which is clearly impractical.

The current implementation returns a gml:RectifiedGrid pseudo-element that is based on the gml:AbstractGrid structure and has about the following structure:

```
<gml:ReferenceableGrid dimension="2" gml:id="some_id">
    <gml:limits>
        <gml:GridEnvelope>
            <gml:low>0 0</gml:low>
            <gml:high>999 999</gml:high>
        </gml:GridEnvelope>
    </gml:limits>
    <gml:axisLabels>lon lat</gml:axisLabels>
</gml:ReferenceableGrid>
```

encodeSubsetDomainSet(*coverage*, *srid*, *size*, *extent*)
This method returns a xml.dom.minidom[30] element containing the GMLCOV representation of a domain set for subsets of rectified or referenceable coverages. Whereas encodeDomainSet() (page 179) computes the grid metadata based on the spatial reference system, extent and pixel size of the whole coverage, this method can be customized with parameters related to a subset of the coverage.

The method expects four parameters: coverage shall be an object implementing EOCoverageInterface (page 200); srid shall be the EPSG ID of the subset CRS (which does not have to be the same as the coverage CRS); size shall be a 2-tuple of width and height of the subset; finally the extent shall be represented by a 4-tuple (minx, miny, maxx, maxy).

class eoxserver.services.ows.wcs.encoders.**EOPEncoder**(*schemas=None*)
This encoder implements some encodings of EO O&M. It inherits from GMLEncoder (page 181).

[26] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[27] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[28] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[29] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[30] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom

encodeEarthObservation(*eo_metadata*, *contributing_datasets=None*, *poly=None*)
This method returns a xml.dom.minidom[31] element containing the EO O&M representation of an Earth Observation. It takes an eo_metadata object as an input that implements the EOMetadataInterface (page 201).

Note that the return value is only a minimal encoding with the mandatory elements.

encodeFootprint(*footprint*, *eo_id*)
Returns a xml.dom.minidom[32] element containing the EO O&M representation of a footprint. The footprint argument shall contain a GeoDjango GEOSGeometry[33] object containing the footprint as a polygon or multipolygon. The eo_id argument is passed on to the GML encoder as a base ID for generating required gml:id attributes.

encodeMetadataProperty(*eo_id*, *contributing_datasets=None*)
This method returns a xml.dom.minidom[34] element containing the EO O&M representation of an eop:metaDataProperty element.

The eo_id element is reported in the eop:identifier element. If provided, a list of contributing_datasets descriptions will be included in the eop:composedOf element.

class eoxserver.services.ows.wcs.encoders.**GMLEncoder**(*schemas=None*)
This encoder provides methods for encoding basic GML objects.

Note that the axis order for the input point coordinates used in geometry representations is always (x, y) or (lon, lat). The axis order in the output coordinates on the other hand will be the order as mandated by the EPSG definition of the respective spatial reference system. This may be (y, x) for some projected CRSes (e.g. EPSG:3035, the European Lambert Azimuthal Equal Area projection used for many datasets covering Europe) and (lat,lon) for most geographic CRSes including EPSG:4326 (WGS 84).

encodeLinearRing(*ring*, *srid*)
Returns a xml.dom.minidom[35] element containing the GML representation of a linear ring. The ring argument is expected to be a list of tuples which represent 2-D point coordinates with (x,y)/(lon,lat) axis order. The srid argument shall contain the EPSG ID of the spatial reference system as an integer.

encodeMultiPolygon(*geom*, *base_id*)
This method returns a xml.dom.minidom[36] element containing the GML represenation of a multipolygon. The geom argument is expected to be a GeoDjango GEOSGeometry[37] object. The base_id string is used to generate the required gml:id attributes on different elements of the multipolygon encoding.

encodePolygon(*poly*, *base_id*)
This method returns a xml.dom.minidom[38] element containing the GML representation of a polygon. The poly argument is expected to be a GeoDjango Polygon[39] or GEOSGeometry[40] object containing a polygon. The base_id string is used to generate the required gml:id attributes on different elements of the polygon encoding.

class eoxserver.services.ows.wcs.encoders.**WCS20EOAPEncoder**(*schemas=None*)
This encoder provides methods for generating EO-WCS compliant XML descriptions.

encodeContents()
Returns an empty wcs:Contents element as xml.dom.minidom[41] element.

[31] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[32] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[33] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/geos/#django.contrib.gis.geos.GEOSGeometry
[34] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[35] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[36] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[37] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/geos/#django.contrib.gis.geos.GEOSGeometry
[38] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[39] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/geos/#django.contrib.gis.geos.Polygon
[40] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/geos/#django.contrib.gis.geos.GEOSGeometry
[41] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom

encodeContributingDatasets (*coverage*, *poly=None*)
This method returns a list of `xml.dom.minidom`[42] elements containing wcseo:dataset descriptions of contributing datasets. This is used for coverage descriptions of Rectified Stitched Mosaics. The `coverage` parameter shall refer to a `RectifiedStitchedMosaicWrapper` (page 224) object. The optional `poly` argument may contain a GeoDjango `GEOSGeometry`[43] object describing the polygon. If it is provided, the set of contributing datasets will be restricted to those intersecting the given polygon.

encodeCountDefaultConstraint (*count*)
This method returns a ows:Constraint element representing the default maximum of descriptions in an EO-WCS DescribeEOCoverage response for use in WCS 2.0 GetCapabilities responses. The `count` argument is expected to contain a positive integer.

encodeCoverageDescription (*coverage*, *is_root=False*)
This method returns a wcs:CoverageDescription element including EO Metadata as `xml.dom.minidom`[44] element. It expects one mandatory argument, `coverage`, which shall implement `EOCoverageInterface` (page 200). The optional `is_root` flag indicates whether the returned element will be the document root of the response. If yes, a xsi:schemaLocation attribute pointing to the EO-WCS schema will be added to the root element. It defaults to `False`.

encodeCoverageSummary (*coverage*)
This method returns a wcs:CoverageSummary element as `xml.dom.minidom`[45] element. It expects a `coverage` object implementing `EOCoverageInterface` (page 200) as input.

encodeDatasetSeriesDescription (*dataset_series*)
This method returns a `xml.dom.minidom`[46] element representing a Dataset Series description. The method expects a `DatasetSeriesWrapper` (page 226) object `dataset_series` as its only input.

encodeDatasetSeriesDescriptions (*datasetseriess*)
This method returns a wcs:DatasetSeriesDescriptions element as a `xml.dom.minidom`[47] element. The element contains the descriptions of a list of Dataset Series contained in the `datasetseriess` parameter.

encodeDatasetSeriesSummary (*dataset_series*)
This method returns a wcseo:DatasetSeriesSummary element referring to `dataset_series`, a `DatasetSeriesWrapper` (page 226) object.

encodeDescribeEOCoverageSetOperation (*http_service_url*)
This method returns an ows:Operation element describing the additional EO-WCS DescribeEOCoverageSet operation for use in the WCS 2.0 GetCapabilities response. The return value is - as always - a `xml.dom.minidom`[48] element.

The only parameter is the HTTP service URL of the EOxServer instance.

encodeEOCoverageSetDescription (*datasetseriess*, *coverages*, *numberMatched=None*, *numberReturned=None*)
This method returns a wcseo:EOCoverageSetDescription element (the response to a EO-WCS DescribeEOCoverageSet request) as a `xml.dom.minidom`[49] element.

`datasetseriess` shall be a list of `DatasetSeriesWrapper` (page 226) objects. The `coverages` argument shall be a list of objects implementing `EOCoverageInterface` (page 200). The optional `numberMatched` and `numberReturned` arguments are used in responses for pagination.

[42] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[43] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/geos/#django.contrib.gis.geos.GEOSGeometry
[44] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[45] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[46] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[47] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[48] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[49] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom

encodeEOMetadata(*coverage*, *req=None*, *include_composed_of=False*, *poly=None*)
This method returns a xml.dom.minidom[50] element containing the EO Metadata description of a coverage as needed for EO-WCS descriptions. The method requires one argument, coverage, that shall implement EOCoverageInterface (page 200).

Moreover, a OWSRequest (page 174) object req can be provided. If it is present, a wcseo:lineage element that describes the request arguments will be added to the metadata description.

The include_composed_of and poly arguments are ignored at the moment.

encodeEOProfiles()
Returns a list of ows:Profile elements referring to the WCS 2.0 profiles implemented by EOxServer (EO-WCS and its GET KVP binding as well as the CRS extension of WCS 2.0). The resulting xml.dom.minidom[51] elements can be used in WCS 2.0 GetCapabilities responses.

encodeRangeSet(*reference*, *mimeType*)
This method returns a xml.dom.minidom[52] element containing a reference to the range set of the coverage. The reference parameter shall refer to the file part of a multipart message. The mime_type shall contain the MIME type of the delivered coverage.

encodeRectifiedDataset(*dataset*, *req=None*, *nodes=None*, *poly=None*)
This method returns a wcseo:RectifiedDataset element describing the dataset object of type RectifiedDatasetWrapper (page 219). The nodes parameter may contain a list of xml.dom.minidom[53] nodes to be appended to the root element. The req and poly arguments are passed on to encodeEOMetadata() (page 182).

encodeRectifiedStitchedMosaic(*mosaic*, *req=None*, *nodes=None*, *poly=None*)
This method returns a wcseo:RectifiedStitchedMosaic element describing the mosaic object of type RectifiedStitchedMosaicWrapper (page 224). The nodes parameter may contain a list of xml.dom.minidom[54] nodes to be appended to the root element. The req and poly arguments are passed on to encodeEOMetadata() (page 182).

encodeReferenceableDataset(*coverage*, *reference*, *mimeType*, *is_root=False*, *subset=None*)
This method returns the description of a Referenceable Dataset as a xml.dom.minidom[55] element. It expects three input arguments: coverage shall be a ReferenceableDatasetWrapper (page 221) instance; reference shall be a string containing a reference to the coverage data; mime_type shall be a string containing the MIME type of the coverage data.

The is_root flag indicates that the returned element is the document root and an xsi:schemaLocation attribute pointing to the EO-WCS schemas shall be added. It defaults to False. The subset argument is optional. In case it is provided it indicates that the description relates to a subset of the dataset only and thus the metadata (domain set) shall be changed accordingly. It is expected to be a 4-tuple of (srid, size, extent, footprint). The srid represents the integer EPSG ID of the CRS description. The size contains a 2-tuple of width and height. The extent is a 4-tuple of (minx, miny, maxx, maxy); the coordinates shall be expressed in the CRS denoted by srid. The footprint part is not used.

encodeSubsetCoverageDescription(*coverage*, *srid*, *size*, *extent*, *footprint*, *is_root=False*)
This method returns a xml.dom.minidom[56] element containing a coverage description for a subset of a coverage according to WCS 2.0. The coverage parameter shall implement EOCoverageInterface (page 200). The srid shall contain the integer EPSG ID of the output (subset) CRS. The size parameter shall be a 2-tuple of width and height. The extent shall be a 4-tuple of (minx, miny, maxx, maxy) expressed in the CRS described by srid. The footprint argument shall be a GeoDjango GEOSGeometry[57] object containing a polygon. The is_root flag indicates whether the resulting wcs:CoverageDescription element is the document root

[50] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[51] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[52] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[53] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[54] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[55] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[56] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[57] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/geos/#django.contrib.gis.geos.GEOSGeometry

of the response. In that case a xsi:schemaLocation attribute pointing to the EO-WCS schema will be added. It defaults to `False`.

encodeSupportedCRSs()
This method returns list of `xml.dom.minidom`[58] elements containing the supported CRSes for a service. The CRSes are retrieved using `eoxserver.resources.coverages.crss.getSupportedCRS_WCS()` (page 189). They are encoded as crsSupported elements in the namespace of the WCS 2.0 CRS extension.

encodeTimePeriod(*dataset_series*)
This method returns a gml:TimePeriod element referring to the time period of a Dataset Series. The input argument is expected to be a `DatasetSeriesWrapper` (page 226) object.

encodeWGS84BoundingBox(*dataset_series*)
This element returns the ows:WGS84BoundingBox for a Dataset Series. The input parameter shall be a `DatasetSeriesWrapper` (page 226) object.

class `eoxserver.services.ows.wcs.encoders.`**WCS20Encoder**(*schemas=None*)
This encoder class provides methods for generating XML needed by WCS 2.0. It inherits from `CoverageGML10Encoder` (page 179).

encodeCoverageDescription(*coverage*)
Returns a `xml.dom.minidom`[59] element representing a coverage description. The method expects one parameter, `coverage`, which shall implement the `EOCoverageInterface` (page 200).

encodeCoverageDescriptions(*coverages*, *is_root=False*)
Returns a `xml.dom.minidom`[60] element representing a wcs:CoverageDescriptions element. The `coverages` argument shall be a list of objects implementing `EOCoverageInterface` (page 200) whereas the optional `is_root` flag indicates that the element will be the document root and thus should include an xsi:schemaLocation attribute pointing to the EO-WCS schema; it defaults to `False`.

encodeExtension()
Returns an empty wcs:Extension element as an `xml.dom.minidom`[61] element.

Module eoxserver.services.auth.base

This module contains basic classes and functions for the security layer (which is integrated in the service layer for now).

class `eoxserver.services.auth.base.`**AuthorizationResponse**(*content=''*, *content_type='text/xml'*, *headers={}*, *status=None*, *authorized=False*)
A simple base class that contains a response text, content type, headers and status, as well as an `authorized` flag. It inherits from `Response` (page 174).

class `eoxserver.services.auth.base.`**BasePDP**
This is the base class for PDP implementations. It provides a skeleton for authorization request handling.

authorize(*ows_req*)
This method handles authorization requests according to the requirements given in the `PolicyDecisionPointInterface` (page 184) declaration.

Internally, it invokes the `_decide()` method that implements the actual authorization decision logic.

class `eoxserver.services.auth.base.`**PolicyDecisionPointInterface**
This is the interface for Policy Decision Point (PDP) implementations.

[58] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[59] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[60] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom
[61] http://docs.python.org/2.7/library/xml.dom.minidom.html#xml.dom.minidom

authorize(*ows_req*)
This method takes an OWSRequest (page 174) object as input and returns an AuthorizationResponse (page 184) instance. It is expected to check if the authenticated user (if any) is authorized to access the requested resource and set the authorized flag of the response accordingly.

In case the user is not authorized, the content and status of the response shall be filled with an error message and the appropriate HTTP Status Code (403).

The method shall not raise any exceptions.

2.12.4 Processing Layer

Module eoxserver.resources.processes.tracker

This module contains the process tracker API. Process tracker is an essential part of the ATP (Asynchronous Task Processing) subsystem.

Table of Contents

- Module eoxserver.resources.processes.tracker (page 185)
  - Basic API (page 185)
    - Task Type Registration (page 185)
    - Task Creation (page 186)
    - Task Handler Subroutine (page 186)
  - Advanced API (page 187)
    - Task Manipulation (page 187)
    - Task Processing History (page 188)
    - Task Response (page 188)
    - Clean-up Tools (page 188)
    - DB Access and Locking (page 188)
    - Auxiliary Subroutines (page 189)
    - Auxiary Data (page 189)
    - Exceptions (page 189)

Basic API

The *User API* section contains the basic functions and classes required by an actual ATP user to implement a new asynchronous application.

Task Type Registration

eoxserver.resources.processes.tracker.**registerTaskType**(*identifier*, *handler*, *timeout=3600*, *timeret=-1*, *maxrestart=2*)
Register new task type.

The task type 'identifier' string and 'handler' subroutine must be specified. The string identifier must uniquely identify the created task type.

Optionally, the parameters such as: task 'timeout' in sec. after which the task is restarted (re-enqueued for new processing), retention time ('timeret'), i.e., the time to keep finished tasks stored in DB, for any non-positive number the task is kept forever), and finally the max. allowed number of task's restarts caused by task time-out ('maxrestart'). When the number of restarts is exceeded, the task is labelled as FAILED and not re-enqueued any more).

When called repeatedly with the same task identifier, the first run creates new task types and the subsequent calls update the task type parameters.

eoxserver.resources.processes.tracker.**unregisterTaskType**(*identifier*, *force=False*)

Unregister (remove) an existing task Type.

By default, the task Type removal will fail as long as there is an existing task Instance raising 'django.db.models.ProtectedError' exception (a subclass of django.db.IntegrityError).

To force the Type removal wiping out all the linked Instances set the 'force' parameter to True.

Task Creation

eoxserver.resources.processes.tracker.**enqueueTask**(*type*, *identifier*, *input*, *message=''*)

Create new task Instance of the given Type using the given identifier and task inputs and enqueue this task for processing. The task status is set to ACCEPTED.

The 'type' parameter should be the string identifier of a registered task type. The string 'identifier' shall uniquely identify the created task.

The 'input' can be any Python object serializable by the 'pickle' module.

The optional log 'message' can be specified.

In case of full task queue the QueueFull exception is risen.

Task Handler Subroutine

eoxserver.resources.processes.tracker.**dummyHandler**(*taskStatus*, *input*)

Dummy ATP handler. No action implemented.

Prototype of an ATP handler subroutine.

Any ATP handler receives two parameters:

- 'taskStatus' - an instance of TaskStatus class providing access the the actual task,
- 'input' - input parameters specified during the task enqueueing.

class eoxserver.resources.processes.tracker.**TaskStatus**(*task_id*, *dbLock=None*)

TaskStatus provides an interface to current asynchronous task. An instance of this class is exepected to be passed as an input parameter to the ATP handler function when executed by the ATPD.

- 'task_id' in an unique task identifier (string).
- dbLock' can be None or any class instance providing two members: 'dbLock.acquire()' and 'dbLock.release()'.

The status changing member function internally lock the access to the DB using the user provided 'dbLock'. In case the 'dbLock' is not provided the locking is not performed (see also 'DummyLock' class).

getIdentifier()

Get tuple of task Type and Instance identifiers.

getInfo()

Get short info about the task. Returns tuple of Type identifier, Instance identifier, status, and status string.

getStatus()

Get task status as tuple of the integer code and the string label.

setFailure(*message=''*)

Set task status to FAILED.

setPaused(*message=''*)

Set task status to PAUSED.

setRunning(*message=''*)

Set task status to RUNNING.

setSuccess(*message=''*)
Set task status to FINISHED (i.e., success).

storeResponse(*response*, *mimeType='text/xml'*)
Store the task response.

The response is expected to be python string (Text). However binary data (such as pickled data) may be used as well.

Advanced API

The *ATPD API* section contains additional functions and classes required for creation of ATP daemon.

Task Manipulation Note: These functions are NOT granted to have an exlusive access to the DB. When DB locking is required call these function through the 'dbLocker' wrapper.

eoxserver.resources.processes.tracker.**getTaskInfo**(*task_id*)
Get tuple of Type identifier, Instance identifiers, Instance status and corresponding status string

eoxserver.resources.processes.tracker.**getTaskIdentifier**(*task_id*)
Get tuple of Type and Instance identifiers.

eoxserver.resources.processes.tracker.**getTaskStatus**(*task_id*)
Get tuple of Instance status and corresponding status string. 'task_id' is the DB record ID.

eoxserver.resources.processes.tracker.**getTaskStatusByIdentifier**(*type*, *identifier*)
Get tuple of Instance status and corresponding status string. 'type' is the Type string ID and 'identifier' is the Instance string ID.

eoxserver.resources.processes.tracker.**reenqueueTask**(*task_id*, *message=''*)
Re-enqueue an existing task Instance identified by the given DB record ID and set its status to ACCEPTED. The optional log message can be specified.

The task is always enqueued and can possibly increase the task queue size beyond queue size limit.

eoxserver.resources.processes.tracker.**dequeueTask**(*serverID*, *message=''*)
Attempt to dequeue a single task from the task queue. An unique serverID must be provided to prevent collisions with the other ATPDs pulling tasks from the same queue.

The function returns list of the dequeue tasks. There is rare but still possible chance that the function returns either zero or more than one tasks and the user must take this into consideration.

The returned dequeued tasks' status is set to SCHEDULED.

In case of an empty queue the QueueEmpty exception is risen.

eoxserver.resources.processes.tracker.**startTask**(*task_id*, *message=''*)
Get the inputs of the task Instance identified by the given DB record ID and set the task's status to RUNNING

eoxserver.resources.processes.tracker.**pauseTask**(*task_id*, *message=''*)
Set status of task instance identified by the given DB record ID to PAUSED.

eoxserver.resources.processes.tracker.**resumeTask**(*task_id*, *message=''*)
Set status of task instance identified by the given DB record ID to RUNNING.

eoxserver.resources.processes.tracker.**stopTaskSuccessIfNotFinished**(*task_id*, *message=''*)
Set status of task Instance identified by the given DB record ID to FINISHED if its status has not been set to FINISHED or FAILED yet.

eoxserver.resources.processes.tracker.**stopTaskSuccess**(*task_id*, *message=''*)
Set status of task Instance identified by the given DB record ID to FINISHED.

eoxserver.resources.processes.tracker.**stopTaskFailure**(*task_id*, *message=''*)
Set status of task instance identified by the given DB record ID to FAILED.

eoxserver.resources.processes.tracker.**deleteTask**(*task_id*)
Delete task Instance. 'task_id' is the DB record ID.

eoxserver.resources.processes.tracker.**deleteTaskByIdentifier**(*type*, *identifier*)
Delete task Instance. 'type' is the Type string ID and 'identifier' is the Instance string ID.

Task Processing History

eoxserver.resources.processes.tracker.**getTaskLog**(*type*, *identifier*)
Return list of log records sorted by time for the task identified by the task Type and Instance identifiers. Each log record is a tuple of three fields: time-stamp, status tuple (see get task status), and logged message.

Task Response

eoxserver.resources.processes.tracker.**setTaskResponse**(*task_id*, *response*, *mimeType='text/xml'*)
Set response of task Instance identified by the given DB record ID.

The response is expected to be python string (Text). However binary data (such as pickled data) may be used as well.

It is safe to call this function repeatedly. First call creates a new Response record and the successive calls update the existing Response record.

eoxserver.resources.processes.tracker.**getTaskResponse**(*type*, *identifier*)
Return a tuple of task response and its MIME type. Task Instance is identified by an unique pair of Type and Instance string identifiers 'type' and 'identifier', respectively.

The response is expected to be python string (Text). However binary data (such as pickled data) may be used as well.

Clean-up Tools

eoxserver.resources.processes.tracker.**reenqueueZombieTasks**(*message=''*)
Find all tasks exceeding their time-out and try to re-enqueue them again. Tasks exceeding the number of allowed start are rejected and marked as FAILED.

eoxserver.resources.processes.tracker.**deleteRetiredTasks**()
Find all FINISHED or FAILED task Instances exceeding their retention time and remove them.

DB Access and Locking In certain cases it my be necessary to assure mutually exclusive access to the underlaying DB. The proper logging mechanism is dependent on the actual concurrent processing implementation.

class eoxserver.resources.processes.tracker.**DummyLock**
Dummy (default) lock class implementing lock interface.

acquire()
Acquire DB lock. No action impelemented!

release()
Release DB lock. No action impelemented!

eoxserver.resources.processes.tracker.**dbLocker**(*dbLock*, *func*, **prm*, ***kprm*)
Grant exclusive DB access while executing the passed function. The 'dbLocker' function executes the 'dbLock.acquire()' and 'dbLock.release()' methods on entry and exit, respectively, assuring the executed function 'func' has an exclusive access to the DB. 'prm' and 'kprm' are the optional 'func' function parameters. The 'dbLocker' function returns the returning value of the passed 'func' function.

Auxiliary Subroutines

eoxserver.resources.processes.tracker.**getQueueSize**()
Get number of enqueued tasks.

eoxserver.resources.processes.tracker.**getMaxQueueSize**()
Get the maximum allowed number of task the queue can hold.

Auxiary Data

eoxserver.resources.processes.tracker.**MAX_QUEUE_SIZE = 64**
Actual queue size limit. Note may be removed in the future. Use 'getMaxQueueSize()' instead.

eoxserver.resources.processes.models.**STATUS2TEXT = {0: 'UNDEFINED', 1: 'ACCEPTED', 2: 'SCHEDUL**
status code to text conversion dictionary

eoxserver.resources.processes.models.**TEXT2STATUS = {'SCHEDULED': 2, 'UNDEFINED': 0, 'FINISHED'**
status text to code reverse conversion dictionary (filled dynamically)

Exceptions

class eoxserver.resources.processes.tracker.**QueueException**
Task queue base exception.

class eoxserver.resources.processes.tracker.**QueueEmpty**
Queue exception signalising that the task queue is empty and no task can be pulled from it.

class eoxserver.resources.processes.tracker.**QueueFull**
Queue exception signalising that the task queue is full and no task can be pushed to it.

2.12.5 Data Integration Layer

Module eoxserver.resources.coverages.crss

This module provides CRS handling utilities.

Getting List of Supported CRSes

eoxserver.resources.coverages.crss.**getSupportedCRS_WMS**(*format_function=<function asShortCode at 0x5f941b8>*)
Get list of CRSes supported by WMS. The format_function is used to format individual list items.

eoxserver.resources.coverages.crss.**getSupportedCRS_WCS**(*format_function=<function asShortCode at 0x5f941b8>*)
Get list of CRSes supported by WCS. The format_function is used to format individual list items.

Utilities

eoxserver.resources.coverages.crss.**hasSwappedAxes**(*epsg*)
Decide whether the coordinate system given by the passed EPSG code is displayed with swapped axes (True) or not (False).

eoxserver.resources.coverages.crss.**getAxesSwapper**(*epsg*, *swapAxes=None*)
Second order function returning point tuple axes swaper f(x,y) -> (x,y) or f(x,y) -> (y,x). The axes order is determined by the provided EPSG code. (Or exlicitely by the swapAxes boolean flag.

eoxserver.resources.coverages.crss.**isProjected**(*epsg*)
Is the coordinate system projected (True) or Geographic (False)?

eoxserver.resources.coverages.crss.**validateEPSGCode**(*string*)
Check whether the given string is a valid EPSG code (True) or not (False)

EPSG Code Parsing

This is the top level EPSG parser able to use one or more elementary parsers listed below.

`eoxserver.resources.coverages.crss.`**`parseEPSGCode`**(*string*, *parsers*)
parse EPSG code using provided sequence of EPSG parsers

These are the elementary EPSG code parser:

`eoxserver.resources.coverages.crss.`**`fromInteger`**(*string*)
parse EPSG code from simple integer string

`eoxserver.resources.coverages.crss.`**`fromShortCode`**(*string*)
parse EPSG code from given string in short CRS `EPSG:<code>` notation

`eoxserver.resources.coverages.crss.`**`fromURL`**(*string*)
parse EPSG code from given string in OGC URL CRS `http://www.opengis.net/def/crs/EPSG/0/<code>` notation

`eoxserver.resources.coverages.crss.`**`fromURN`**(*string*)
parse EPSG code from given string in OGC URN CRS `urn:ogc:def:crs:epsg::<code>` notation

`eoxserver.resources.coverages.crss.`**`fromProj4Str`**(*string*)
parse EPSG code from given string in OGC Proj4Str CRS `+init=epsg:<code>` notation

EPSG Code Formating

These formating functions are used to get the CRSes in different notations.

`eoxserver.resources.coverages.crss.`**`asInteger`**(*epsg*)
convert EPSG code to integer

`eoxserver.resources.coverages.crss.`**`asShortCode`**(*epsg*)
convert EPSG code to short CRS `EPSG:<code>` notation

`eoxserver.resources.coverages.crss.`**`asURL`**(*epsg*)
convert EPSG code to OGC URL CRS `http://www.opengis.net/def/crs/EPSG/0/<code>` notation

`eoxserver.resources.coverages.crss.`**`asURN`**(*epsg*)
convert EPSG code to OGC URN CRS `urn:ogc:def:crs:epsg::<code>` notation

`eoxserver.resources.coverages.crss.`**`asProj4Str`**(*epsg*)
convert EPSG code to *proj4* `+init=epsg:<code>` notation

Static Data

`eoxserver.resources.coverages.crss.`**`EPSG_AXES_REVERSED = set([25884, 2036, 22185, 2044, 2045, 22187, 206`**
Set (Python `set` type) of EPSG codes of CRS whose axes are displayed in reversed order.

Module eoxserver.resources.coverages.data

class `eoxserver.resources.coverages.data.`**`DataPackageFactory`**
This factory gives access to data package wrappers. It inherits from `RecordWrapperFactory` (page 143).

class `eoxserver.resources.coverages.data.`**`DataPackageWrapper`**
This is the common base class for data package wrappers. It derives from `RecordWrapper` (page 143).

`setAttrs`(***kwargs*)
`DataPackageWrapper` (page 190) defines three attributes that can be assigned to any data package instance:

- `location`: the location of the data; the location type depends on the concrete data package subclass
- `metadata_location`: the location of the metadata; the location type depends on the concrete data package subclass
- `metadata_format_name`: the name of the metadata format; can be derived automatically from the metadata using `readEOMetadata()` (page 191)

sync()
See `RecordWrapper.sync()` (page 143).

getRecord()
See `RecordWrapper.getRecord()` (page 143)

getAccessibleLocation()
Get an accessible location of the underlying dataset. A previous successful call to `prepareAccess()` (page 191) may be necessary for this method to yield a meaningful result. Concrete subclasses have to override this. By default `InternalError` (page 133) is raised.

getCoverages()
Return the coverages that use this data package.

getGDALDatasetIdentifier()
Get a GDAL dataset identifier for the underlying dataset, i.e. the string to be passed on to the `gdal.Open()` function. A previous successful call to `prepareAccess()` (page 191) may be necessary for this method to yield a meaningful result. Concrete subclasses have to override this. By default `InternalError` (page 133) is raised.

getLocation()
Return the location of the data, i.e. an object implementing `LocationInterface` (page 237). The location type depends on the concrete data package subclass.

getMetadataLocation()
Return the location of the metadata, i.e. an object implementing `LocationInterface` (page 237). The location type depends on the concrete data package subclass.

getSourceFormat()
Return the source data file format.

open()
Open the underlying dataset with GDAL and return a `osgeo.gdal.Dataset` object. This method raises `EngineError` if GDAL was not able to open the dataset. It raises `DataAccessError` if the dataset could not be made accessible to GDAL (e.g. download of a remote FTP resource failed).

prepareAccess()
Prepare access to the underlying dataset. This makes the underlying dataset accessible so that `getAccessibleLocation()` (page 191) and `getGDALDatasetIdentifier()` (page 191) can yield meaningful results. Concrete subclasses have to override this. By default `InternalError` (page 133) is raised.

readEOMetadata()
Read EO Metadata from the metadata location and return an `EOMetadata` (page 214) instance. `DataAccessError` may be raised if the metadata location cannot be made accessible (e.g. an XML metadata file cannot be retrieved from a remote location). `MetadataException` will be raised if the metadata cannot be read (e.g. because a metadata file does not contain valid XML).

readGeospatialMetadata(*default_srid=None*)
Read geospatial metadata from the underlying dataset. The return value is a `GeospatialMetadata` instance. The method accepts an optional integer `default_srid` argument which predefines the output SRID if it cannot be retrieved from the dataset; see `readFromDataset()`.

The dataset is opened using `open()` (page 191); it may raise `DataAccessError` or `EngineError` in the error cases described there.

class eoxserver.resources.coverages.data.**DataSourceFactory**
This is a factory for DataSourceWrapper (page 192) objects. It inherits from RecordWrapperFactory (page 143).

class eoxserver.resources.coverages.data.**DataSourceWrapper**
This class implements DataSourceInterface (page 200). It inherits from RecordWrapper (page 143).

setAttrs(***kwargs*)
DataSourceWrapper (page 192) defines two attributes that can be assigned to an instance:

- location: the location of the data source (a local or remote path to a directory)
- search_pattern: the search pattern defined for the data source (optional)

sync()
See RecordWrapper.sync() (page 143).

getRecord()
See RecordWrapper.getRecord() (page 143)

contains(*wrapper*)
Check if the DataSource contains a coverage with a certain ID

detect()
Detect files at the location that match the given search pattern. Returns a list of locations. If no location has been defined yet, return an empty list.

getType()
Returns "data_source".

class eoxserver.resources.coverages.data.**LocalDataPackageWrapper**(***kwargs*)
This is a wrapper for data packages stored in files on the local file system. It inherits from DataPackageWrapper (page 190). See there for the inherited methods.

getAccessibleLocation()
Returns the same as getLocation().

getDataStructureType()
Returns "file".

getGDALDatasetIdentifier()
Returns the path to the data file.

Note: This does not account for data formats where the dataset is structured into subdatasets. This is future work

---

getType()
Returns "local".

prepareAccess()
Nothing to be done here as locations on the local file system are accessible by themselves.

class eoxserver.resources.coverages.data.**RasdamanDataPackageWrapper**
This is a wrapper for rasdaman data packages. It inherits from DataPackageWrapper (page 190). See there for the inherited methods.

getAccessibleLocation()
Return the rasdaman array location.

getDataStructureType()
Returns "rasdaman_array".

getGDALDatasetIdentifier()
Returns a connection string to the rasdaman database combined with a query indicating the given dataset. This is the format GDAL expects for reading data from a rasdaman array.

getSourceFormat()
Return the source data file format.

getType()
Returns "rasdaman_array".

prepareAccess()
Nothing to be done here. Though not necessarily local the rasdaman data is always accessible in the sense that its always possible to connect to it without further preconditions.

class eoxserver.resources.coverages.data.**RemoteDataPackageWrapper**
This is a wrapper for data stored in a remote repository accessible via FTP. It inherits from DataPackageWrapper (page 190). See there for the inherited methods.

This class wraps not only the (remote) locations of data and metadata, but also CacheFileWrapper (page 235) instances for locally cached copies of the respective files.

initialize(***kwargs*)
In addition to the attributes declared in DataPackageWrapper.initialize() this method accepts an optional cache_file keyword argument which is expected to be an instance of CacheFileWrapper (page 235).

getAccessibleLocation()
Returns the location of the locally cached data file.

getDataStructureType()
Returns "file".

getGDALDatasetIdentifier()
Returns the path to the location of the locally cached data file.

getType()
Returns "remote".

prepareAccess()
Loads a remote data file into the local cache, if necessary. Never omit the call to prepareAccess() (page 193) when attempting to access a remote dataset, subsequent method calls to open(), getAccessibleLocation() (page 193) and getGDALDatasetIdentifier() (page 193) may fail.

class eoxserver.resources.coverages.data.**TileIndexFactory**
This is a factory for TileIndexWrapper (page 193) objects. It inherits from RecordWrapperFactory (page 143).

class eoxserver.resources.coverages.data.**TileIndexWrapper**
This class wraps a tile index. It inherits from RecordWrapper (page 143).

initialize(***kwargs*)
Apart from the mandatory record keyword argument, this method accepts a storage_dir argument which will be saved as instance attribute. An InternalError (page 133) will be raised if neither of the two is given. The storage_dir denotes the path to the local directory where to find a tile index shape file as well as the actual tiles (usually stored in a directory tree under that directory).

getDataStructureType()
Returns "index".

getShapeFilePath()
Returns the path to the tile index shape file.

getSourceFormat()
Return the source data file format.

getStorageDir()
Returns the path to the directory where to find the tile index shape file as well as the actual tiles.

getType()
Returns "index".

Module eoxserver.resources.coverages.filters

This module defines filters and filter expressions for EO Coverages. For more information on filters, see `eoxserver.core.filters` (page 134).

Helper Classes

class `eoxserver.resources.coverages.filters.`**`TimeInterval`**(*begin*, *end*)
This class contains information about a time interval. The constructor accepts two arguments: `begin` and `end` which must be set either to the string `"unbounded"` or a `datetime.datetime`[62] object.

`InternalError` (page 133) is raised if the arguments do not validate. `InvalidExpressionError` (page 133) is raised if the begin time is later than the end time.

class `eoxserver.resources.coverages.filters.`**`Slice`**(*crs_id*, *axis_label*, *slice_point*)
This class contains information about a slice subsetting. The constructor accepts three arguments:

- `crs_id`: either `"imageCRS"` or an integer EPSG SRID,
- `axis_label`: the axis label the slicing operation refers to,
- `slice_point`: a `float` or `int` containing the slice point information

`InternalError` (page 133) is raised if arguments do not validate.

class `eoxserver.resources.coverages.filters.`**`BoundedArea`**(*crs_id*, *minx*, *miny*, *maxx*, *maxy*)
This class contains information about a bounded area. The constructor accepts a `crs_id` and four bounds arguments `minx, miny, maxx, maxy`. The `crs_id` parameter may be set to `"imageCRS"` or an integer EPSG SRID. The bounds parameters may be set to a `float` or `int` value designating the bound in the given coordinate system or to `"unbounded"`.

`InternalError` (page 133) is raised if the arguments do not validate. `InvalidExpressionError` (page 133) is raised if the lower bounds of an axis are greater than the upper bounds.

Filter Expressions

Filter expressions (i.e. implementations of `FilterExpressionInterface` (page 134)) define certain search constraints for resource factories. Filter expressions should be created using the `get()` or `find()` methods of `CoverageExpressionFactory` (page 196). They will be translated into the corresponding filters by the resource factory.

class `eoxserver.resources.coverages.filters.`**`TimeSliceExpression`**
Filter expression implementation representing a time slice. Expects one operand: a `datetime.datetime`[63] object representing the slice point in time.

class `eoxserver.resources.coverages.filters.`**`TimeIntervalExpression`**
Filter expression implementation representing a time interval. It expects one operand of type `TimeInterval` (page 194).

class `eoxserver.resources.coverages.filters.`**`IntersectingTimeIntervalExpression`**
Filter expression implementation that matches if a time or time interval intersects with the time interval specified in the expression. Inherits from `TimeIntervalExpression` (page 194).

class `eoxserver.resources.coverages.filters.`**`ContainingTimeIntervalExpression`**
Filter expression implementation that matches if a time or time interval is contained in the time interval specified in the expression. Inherits from `TimeIntervalExpression` (page 194).

class `eoxserver.resources.coverages.filters.`**`SpatialSliceExpression`**
Filter expression implementation that represents a slice subsetting. It expects one operand of type `Slice` (page 194).

[62] http://docs.python.org/2.7/library/datetime.html#datetime.datetime
[63] http://docs.python.org/2.7/library/datetime.html#datetime.datetime

class eoxserver.resources.coverages.filters.**BoundedAreaExpression**
Filter expression implementation that represents a trim or BBOX subsetting. It expects one operand of type BoundedArea (page 194).

class eoxserver.resources.coverages.filters.**FootprintIntersectsAreaExpression**
Filter expression implementation that matches if the footprint of an object intersects the given BoundedArea (page 194). Inherits from BoundedAreaExpression (page 195).

class eoxserver.resources.coverages.filters.**FootprintWithinAreaExpression**
Filter expression implementation that matches if the footprint of an object is contained within the given BoundedArea (page 194). Inherits from BoundedAreaExpression (page 195).

Filters

Filters (i.e. implementations of FilterInterface (page 135)) are used primarily to select EO Coverages matching certain criteria. In general developers will not use filters directly, but define filter expressions instead which will be applied to the EO Coverages when invoking the find() (page 153) method of a resource factory.

class eoxserver.resources.coverages.filters.**TimeSliceFilter**
Filter class for time slice operations.

class eoxserver.resources.coverages.filters.**RectifiedDatasetTimeSliceFilter**
Filter which matches Rectified Datasets whose acquisition time interval contains a given timestamp.

class eoxserver.resources.coverages.filters.**ReferenceableDatasetTimeSliceFilter**
Filter which matches Referenceable Datasets whose acquisition time interval contains a given timestamp.

class eoxserver.resources.coverages.filters.**RectifiedStitchedMosaicTimeSliceFilter**
Filter which matches Rectified Stitched Mosaics whose acquisition time interval contains a given timestamp.

class eoxserver.resources.coverages.filters.**IntersectingTimeIntervalFilter**
Filter class for 'time_intersects' operations.

class eoxserver.resources.coverages.filters.**RectifiedDatasetIntersectingTimeIntervalFilter**
Filter which matches Rectified Datasets whose acquisition time interval intersects a given time interval.

class eoxserver.resources.coverages.filters.**ReferenceableDatasetIntersectingTimeIntervalFi**
Filter which matches Referenceable Datasets whose acquisition time interval intersects a given time interval.

class eoxserver.resources.coverages.filters.**RectifiedStitchedMosaicIntersectingTimeInterva**
Filter which matches Rectified Stitched Mosaics whose acquisition time interval intersects a given time interval.

class eoxserver.resources.coverages.filters.**ContainingTimeIntervalFilter**
Filter class for 'time_within' operations.

class eoxserver.resources.coverages.filters.**RectifiedDatasetContainingTimeIntervalFilter**
Filter which matches Rectified Datasets whose acquisition time interval is contained within a given time interval.

class eoxserver.resources.coverages.filters.**ReferenceableDatasetContainingTimeIntervalFilt**
Filter which matches Referenceable Datasets whose acquisition time interval is contained within a given time interval.

class eoxserver.resources.coverages.filters.**RectifiedStitchedMosaicContainingTimeIntervalF**
Filter which matches Rectified Stitched Mosaics whose acquisition time interval is contained within a given time interval.

class eoxserver.resources.coverages.filters.**SpatialFilter**
Common base class for spatial filters.

class eoxserver.resources.coverages.filters.**SpatialSliceFilter**
Common base class for spatial slice filters.

class eoxserver.resources.coverages.filters.**RectifiedDatasetSpatialSliceFilter**
Filter which matches Rectified Datasets whose footprint intersects a given spatial slice.

class eoxserver.resources.coverages.filters.**ReferenceableDatasetSpatialSliceFilter**
Filter which matches Referenceable Datasets whose footprint intersects a given spatial slice.

class eoxserver.resources.coverages.filters.**RectifiedStitchedMosaicSpatialSliceFilter**
Filter which matches Rectified Stitched Mosaics whose footprint intersects a given spatial slice.

class eoxserver.resources.coverages.filters.**FootprintFilter**
Common base class for footprint-related filters.

class eoxserver.resources.coverages.filters.**FootprintIntersectsAreaFilter**
Base filter class matching EO Coverages whose footprint intersects a given area.

class eoxserver.resources.coverages.filters.**RectifiedDatasetFootprintIntersectsAreaFilter**
Filter which matches Rectified Datasets whose footprint intersects a given bounded area.

class eoxserver.resources.coverages.filters.**ReferenceableDatasetFootprintIntersectsAreaFil**
Filter which matches Referenceable Datasets whose footprint intersects a given bounded area.

class eoxserver.resources.coverages.filters.**RectifiedStitchedMosaicFootprintIntersectsArea**
Filter which matches Rectified Stitched Mosaics whose footprint intersects a given bounded area.

class eoxserver.resources.coverages.filters.**FootprintWithinAreaFilter**
Filter matching EO Coverages whose footprint lies within a given area.

class eoxserver.resources.coverages.filters.**RectifiedDatasetFootprintWithinAreaFilter**
Filter which matches Rectified Datasets whose footprint is contained within a given bounded area.

class eoxserver.resources.coverages.filters.**ReferenceableDatasetFootprintWithinAreaFilter**
Filter which matches Referenceable Datasets whose footprint is contained within a given bounded area.

class eoxserver.resources.coverages.filters.**RectifiedStitchedMosaicFootprintWithinAreaFilt**
Filter which matches Rectified Stiched Mosaics whose footprint is contained within given bounded area.

class eoxserver.resources.coverages.filters.**ContainedRectifiedDatasetFilter**
Filter which matches RectifiedDatasets contained in a given RectifiedStitchedMosaic or Dataset series.

class eoxserver.resources.coverages.filters.**ContainedReferenceableDatasetFilter**
Filter which matches ReferenceableDatasets contained in a given DatasetSeries.

Factories

class eoxserver.resources.coverages.filters.**CoverageExpressionFactory**
This is the factory which gives access to the filter expressions defined in this module. It inherits from SimpleExpressionFactory (page 136).

Module eoxserver.resources.coverages.formats

This module contains format handling utilities.

Getting Format Registry

The format registry, although it can be instantiated by the user's code, shall be retrieved by the following function:

eoxserver.resources.coverages.formats.**getFormatRegistry**()
Get initialised instance of the FormatRegistry class. This is the preferable way to get the Format Registry.

Format Record

class eoxserver.resources.coverages.formats.**Format**(*mime_type*, *driver*, *extension*, *is_writeable*)

Format record class. The class is rather structure with read-only properties (below). The class implements `__str__()` and `__eq__()` methods.

defaultExt
default extension (including dot)

driver
library/driver identifier

isWriteable
boolean flag indicating that output can be produced

mimeType
MIME-type

wcs10name
get WCS 1.0 format name

Format Registry

class eoxserver.resources.coverages.formats.**FormatRegistry**(*config*)

The `FormatRegistry` (page 197) class represents cofiguration of file supported formats and of the auxiliary methods. The formats' configuration relies on two configuration files:

- the default formats' configuration (`eoxserver/conf/default_formats.conf`)
- the optional instance configuration (`conf/format.conf` in the instance directory)

Configuration values are read from these files.

getFormatByMIME(*mime_type*)
Get format record for the given MIME type. In case of no match None is returned.

getFormatsAll()
Get list of all registered formats

getFormatsByDriver(*driver_name*)
Get format records for the given GDAL driver name. In case of no match empty list is returned.

getFormatsByWCS10Name(*wcs10name*)
Get format records for the given GDAL driver name. In case of no match an empty list is returned.

getSupportedFormatsWCS()
Get list of formats to be announced as supported WCS formats.

The the listed formats must be: * defined in EOxServers configuration (section "services.ows.wcs", item "supported_formats") * defined in the formats' configuration ("default_formats.conf" or "formats.conf") * supported by the used GDAL installation

getSupportedFormatsWMS()
Get list of formats to be announced as supported WMS formats.

The the listed formats must be: * defined in EOxServers configuration (section "services.ows.wms", item "supported_formats") * defined in the formats' configuration ("default_formats.conf" or "formats.conf") * supported by the used GDAL installation

mapSourceToNativeWCS20(*format*)
Map source format to WCS 2.0 native format.

Both the input and output shall be instances of `Formats` class. The input format can be obtained, e.g., by the *getFormatByDriver* or *getFormatByMIME* method.

To force the default native format use None as the source format.

The format mapping follows these rules:

1. Mapping based on the explicite rules is applied if possible (defined in EOxServers configuration, section "services.ows.wcs20", item "source_to_native_format_map"). If there is no mapping available the source format is kept.
2. If the format resulting from step 1 is not a writable GDAL format or it is not among the supported WCS formats than it is replaced by the default native format (defined in EOxServers configuration, section "services.ows.wcs20", item "default_native_format"). In case of writable GDAL format, the result of step 1 is returned.

Utilities

eoxserver.resources.coverages.formats.**valMimeType**(*string*)
MIME type reg.ex. validator. If pattern not matched 'None' is returned otherwise the input is returned.

eoxserver.resources.coverages.formats.**valDriver**(*string*)
Driver identifier reg.ex. validator. If pattern not matched 'None' is returned otherwise the input is returned.

eoxserver.resources.coverages.formats.**_gerexValMime = <_sre.SRE_Pattern object at 0x4c6da90>**
MIME-type regular expression validator (compiled reg.ex. pattern)

eoxserver.resources.coverages.formats.**_gerexValDriv = <_sre.SRE_Pattern object at 0x55ee080>**
library driver regular expression validator (compiled reg.ex. pattern)

class eoxserver.resources.coverages.formats.**FormatLoaderStartupHandler**
This class is the implementation of the `StartupHandlerInterface` responsible for loading and intialization of the format registry.

reset(*config*, *registry*)
reset handler

startup(*config*, *registry*)
start-up handler

eoxserver.resources.coverages.formats.**FormatLoaderStartupHandlerImplementation = <class 'eox**

Module eoxserver.resources.coverages.interfaces

This module contains the definition of coverage and dataset series interfaces. These provide a harmonized interface to coverage data that can be stored in different formats and has different types.

class eoxserver.resources.coverages.interfaces.**ContainerInterface**
This is the common interface for coverages and series containing EO Coverages.

contains(*wrapper*)
Returns a boolean value describing if the container contains the resource specified by the given `wrapper`.

addCoverage(*wrapper*)
Add resource specified by the given `wrapper`.

removeCoverage(*wrapper*)
Remove resource specified by the given `wrapper`.

getDataSources()
This method shall return a list of data sources, i.e. objects implementing `DataSourceInterface` (page 200) for the given container. It is intended for use in `eoxserver.resources.coverages.synchronize`.

class eoxserver.resources.coverages.interfaces.**CoverageDataInterface**
This is the common base interface for coverage data.

getDataStructureType()
This method shall return a string denoting the data structure type of the data.

class eoxserver.resources.coverages.interfaces.**CoverageInterface**
The parent class of all coverage interfaces. It defines methods for access to coverage data. It inherits from ResourceInterface (page 155).

Interface ID resources.coverages.interfaces.Coverage

getCoverageId()
This method shall return the coverage id of the coverage resource wrapped by the implementation

getCoverageSubtype()
This method shall return the GML coverage subtype of the coverage resource wrapped by the implementation

getType()
This method shall return the EOxServer coverage type of the coverage wrapped by the implementation. Current choices are:

- file
- eo.rect_dataset
- eo.ref_dataset
- eo.rect_stitched_mosaic

getSize()
This method shall return the size of the coverage wrapped by the implementation. The return value is expected to be a 2-tuple of integers (xsize, ysize).

getRangeType()
This method shall return a RangeType (page 216) instance containing the data type and band structure of the coverage wrapped by the implementation

getDataStructureType()
This method shall return the type of the data structure that contains the coverage's data. See CoverageDataInterface.getDataStructureType() (page 198). Note that this does not define the implementation of the coverage data object returned with getData() (page 199).

getData()
This method shall return an object that provides access to the coverage data, i.e. an implementation of CoverageDataInterface (page 198).

getLayerMetadata()
This method shall return a list containing 2-tuples of MapServer metadata key-value-pairs that will be tagged on the MapServer layer representing this coverage.

class eoxserver.resources.coverages.interfaces.**DataPackageInterface**
This interface shall be implemented by Data Packages. Data Packages provide an abstraction layer for various kinds of file-based or database-based datasets. Internally, data packages store information about the location of the original data and (for remote backends) the location of a locally accessible copy.

Methods for high-level data access:

open()
This method shall open the data package. It shall return an object representing the underlying dataset in the engine defined by the data format of the data package.

getLocation()
Returns the location of the data, i.e. an object that implements LocationInterface (page 237). Note that this location is not necessarily directly accessible from the local file system or operating system, but may be remote. For fetching an accessible location, see prepareAccess() (page 200) and getAccessibleLocation() (page 200).

getMetadataLocation()
Returns the location of the metadata, i.e. an object that implements LocationInterface (page 237).

readGeospatialMetadata(*default_srid=None*)
This method shall return an object containing the geospatial metadata stored with the data package. It accepts an optional `default_srid` parameter which indicates the SRID to use if it cannot be read from the data package.

readEOMetadata()
This method shall return an object containing the EO metadata required by EOxServer and stored with the data package.

Methods for low-level data access; use these with care:

prepareAccess()
This method has to be called before any attempt to actually access the data. It shall prepare access, e.g. by retrieving remote data or unpacking complex packages, so that subsequent calls to `getAccessibleLocation()` (page 200) and `getAccessiblePath()` can return meaningful results. It shall raise `DataAccessError` in case of an error.

getAccessibleLocation()
This method shall return a location, i.e. an object implementing `LocationInterface`. An `InternalError` shall be raised if the data package is not accessible (e.g. because `prepareAccess()` (page 200) has not been called or the call failed)

getGDALDatasetIdentifier()
This method shall return a string to be used to open the data package in GDAL. It shall raise `InternalError` (page 133) if the data package cannot be opened in GDAL.

class `eoxserver.resources.coverages.interfaces.`**DataSourceInterface**
This interface shall be implemented by Data Sources. They represent locations where information about a collection of Data Packages can be retrieved.

detect()
This method shall return a list of Data Packages, i.e. objects implementing `DataPackageInterface` (page 199), related to the Data Source.

contains()
This method shall return `True` if a data source references a dataset, `False` otherwise.

class `eoxserver.resources.coverages.interfaces.`**DatasetSeriesInterface**
This interface is intended for implementations of Dataset Series according to the WCS 2.0 EO-AP (EO-WCS). It inherits from `ResourceInterface` (page 155) and `EOWCSObjectInterface` (page 202).

Interface ID resources.coverages.interfaces.DatasetSeries

getType()
Shall return `"eo.dataset_series"`.

getEOCoverages(*filter_exprs=None*)
This method shall return a list of EOCoverage wrappers for the datasets and stitched mosaics contained in the dataset series wrapped by the implementation. The optional `filter_exprs` argument is expected to be a list of filter expressions to be applied to the datasets or `None`. In case no contained dataset matches the filter expressions an empty list shall be returned.

getDatasets(*filter_exprs=None*)
This method shall return a list of RectifiedDataset and ReferenceableDataset wrappers contained in the dataset series. The optional `filter_exprs` argument is expected to be a list of filter expressions to be applied to the datasets or `None`. In case no contained dataset matches the filter expressions an empty list shall be returned.

contains(*wrapper*)
This method shall return `True` if the EO Coverage specified by `wrapper` is contained in the Dataset Series, `False` otherwise.

class `eoxserver.resources.coverages.interfaces.`**EOCoverageInterface**
This interface is the base interface for implementations of EO Coverages according to the WCS 2.0 EO-AP (EO-WCS). It inherits from `CoverageInterface` (page 199) and class:*EOWCSObjectInterface*. It is not intended to be implemented directly; rather one of its descendants shall be used.

Interface ID resources.coverages.interfaces.EOCoverage

getEOCoverageSubtype()
This method shall return the EO coverage subtype of the coverage wrapped by the implementation

getDatasets(*filter_exprs=None*)
This method shall return a list of dataset wrappers for the datasets contained in the coverage wrapped by the implemention. The optional `filter_exprs` argument is expected to be a list of filter expressions to be applied to the datasets or `None`. In case no contained dataset matches the filter expressions an empty list shall be returned.

In case of atomic coverages which do not contain any datasets (e.g. RectifiedDatasets themselves) a list containing the coverage wrapper itself shall be returned. In case filter expressions are provided with the call these shall be applied; if the coverage does not match them an empty list shall be returned.

getLineage()
This method shall return the content of the lineage object stored with the EO coverage wrapped by the implementation. Note that this element is not yet specified in detail in the specification at the moment (2011-05-26). If no lineage is available, None shall be returned.

getContainers()
This method shall return a list of container wrappers (Stitched Mosaic or Dataset Series wrappers) the coverage is contained in. The empty list shall be returned if the coverage is not related to any container object.

getContainerCount()
This method shall return the number of container objects the EO Coverage is contained in.

containedIn(*res_id*)
This method shall return `True` if the EO coverage is contained in the container object (Stitched Mosaic or Dataset Series) specified by the `wrapper`, `False` otherwise.

contains(*res_id*)
This method shall return `True` if the EO coverage is a container object and contains the coverage with resource specified by the `wrapper`, `False` otherwise.

class eoxserver.resources.coverages.interfaces.**EOMetadataFormatInterface**
This interface is intended for EO metadata formats and extends `MetadataFormatInterface` (page 203).

getEOMetadata(*raw_metadata*)
This method shall decode the raw metadata passed to it and return an EO Metadata object, i.e. an implementation of `EOMetadataInterface` (page 201).

The method shall raise `InternalError` (page 133) if the format cannot decode the raw metadata content, e.g. because it is in the wrong data format.

class eoxserver.resources.coverages.interfaces.**EOMetadataInterface**
This an interface for objects carrying basic EO Metadata. It is the base for metadata reader interfaces as well as EO-WCS object interfaces. Note that it does NOT inherit from `MetadataInterface`, so key-value-pair access to metadata values is not possible.

getEOID()
This method shall return the EO ID of the coverage wrapped by the implementation

getBeginTime()
This method shall return the acquisition begin date and time of the EO coverage wrapped by the implementation. The type of the return value is expected to be `datetime.datetime`[64].

getEndTime()
This method shall return the acquisition end date and time of the EO coverage wrapped by the implementation. The type of the return value is expected to be `datetime.datetime`[65].

[64] http://docs.python.org/2.7/library/datetime.html#datetime.datetime
[65] http://docs.python.org/2.7/library/datetime.html#datetime.datetime

getFootprint()
This method shall return the acquisition footprint of the EO coverage wrapped by the implementation. The type of the return value is expected to be `django.contrib.gis.geos.GEOSGeometry`[66].

class `eoxserver.resources.coverages.interfaces.`**EOMetadataReaderInterface**
This interfaces shall be implemented by objects that can read EO Metadata sources and translate them to EO Metadata objects. Implementations can be found in the registry by key-value-pair matching. The interface defines two registry keys:

- `resources.coverages.interfaces.location_type`: the type of the location, e.g. `local` (this is two abstract from file or catalogue record access)
- `resources.coverages.interfaces.encoding_type`: the way how the metadata was encoded; most common are XML encoding in a metadata file or catalogue record and metadata tags in a data file

readEOMetadata(*location*)
This method shall read the object at the given `location` and return the decoded EO Metadata found in it. It shall raise `InternalError` (page 133) if the location or encoding type do not match the implementation specification or `DataAccessError` if the underlying resource cannot be accessed.

class `eoxserver.resources.coverages.interfaces.`**EOWCSObjectInterface**
This is the interface for EO Coverage subtypes as defined by the Earth Observation Application Profile for WCS 2.0. It inherits from `EOMetadataInterface` (page 201). It should not be implemented directly; you'd rather use its descendants.

getWGS84Extent()
This method shall return the WGS 84 extent of the EO coverage wrapped by the implementation. The return value shall be a 4-tuple of floating point coordinates (minlon, minlat, maxlon, maxlat) given in the WGS 84 coordinate system (EPSG:4326).

getEOGML()
This method shall return the EO GML (EO O&M) conformant metadata stored with the EO coverage. If no EO O&M metadata is available, the empty string will be returned

class `eoxserver.resources.coverages.interfaces.`**GenericEOMetadataInterface**
An interface combining generic and EO metadata access. Inherits from `MetadataInterface` and `EOMetadataInterface` (page 201).

class `eoxserver.resources.coverages.interfaces.`**GenericMetadataInterface**
This is an interface for objects containing metadata of any kind. They can be retrieved using a key-value-pair schema.

getMetadataFormat()
This method shall return the metadata format, i.e. an object implementing `MetadataFormatInterface` (page 203).

getMetadataKeys()
This method shall return a list of metadata keys that are understood by the metadata interface.

getMetadataValues(*keys*)
This method shall return a dictionary of metadata keys and values. The dictionary keys shall correspond to the keys conveyed with the request, the dictionary values shall be the respective metadata values, or `None` if the key is not known or no metadata value is defined for the specific metadata instance.

class `eoxserver.resources.coverages.interfaces.`**ManagerInterface**
This is an interface for coverage and dataset series managers. These managers shall facilitate registration of data in the database providing an easy-to-use interface for application programmers.

Managers are bound to a certain resource type, e.g. a DatasetSeries or a RectifiedStitchedMosaic. It suffices to have one manager per resource type as it can be invoked for many objects of this type.

[66] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/geos/#django.contrib.gis.geos.GEOSGeometry

acquireID(*obj_id=None*, *fail=False*)
This method shall acquire a valid and unique object ID and return it. The caller can provide a suggestion `obj_id`. In this case, the method shall try to acquire this object ID. The optional `fail` argument determines what the method shall do in case it cannot acquire the given `obj_id`. If it is set to `True`, the method shall raise an exception, otherwise it shall degrade gracefully returning a newly generated object ID.

The implementation should be able to guarantee that the acquired ID cannot be used by other threads of execution unless it is released and left unused. If it cannot assure this, the deviation shall be documented with a warning.

releaseID(*obj_id*)
This method shall release an object ID `obj_id` that has been acquired with `acquireID()` (page 202) beforehand. In case the object ID has been left unused, it shall be free to be acquired again.

create(*obj_id=None*, ***kwargs*)
This method shall create and return a coverage or dataset series wrapper from the attributes given in `kwargs`. The actual range of keyword arguments accepted may depend on the resource type and the implementation.

If the `obj_id` argument is omitted a new object ID shall be generated using the same mechanism as `acquireID()` (page 202). If the provided object ID is invalid or already in use, appropriate exceptions shall be raised.

update(*obj_id*, ***kwargs*)
This method shall update the coverage or dataset series with ID `obj_id` with new parameters provided as keyword arguments. The actual range of keyword arguments accepted may depend on the resource type and the implementation and should correlate with the arguments accepted by `create()` (page 203) as far as possible.

The method shall return the updated coverage or dataset series wrapper.

It shall raise `NoSuchCoverage` if there is no coverage or dataset series with ID `obj_id`.

delete(*obj_id*)
This method shall delete the coverage or dataset with ID `obj_id`.

It shall raise `NoSuchCoverage` if there is no coverage or dataset series with ID `obj_id`.

class `eoxserver.resources.coverages.interfaces.`**MetadataFormatInterface**
This is a (very basic) interface for metadata formats. So far, it defines only one method:

getName()
This method shall return the name of the format.

getMetadataKeys()
This method shall return a list of metadata keys that are known to the metadata format

getMetadataValues(*keys*, *raw_metadata*)
This method shall return a dictionary of metadata keys and values. The dictionary keys shall correspond to the keys conveyed with the request, the dictionary values shall be the respective metadata values, or `None` if the key is not known or no metadata value is defined for the specific metadata instance.

The `raw_metadata` argument shall point to the raw metadata input. The method shall raise `InternalError` (page 133) if the format cannot decode the raw metadata content, e.g. because it is in the wrong data format.

class `eoxserver.resources.coverages.interfaces.`**RectifiedDatasetInterface**
This class is intended for implementations of RectifiedDataset objects according to the WCS 2.0 EO-AP (EO-WCS). It inherits from `EODatasetInterface` and `RectifiedGridInterface` (page 203).

Interface ID resources.coverages.interfaces.RectifiedDataset

class eoxserver.resources.coverages.interfaces.**RectifiedGridInterface**
This interface defines methods to access rectified grid information, namely the coordinate reference system ID and the geographical extent of the coverage. It is intended to be used as mix-in for coverage interfaces.

Interface ID resources.coverages.interfaces.RectifiedGrid

getSRID()
This method shall return the EPSG SRID of the coverage's coordinate reference system (CRS)

getExtent()
This method shall return the extent of the coverage wrapped by the implementation. The return value is expected to be a 4-tuple of floating point coordinates (minx, miny, maxx, maxy) expressed in the CRS described by the SRID returned with getSRID() (page 204).

class eoxserver.resources.coverages.interfaces.**RectifiedStitchedMosaicInterface**
This class is intended for implementations of Rectified Stitched Mosaic objects according to WCS 2.0 EO-AP (EO-WCS). It inherits from EOCoverageInterface (page 200), RectifiedGridInterface (page 203) and TileIndexInterface (page 204).

Interface ID resources.coverages.interfaces.RectifiedStitchedMosaic

getDataDirs:
This method shall return a list of directories which hold the stitched mosaic data.

getImagePattern()
This method shall return the filename pattern for image files to be included in the stitched mosaic.

class eoxserver.resources.coverages.interfaces.**ReferenceableDatasetInterface**
This class is intended for implementations of RectifiedDataset objects according to the WCS 2.0 EO-AP (EO-WCS). It inherits from EODatasetInterface and ReferenceableGridInterface (page 204).

Note: the design of this interface is still TBD

Interface ID resources.coverages.interfaces.ReferenceableDataset

class eoxserver.resources.coverages.interfaces.**ReferenceableGridInterface**
This interface defines methods for access to referenceable grid information.

Interface ID resources.coverages.interfaces.ReferenceableGrid

getSRID()
This method shall return the EPSG SRID of the coordinate reference system (CRS) of the coverages tie-points.

getExtent()
This method shall return the extent of the coverage wrapped by the implementation. The return value is expected to be a 4-tuple of floating point coordinates (minx, miny, maxx, maxy) expressed in the CRS described by the SRID returned with getSRID() (page 204).

class eoxserver.resources.coverages.interfaces.**TileIndexInterface**
This interface provides the methods necessary to access tile index information for coverages.

Interface ID resources.coverages.interfaces.TileIndex

getShapeFilePath()
This method shall return the path to the shape file that holds information about the tiles the coverage is split up into.

Module eoxserver.resources.coverages.managers

Table of Contents

- Module eoxserver.resources.coverages.managers (page 204)
  - Coverage ID Manager (page 205)
  - Wrapper Manager Implementations (page 206)
    - Rectified Dataset Manager (page 206)
    - Referenceable Dataset Manager (page 208)
    - Rectified Stitched Mosaic Manager (page 209)
    - Dataset Series Manager (page 211)
  - Abstract Manager (page 213)

This module implements variaous managers providing API for operation over the stored datasets. For details of the provided functionality see the documentation of the individual manager classes.

Coverage ID Manager

class `eoxserver.resources.coverages.managers.`**`CoverageIdManager`**

Manager for Coverage IDs. The purpose of this manager class is to help:

- During registration of a new EO-entities/coverage when the uniqueness of the ID must be guarantied. Further, the manager provies means for time limitted reservation (booking) of IDs preventing any parallel process to `steal` the ID while the new coverage is being registered.
- During inspection of an existing ID. The manager determines wehther the inspected ID belongs to an existing EO-entity/coverage or is reserved for a new one. Further, it helps to determine type of the EO-entity/coverage associated to it so that a proper specific manager class can be selected for further action.

Note: EOIDs of DatasetSeries are now included. The name `CoverageIdManager` is therefore misleading as the EO-IDs are involved in the checks.

`available`(*coverage_id*)

> **Warning:** This method has been deprecated. Use `isAvailable()` (page 206) instead.

`check`(*coverage_id*)

> **Warning:** This method has been deprecated. Use `isUsed()` (page 206) instead.

`getAllReservedIds`()

Returns a list of all reserved IDs associated to a specific request ID.

`getCoverageIds`(*request_id*)

> **Warning:** This method has been deprecated. Use `getReservedIds()` (page 205) instead.

`getCoverageType`(*coverage_id*)

> **Warning:** This method has been deprecated. Use `getType()` (page 205) instead.

`getRequestId`(*coverage_id*)

Returns the request ID associated with a `ReservedCoverageIdRecord` or *None* if the no record with that ID is available.

`getReservedIds`(*request_id*)

Returns a list of all reserved IDs associated to a specific request ID.

getType(*coverage_id*)
Returns string, type name of the entity identified by the given ID. In case there is no entity corresponding to the given ID None is returned.

Possible return values are: None, 'PlainCoverage', 'RectifiedDataset', 'ReferenceableDataset', 'RectifiedStitchedMosaic', 'DatasetSeries', and 'Reserved'

isAvailable(*coverage_id*)
Returns a boolean value, indicating if the `coverage_id` is identifier of an existing entity (coverage, eo-dataset, rs-mosaic or ds-series) or it is a reserved ID.

Note: The check also involves EO-IDs!

---

isReserved(*coverage_id*)
Returns a boolean value, indicating if the `coverage_id` is reserved for an entity being currently created.

isUsed(*coverage_id*)
Returns a boolean value, indicating if the `coverage_id` is identifier of an existing entity (coverage, eo-dataset, rs-mosaic or ds-series).

Note: The check also involves EO-IDs!

---

release(*coverage_id*, *fail=False*)
Releases a previously reserved `coverage_id`.

If `fail` is set to `True`, an exception is raised when the ID was not previously reserved.

reserve(*coverage_id*, *request_id=None*, *until=None*)
Tries to reserve a `coverage_id` until a given datetime. If `until` is omitted, the configuration value `resources.coverages.coverage_id.reservation_time` is used.

If the ID is already reserved and the `resource_id` is not equal to the reserved `resource_id`, a `CoverageIdReservedError` is raised. If the ID is already taken by an existing coverage a `CoverageIdInUseError` is raised. These exceptions are sub-classes of `CoverageIdError`.

Wrapper Manager Implementations

Rectified Dataset Manager

class `eoxserver.resources.coverages.managers.`**RectifiedDatasetManager**
Coverage Manager for *RectifiedDatasets*. The following parameters can be used for the `create()` (page 213) and `update()` (page 213) methods.

To define the data and metadata location, the `location` and `md_location` parameters can be used, where the value has to implement the `LocationInterface` (page 237). Alternatively `local_path` and `md_local_path` can be used to define local locations. For when the data and metadata is located on an FTP server, use `remote_path` and `md_remote_path` instead, which also requires the `ftp_host` parameter (`ftp_port`, `ftp_user` and `ftp_passwd` are optional). When the data is located in a rasdaman database use the `collection` and `ras_host` parameters. `oid`, `ras_port`, `ras_user`, `ras_passwd`, and `ras_db` can be used to further specify the location. Currently, these parameters can only be used within the `create()` (page 213) method and not within the `update()` (page 213) method

To specify geospatial metadata use the `geo_metadata` parameter, which has to be an instance of `GeospatialMetadata`. Optionally `default_srid` can be used to declare a default SRID. When updating, it has to be placed within the `set` dict.

To specify earth observation related metadata use the `eo_metadata` parameter which has to be of the type `EOMetadata` (page 214). When updating, it has to be placed within the `set` dict.

The mandatory parameter `range_type_name` states which range type this coverage is using.

If the created dataset shall be inserted into a *DatasetSeries* or *RectifiedStitchedMosaic* a wrapper instance can be passed with the `container` parameter. Alternatively you can use the `container_ids` parameter, passing a list of IDs referencing either *DatasetSeries* or *RectifiedStitchedMosaics*. When used in the context of an `update()` (page 213), both parameters can be placed within the `link` or the `unlink` dict, to either add or remove a reference to the container.

Additional metadata can be added with the `abstract`, `title`, and `keywords` parameters.

For additional `set` parameters for the `update()` (page 213) method please refer to the `FIELDS` (page 219) attribute of the according wrapper.

check_id(*obj_id*)
Check whether the `obj_id` identifies an existing rectified dataset.

Return type boolean

create(*obj_id=None*, *request_id=None*, ***kwargs*)
Creates a new instance of the underlying type and returns an according wrapper object. The optional parameter `obj_id` is used as CoverageID/EOID and a UUID is generated automatically when omitted.

If the ID was previously reserved by a specific `request_id` this parameter must be set.

The other parameters depend on the actual coverage manager type and will be documented there.

If the given ID is already in use, an `CoverageIdInUseError` exception is raised. If the ID is already reserved by another `request_id`, an `CoverageIdReservedError` is raised. These exceptions are sub-classes of `CoverageIdError`.

Parameters

- **obj_id** (*string*[67]) – the ID (CoverageID or EOID) of the object to be created
- **request_id** (*string*[68]) – an optional request ID for the acquisition of the CoverageID/EOID.
- **kwargs** – the arguments

Return type a wrapper of the created object

delete(*obj_id*)
Remove a rectified dataset identified by the `obj_id` parameter.

Parameters **obj_id** – the ID (CoverageID or EOID) of the object to be deleted

Return type no output returned

get_all_ids()
Get CoverageIDs of all registered rectified datasets.

Return type list of CoverageIDs (strings)

is_automatic(*obj_id*)
For the dataset identified by the `obj_id` parameter return value of the `automatic` boolean flag. Returns

Parameters **obj_id** – the ID (CoverageID or EOID) of the object to be deleted

Return type boolean value, `True` if the dataset is automatic

update(*obj_id*, *link=None*, *unlink=None*, *set=None*)
Updates the coverage/dataset series identified by `obj_id`. The `link` and `unlink` dicts are used to add or remove references to other objects, whereas the `set` dict values are used to set attributes of the objects. This can be either a set of values (like `geo_metadata` or `eo_metadata`) or single values as defined in the `FIELDS` dict of the according wrapper.

For all supported attributes please refer to the actually used manager.

[67] http://docs.python.org/2.7/library/string.html#string
[68] http://docs.python.org/2.7/library/string.html#string

Parameters

- **obj_id** (*string*[69]) – the ID (CoverageID or EOID) of the object to be updated
- **link** (*dict or None*) – objects to be linked with
- **unlink** (*dict or None*) – objects to be unlinked
- **set** (*dict or None*) – attributes to be set

Return type a wrapper of the altered object

Referenceable Dataset Manager

class `eoxserver.resources.coverages.managers.`**`ReferenceableDatasetManager`**

Coverage Manager for *ReferenceableDatasets*.

Sorry, but no one has bothered to document this class yet.

`check_id`(*obj_id*)

Check whether the `obj_id` identifies an existing referenceable dataset.

Return type boolean

`create`(*obj_id=None*, *request_id=None*, ***kwargs*)

Creates a new instance of the underlying type and returns an according wrapper object. The optional parameter `obj_id` is used as CoverageID/EOID and a UUID is generated automatically when omitted.

If the ID was previously reserved by a specific `request_id` this parameter must be set.

The other parameters depend on the actual coverage manager type and will be documented there.

If the given ID is already in use, an `CoverageIdInUseError` exception is raised. If the ID is already reserved by another `request_id`, an `CoverageIdReservedError` is raised. These exceptions are sub-classes of `CoverageIdError`.

Parameters

- **obj_id** (*string*[70]) – the ID (CoverageID or EOID) of the object to be created
- **request_id** (*string*[71]) – an optional request ID for the acquisition of the CoverageID/EOID.
- **kwargs** – the arguments

Return type a wrapper of the created object

`delete`(*obj_id*)

Remove a referenceable dataset identified by the `obj_id` parameter.

Parameters **obj_id** – the ID (CoverageID or EOID) of the object to be deleted

Return type no output returned

`get_all_ids`()

Get CoverageIDs of all registered referenceable datasets.

Return type list of CoverageIDs (strings)

`is_automatic`(*obj_id*)

For the dataset identified by the `obj_id` parameter return value of the `automatic` boolean flag. Returns

Parameters **obj_id** – the ID (CoverageID or EOID) of the object to be deleted

Return type boolean value, `True` if the dataset is automatic

[69] http://docs.python.org/2.7/library/string.html#string
[70] http://docs.python.org/2.7/library/string.html#string
[71] http://docs.python.org/2.7/library/string.html#string

update(*obj_id*, *link=None*, *unlink=None*, *set=None*)

Updates the coverage/dataset series identified by `obj_id`. The `link` and `unlink` dicts are used to add or remove references to other objects, whereas the `set` dict values are used to set attributes of the objects. This can be either a set of values (like `geo_metadata` or `eo_metadata`) or single values as defined in the `FIELDS` dict of the according wrapper.

For all supported attributes please refer to the actually used manager.

Parameters

- **obj_id** (*string*[72]) – the ID (CoverageID or EOID) of the object to be updated
- **link** (*dict or None*) – objects to be linked with
- **unlink** (*dict or None*) – objects to be unlinked
- **set** (*dict or None*) – attributes to be set

Return type a wrapper of the altered object

Rectified Stitched Mosaic Manager

class `eoxserver.resources.coverages.managers.`**RectifiedStitchedMosaicManager**

Coverage Manager for *RectifiedStitchedMosaics*

To add data sources to the `RectifiedStitchedMosaic` at the time it is created the `data_sources` and `data_dirs` parameters can be used. The `data_sources` parameter shall be a list of objects implementing the `DataSourceInterface` (page 200). Alternatively the `data_dirs` parameter shall be a list of dictionaries consisting of the following arguments:

- `search_pattern`: a regular expression to specify what files in the directory are considered as data files.
- `path`: for local or FTP data sources, this parameter shall be a path to a valid directory, containing the data files.
- `type`: defines the type of the location describing the data source. This can either be *local* or *remote*.

These parameters can also be used in the context of an `update()` (page 213) within the *link* or *unlink* dict.

To specify geospatial metadata use the `geo_metadata` parameter, which has to be an instance of `GeospatialMetadata`. Optionally `default_srid` can be used to declare a default SRID. When updating, it has to be placed within the `set` dict.

To specify earth observation related metadata use the `eo_metadata` parameter which has to be of the type `EOMetadata` (page 214). When updating, it has to be placed within the `set` dict.

The mandatory parameter `range_type_name` states which range type this coverage is using.

If the created dataset shall be inserted into a *DatasetSeries* or *RectifiedStitchedMosaic* a wrapper instance can be passed with the `container` parameter. Alternatively you can use the `container_ids` parameter, passing a list of IDs referencing either *DatasetSeries* or *RectifiedStitchedMosaics*. These parameters can also be used in the context of an `update()` (page 213) within the *link* or *unlink* dict.

Additional metadata can be added with the `abstract`, `title`, and `keywords` parameters.

For additional `set` parameters for the `update()` (page 213) method please refer to the `FIELDS` (page 224) attribute of the according wrapper.

synchronize(*obj_id*)

This method synchronizes a `RectifiedStitchedMosaicRecord` identified by the `obj_id` with the file system. It does three tasks:

- It scans through all directories specified by its data sources and checks if data files exist which do not yet have an according record. For each, a `RectifiedDatasetRecord` is created and linked with the `RectifiedStitchedMosaicRecord`. Also all existing, but previously not contained datasets are linked to the *Rectified Stitched Mosaic*.

[72] http://docs.python.org/2.7/library/string.html#string

- All contained instances of `RectifiedDatasetRecord` are checked if their data file still exists. If not, the according record is unlinked from the *Rectified Stitched Mosaic* and deleted.
- All instances of `RectifiedDatasetRecord` associated with the `RectifiedStitchedMosaicRecord` which are not referenced by a data source anymore are unlinked from the *Rectified Stitched Mosaic*.

delete(*obj_id*)
This deletes a *RectifiedDataset* record specified by its `obj_id`. If no coverage with this ID can be found, an `NoSuchCoverage` exception will be raised.

check_id(*obj_id*)
Check whether the `obj_id` identifies an existing rectified stitched mosaic.

Return type boolean

create(*obj_id=None*, *request_id=None*, ***kwargs*)
Creates a new instance of the underlying type and returns an according wrapper object. The optional parameter `obj_id` is used as CoverageID/EOID and a UUID is generated automatically when omitted.

If the ID was previously reserved by a specific `request_id` this parameter must be set.

The other parameters depend on the actual coverage manager type and will be documented there.

If the given ID is already in use, an `CoverageIdInUseError` exception is raised. If the ID is already reserved by another `request_id`, an `CoverageIdReservedError` is raised. These exceptions are sub-classes of `CoverageIdError`.

Parameters

- **obj_id** (*string*[73]) – the ID (CoverageID or EOID) of the object to be created
- **request_id** (*string*[74]) – an optional request ID for the acquisition of the CoverageID/EOID.
- **kwargs** – the arguments

Return type a wrapper of the created object

delete(*obj_id*)
Remove a referenceable dataset identified by the `obj_id` parameter.

Parameters **obj_id** – the ID (CoverageID or EOID) of the object to be deleted

Return type no output returned

get_all_ids()
Get CoverageIDs of all registered rectified stitched mosaics.

Return type list of CoverageIDs (strings)

synchronize(*obj_id*)
This method synchronizes a `RectifiedStitchedMosaicRecord` identified by the `obj_id` with the file system. It does three tasks:

- It scans through all directories specified by its data sources and checks if data files exist which do not yet have an according record. For each, a `RectifiedDatasetRecord` is created and linked with the `RectifiedStitchedMosaicRecord`. Also all existing, but previously not contained datasets are linked to the *Rectified Stitched Mosaic*.
- All contained instances of `RectifiedDatasetRecord` are checked if their data file still exists. If not, the according record is unlinked from the *Rectified Stitched Mosaic* and deleted.
- All instances of `RectifiedDatasetRecord` associated with the `RectifiedStitchedMosaicRecord` which are not referenced by a data source anymore are unlinked from the *Rectified Stitched Mosaic*.

[73] http://docs.python.org/2.7/library/string.html#string
[74] http://docs.python.org/2.7/library/string.html#string

Parameters **obj_id** – the ID (CoverageID or EOID) of the object to be synchronised

Return type no output returned

update(*obj_id*, *link=None*, *unlink=None*, *set=None*)

Updates the coverage/dataset series identified by `obj_id`. The `link` and `unlink` dicts are used to add or remove references to other objects, whereas the `set` dict values are used to set attributes of the objects. This can be either a set of values (like `geo_metadata` or `eo_metadata`) or single values as defined in the `FIELDS` dict of the according wrapper.

For all supported attributes please refer to the actually used manager.

Parameters

- **obj_id** (*string*[75]) – the ID (CoverageID or EOID) of the object to be updated
- **link** (*dict or None*) – objects to be linked with
- **unlink** (*dict or None*) – objects to be unlinked
- **set** (*dict or None*) – attributes to be set

Return type a wrapper of the altered object

Dataset Series Manager

class `eoxserver.resources.coverages.managers.`**DatasetSeriesManager**

This manager handles interactions with `DatasetSeries`.

If the `obj_id` argument is omitted a new object ID shall be generated using the same mechanism as `acquireID()`. If the provided object ID is invalid or already in use, appropriate exceptions shall be raised.

To add data sources to the `DatasetSeries` at the time it is created the `data_sources` and `data_dirs` parameters can be used. The `data_sources` parameter shall be a list of objects implementing the :class:`~.DataSourceInterface`. Alternatively the `data_dirs` parameter shall be a list of dictionaries consisting of the following arguments:

- `search_pattern`: a regular expression to specify what files in the directory are considered as data files.
- `path`: for local or FTP data sources, this parameter shall be a path to a valid directory, containing the data files.
- `type`: defines the type of the location describing the data source. This can either be *local* or *remote*.

These parameters can also be used in the context of an `update()` (page 213) within the *link* or *unlink* dict.

To specify earth observation related metadata use the `eo_metadata` parameter which has to be of the type `EOMetadata` (page 214). When updating, it has to be placed within the `set` dict.

For additional `set` parameters for the `update()` (page 213) method please refer to the `FIELDS` (page 226) attribute of the according wrapper.

synchronize(*obj_id*)

This method synchronizes a `DatasetSeriesRecord` identified by the `obj_id` with the file system. It does three tasks:

- It scans through all directories specified by its data sources and checks if data files exist which do not yet have an according record. For each, a `RectifiedDatasetRecord` is created and linked with the `DatasetSeriesRecord`. Also all existing, but previously not contained datasets are linked to the *Dataset Series*.
- All contained instances of `RectifiedDatasetRecord` are checked if their data file still exists. If not, the according record is unlinked from the *Dataset Series* and deleted.

[75] http://docs.python.org/2.7/library/string.html#string

- All instances of `RectifiedDatasetRecord` associated with the `DatasetSeriesRecord` which are not referenced by a data source anymore are unlinked from the *Dataset Series*.

delete(*obj_id*)
This deletes a *RectifiedDataset* record specified by its `obj_id`. If no coverage with this ID can be found, an `NoSuchCoverage` exception will be raised.

check_id(*obj_id*)
Check whether the `obj_id` identifies an existing dataset series.

Return type boolean

create(*obj_id=None*, *request_id=None*, ***kwargs*)
Creates a new instance of the underlying type and returns an according wrapper object. The optional parameter `obj_id` is used as CoverageID/EOID and a UUID is generated automatically when omitted.

If the ID was previously reserved by a specific `request_id` this parameter must be set.

The other parameters depend on the actual coverage manager type and will be documented there.

If the given ID is already in use, an `CoverageIdInUseError` exception is raised. If the ID is already reserved by another `request_id`, an `CoverageIdReservedError` is raised. These exceptions are sub-classes of `CoverageIdError`.

Parameters

- **obj_id** (*string*[76]) – the ID (CoverageID or EOID) of the object to be created
- **request_id** (*string*[77]) – an optional request ID for the acquisition of the CoverageID/EOID.
- **kwargs** – the arguments

Return type a wrapper of the created object

delete(*obj_id*)
Remove a dataset series identified by the `obj_id` parameter.

Parameters **obj_id** – the EOID of the object to be deleted

Return type no output returned

get_all_ids()
Get EOIDs of all registered dataset series.

Return type list of EOIDs (strings)

synchronize(*obj_id*)
Synchronise a dataset series identified by the `obj_id` parameter.

Parameters **obj_id** – the ID (EOID) of the object to be synchronised

Return type no output returned

update(*obj_id*, *link=None*, *unlink=None*, *set=None*)
Updates the coverage/dataset series identified by `obj_id`. The `link` and `unlink` dicts are used to add or remove references to other objects, whereas the `set` dict values are used to set attributes of the objects. This can be either a set of values (like `geo_metadata` or `eo_metadata`) or single values as defined in the `FIELDS` dict of the according wrapper.

For all supported attributes please refer to the actually used manager.

Parameters

- **obj_id** (*string*[78]) – the ID (CoverageID or EOID) of the object to be updated

[76] http://docs.python.org/2.7/library/string.html#string
[77] http://docs.python.org/2.7/library/string.html#string
[78] http://docs.python.org/2.7/library/string.html#string

- **link** (*dict or None*) – objects to be linked with
- **unlink** (*dict or None*) – objects to be unlinked
- **set** (*dict or None*) – attributes to be set

Return type a wrapper of the altered object

Abstract Manager

class `eoxserver.resources.coverages.managers.`**`BaseManager`**

`create`(*obj_id=None*, *request_id=None*, ***kwargs*)

Creates a new instance of the underlying type and returns an according wrapper object. The optional parameter `obj_id` is used as CoverageID/EOID and a UUID is generated automatically when omitted.

If the ID was previously reserved by a specific `request_id` this parameter must be set.

The other parameters depend on the actual coverage manager type and will be documented there.

If the given ID is already in use, an `CoverageIdInUseError` exception is raised. If the ID is already reserved by another `request_id`, an `CoverageIdReservedError` is raised. These exceptions are sub-classes of `CoverageIdError`.

Parameters

- **obj_id** (*string*[79]) – the ID (CoverageID or EOID) of the object to be created
- **request_id** (*string*[80]) – an optional request ID for the acquisition of the CoverageID/EOID.
- **kwargs** – the arguments

Return type a wrapper of the created object

`update`(*obj_id*, *link=None*, *unlink=None*, *set=None*)

Updates the coverage/dataset series identified by `obj_id`. The `link` and `unlink` dicts are used to add or remove references to other objects, whereas the `set` dict values are used to set attributes of the objects. This can be either a set of values (like `geo_metadata` or `eo_metadata`) or single values as defined in the `FIELDS` dict of the according wrapper.

For all supported attributes please refer to the actually used manager.

Parameters

- **obj_id** (*string*[81]) – the ID (CoverageID or EOID) of the object to be updated
- **link** (*dict or None*) – objects to be linked with
- **unlink** (*dict or None*) – objects to be unlinked
- **set** (*dict or None*) – attributes to be set

Return type a wrapper of the altered object

Module eoxserver.resources.coverages.metadata

This module contains the implementation of basic XML EO metadata formats and EO metadata objects.

[79] http://docs.python.org/2.7/library/string.html#string
[80] http://docs.python.org/2.7/library/string.html#string
[81] http://docs.python.org/2.7/library/string.html#string

class eoxserver.resources.coverages.metadata.**DatasetMetadataFileReader**
This is an implementation of EOMetadataReaderInterface (page 202) for local dataset files, i.e. resources.coverages.interfaces.location_type is local and resources.coverages.interfaces.encoding_type is dataset.

readEOMetadata(*location*)
Returns an EOMetadata (page 214) object for the dataset file at the given local path. Raises InternalError (page 133) if the location is not a path on the local file system or DataAccessError if it cannot be opened. MetadataException is raised if the file content is not valid or if the metadata format is unknown.

class eoxserver.resources.coverages.metadata.**EOMetadata**(*eo_id*, *begin_time*, *end_time*, *footprint*, *md_format=None*, *raw_metadata=None*)
This is an implementation of GenericEOMetadataInterface (page 202). It is an object containing the basic set of EO Metadata required by EOxServer. Additional metadata is available using the generic metadata access methods.

Instances of this object are returned by metadata format implementations.

getBeginTime()
Returns the acquisition begin time as datetime.datetime[82] object.

getEOID()
Returns the EO ID of the object.

getEndTime()
Returns the acquisition end time as datetime.datetime[83] object.

getFootprint()
Returns the acquisition footprint as django.contrib.gis.geos.GEOSGeometry[84] object.

getMetadataFormat()
Returns the metadata format object, i.e. an implementation of EOMetadataFormatInterface (page 201) if one was defined when creating the object, None otherwise.

getMetadataKeys()
Returns the keys of the metadata key-value-pairs that can be retrieved from this instance or an empty list if no metadata format has been specified that can decode the raw metadata.

getMetadataValues(*keys*)
Returns a dictionary of metadata key-value-pairs for the given keys. If there is no metadata format and/or no raw metadata object defined for the instance a dictionary mapping the keys to None is returned.

class eoxserver.resources.coverages.metadata.**EOOMFormat**
This is a basic implementation of the OGC (and ESA HMA) EO O&M metadata format.

getName()
Returns "eogml".

test(*test_params*)
This method is required by the Registry (page 148). It tests whether XML input can be interpreted as EOxServer native XML. It expects one dictionary entry root_name in the test_params dictionary. It will raise InternalError (page 133) if it is missing.

The method will return True if the root_name is "Metadata", False otherwise.

class eoxserver.resources.coverages.metadata.**EnvisatDatasetMetadataFormat**
Metadata format for ENVISAT datasets.

[82] http://docs.python.org/2.7/library/datetime.html#datetime.datetime
[83] http://docs.python.org/2.7/library/datetime.html#datetime.datetime
[84] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/geos/#django.contrib.gis.geos.GEOSGeometry

getMetadataKeys()
Returns the keys for key-value-pair metadata access.

test(*test_params*)
This metadata format is applicable, if all metadata tags (MPH_PRODUCT, MPH_SENSING_START, MPH_SENSING_STOP) are found within the metadata entries of the dataset and the dataset contains at least one GCP.

class eoxserver.resources.coverages.metadata.**MetadataFormat**
Abstract base class for metada formats. A blueprint for implementing MetadataFormatInterface (page 203).

getMetadataKeys()
Not implemented. Raises InternalError (page 133).

getMetadataValues(*keys*, *raw_metadata*)
Not implemented. Raises InternalError (page 133).

getName()
Not implemented. Raises InternalError (page 133).

test(*test_params*)
Not implemented. Raises InternalError (page 133).

class eoxserver.resources.coverages.metadata.**NativeMetadataFormat**
This is an implementation of an EOxServer native metadata format. This format was designed to be as simple as possible and is intended for use in a testing environment. A template XML snippet looks like:

```
<Metadata>
    <EOID>some_unique_eoid</EOID>
    <BeginTime>YYYY-MM-DDTHH:MM:SSZ</BeginTime>
    <EndTime>YYYY-MM-DDTHH:MM:SSZ</EndTime>
    <Footprint>
        <Polygon>
            <Exterior>Mandatory - some_pos_list as all-space-delimited Lat Lon pairs (closed
            [
             <Interior>Optional - some_pos_list as all-space-delimited Lat Lon pairs (closed
             ...
            ]
        </Polygon>
    </Footprint>
</Metadata>
```

getName()
Returns "native".

test(*test_params*)
This method is required by the Registry (page 148). It tests whether XML input can be interpreted as EOxServer native XML. It expects one dictionary entry root_name in the test_params dictionary. It will raise InternalError (page 133) if it is missing.

The method will return True if the root_name is "Metadata", False otherwise.

class eoxserver.resources.coverages.metadata.**NativeMetadataFormatEncoder**(*schemas=None*)
Encodes EO Coverage metadata

class eoxserver.resources.coverages.metadata.**XMLEOMetadataFileReader**
This is an implementation of EOMetadataReaderInterface (page 202) for local XML files, i.e. resources.coverages.interfaces.location_type is local and resources.coverages.interfaces.encoding_type is xml.

readEOMetadata(*location*)
Returns an EOMetadata (page 214) object for the XML file at the given local path. Raises InternalError (page 133) if the location is not a path on the local file system or DataAccessError if it cannot be opened. MetadataException is raised if the file content is not valid XML or if the XML metadata format is unknown.

class eoxserver.resources.coverages.metadata.**XMLEOMetadataFormat**
This is the base class for XML EO Metadata formats implementing EOMetadataFormatInterface (page 201). It adds getEOMetadata() (page 216) to the XMLMetadataFormat (page 216) implementation it inherits from.

getEOMetadata(*raw_metadata*)
This method decodes the raw XML metadata passed to it and returns an EOMetadata (page 214) instance. The method raises InternalError (page 133) if raw_metadata is not a string or MetadataException if it cannot be parsed as valid XML.

class eoxserver.resources.coverages.metadata.**XMLMetadataFormat**
This is a base class for XML based metadata formats. It inherits from MetadataFormat (page 215).

getMetadataKeys()
Returns the keys for key-value-pair metadata access.

getMetadataValues(*keys*, *raw_metadata*)
Returns metadata key-value-pairs for the given keys. The argument raw_metadata is expected to be a string containing valid XML. This method raises InternalError (page 133) if raw_metadata is not a string or MetadataException if it cannot be parsed as valid XML.

Module eoxserver.resources.coverages.rangetype

Table of Contents

- Module eoxserver.resources.coverages.rangetype (page 216)
  - Helper Subroutines (page 216)
  - Range Type Classes (page 216)
    - RangeType (page 216)
    - Band (page 217)
    - NilValue (page 218)

Helper Subroutines

eoxserver.resources.coverages.rangetype.**getAllRangeTypeNames**()
Return a list of identifiers of all registered range-types.

eoxserver.resources.coverages.rangetype.**isRangeTypeName**(*name*)
Check whether there is (True) or is not (False) a registered range-type with given identifier‘‘name‘‘.

eoxserver.resources.coverages.rangetype.**getRangeType**(*name*)
Return RangeType object for given name. The object properties are loaded from the DB. If there is no RangeTypeRecord corresponding to the given name None is returned.

eoxserver.resources.coverages.rangetype.**setRangeType**(*rtype*)
Save range-type record to the DB. The range-type record is created from the rtype which can be either a RangeType object or parsed JSON dictionary.

Range Type Classes

RangeType

class eoxserver.resources.coverages.rangetype.**RangeType**(*name*, *data_type*, *bands=None*)
RangeType contains range type information of a coverage. The constructor accepts the mandatory name and data_type parameters as well as an optional bands parameter. If no bands are specified they shall be added with addBands().

The data_type parameter may be set to one of the following constants defined in osgeo.gdalconst:

- GDT_Byte
- GDT_UInt16
- GDT_Int16
- GDT_UInt32
- GDT_Int32
- GDT_Float32
- GDT_Float64
- GDT_CInt16
- GDT_CInt32
- GDT_CFloat32
- GDT_CFloat64

addBand(*band*)
Append a new band to the band list.

asDict()
return object as a tupe to be passed to JSON serializer

getAllowedValues()
Get interval bounds of the currently used type.

getDataTypeAsString()
Return string representation of the data_type.

getSignificantFigures()
Get significant figures of the currently used type.

Band

class eoxserver.resources.coverages.rangetype.**Band**(*name*, *identifier=''*, *description=''*, *definition='http://opengis.net/def/property/OGC/0/Radiance'*, *nil_values=None*, *uom='W.m-2.sr-1.nm-1'*, *gdal_interpretation=0*)

Band represents a band configuration.

The gdal_interpretation parameter contains the GDAL BandInterpretation value which may be assigned to a band. It may be set to one of the following constants defined in osgeo.gdalconst:

- GCI_Undefined
- GCI_GrayIndex
- GCI_PaletteIndex
- GCI_RedBand
- GCI_GreenBand
- GCI_BlueBand
- GCI_AlphaBand
- GCI_HueBand
- GCI_SaturationBand
- GCI_LightnessBand
- GCI_CyanBand
- GCI_MagentaBand

- `GCI_YellowBand`
- `GCI_BlackBand`

It defaults to `GCI_Undefined`.

`asDict``()`
Return object's data as a dictionary to be passed to a JSON serializer.

`getGDALInterpretationAsString``()`
Return string representation of the `gdal_interpretation`.

NilValue

class `eoxserver.resources.coverages.rangetype.`**`NilValue`**(*reason*, *value*)
This class represents nil values of a coverage band.

The constructor accepts the nil value itself and a reason. The reason shall be one of:

- `http://www.opengis.net/def/nil/OGC/0/inapplicable`
- `http://www.opengis.net/def/nil/OGC/0/missing`
- `http://www.opengis.net/def/nil/OGC/0/template`
- `http://www.opengis.net/def/nil/OGC/0/unknown`
- `http://www.opengis.net/def/nil/OGC/0/withheld`
- `http://www.opengis.net/def/nil/OGC/0/AboveDetectionRange`
- `http://www.opengis.net/def/nil/OGC/0/BelowDetectionRange`

See http://www.opengis.net/def/nil/ for the official description of the meanings of these values.

`asDict``()`
Return object's data as a dictionary to be passed to a JSON serializer.

Module eoxserver.resources.coverages.wrappers

Table of Contents

- Module eoxserver.resources.coverages.wrappers (page 218)
  - Top Level Wrappers (page 218)
    - Rectified Dataset (page 219)
    - Referenceable Datasets (page 221)
    - Rectified Stitched Mosaic (page 224)
    - Dataset Series (page 226)
  - Factory Classes (page 228)
  - Wrappers' Parent Classes (page 232)
  - Wrappers' Mix-In Classes (page 233)

This module provides implementations of coverage interfaces as defined in `eoxserver.resources.coverages.interfaces` (page 198). These classes wrap the resources stored in the database and augment them with additional application logic.

Top Level Wrappers

The top level wrappers are displayed including the inherited members.

Rectified Dataset

class eoxserver.resources.coverages.wrappers.**RectifiedDatasetWrapper**

This is the wrapper for Rectified Datasets. It inherits from EODatasetWrapper (page 232) and RectifiedGridWrapper (page 233). It implements RectifiedDatasetInterface (page 203).

FIELDS

- eo_id: the EO ID of the dataset; value must be a string
- begin_time: the begin time of the eo metadata entry
- end_time: the end time of the eo metadata entry
- footprint: the footprint of the dataset
- srid: the SRID of the dataset's CRS; value must be an integer
- size_x: the width of the coverage in pixels; value must be an integer
- size_y: the height of the coverage in pixels; value must be an integer
- minx: the left hand bound of the dataset's extent; value must be numeric
- miny: the lower bound of the dataset's extent; value must be numeric
- maxx: the right hand bound of the dataset's extent; value must be numeric
- maxy: the upper bound of the dataset's extent; value must be numeric
- visible: the visibility of the coverage (for DescribeCoverage requests); boolean
- automatic: if the dataset was automatically created or by hand; boolean

containedIn(*wrapper*)

Returns True if this Rectified Dataset is contained in the Rectified Stitched Mosaic or Dataset Series specified by its wrapper, False otherwise.

contains(*wrapper*)

Always returns False. A Dataset does not contain other Datasets.

createModel(*params*)

This method shall be used to create models for the concrete coverage type.

deleteModel()

Delete the coverage model.

getAttrField(*attr_name*)

Returns the field name for the attribute named attr_name. An UnknownAttribute (page 134) exception is raised if there is no attribute with the given name.

getAttrNames()

Returns a list of names of the accessible attributes of the resource.

getAttrValue(*attr_name*)

Returns the value of the attribute named attr_name. An UnknownAttribute exception is raised in case there is no attribute with the given name.

getBeginTime()

Returns the acquisition begin time as datetime.datetime[85] object.

getContainerCount()

This method returns the number of Dataset Series and Rectified Stitched Mosaics containing this Rectified Dataset.

getContainers()

This method returns a list of DatasetSeriesWrapper (page 226) and RectifiedStitchedMosaicWrapper (page 224) objects containing this Rectified Dataset, or an empty list.

[85] http://docs.python.org/2.7/library/datetime.html#datetime.datetime

getCoverageId()
Returns the Coverage ID.

getCoverageSubtype()
Returns `RectifiedGridCoverage`.

getData()
Return the data package wrapper associated with the coverage, i.e. an instance of a subclass of `DataPackageWrapper` (page 190).

getDataStructureType()
Returns the data structure type of the underlying data package

getDatasets(*filter_exprs=None*)
This method applies the given filter expressions to the model and returns a list containing the wrapper in case the filters are matched or an empty list otherwise.

getEOCoverageSubtype()
Returns `RectifiedDataset`.

getEOGML()
Returns the EO O&M XML text stored in the metadata.

getEOID()
Returns the EO ID of the object.

getEndTime()
Returns the acquisition end time as `datetime.datetime`[86] object.

getExtent()
Returns the coverage extent as a 4-tuple of floating point coordinates `(minx, miny, maxx, maxy)` expressed in the coverage CRS as defined by the SRID returned by `getSRID()` (page 220).

getFootprint()
Returns the acquisition footprint as `django.contrib.gis.geos.GEOSGeometry`[87] object in the EPSG:4326 CRS.

getId()
This method shall return the model ID, i.e. the content of its `id_field` field. Child classes may override it in order to implement more efficient data access.

getLayerMetadata()
Returns a list of `(metadata_key, metadata_value)` pairs that represent MapServer metadata tags to be attached to MapServer layers.

getLineage()
Returns `None`.

Note: The lineage element has yet to be specified in detail in the WCS 2.0 EO-AP (EO-WCS).

---

getModel()
Returns the model wrapped by this implementation.

getRangeType()
This method returns the range type of the coverage as `RangeType` (page 216) object.

getResolution()
Returns the coverage resolution as a 2-tuple of float values for the x and y axes `(resx, resy)` expressed in the unit of measure of the coverage CRS as defined by the SRID returned by `getSRID()` (page 220).

getSRID()
Returns the SRID of the coverage CRS.

[86] http://docs.python.org/2.7/library/datetime.html#datetime.datetime
[87] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/geos/#django.contrib.gis.geos.GEOSGeometry

getSize()
Returns the pixel size of the dataset as 2-tuple of integers `(size_x, size_y)`.

getType()
Returns `eo.rect_dataset`

getWGS84Extent()
Returns the WGS 84 extent as 4-tuple of floating point coordinates `(minlon, minlat, maxlon, maxlat)`.

isAutomatic()
Returns `True` if the coverage is automatic or `False` otherwise.

matches(*filter_exprs*)
Returns True if the Coverage matches the given filter expressions and False otherwise.

saveModel()
Save the coverage model to the database.

setAttrValue(*attr_name*, *value*)
Sets the value of the attribute named `attr_name` to `value`. An `InternalError` (page 133) is raised if the resource is not mutable.

setModel(*model*)
Use this function to set the coverage model that shall be wrapped.

setMutable(*mutable=True*)
This method sets the mutability status of the resource. It accepts one optional boolean argument `mutable` which defaults to `True`. The mutability status can be set only once for each resource, attempts to change it will cause an `InternalError` (page 133) to be raised.

Referenceable Datasets

class `eoxserver.resources.coverages.wrappers.`**ReferenceableDatasetWrapper**
This is the wrapper for Referenceable Datasets. It inherits from `EODatasetWrapper` (page 232) and `ReferenceableGridWrapper` (page 234).

FIELDS

- `eo_id`: the EO ID of the dataset; value must be a string
- `begin_time`: the begin time of the eo metadata entry
- `end_time`: the end time of the eo metadata entry
- `footprint`: the footprint of the dataset
- `filename`: the path to the dataset; value must be a string
- `metadata_filename`: the path to the accompanying metadata file; value must be a string
- `srid`: the SRID of the dataset's CRS; value must be an integer
- `size_x`: the width of the coverage in pixels; value must be an integer
- `size_y`: the height of the coverage in pixels; value must be an integer
- `minx`: the left hand bound of the dataset's extent; value must be numeric
- `miny`: the lower bound of the dataset's extent; value must be numeric
- `maxx`: the right hand bound of the dataset's extent; value must be numeric
- `maxy`: the upper bound of the dataset's extent; value must be
- `visible`: the `visible` attribute of the dataset; value must be boolean
- `automatic`: the `automatic` attribute of the dataset; value must be boolean

Note: The design of Referenceable Datasets is still TBD.

containedIn(*wrapper*)
This method returns `True` if this Referenceable Dataset is contained in the Dataset Series specified by its `wrapper`, `False` otherwise.

contains(*wrapper*)
Always returns `False`. A Dataset cannot contain other Datasets.

createModel(*params*)
This method shall be used to create models for the concrete coverage type.

deleteModel()
Delete the coverage model.

getAttrField(*attr_name*)
Returns the field name for the attribute named `attr_name`. An `UnknownAttribute` (page 134) exception is raised if there is no attribute with the given name.

getAttrNames()
Returns a list of names of the accessible attributes of the resource.

getAttrValue(*attr_name*)
Returns the value of the attribute named `attr_name`. An `UnknownAttribute` exception is raised in case there is no attribute with the given name.

getBeginTime()
Returns the acquisition begin time as `datetime.datetime`[88] object.

getContainerCount()
This method returns the number of Dataset Series containing this Referenceable Dataset.

getContainers()
This method returns a list of `DatasetSeriesWrapper` (page 226) objects containing this Referenceable Dataset, or an empty list.

getCoverageId()
Returns the Coverage ID.

getCoverageSubtype()
Returns `ReferenceableGridCoverage`.

getData()
Return the data package wrapper associated with the coverage, i.e. an instance of a subclass of `DataPackageWrapper` (page 190).

getDataStructureType()
Returns the data structure type of the underlying data package

getDatasets(*filter_exprs=None*)
This method applies the given filter expressions to the model and returns a list containing the wrapper in case the filters are matched or an empty list otherwise.

getEOCoverageSubtype()
Returns `ReferenceableDataset`.

getEOGML()
Returns the EO O&M XML text stored in the metadata.

getEOID()
Returns the EO ID of the object.

[88] http://docs.python.org/2.7/library/datetime.html#datetime.datetime

getEndTime()
Returns the acquisition end time as `datetime.datetime`[89] object.

getExtent()
Returns the coverage extent as a 4-tuple of floating point coordinates `(minx, miny, maxx, maxy)` expressed in the coverage CRS as defined by the SRID returned by `getSRID()` (page 223).

getFootprint()
Returns the acquisition footprint as `django.contrib.gis.geos.GEOSGeometry`[90] object in the EPSG:4326 CRS.

getId()
This method shall return the model ID, i.e. the content of its `id_field` field. Child classes may override it in order to implement more efficient data access.

getLayerMetadata()
Returns a list of `(metadata_key, metadata_value)` pairs that represent MapServer metadata tags to be attached to MapServer layers.

getLineage()
Returns `None`.

Note: The lineage element has yet to be specified in detail in the WCS 2.0 EO-AP (EO-WCS).

---

getModel()
Returns the model wrapped by this implementation.

getRangeType()
This method returns the range type of the coverage as `RangeType` (page 216) object.

getSRID()
Returns the SRID of the coverage CRS.

getSize()
Returns the pixel size of the dataset as 2-tuple of integers `(size_x, size_y)`.

getType()
Returns `eo.ref_dataset`

getWGS84Extent()
Returns the WGS 84 extent as 4-tuple of floating point coordinates `(minlon, minlat, maxlon, maxlat)`.

isAutomatic()
Returns `True` if the coverage is automatic or `False` otherwise.

matches(*filter_exprs*)
Returns True if the Coverage matches the given filter expressions and False otherwise.

saveModel()
Save the coverage model to the database.

setAttrValue(*attr_name*, *value*)
Sets the value of the attribute named `attr_name` to `value`. An `InternalError` (page 133) is raised if the resource is not mutable.

setModel(*model*)
Use this function to set the coverage model that shall be wrapped.

setMutable(*mutable=True*)
This method sets the mutability status of the resource. It accepts one optional boolean argument `mutable` which defaults to `True`. The mutability status can be set only once for each resource, attempts to change it will cause an `InternalError` (page 133) to be raised.

[89] http://docs.python.org/2.7/library/datetime.html#datetime.datetime
[90] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/geos/#django.contrib.gis.geos.GEOSGeometry

Rectified Stitched Mosaic

class eoxserver.resources.coverages.wrappers.**RectifiedStitchedMosaicWrapper**

This is the wrapper for Rectified Stitched Mosaics. It inherits from EOCoverageWrapper (page 232) and RectifiedGridWrapper (page 233). It implements RectifiedStitchedMosaicInterface (page 204).

FIELDS

- eo_id: the EO ID of the mosaic; value must be a string
- begin_time: the begin time of the eo metadata entry
- end_time: the end time of the eo metadata entry
- footprint: the footprint of the mosaic
- srid: the SRID of the mosaic's CRS; value must be an integer
- size_x: the width of the coverage in pixels; value must be an integer
- size_y: the height of the coverage in pixels; value must be an integer
- minx: the left hand bound of the mosaic's extent; value must be numeric
- miny: the lower bound of the mosaic's extent; value must be numeric
- maxx: the right hand bound of the mosaic's extent; value must be numeric
- maxy: the upper bound of the mosaic's extent; value must be numeric

addCoverage(*wrapper*)

Adds a Rectified Dataset specified by its wrapper. An InternalError is raised if the wrapper type is not equal to eo.rect_dataset or if the grids of the dataset is not compatible to the grid of the Rectified Stitched Mosaic.

containedIn(*wrapper*)

This method returns True if this Stitched Mosaic is contained in the Dataset Series specified by its wrapper, False otherwise.

contains(*wrapper*)

This method returns True if the a Rectified Dataset specified by its wrapper is contained within this Stitched Mosaic, False otherwise.

createModel(*params*)

This method shall be used to create models for the concrete coverage type.

deleteModel()

Delete the coverage model.

getAttrField(*attr_name*)

Returns the field name for the attribute named attr_name. An UnknownAttribute (page 134) exception is raised if there is no attribute with the given name.

getAttrNames()

Returns a list of names of the accessible attributes of the resource.

getAttrValue(*attr_name*)

Returns the value of the attribute named attr_name. An UnknownAttribute exception is raised in case there is no attribute with the given name.

getBeginTime()

Returns the acquisition begin time as datetime.datetime[91] object.

getContainerCount()

This method returns the number of Dataset Series containing this Stitched Mosaic.

[91] http://docs.python.org/2.7/library/datetime.html#datetime.datetime

getContainers()
This method returns a list of DatasetSeriesWrapper (page 226) objects containing this Stitched Mosaic or an empty list.

getCoverageId()
Returns the Coverage ID.

getCoverageSubtype()
Returns RectifiedGridCoverage.

getData()
Returns a TileIndexWrapper instance.

getDataDirs()
This method returns a list of directories which hold the stitched mosaic data.

getDataStructureType()
Returns "index".

getDatasets(*filter_exprs=None*)
Returns a list of RectifiedDatasetWrapper (page 219) objects contained in the stitched mosaic wrapped by the implementation. It accepts an optional filter_exprs parameter which is expected to be a list of filter expressions (see module eoxserver.resources.coverages.filters (page 194)) or None. Only the datasets matching the filters will be returned; in case no matching coverages are found an empty list will be returned.

getEOCoverageSubtype()
Returns RectifiedStitchedMosaic.

getEOGML()
Returns the EO O&M XML text stored in the metadata.

getEOID()
Returns the EO ID of the object.

getEndTime()
Returns the acquisition end time as datetime.datetime[92] object.

getExtent()
Returns the coverage extent as a 4-tuple of floating point coordinates (minx, miny, maxx, maxy) expressed in the coverage CRS as defined by the SRID returned by getSRID() (page 226).

getFootprint()
Returns the acquisition footprint as django.contrib.gis.geos.GEOSGeometry[93] object in the EPSG:4326 CRS.

getId()
This method shall return the model ID, i.e. the content of its id_field field. Child classes may override it in order to implement more efficient data access.

getImagePattern()
Returns the filename pattern for image files to be included into the stitched mosaic. The pattern is expressed in the format accepted by fnmatch.fnmatch()[94].

getLayerMetadata()
Returns a list of (metadata_key, metadata_value) pairs that represent MapServer metadata tags to be attached to MapServer layers.

getLineage()
Returns None.

Note: The lineage element has yet to be specified in detail in the WCS 2.0 EO-AP (EO-WCS).

---

[92] http://docs.python.org/2.7/library/datetime.html#datetime.datetime
[93] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/geos/#django.contrib.gis.geos.GEOSGeometry
[94] http://docs.python.org/2.7/library/fnmatch.html#fnmatch.fnmatch

getModel()
Returns the model wrapped by this implementation.

getRangeType()
This method returns the range type of the coverage as RangeType (page 216) object.

getResolution()
Returns the coverage resolution as a 2-tuple of float values for the x and y axes (resx, resy) expressed in the unit of measure of the coverage CRS as defined by the SRID returned by getSRID() (page 226).

getSRID()
Returns the SRID of the coverage CRS.

getShapeFilePath()
Returns the path to the shape file.

getSize()
Returns the pixel size of the mosaic as 2-tuple of integers (size_x, size_y).

getType()
Returns eo.rect_stitched_mosaic

getWGS84Extent()
Returns the WGS 84 extent as 4-tuple of floating point coordinates (minlon, minlat, maxlon, maxlat).

isAutomatic()
Returns True if the coverage is automatic or False otherwise.

matches(*filter_exprs*)
Returns True if the Coverage matches the given filter expressions and False otherwise.

removeCoverage(*wrapper*)
Removes a Rectified Dataset specified by its wrapper. An InternalError is raised if the wrapper type is not equal to eo.rect_dataset.

saveModel()
Save the coverage model to the database.

setAttrValue(*attr_name*, *value*)
Sets the value of the attribute named attr_name to value. An InternalError (page 133) is raised if the resource is not mutable.

setModel(*model*)
Use this function to set the coverage model that shall be wrapped.

setMutable(*mutable=True*)
This method sets the mutability status of the resource. It accepts one optional boolean argument mutable which defaults to True. The mutability status can be set only once for each resource, attempts to change it will cause an InternalError (page 133) to be raised.

Dataset Series

class eoxserver.resources.coverages.wrappers.**DatasetSeriesWrapper**
This is the wrapper for Dataset Series. It inherits from EOMetadataWrapper (page 233). It implements DatasetSeriesInterface.

FIELDS

- eo_id: the EO ID of the dataset series; value must be a string
- begin_time: the begin time of the eo metadata entry
- end_time: the end time of the eo metadata entry
- footprint: the footprint of the mosaic

addCoverage(*wrapper*)
Adds the EO coverage of type `res_type` with primary key `res_id` to the dataset series. An `InternalError` is raised if the type cannot be handled by Dataset Series. Supported wrapper types are:

- `eo.rect_dataset`
- `eo.ref_dataset`
- `eo.rect_stitched_mosaic`

contains(*wrapper*)
This method returns `True` if the Dataset Series contains the EO Coverage specifiec by its `wrapper`, `False` otherwise.

createModel(*params*)
This method shall be used to create models for the concrete coverage type.

deleteModel()
Delete the coverage model.

getAttrField(*attr_name*)
Returns the field name for the attribute named `attr_name`. An `UnknownAttribute` (page 134) exception is raised if there is no attribute with the given name.

getAttrNames()
Returns a list of names of the accessible attributes of the resource.

getAttrValue(*attr_name*)
Returns the value of the attribute named `attr_name`. An `UnknownAttribute` exception is raised in case there is no attribute with the given name.

getBeginTime()
Returns the acquisition begin time as `datetime.datetime`[95] object.

getDataDirs()
This method returns a list of directories which hold the dataset series data.

getDatasets(*filter_exprs=None*)
This method returns a list of RectifiedDataset or ReferenceableDataset wrappers associated with the dataset series. It accepts an optional `filter_exprs` parameter which is expected to be a list of filter expressions (see module `eoxserver.resources.coverages.filters` (page 194)) or `None`. Only the Datasets matching the filters will be returned; in case no matching Datasets are found an empty list will be returned.

getEOCoverages(*filter_exprs=None*)
This method returns a list of EOCoverage wrappers (for datasets and stitched mosaics) associated with the dataset series wrapped by the implementation. It accepts an optional `filter_exprs` parameter which is expected to be a list of filter expressions (see module `eoxserver.resources.coverages.filters` (page 194)) or `None`. Only the EOCoverages matching the filters will be returned; in case no matching coverages are found an empty list will be returned.

getEOGML()
Returns the EO O&M XML text stored in the metadata.

getEOID()
Returns the EO ID of the object.

getEndTime()
Returns the acquisition end time as `datetime.datetime`[96] object.

getFootprint()
Returns the acquisition footprint as `django.contrib.gis.geos.GEOSGeometry`[97] object in

[95] http://docs.python.org/2.7/library/datetime.html#datetime.datetime
[96] http://docs.python.org/2.7/library/datetime.html#datetime.datetime
[97] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/geos/#django.contrib.gis.geos.GEOSGeometry

the EPSG:4326 CRS.

getId()
This method shall return the model ID, i.e. the content of its `id_field` field. Child classes may override it in order to implement more efficient data access.

getImagePattern()
Returns the filename pattern for image files to be included into the stitched mosaic. The pattern is expressed in the format accepted by `fnmatch.fnmatch()`[98].

getLayerMetadata()
Returns a list of `(metadata_key, metadata_value)` pairs that represent MapServer metadata tags to be attached to MapServer layers.

getModel()
Returns the model wrapped by this implementation.

getType()
Returns `"eo.dataset_series"`.

getWGS84Extent()
Returns the WGS 84 extent as 4-tuple of floating point coordinates `(minlon, minlat, maxlon, maxlat)`.

removeCoverage(*wrapper*)
Removes the EO coverage specified by its `wrapper` from the dataset series. An `InternalError` is raised if the type cannot be handled by Dataset Series. Supported wrapper types are:

- `eo.rect_dataset`
- `eo.ref_dataset`
- `eo.rect_stitched_mosaic`

saveModel()
Save the coverage model to the database.

setAttrValue(*attr_name*, *value*)
Sets the value of the attribute named `attr_name` to `value`. An `InternalError` (page 133) is raised if the resource is not mutable.

setModel(*model*)
Use this function to set the coverage model that shall be wrapped.

setMutable(*mutable=True*)
This method sets the mutability status of the resource. It accepts one optional boolean argument `mutable` which defaults to `True`. The mutability status can be set only once for each resource, attempts to change it will cause an `InternalError` (page 133) to be raised.

Factory Classes

class `eoxserver.resources.coverages.wrappers.`**EOCoverageFactory**

create(***kwargs*)
This method creates a resource according to the given parameters and returns it to the caller. It accepts one mandatory and two optional parameters:

- `subj_id`: the id of the calling component (optional)
- `impl_id`: the implementation ID of the resource to be created (mandatory)
- `params`: a dictionary of parameters to initialize the resource with; the format of this dictionary is specific to the resource class

[98] http://docs.python.org/2.7/library/fnmatch.html#fnmatch.fnmatch

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

`delete`(***kwargs*)
This method deletes a selection of resources. It accepts the following parameters:

- `subj_id`: the id of the calling component
- `obj_id`: the resource ID of the resource
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

The `obj_id` argument and the `impl_ids` and `filter_exprs` arguments on the other hand are mutually exclusive. `InternalError` is raised if these conditions are not met.

`exists`(***kwargs*)
Returns `True` if there are resources matching the given criteria, or `False` otherwise.

- `subj_id`: the id of the calling component
- `obj_id`: the id of the requested resource
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources

Note that `filter_exprs` will not be taken into account when `obj_id` is given.

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

`find`(***kwargs*)
Returns a list of resource instances matching the given search criteria. This method accepts three optional arguments:

- `subj_id`: the id of the calling component
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

`get`(***kwargs*)
Returns the resource instance wrapping the resource model defined by the input parameters. This method accepts three optional keyword arguments:

- `subj_id`: the id of the calling component
- `obj_id`: the resource ID of the resource
- `filter_exprs`: a list of filter expressions that define the resource

Note that `obj_id` and `filter_exprs` are mutually exclusive, but exactly one of them must be provided. The `subj_id` argument will be used to check for relations to the resource (not yet implemented).

`getAttrValues`(***kwargs*)
This method returns the values of a given attribute for a selection of resources.

- `subj_id`: the id of the calling component
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources
- `attr_name`: the attribute name (mandatory)

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

Raises `InternalError` (page 133) if the `attr_name` argument is missing, or `UnknownAttribute` (page 134) if the attribute name is not known to a resource.

getIds(***kwargs*)
This method returns the IDs of a selection of resources. It accepts the following parameters:

- `subj_id`: the id of the calling component
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

update(***kwargs*)
This method runs updates on a selection of resources and returns the updated resources. It accepts the following parameters:

- `subj_id`: the id of the calling component
- `obj_id`: the resource ID of the resource
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources
- `attrs`: a dictionary of attribute names and values; the attribute names are specific to the resource classes
- `params`: a dictionary of parameters to update the resource with; the format of this dictionary is specific to the resource classes

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

The `obj_id` argument and the `impl_ids` and `filter_exprs` arguments on the other hand are mutually exclusive. The `attrs` and `params` arguments are mutually exclusive as well, exactly one of them has to be specified. `InternalError` is raised if these conditions are not met.

class `eoxserver.resources.coverages.wrappers.`**DatasetSeriesFactory**

create(***kwargs*)
This method creates a resource according to the given parameters and returns it to the caller. It accepts one mandatory and two optional parameters:

- `subj_id`: the id of the calling component (optional)
- `impl_id`: the implementation ID of the resource to be created (mandatory)
- `params`: a dictionary of parameters to initialize the resource with; the format of this dictionary is specific to the resource class

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

delete(***kwargs*)
This method deletes a selection of resources. It accepts the following parameters:

- `subj_id`: the id of the calling component
- `obj_id`: the resource ID of the resource
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

The `obj_id` argument and the `impl_ids` and `filter_exprs` arguments on the other hand are mutually exclusive. `InternalError` is raised if these conditions are not met.

exists(***kwargs*)
Returns `True` if there are resources matching the given criteria, or `False` otherwise.

- `subj_id`: the id of the calling component
- `obj_id`: the id of the requested resource

- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources

Note that `filter_exprs` will not be taken into account when `obj_id` is given.

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

find(***kwargs*)
Returns a list of resource instances matching the given search criteria. This method accepts three optional arguments:

- `subj_id`: the id of the calling component
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

get(***kwargs*)
Returns the resource instance wrapping the resource model defined by the input parameters. This method accepts three optional keyword arguments:

- `subj_id`: the id of the calling component
- `obj_id`: the resource ID of the resource
- `filter_exprs`: a list of filter expressions that define the resource

Note that `obj_id` and `filter_exprs` are mutually exclusive, but exactly one of them must be provided. The `subj_id` argument will be used to check for relations to the resource (not yet implemented).

getAttrValues(***kwargs*)
This method returns the values of a given attribute for a selection of resources.

- `subj_id`: the id of the calling component
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources
- `attr_name`: the attribute name (mandatory)

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

Raises `InternalError` (page 133) if the `attr_name` argument is missing, or `UnknownAttribute` (page 134) if the attribute name is not known to a resource.

getIds(***kwargs*)
This method returns the IDs of a selection of resources. It accepts the following parameters:

- `subj_id`: the id of the calling component
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

update(***kwargs*)
This method runs updates on a selection of resources and returns the updated resources. It accepts the following parameters:

- `subj_id`: the id of the calling component
- `obj_id`: the resource ID of the resource
- `impl_ids`: the implementation IDs of the resource classes to be taken into account
- `filter_exprs`: a list of filter expressions that constrain the resources

- `attrs`: a dictionary of attribute names and values; the attribute names are specific to the resource classes
- `params`: a dictionary of parameters to update the resource with; the format of this dictionary is specific to the resource classes

The `subj_id` argument will be used to check for relations to the resources (not yet implemented).

The `obj_id` argument and the `impl_ids` and `filter_exprs` arguments on the other hand are mutually exclusive. The `attrs` and `params` arguments are mutually exclusive as well, exactly one of them has to be specified. `InternalError` is raised if these conditions are not met.

Wrappers' Parent Classes

class `eoxserver.resources.coverages.wrappers.`**`EODatasetWrapper`**

This is the base class for EO Dataset wrapper implementations. It inherits from `EOCoverageWrapper` (page 232) and `PackagedDataWrapper` (page 234).

`getDatasets`(*filter_exprs=None*)

This method applies the given filter expressions to the model and returns a list containing the wrapper in case the filters are matched or an empty list otherwise.

class `eoxserver.resources.coverages.wrappers.`**`EOCoverageWrapper`**

This is a partial implementation of `EOCoverageInterface` (page 200). It inherits from `CoverageWrapper` (page 232) and `EOMetadataWrapper` (page 233).

`getDatasets`(*filter_exprs=None*)

This method shall return the datasets associated with this coverage, possibly filtered by the optional filter expressions. It must be overridden by child implementations. By default `InternalError` (page 133) is raised.

`getEOCoverageSubtype`()

This method shall return the EO Coverage subtype according to the WCS 2.0 EO-AP (EO-WCS). It must be overridden by child implementations. By default `InternalError` (page 133) is raised.

`getLineage`()

Returns `None`.

Note: The lineage element has yet to be specified in detail in the WCS 2.0 EO-AP (EO-WCS).

---

class `eoxserver.resources.coverages.wrappers.`**`CoverageWrapper`**

This is the base class for all coverage wrappers. It is a partial implementation of `CoverageInterface` (page 199). It inherits from `ResourceWrapper` (page 156).

`getCoverageId`()

Returns the Coverage ID.

`getCoverageSubtype`()

This method shall return the coverage subtype as defined in the WCS 2.0 EO-AP (EO-WCS). It must be overridden by concrete coverage wrappers. By default this method raises `InternalError` (page 133).

See the definition of `getCoverageSubtype()` (page 199) in `CoverageInterface` (page 199) for possible return values.

`getData`()

Returns the a `CoverageDataWrapper` object that wraps the coverage data, raises `InternalError` (page 133) by default.

`getDataStructureType`()

Returns the data structure type of the coverage. To be implemented by subclasses, raises `InternalError` (page 133) by default.

getLayerMetadata()
Returns a list of `(metadata_key, metadata_value)` pairs that represent MapServer metadata tags to be attached to MapServer layers.

getRangeType()
This method returns the range type of the coverage as `RangeType` (page 216) object.

getSize()
This method shall return a tuple `(xsize, ysize)` for the coverage wrapped by the implementation. It has to be overridden by concrete coverage wrappers. By default this method raises `InternalError` (page 133).

getType()
This method shall return the internal coverage type code. It must be overridden by concrete coverage wrappers. By default this method raises `InternalError` (page 133).

See the definition of `getType()` (page 199) in `CoverageInterface` (page 199) for possible return values.

isAutomatic()
Returns `True` if the coverage is automatic or `False` otherwise.

matches(*filter_exprs*)
Returns True if the Coverage matches the given filter expressions and False otherwise.

Wrappers' Mix-In Classes

class `eoxserver.resources.coverages.wrappers.`**EOMetadataWrapper**
This wrapper class is intended as a mix-in for EO coverages and dataset series as defined in the WCS 2.0 EO-AP (EO-WCS).

getBeginTime()
Returns the acquisition begin time as `datetime.datetime`[99] object.

getEOGML()
Returns the EO O&M XML text stored in the metadata.

getEOID()
Returns the EO ID of the object.

getEndTime()
Returns the acquisition end time as `datetime.datetime`[100] object.

getFootprint()
Returns the acquisition footprint as `django.contrib.gis.geos.GEOSGeometry`[101] object in the EPSG:4326 CRS.

getWGS84Extent()
Returns the WGS 84 extent as 4-tuple of floating point coordinates `(minlon, minlat, maxlon, maxlat)`.

class `eoxserver.resources.coverages.wrappers.`**RectifiedGridWrapper**
This wrapper is intended as a mix-in for coverages that rely on a rectified grid. It implements `RectifiedGridInterface` (page 203).

getExtent()
Returns the coverage extent as a 4-tuple of floating point coordinates `(minx, miny, maxx, maxy)` expressed in the coverage CRS as defined by the SRID returned by `getSRID()` (page 234).

getResolution()
Returns the coverage resolution as a 2-tuple of float values for the x and y axes `(resx, resy)`

[99] http://docs.python.org/2.7/library/datetime.html#datetime.datetime
[100] http://docs.python.org/2.7/library/datetime.html#datetime.datetime
[101] https://docs.djangoproject.com/en/1.4/ref/contrib/gis/geos/#django.contrib.gis.geos.GEOSGeometry

expressed in the unit of measure of the coverage CRS as defined by the SRID returned by `getSRID()` (page 234).

getSRID()
Returns the SRID of the coverage CRS.

class eoxserver.resources.coverages.wrappers.**ReferenceableGridWrapper**
This wrapper is intended as a mix-in for coverages that rely on referenceable grids. It implements `ReferenceableGridInterface` (page 204).

getExtent()
Returns the coverage extent as a 4-tuple of floating point coordinates `(minx, miny, maxx, maxy)` expressed in the coverage CRS as defined by the SRID returned by `getSRID()` (page 234).

getSRID()
Returns the SRID of the coverage CRS.

class eoxserver.resources.coverages.wrappers.**PackagedDataWrapper**
This wrapper is intended as a mix-in for coverages that are stored as data packages.

getData()
Return the data package wrapper associated with the coverage, i.e. an instance of a subclass of `DataPackageWrapper` (page 190).

getDataStructureType()
Returns the data structure type of the underlying data package

class eoxserver.resources.coverages.wrappers.**TiledDataWrapper**
This wrapper is intended as a mix-in for coverages that are stored in tile indices.

getData()
Returns a `TileIndexWrapper` instance.

getDataStructureType()
Returns `"index"`.

2.12.6 Data Access Layer

Module eoxserver.backends.base

class eoxserver.backends.base.**LocationWrapper**
This is the base class for location wrappers. It inherits from `RecordWrapper` (page 143). It should not be instantiated directly, but one of its subclasses should be used.

detect(*search_pattern=None*)
Searches the location for objects that match `search_pattern`. If the parameter is omitted, all found objects are returned. It returns a list of locations of the same type that point to these objects. Raises `InternalError` (page 133) if the corresponding storage is not capable of auto-detection. See `StorageInterface.detect()` (page 239) for details.

getLocalCopy(*target*)
Copies the resource to the path `target` on the local file system. Raises `InternalError` (page 133) if the corresponding storage is not capable of copying data. See `StorageInterface.getLocalCopy()` (page 239) for details.

getSize()
Returns the size (in bytes) of the object at the location. Raises `InternalError` (page 133) if the corresponding storage is not capable of retrieving the size. See `StorageInterface.getSize()` (page 238) for details.

getStorageCapabilities()
Returns the capabilities of the corresponding storage. See `StorageInterface.getStorageCapabilities()` (page 238) for details.

Module eoxserver.backends.cache

This module provides an implementation of a cache, intended primarily for caching content from remote backends.

Warning: The current implementation of the `Cache` class is not functional and shall not be used. A future implementation must be able to work properly in a multi-process multi-threaded environment (i.e. provide some kind of data access synchronization). This requires inter-process communication to be implemented and is thus too much of an effort for the time being.

class `eoxserver.backends.cache.`**`CacheFileWrapper`**(*model*)
This class wraps `CacheFile` (page 240) records and adds the logic to handle them to the database model.

`access`()
This method shall be called every time a cache file is accessed. It updates the access timestamp of the model that should be used by cache implementations to determine which cache files can be removed.

`copy`(*location*)
Copy the file from its current `location` to the cache. This may raise `InternalError` (page 133) if the storage implementation for the location does not support the `getSize()` (page 238) and/or `getLocalCopy()` (page 239) methods or `DataAccessError` if there was a fault when retrieving the original file.

classmethod `create`(*filename*)
This class method creates a `CacheFileWrapper` (page 235) instance for the given file name. It makes a database record for the cache file, but does NOT copy it from its location to the cache. You have to call `copy()` (page 235) on the instance for that.

`getLocation`()
Returns the a `LocalPathWrapper` (page 239) object pointing to the location of the cache file.

`getModel`()
Returns the model record wrapped by the implementation.

`getSize`()
Returns the size of the cache file in bytes. Note that the return value is `None` if the `CacheFileWrapper` (page 235) instance has been initialized already, but `copy()` (page 235) has not been called yet.

`purge`()
Delete the cache file from the local file system and delete the associate `CacheFile` (page 240) database record. Raises `DataAccessError` if the file could not be deleted.

class `eoxserver.backends.cache.`**`CacheConfigReader`**
This is the configuration reader for the cache configuration. It should be used by cache implementations.

The cache can be configured by config file entries in the section `backends.cache`. There are three of them:

- `cache_dir`: if you want to use the cache you have to define this setting; it tells under which directory tree the cache files shall be stored. Note that if you change this setting, the cached files at the old location will not be forgotten.
- `max_size`: the maximum size of the cache in bytes; be sure to set this to a value that exceeds maximum traffic within the given retention time, otherwise you will get `CacheOverflow` errors at runtime
- `retention_time`: the minimum time cache files will be kept expressed in hours. At your own risk you can set it to 0, but strange things may occur then due to one thread deleting the data another one needs. A minimum of 1 hour is recommended, the default is 168 (a week).

`getCacheDir`()
Returns the `cache_dir` config file setting.

getMaxSize()
Returns the `max_size` config file setting.

getRetentionTime()
Returns the `retention_time` config file setting.

validate(*config*)
Returns `True`.

Module eoxserver.backends.ftp

This module provides the implementation of the FTP remote file backend.

class `eoxserver.backends.ftp.`**FTPStorage**
This is an implementation of the `StorageInterface` (page 238) for accessing files on a remote FTP server.

Note that internally, it creates a persistent connection that may be used for multiple requests on the same location or requests for multiple locations on the same server. DO NOT try to connect to different servers using the same `FTPStorage` (page 236) instance however, this will cause trouble and most definitely not work!

detect(*location*, *search_pattern=None*)
Recursively detects files in a directory tree and returns their locations. This will raise `DataAccessError` if the object at `location` is not a directory.

exists(*location*)
Checks the existance of a certain location within the storage. Returns *True* if the location exists and *False* if not or the location is not accessible.

getLocalCopy(*location*, *target*)
Copies the file at the remote `location` to the `target` path on the local file system. The parameter `location` is expected to be an instance of `RemotePathWrapper` (page 236), `target` is expected to be a string denoting the destination path.

The method raises `InternalError` (page 133) in case the location is not of appropriate type and `DataAccessError` in case an error occurs while copying the resources.

getSize(*location*)
Returns the size of the object at `location`. Note that not all FTP implementations are able to respond to this call. In that case `None` will be returned.

getStorageCapabilities()
Returns the storage capabilities, i.e. the names of the optional methods implemented by the storage. Currently `("getSize", "getLocalCopy", "detect")`.

getType()
Returns `"ftp"`.

class `eoxserver.backends.ftp.`**RemotePathWrapper**
This is a wrapper class for remote paths. It inherits from `LocationWrapper` (page 234).

setAttrs(***kwargs*)
This method is called to initialize the wrapper. The following attribute keyword arguments are accepted:

- `host` (required): the FTP host name
- `port` (optional): the FTP port number
- `user` (optional): the user name to be used for login
- `passwd` (optional): the password to be used for login
- `path` (required): the path to the location on the remote server

getHost()
Returns the FTP host name.

getPassword()
Returns the password to be used for login to the remote server or `None` if it has not been defined.

getPath()
Returns the path on the remote server.

getPort()
Returns the FTP port number or `None` if it has not been defined.

getStorageType()
Returns `"ftp"`.

getType()
Returns `"ftp"`.

getUser()
Returns the user name to be used for login to the remote server or `None` if it has not been defined.

Module eoxserver.backends.interfaces

This module defines interfaces for the Data Access Layer.

class `eoxserver.backends.interfaces.`**DatabaseLocationInterface**
This is the interface for raster data stored in a database. It inherits from `LocationInterface` (page 237) and adds some methods.

getHost()
This method shall return the hostname of the database manager.

getPort()
This method shall return the number of the port where the database manager listens for connections, or `None` if the port is undefined.

getDBName()
This method shall return the database name, or `None` if it is undefined.

getUser()
This method shall return the user name to be used for opening database connections, or `None` if it is undefined.

getPassword()
This method shall return the password to be used for opening database connections, or `None` if it is undefined.

class `eoxserver.backends.interfaces.`**LocalPathInterface**
This is the interface for locations on the local file system. It inherits from `LocationInterface` (page 237).

getPath()
This method shall return the path to the resource on the local file system.

open()
This method shall attempt to open the file at this location and return a `file` object. It accepts one optional parameter `mode` which is passed on to the builtin `open()` (page 237) command (defaults to `'r'`). The method shall raise `DataAccessError` if the file cannot be opened, or if the object at the location is not a file.

class `eoxserver.backends.interfaces.`**LocationInterface**
This is the base interface for locations where to find data, metadata or resources in general. It is not intended to be instantiated directly, but rather through its descendant interfaces. It inherits from `RecordWrapperInterface` (page 144).

getStorageCapabilities()
This method shall return the capabilities of the underlying storage implementation. See StorageInterface.getStorageCapabilities() (page 238).

getSize()
This method shall return the size of the object at the location or None if it cannot be retrieved. Note that InternalError (page 133) will be raised if this operation is not supported by the underlying storage implementation. See StorageInterface.getSize() (page 238).

getLocalCopy(*target*)
This method shall retrieve a local copy of the object at the location and save it to target. This parameter may be a path to a file or directory. The method shall return the location of the local copy, i.e. an implementation of LocalPathInterface (page 237).

detect(search_pattern=None):
This method shall return a list of locations of objects matching the given search_pattern to be found under the location, which is expected to be a tree-like object, most commonly a directory. If search_pattern is omitted all the locations shall be returned.

class eoxserver.backends.interfaces.**RemotePathInterface**
This is the interface for data and metadata files stored on a remote server. It inherits from LocationInterface (page 237).

getStorageType()
This method shall return the type of the remote storage, e.g. "ftp".

getHost()
This method shall return the host name of the remote storage.

getPort()
This method shall return the port number of the remote storage, or None if it is not defined.

getUser()
This method shall return the user name to be used for access to the remote storage, or None if it is not defined.

getPasswd()
This method shall return the password to be used for access to the remote storage, or None if it is not defined.

getPath()
This method shall return the path to the resource on the remote storage.

class eoxserver.backends.interfaces.**StorageInterface**
This is the interface for any kind of storage (local file system, remote repositories, databases, ...). It defines three methods:

getType()
This method shall return a string designating the type of the storage wrapped by the implementation. Current choices are:

- local
- ftp
- rasdaman

Additional modules may add more choices in the future.

getStorageCapabilities()
This method shall return which of the optional methods a storage implements.

The following methods are optional in the sense that they are not needed to be implemented in a meaningful way, either because the storage type does not support it, or because it is not needed. Even in this case, they need to be present and should raise InternalError (page 133).

getSize(*location*)
This method shall return the size in bytes of the object at `location` or `None` if it cannot be retrieved (e.g. for some FTP server implementations).

getLocalCopy(*location*, *target*)
This method shall make a local copy of the object at `location` at the path `target`.

The method shall return the location of the local copy, i.e. an implementation of a descendant of `LocationInterface` (page 237).

In case the type of `location` cannot be handled by the specific storage `InternalError` (page 133) shall be raised. In case the copying of resources fails `DataAccessError` shall be raised.

detect(*location*, *search_pattern=None*)
This method shall return a list of object locations found at the given `location` (which may designate some kind of collection, like a directory) that match the given `search_pattern`. If `search_pattern` is omitted any object location shall be returned.

Module eoxserver.backends.local

This module implements the local storage backend for EOxServer.

class `eoxserver.backends.local.`**LocalPathWrapper**
This a wrapper for locations on the local file system. It inherits from `LocationWrapper` (page 234).

setAttrs(***kwargs*)
The `path` keyword argument is mandatory; it is expected to contain the path to the location on the local file system.

getPath()
Returns the path to the location on the local file system.

getType()
Returns `"local"`.

open(*mode='r'*)
Opens the file at the location on the local file system and return the `file` object. The `mode` flag is passed to the builtin `open()` (page 239) function and defaults to `'r'` read only. Raises `DataAccessError` if the object at the location is not a file or cannot be opened for some other reason.

class `eoxserver.backends.local.`**LocalStorage**
This is a wrapper for the storage on the local file system.

detect(*location*, *search_pattern=None*)
Recursively detects files whose name matches `search_pattern` in the directory tree under `location` and returns their locations as a list of `LocalPathWrapper` (page 239) instances. If `search_pattern` is omitted all files found are returned.

getLocalCopy(*location*, *target*)
Makes a local copy of the file at `location` at the path `target` and returns the location of the copy (i.e. a `LocalPathWrapper` (page 239) instance). Raises `InternalError` (page 133) if the location does not refer to an object on the local file system or `DataAccessError` if copying fails.

getSize(*location*)
Returns the size (in bytes) of the object at the location or `None` if it cannot be retrieved.

getStorageCapabilities()
Returns the names of the optional methods implemented by the storage. Currently `("getSize", "getLocalCopy", "detect")`.

getType()
Returns `"local"`.

Module eoxserver.backends.models

class eoxserver.backends.models.**CacheFile**(**args*, ***kwargs*)
CacheFile (page 240) stores the whereabouts of a file held in the cache. Fields:

- location: a link to a LocalPath (page 240) denoting the path to the cached file
- size: the size of the file in bytes, null if it is not known
- access_timestamp: the time of the last access

class eoxserver.backends.models.**FTPStorage**(**args*, ***kwargs*)
This class describes an FTP repository. It inherits from Storage (page 240). Additional fields:

- host: the host name
- port (optional): the port number
- user (optional): the user name to use
- passwd (optional): the password to use

class eoxserver.backends.models.**LocalPath**(**args*, ***kwargs*)
LocalPath (page 240) describes a path on the local file system. It inherits from Location (page 240). Fields:

- path: a path on the local file system

class eoxserver.backends.models.**Location**(**args*, ***kwargs*)
Location (page 240) is the base class for describing the physical or logical location of a (general) resource relative to some storage. Fields:

- location_type: a string denoting the type of location

class eoxserver.backends.models.**RasdamanLocation**(**args*, ***kwargs*)
RasdamanLocation (page 240) describes the parameters for accessing a rasdaman array. It inherits from Location (page 240). Fields:

- storage: a foreign key of a RasdamanStorage (page 240) entry
- collection: name of the rasdaman collection that contains the array
- oid: rasdaman OID of the array (note that this is a float)

class eoxserver.backends.models.**RasdamanStorage**(**args*, ***kwargs*)
This class describes a rasdaman database access. It inherits from Storage (page 240). Additional fields:

- host: the host name
- port (optional): the port number
- user (optional): the user name to use
- passwd (optional): the password to use

class eoxserver.backends.models.**RemotePath**(**args*, ***kwargs*)
RemotePath (page 240) describes a path on an FTP repository. It inherits from Location (page 240). Fields:

- storage: a foreign key of an FTPStorage (page 240) entry.
- path: path on the repository

class eoxserver.backends.models.**Storage**(**args*, ***kwargs*)
This class describes the storage facility a collection of data is stored on. Fields:

- storage_type: a string denoting the storage type
- name: a string denoting the name of the storage

Module eoxserver.backends.rasdaman

This module implements the rasdaman database backend for EOxServer.

class `eoxserver.backends.rasdaman.`**`RasdamanArrayWrapper`**
This is a wrapper for rasdaman database locations. It inherits from `LocationWrapper` (page 234).

`setAttrs`(***kwargs*)
The following attribute keyword arguments are accepted:

- `host` (required): the host name of the server rasdaman runs on
- `port` (optional): the port number where to reach rasdaman
- `user` (optional): the user name to be used for login
- `db_name` (optional): the database name
- `passwd` (optional): the password to be used for login
- `collection` (required): the name of the collection in the database
- `oid` (optional): the `oid` of the array within the collection

`getCollection`()
Returns the collection name.

`getDBName`()
Returns the rasdaman database name, or `None` if it has not been defined.

`getHost`()
Returns the host name of the server rasdaman runs on.

`getOID`()
Returns the oid of the array within the collection.

`getPassword`()
Returns the password used to login to the database, or `None` if it has not been defined.

`getPort`()
Returns the port number where to reach rasdaman, or `None` if it has not been defined.

`getType`()
Returns `"rasdaman"`.

`getUser`()
Returns the user name used to login to the database, or `None` if it has not been defined.

class `eoxserver.backends.rasdaman.`**`RasdamanStorage`**
This class implements the rasdaman storage.

`detect`(*location*, *search_pattern=None*)
Not supported; raises `InternalError` (page 133).

`getLocalCopy`(*location*, *target*)
Not supported; raises `InternalError` (page 133).

`getSize`(*location*)
Not supported; raises `InternalError` (page 133).

`getStorageCapabilities`()
Returns the storage capabilities, i.e. the names of the optional methods implemented by the storage. Currently none are supported.

`getType`()
Returns `"rasdaman"`

2.12.7 Testing

Module eoxserver.testing.core

class `eoxserver.testing.core.`**`CommandFaultTestCase`**(*methodName='runTest'*)
Base class for CLI tool tests that expect failures (CommandErrors) to be raised.

`execute_command`(*args*)
Specialized implementation of the command execution. A failure is raised if no error occurs.

class `eoxserver.testing.core.`**`CommandTestCase`**(*methodName='runTest'*)
Base class for testing CLI tools.

`execute_command`(*args*)
This function actually executes the given command. It raises a failure if the command prematurely quits.

class `eoxserver.testing.core.`**`EOxServerTestCase`**(*methodName='runTest'*)
Test are carried out in a transaction which is rolled back after each test.

class `eoxserver.testing.core.`**`EOxServerTestRunner`**(*verbosity=1*, *interactive=True*, *failfast=True*, ***kwargs*)
Custom test runner. It extends the standard `django.test.simple.DjangoTestSuiteRunner`[102] with automatic test case search for a given regular expression.

Activate by including `TEST_RUNNER = 'eoxserver.testing.core.EOxServerTestRunner'` in *settings.py*.

For example *services.WCS20* would get expanded to all test cases of the *service* app starting with *WCS20*.

Note that we're using regex and thus *services.WCS20** would get expanded to all test cases of the *services* app starting with *WCS2* and followed by any number of *0*s.

Add test cases to exclude after a "|" character e.g. services.WCS20GetCoverage|WCS20GetCoverageReprojectedEPSG3857D

Module eoxserver.testing.xcomp

Simple XML documets' comparator.

XML Comparison

`eoxserver.testing.xcomp.`**`xmlCompareFiles`**(*src0*, *src1*, *verbose=False*)
Compare two XML documents passed as filenames, file or file-like objects.

`eoxserver.testing.xcomp.`**`xmlCompareStrings`**(*str0*, *str1*, *verbose=False*)
Compare two XML documents passed as strings.

`eoxserver.testing.xcomp.`**`xmlCompareDOMs`**(*xml0*, *xml1*, *verbose=False*)
Compare two XML documents passed as DOM trees (xml.dom.minidom).

Exceptions

Thies are the excetions raised by the XML comparison:

class `eoxserver.testing.xcomp.`**`XMLError`**
XML base error error

class `eoxserver.testing.xcomp.`**`XMLParseError`**
XML parse error

[102] https://docs.djangoproject.com/en/1.4/topics/testing/#django.test.simple.DjangoTestSuiteRunner

class eoxserver.testing.xcomp.**XMLMismatchError**
XML mismatch error

2.13 Testing

TBD

eoxserver.testing.core (page 242)

CHAPTER

THREE

EOXSERVER REQUESTS FOR COMMENTS

EOxServer Requests for Comments (RFCs) are a means for EOxServer developers to share their ideas and feature requests, propose enhancements, and discuss high-level issues concerning the further development of the software.

3.1 RFC Procedures

See the *RFC Policies* (page 327) for details.

3.2 Writing RFCs

If you want to write a Request for Comments, please read the *Guidelines for Requests for Comments* (page 328) first.

3.3 RFCs

Table: "*List of accepted EOxServer RFCs* (page 245)" below lists all accepted EOxServer RFCs and their implementation status [1].

Table 3.1: List of accepted EOxServer RFCs

No.	Title	Status
0	*RFC 0: Project Steering Committee Guidelines* (page 246)	Effective
1	*RFC 1: An Extensible Software Architecture for EOxServer* (page 248)	Implemented in version 0.2
2	*RFC 2: Extension Mechanism for EOxServer* (page 270)	Implemented in version 0.2
6	*RFC 6: Directory Structure* (page 280)	Implemented in version 0.2
7	*RFC 7: Release Guidelines* (page 281)	Effective
8	*RFC 8: SVN Commit Management* (page 283)	Effective
9	*RFC 9: SOAP Binding of WCS GetCoverage Response* (page 286)	Implemented in SOAP Proxy
10	*RFC 10: SOAP Proxy* (page 287)	Implemented in SOAP Proxy
12	*RFC 12: Backends for the Data Access Layer* (page 291)	Implemented in version 0.2
13	*RFC 13: WCS-T 1.1 Interface Prototype* (page 295)	Implemented in version 0.2
14	*RFC 14: Asynchronous Task Processing (ATP)* (page 299)	Implemented in version 0.2
15	*RFC 15: Access Control Support* (page 303)	Implemented in version 0.2
16	*RFC 16: Referenceable Grid Coverages* (page 305)	Implemented in version 0.2
17	*RFC 17: Configuration of Supported Output Formats and CRSes* (page 308)	Implemented in version 0.3
19	*RFC 19: Migrate project repository from svn to git* (page 324)	Effective

[1] Note that this list might not be fully up to date although we try hard.

The list below provides links to all EOxServer RFCs available:

3.3.1 RFC 0: Project Steering Committee Guidelines

Author Stephan Meißl

Created 2011-03-02

Last Edit 2011-05-17

Status ACCEPTED

Discussion http://www.eoxserver.org/wiki/DiscussionRfc0

Overview

This RFC documents the EOxServer Project Steering Committee Guidelines.

(Credit: Inspired by the MapServer PSC guidelines at: http://mapserver.org/development/rfc/ms-rfc-23.html)

Introduction

This RFC describes how the EOxServer Project Steering Committee (PSC) handles membership and makes decisions on all aspects, technical and non-technical, of the EOxServer project.

The PSC duties include:

- defining and deciding on the overall development road map
- defining and deciding on technical standards and policies like file naming conventions, coding standards, etc.
- establishing a regular release cycle
- reviewing and voting on RFCs

The PSC members vote on proposals, RFCs, etc. via e-mail on the dev mailing list. Proposals shall be available for review for at least two days where a single veto delays the progress but at the end a majority of members may adopt a proposal.

Voting

Voting Procedure

The following steps shall be followed in any voting:

- Any interested person may submit a proposal to the dev mailing list for discussion. Note that this is explicitly not limited to PSC members.
- Voting on proposals shall not be closed earlier than two business days after the proposal has been submitted.
- The following voting options shall be used:
  - "+1" .. support willingness to support implementation
  - "+0" .. low support
  - "0" .. no opinion
  - "-0" .. low disagreement
  - "-1" .. veto
- A veto shall include clear reasoning and alternative approaches to solve the problem at hand.

- Any interested person may comment on proposals but only votes from PSC members will be counted.
- A proposal may be declared accepted if it receives at least +2 and not vetoes (-1).
- Vetoed proposals that cannot be revised to satisfy all PSC members may be submitted for an override vote. The proposal may be declared accepted if a simple majority of eligible voters votes in favor (+1). Eligible voters are all PSC members that have not been declared inactive. However, it is intended that in normal circumstances vetoers are convinced to withdraw their veto. We are trying to reach consensus.
- Any eligible voter who has not cast a vote in the last two votes shall be considered inactive. Casting a vote immediately turns the status to active.
- Upon completion of discussion and voting the author shall announce the new status of the proposal (accepted, withdrawn, rejected, postponed, obsolete).
- The PSC Chair is responsible for keeping track of who is a member of the PSC Membership.
- Addition and removal of members from the PSC, as well as selection of a Chair should be handled as a proposal to the PSC.
- The PSC Chair adjudicates in cases of disputes about voting.

Voting is Required for

- any change to committee membership (adding members, removing inactive members).
- creating and dissolving of sub-committees (e.g. to manage conferences, documentation, or web sites).
- changes to project infrastructure (e.g. tool, location, or substantive configuration).
- anything that could cause backward compatibility issues.
- adding substantial amounts of new code.
- changing inter-subsystem APIs, or objects.
- issues of procedure.
- when releases should take place.
- anything dealing with relationships with external entities such as MapServer or OSGeo.
- anything that might be controversial.

PSC Membership

The PSC is made up of individuals consisting of technical contributors (e.g. developers) and prominent members of the EOxServer user community. There is no fixed number of members for the PSC.

Adding Members

Any member of the dev mailing list may nominate someone for committee membership at any time. Only existing PSC committee members may vote on new members. Nominees must receive a majority vote from existing members to be added to the PSC.

Stepping Down

If, for any reason, a PSC member is not able to fully participate then they certainly are free to step down. If a member is not active (e.g. no voting, no IRC, or e-mail participation) for a period of two months then the committee reserves the right to vote to cease membership. Should that person become active again then they are certainly welcome, but require a nomination.

Membership Responsibilities

Guiding Development

Members should take an active role guiding the development of new features they feel passionate about. Once a change request has been accepted and given a green light to proceed does not mean the members are free of their obligation. PSC members voting "+1" for a change request are expected to stay engaged and ensure the change is implemented and documented in a way that is most beneficial to users. Note that this applies not only to change requests that affect code, but also those that affect the web site, technical infrastructure, policies, and standards.

IRC Meeting Attendance

PSC members are expected to participate in pre-scheduled IRC development meetings. If known in advance that a member cannot attend a meeting, the member should let the meeting organizer know via e-mail.

Mailing List Participation

PSC members are expected to be active on both the users and dev mailing lists, subject to open source mailing list etiquette. Non-developer members of the PSC are not expected to respond to coding level questions on the developer mailing list, however they are expected to provide their thoughts and opinions on user level requirements and compatibility issues when RFC discussions take place.

List of Members

Charter members are (in alphabetical order):

- Arndt Bonitz
- Peter Baumann
- Stephan Krause
- Stephan Meißl
- Milan Novacek
- Martin Paces
- Fabian Schindler

Stephan Meißl is declared initial Chair of the Project Steering Committee.

Voting History

Acceptance All charter members declared their availability via e-mail to the dev mailing list.

Traceability

Requirements N/A

Tickets N/A

3.3.2 RFC 1: An Extensible Software Architecture for EOxServer

Author Stephan Krause

Created 2011-02-18

Last Edit 2011-07-20

Status ACCEPTED

Discussion http://www.eoxserver.org/wiki/DiscussionRfc1

This RFC proposes an extensible software architecture for EOxServer that is based on the following ideas:

- Separation of instance and distribution code
- Structuring of the distribution in layers
- Extensibility through a plugin system

Introduction

EOxServer development has been initiated in the course of two ESA projects that aim at providing a harmonized standard interface to access Earth Observation (EO) products, namely:

- Heterogeneous Mission Accessibility - Follow-On Open Data Access (HMA-FO ODA)
- Open-standard Online Observation Service (O3S)

The specification of a software architecture is required by these projects. From a practical point of view, EOxServer has reached a point where a common framework for a rapidly evolving project is needed.

Summarizing the requirements in a nutshell EOxServer has to integrate:

- different OGC Web Services
- different data and processing resources
- heterogeneous data and metadata formats

This leads to the conclusion that an extensible software architecture is needed. The problems to address are discussed in further detail in the *Requirements* (page 249) section.

The *proposed architecture* (page 254) is modular, extensible and flexible and structured in layers. The following separate components are identified:

- Distribution
  - *Distribution Core* (page 267)
  - *Service Layer* (page 267)
  - *Processing Layer* (page 268)
  - *Data Integration Layer* (page 268)
  - *Data Access Layer* (page 269)
- *Instances* (page 269)

In this architecture the Core shall provide the central logic for the extension mechanism while the layers shall contain interface definitions based on the extension model of the Core that can be implemented by extending modules and plugins.

Requirements

The main sources of requirements for EOxServer at the moment of writing this RFC are:

- the HMA-FO Open Data Access Sofware Requirements Specification (SRS)[2]
- the O3S Software System Specification (SSS)
- the feature requests posted on the EOxServer Trac[3]

[2] http://wiki.services.eoportal.org/tiki-download_wiki_attachment.php?attId=957&download=y
[3] http://www.eoxserver.org

Most of the requirements are related to the features EOxServer shall implement. There is one requirement, however, in the O3S SSS that is directly related to the software architecture; O3S_QUA_004[4] states:

> The O3S3 shall sustain maintainability and reusability by using a modular system architecture
>
> This shall facilitate
>
> - isolation and removal of code defects
> - integration of new functionality, such as the implementation of new interface standard versions
> - extension of the system functionality according to new or modified requirements

Thus, modularity as well as integration and extension of functionality are central issues in the drafting of the EOxServer software architecture. The question remains what considerations shall govern the structuring of the software into modules, what functionality it shall implement and in what way the system shall be able to be extended.

Our approach to this question is to identify different topics of concern for the EOxServer development that shall structure the requirements analysis and give a first hint on the architectural design.

The main goal of EOxServer is to furnish an implementation of OGC[5] Web Services (OWS) intended for use within the Earth Observation (EO) domain. These *services* (page 250) shall provide access to different kinds of *resources* (page 251) and to *processes* (page 251) operating on these resources. The requirements cite different *backends* (page 251) that the software shall implement in order to allow access to local and remote content. Finally, we discuss where and how the software is going to be *deployed* (page 252).

Services

The following OGC Web Services shall be implemented:

Web Coverage Service (WCS)[6] (requirement O3S_CAP_001[7])

> The Web Coverage Service shall be able to present Earth Observation data, e.g. optical satellite imagery, SAR data, etc. The following extensions shall be implemented:
>
> Earth Observation Application Profile for WCS (EO-WCS) (requirement O3S_CAP_100[8])
>
> > This application profile is intended to ease access to large collections of Earth Observation data.
>
> Transactional WCS (WCS-T) (requirement O3S_CAP_150[9])
>
> > This extension of WCS introduces a Transaction operation that supports transfer of data *to* a WCS server.

Web Map Service (WMS)[10] (requirement O3S_CAP_220[11])

> This service shall be used to give to portrayals of the coverages the system presents. The following extension shall be implemented:
>
> WMS Profile for EO Products (EO-WMS) (requirement O3S_CAP_240[12])
>
> > The extension allows access to portrayals of large dataset series.

Web Feature Service (WFS)[13] (requirement O3S_CAP_260[14])

> This service shall be used to present vector data.

[4] https://o3s.eox.at/requirements/ticket/122
[5] http://www.opengeospatial.org
[6] http://www.opengeospatial.org/standards/wcs
[7] https://o3s.eox.at/requirements/ticket/7
[8] https://o3s.eox.at/requirements/ticket/8
[9] https://o3s.eox.at/requirements/ticket/198
[10] http://www.opengeospatial.org/standards/wms
[11] https://o3s.eox.at/requirements/ticket/204
[12] https://o3s.eox.at/requirements/ticket/210
[13] http://www.opengeospatial.org/standards/wfs
[14] https://o3s.eox.at/requirements/ticket/214

Web Processing Service (WPS)[15] (requirement O3S_CAP_200[16])

This service shall be used to make processing resources accessible online.

Processes

EOxServer shall present various processes to the public using WPS. The processes planned for implementation at the moment of writing this RFC are specific to the use cases to be handled in the course of the O3S project. The capability to publish a variety of processes on the other hand is a general requirement for EOxServer.

Being a project focussing on the EO domain EOxServer concentrates on the processing of EO coverage (raster) data. So, the considerations made for coverages regarding the variety of data and metadata *formats* (page 251) are valid for processes as well.

Resources

EOxServer shall enable public access to different kinds of geo-spatial resources in the Earth Observation domain. These are:

- Coverages
- Vector Data
- Processes

Coverages Coverages are defined in a very abstract way. What EOxServer focusses on are coverages dealt with by the Earth Observation Application Profile for WCS (EO-WCS) which is a draft OGC Best Practice Paper as of writing this RFC. The main categories of resources defined in that paper are:

Datasets Datasets are the atomic components EO-WCS objects are composed of. They are coverages that are associated with EO Metadata. EO satellite mission scenes are a good example of Datasets. They can be accessed individually even when being part of a Stitched Mosaic or Dataset Series.

Stitched Mosaics Stitched Mosaics are made up from a collection of Datasets that share a common range type and grid. Other than Dataset Series they are not merely a container for Datasets but coverages themselves. The coverage values are generated from the contributing datasets. This process must follow some rule to decide what value to take into account in the areas where the contributing Datasets overlap. The most common rule is "latest-on-top".

Dataset Series Dataset Series represent collections of Datasets or Stitched Mosaics. They do not impose any constraints on the contained objects, so very heterogeneous data can be included in the same series.

A major problem for the EOxServer implementation is that raster data coverages originating from EO satellite missions are very heterogeneous. They can use a vide variety of data and metadata formats and are often associated with additional data like bitmasks, etc. that should be presented by EOxServer as well. Furthermore, the data packaging is different for every mission.

Vector Data Support for Vector Data handling is required by O3S Use Case 2. In that use case road network data shall be generated from Pléiades satellite data using automated feature detection algorithms. The road network data shall be presented using WFS and WMS.

Backends

EOxServer shall implement various backends to access data it presents to the public via the OGC Web Services:

- Backend for local data (requirement O3S_CAP_013[17])

[15] http://www.opengeospatial.org/standards/wps
[16] https://o3s.eox.at/requirements/ticket/9
[17] https://o3s.eox.at/requirements/ticket/68

- Backends for remote data (requirements: HMA-FO SR_ODA_IF_070, O3S_CAP_014[18])
  - using HTTP/HTTPS
  - using FTP
  - using WCS
- Backend for retrieving data from rasdaman[19] (requirement O3S_CAP_017[20])

Deployment

The only requirements originating from the HMA-FO ODA and O3S projects regarding deployment concern the implementation of the O3S Use Cases. Every use case requires one or more instances of EOxServer to be deployed. The instances have different purposes and thus shall present different services and different resources.

The fact that EOxServer shall be deployed many times in different configurations (possibly on the same server) calls for a strict separation of distribution and instance data.

The ability to activate or deactivate various components of the system implies not only that the architecture must be modular but also that it must be configurable to use different combinations of modules.

Summary

The conclusion of the requirements review is that the EOxServer Architecture shall be:

- modular
- extensible
- flexible in the sense that it must be possible to select different combinations of modules to deploy and activate
- able to present resources using different OGC Web Services
- able to access data from different backends
- able to handle different data, metadata and packaging formats
- separating distribution and instance data

The development of the software architecture will be based on these considerations.

Architecture Overview

The software architecture development for EOxServer does not start at zero. There are already considerations made in the proposal phase of the O3S project and there is the status quo of version 0.1.0. Taking into account this preparational work and the outcomes of the requirements review, the outlines of the *Proposed Architecture* (page 254) will be developed in the last subsection and the following sections.

Draft Architecture

The O3S draft Architectural Design Document (ADD/SDD) has already proposed a software architecture which is, however, outdated in certain aspects due to changes made in the requirements phase of O3S. Here is an overview of the O3S draft architecture:

This identifies four servers and extending modules:

- WPS Server

[18] https://o3s.eox.at/requirements/ticket/69
[19] http://www.rasdaman.com
[20] https://o3s.eox.at/requirements/ticket/183

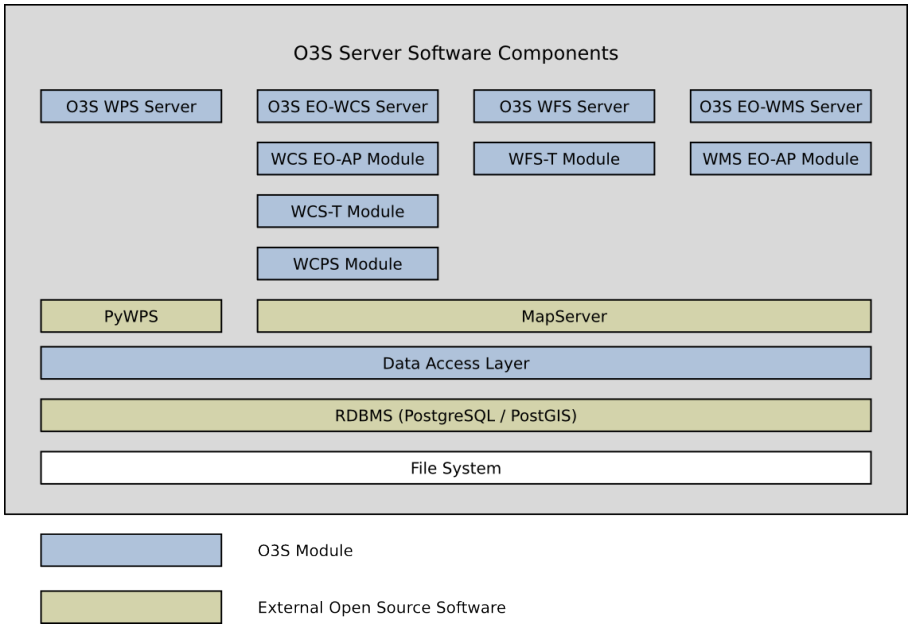


Figure 3.1: *Draft architecture from O3S Proposal*

- WCS Server
  - WCS Earth Observation Application Profile Module
  - WCS-T Module
  - WCPS Module (not included in the requirements any more)
- WFS Server
  - WFS-T Module (not included in the requirements any more)
- WMS Server
  - WMS Profile for EO Products Module

Furthermore the architecture proposes to use PyWPS[21] and MapServer[22] as middleware for handling OGC Web Service requests.

An additional integrating Data Access Layer is foreseen that shall implement storage patterns such as image pyramids and offer an API to read and write data that hides the internal details of data storage from the service and extension modules using it.

PostgreSQL[23] with its geo-spatial extension PostGIS[24] has been planned as relational database backend. Finally, the system relies on the local filesystem as its only storage backend.

During the requirements phase of O3S and the early development of EOxServer many deviations from this original design have been made necessary. Most importantly:

- Django[25] has been added as dependency
- GDAL[26] has been added as dependency
- the implementation of WCPS has been postponed
- the implementation of WFS-T has been postponed

[21] http://pywps.wald.intevation.org/
[22] http://www.mapserver.org
[23] http://www.postgresql.org
[24] http://postgis.refractions.net
[25] http://www.djangoproject.com
[26] http://www.gdal.org

- Django has made use of different geo-spatial database backends possible
- requirements for remote storage backends have been added

Although the basic concepts of the draft architecture remain valid, an updated version is needed for EOxServer to fulfill its requirements and evolve beyond the project horizon of O3S.

Status Quo of Release 0.1.1

As of release 0.1.1 EOxServer is an integrated Django project including a single Django application and additional modules that support OGC Web Service (OWS) request handling and data integration.

The data model is contained in the `eoxserver.server` application. So is the `ows` view, the central entrance point for OWS requests, and the administration client view as well as tools for automatic data ingestion.

Supporting modules are gathered in the `eoxserver.lib` module. These contain the core application logic for OWS request handling, coverage and metadata manipulation as well as utilities e.g. for XML processing.

EOxServer 0.1.1 includes an extension mechanism already which so far is restricted to services. The `eoxserver.lib.registry` module maintains a central registry for the concrete implementations of OWS interfaces which may be published in the `eoxserver.modules` namespace. At the moment there are implementations for WMS 1.0, 1.1 and 1.3, WCS 1.0, 1.1 and 2.0 as well as a preliminary version of EO-WCS. All these modules use MapServer MapScript for image manipulation and part of the request handling in the backend.

This approach fulfills some of the requirements summarized *above* (page 252) already, but further development of the architecture and the code is necessary to be fully compliant. Most importantly:

- extensibility and flexibility have to be enhanced
- WPS must be implemented
- WFS must be implemented
- support for remote backends is necessary

Proposed Architecture

The proposed architecture for EOxServer shall be based on the following principles:

- **Separation of Instance and Distribution**: instance applications shall be separated from EOxServer distribution code in order to facilitate deployment of multiple services on the same machine and to support flexible configurations of services
- **Layered Architecture of the Distribution**: The software architecture shall be structured in layers and a core that contains basic common functionality; each layer builds on the capabilities of the underlying ones to fulfill its tasks
- **Extensibility**: the EOxServer distribution shall be extensible by additional modules and plugins; the distribution core shall provide functionality to enable dynamic binding to extending modules

The identification of different layers is performed based on the structuring of the system components underlying the requirements analysis.

Dependencies The implementation of EOxServer shall use the following dependencies:

- **Python**: Python[27] shall serve as the implementation language; it has been chosen because
  - it facilitates rapid development
  - the geospatial libraries used all have Python bindings
- **Django**: Django[28] has been selected as development framework because

[27] http://www.python.org
[28] http://www.djangoproject.com

  - it provides an object-relational mapper that supports various database backends
  - it supports geospatial databases and integrates vector data handling functionality in the GeoDjango extension
  - it allows for rapid web application development

- **Spatial Database Backend**: using GeoDjango, EOxServer shall support at least the SpatiaLite[29] and PostGIS[30] geospatially enabled RDBMS backends.
- **MapServer**: EOxServer shall build on MapServer[31] MapScript in order to facilitate OGC Web Service handling
- **GDAL/OGR**: For image processing tasks and vector data manipulation the Python binding of the GDAL/OGR[32] libraries shall be used

Concerning the software architecture, the use of Django enforces a Model-View-Controller (MVC) substructure of the distribution layers of EOxServer.

Distribution Core and Layers The breakdown of the distribution into core and layers is as follows:

Core The Core shall contain modules for common use throughout the different components of EOxServer. This includes the global configuration data model, the implementation of the extension mechanism as well as the basic functionality for the EOxServer administration client

Service Layer This layer contains the core request handling logic as well as the implementation of services and service extensions

Processing Layer This layer contains the processing models used internally by EOxServer as well as the data model and the basic handling routines for processes to be published using WPS

Data Integration Layer This layer shall provide data models for resources as well as an abstraction layer for different data formats and data packaging formats

Data Access Layer This layer shall provide backends for local and remote data access

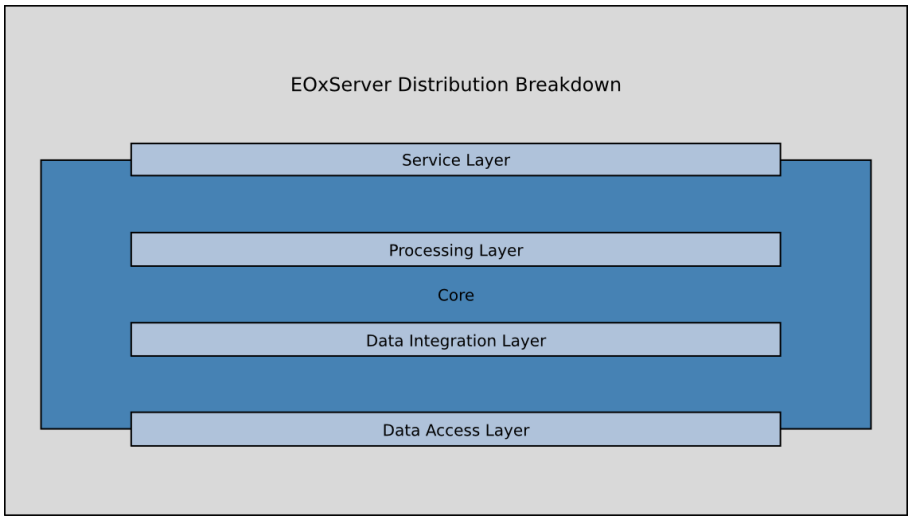


Figure 3.2: *EOxServer Distribution Breakdown*

Each of the four layers shall be sub-structured in:

- data model
- views

[29] http://www.gaia-gis.it/spatialite/
[30] http://postgis.refractions.net
[31] http://www.mapserver.org
[32] http://www.gdal.org

  - for public access (if applicable)
  - for the administration client
- core handling logic
- interface definitions for extensions
- modules implementing the interface definitions

Structure of the Architecture Specification The further specification of the proposed architecture is subdivided into several sections and separate RFCs. This RFC 1 contains a description of the different architectural layers and of EOxServer instances:

- *Distribution Core* (page 267)
- *Service Layer* (page 267)
- *Processing Layer* (page 268)
- *Data Integration Layer* (page 268)
- *Data Access Layer* (page 269)
- *Instances* (page 269)

The following RFCs discuss different aspects of the architecture in further detail:

RFC 2: Extension Mechanism for EOxServer

Author Stephan Krause

Created 2011-02-20

Last Edit 2011-09-15

Status ACCEPTED

Discussion http://www.eoxserver.org/wiki/DiscussionRfc2

This RFC proposes an extension mechanism that allows to integrate extension modules and plugins dynamically into the EOxServer distribution and instances.

Introduction *RFC 1: An Extensible Software Architecture for EOxServer* (page 248) proposes an extensible architecture for EOxServer in order to ensure

- modularity
- extensibility
- flexibility

of the design. It establishes the need for an extension mechanism which acts as a sort of "glue" between different parts of the architecture and enables dynamic binding to these components.

This RFC discusses the extension mechanism in further detail and identifies the architectural principles and components needed to implement it.

The constituent components of the extension mechanism design are interface declarations, the respective implementations and a central registry that contains metadata about interfaces and implementations and enables dynamic binding to the latter ones.

Requirements *RFC 1: An Extensible Software Architecture for EOxServer* (page 248) proposes an extension mechanism for EOxServer. It shall assure extensibility by additional modules and plugins and provide functionality to enable dynamic binding to extending modules.

In the layered architecture of RFC 1 the *Distribution Core* (page 267) shall be the place where the central logic that enables the dynamic extension of system functionality resides. The layers shall provide interface definitions based on the extension model of the Core that can be implemented by extending modules and plugins.

Now which extensions are needed and which requirements do they impose on the extension mechansims? Digging deeper we have a look at the four architectural layers of EOxServer and analyze the interfaces and implementations needed by each of them.

The *Service Layer* (page 267) defines a structured approach to OGC Web Service (OWS) request handling that discerns different levels:

- services
- service versions
- service extensions
- service operations

For all of these levels interfaces are defined that are implemented by extending modules for specific OWS and their different versions and extensions.

The *Processing Layer* (page 268) defines interfaces for processes and processing chains (see *RFC 5: Processing Chains* (page 280)). Some of these are used internally and integrated into the distribution, most will be provided by plugins. While the process interface needs to be generic in order to make the implementation of many different processes possible, it must be concise enough to allow binding between processes in a processing chain. So, this must be sustained by the extension mechanism as well.

The *Data Integration Layer* (page 268) shall provide an abstraction layer for different data formats, metadata formats and data packaging formats. This shall be achieved using common interfaces for coverage data, vector data and metadata respectively.

Data and packaging formats are often not known by the system before ingesting a dataset. Thus, some kind of autodetection of formats is necessary. This is provided partly by the underlying libraries such as GDAL[33], but shall also be considered for the design of the extension mechanism: it must be possible to dynamically bind to the right data, metadata and data packaging format based on evaluations of the data. These tests should be implemented by format extensions and supported by the extension mechansim.

The *Data Access Layer* (page 269) is built around the interface definitions of backends and data sources stored by them.

In addition to modularity and extensibility RFC 1 states that the system shall be

> flexible in the sense that it must be possible to select different combinations of modules to deploy and activate

Modules can be combined to build a specific application. From a user perspective it is essential to be able to activate and deactivate services, service versions and service extensions globally and/or separately for each resource or process. The same applies for other extensible parts of the system such as backends.

The O3S Use Case 2 for instance requires a server setup that consists of:

- local and WCS backends in the Data Access Layer
- a specific combination of coverage, vector data, metadata and packaging formats in the Data Integration Layer
- a feature detection process in the Processing Layer
- WPS and WFS implementations in the Service Layer

All other backends, services and processes shall be disabled.

Summarizing the requirements the extension mechanism shall support:

[33] http://www.gdal.org

- extensibility by additional modules and plugins
- dynamic binding
- interface definitions for extensions
- implementations that can be enabled or disabled
  - globally
  - per resource or per process
- modules that can be configured dynamically to build an application
- autodetection of data, metadata and data packaging formats

Extension Mechanism The basic questions for the design of the extension mechanism are:

- how to declare extensible interfaces
- how to design implementations of these interfaces
- how to advertise them
- how to bind to them

Unlike Java or C++, Python does not have a built-in mechanism to declare interfaces. A method definition always comes with an implementation. With Python 2.6 support for abstract base classes and abstract methods was added, but at the moment it is not an option to use this framework as this would break support for earlier Python versions.

So, two basic design options remain:

- using conventional Python classes and inheritance mechanisms for interfaces and implementations
- customize the interface declaration and implementation creation using Python metaclasses

Whereas the first approach is easier, the second one can provide more control and a clear differentiation between interface declaration and implementation. Both design options are discussed in further detail in the *Interfaces and Implementations* (page 273) section below.

The second major topic is how to find and bind to implementations of an interface if not all implementations are known to the system a priori, as is the case with plugins. Some "glue" is needed that holds the system together and allows for dynamic binding. In the case of EOxServer this is implemented by a central registry that keeps track of implementations by automatically scanning Python modules in certain directories that are supposed to contain EOxServer extending modules or plugins. For more details on the basics of *Registry* (page 275) see below.

In most cases an instance of EOxServer will not need all the functionality provided by the distribution or plugins installed on the system. Dynamic binding allows for enabling and disabling certain services, processes, formats, backends and plugins in an interactive way using the administration client. In order to assure this required functionality a configuration data model is needed that allows to store information about what parts of the system are activated and what resources they may operate on. See the section *Data Model* (page 275) for further details.

Implementations of interfaces are not isolated objects. They depend on libraries, functionality provided by the EOxServer core and layers and, last but not least, on other interface implementations. In order to assure that the dynamically configurable system is in a consistent state, the interdependencies between implementations need to be properly advertised and stored in the configuration data model.

After this short overview, we will go more in depth in the following sections.

Interfaces and Implementations As already discussed before there are two design options for interfaces and implementations:

- interfaces and implementations as conventional Python classes that are linked through inheritance
- interfaces as special Python classes that are linked to implementations by a custom mechanism.

Whereas the first approach is straightforward and easy to implement and handle it has also some serious drawbacks. Most importantly it does not allow for a clear separation between interface declaration and implementation. A method declared in the interface always must contain an implementation, and an implementation may change the signature of the methods it implements in any possible way.

What's more, as the implementation inherits (mostly generic) method code from the interface there is no way to validate if it actually defines concrete methods to override the "abstract" ones the interface class provides.

So, there are also good reasons for the second approach although it is more challenging for developers. The approach proposed here allows to customize class generation and inheritance enabling validation at "compile time" (i.e. when classes are created) and runtime (i.e. when instance methods are invoked) as well as separation of interface definition from implementation.

How can this be achieved? The proposed mechanism relies on an interface base class called `Interface` that concrete interface declarations can derive from, implementing code contained in a conventional Python class and a method called `implement()` that generates a special implementation class from the interface declaration and the class containing the implementing code.

Interface Declaration It has already been said that interface declarations shall derive from a common base class called `Interface`. But that is not the end of the story - one big question remains: how to declare actual methods without implementation? The proposed approach is not to declare methods as such at all, but use classes representing them instead.

For this end three classes are to be defined alongside the `Interface` base class.

- instances of the `Constant` class represent constants defined by the interface
- instances of the `Method` class represent methods
- instances of the `Arg` class represent method arguments; subclasses of `Arg` allow for type validation, e.g. instances of `IntArg` represent integer arguments

Let's have a look at a quick example:

```
from eoxserver.core.interfaces import Interface, Method, Arg

class ServiceInterface(Interface):
    handle = Method(
        Arg("req")
    )
```

Note: Code examples in this RFC are merely informational. The actual implementation may deviate a little bit from them. A reference documentation will be prepared for the definitive extension mechanism.

This snippet of Python code represents a simple and complete interface declaration. The `ServiceInterface` class will be used in further examples as well. It shows a method definition that declares the following: the method `handle` shall take one argument of arbitrary type named `req` that stands for an OWS request.

As you can see the declaration is a class variable containing an instance of the `Method` class. It is not a method (it does not even have to be callable). It serves two purposes:

- documentation of the interface
- validation of the implementation

Thinking of these two goals, the writer of the code could have been more rigorous and declare an argument like this:

```
handle = Method(
    ObjectArg("req", arg_class=OWSRequest)
)
```

That way it is documented what kind of argument is expected. When defining the implementation it is enforced that it have a method `handle` which takes exactly one argument besides `self`, otherwise an exception will be

raised. When invoking an interface of the implementation it can be validated that the argument is of the right type. More on this later under *Validation of Implementations* (page 274). Now let's have a look at implementations.

Implementations The proposed design of interface implementation intends to hide all the complexity of this process from the developers of implementations. They just have to write an implementing class which is a normal new-style Python class, and wrap it with the `implement()` method of the interface, such as in the following example:

```
from eoxserver.services.owscommon import ServiceInterface

class WxSService(object):

    def handle(self, req):

        # ...

        return response

WxSServiceImplementation = ServiceInterface.implement(WxSService)
```

The call to `implement()` ensures validation of the interface and produces an implementation class that inherits all the code of the implementing class and contains information about the interface. This is only the basic functionality of the interface implementation process: more is to be revealed in the following sections.

Validation of Implementations The validation of implementations is performed in two ways:

- at class creation time
- at instance method invocation time

Validation at class creation time checks:

- if all methods declared by the interface are implemented
- if the method arguments of the interface and implementation match

Class creation time validation is performed unconditionally.

Instance method invocation time ("runtime") validation is optional. It can be triggered by the `runtime_validation_level` setting. There are three possible values for this option:

- `trust`: no runtime validation
- `warn`: argument types are checked against interface declaration; in case of mismatch a warning is written to the log file
- `fail`: argument types are checked against interface declaration; in case of mismatch an exception is raised

The `runtime_validation_level` option can be set

- globally (in configuration file)
- per interface
- per implementation

where stricter settings override weaker ones.

Note: The `warn` and `fail` levels are intended for use throughout the development process. In a production setting `trust` should be used.

Registry The Registry is the core component for managing the extension mechanism of EOxServer. It is the central entry point for:

- automated detection of registered interfaces and implementations
- dynamical binding to the implementations
- configuration of components and relations between them

Its functionality shall be discussed in further detail in the following subsections:

- *Data Model* (page 275)
- *Detection* (page 277)
- *Binding* (page 277)

Data Model The data model for the Extension Mechanism including dynamic binding is implemented primarily by the *Registry* (page 275); for persistent information it relies on the configuration files and the database.

As you'd expect, the Registry data model relies on interfaces and implementations. However, not all of them are registered, but only descendants of `RegisteredInterface` (page 152) and their respective implementations. `RegisteredInterface` (page 152) extends the configuration model for interfaces with information relevant to the registration and dynamic binding processes. This is an example for a valid configuration:

```
from eoxserver.core.registry import RegisteredInterface

class SomeInterface(RegisteredInterface):

    REGISTRY_CONF = {
        "name": "Some Interface",
        "intf_id": "somemodule.SomeInterface",
        "binding_method": "direct"
    }
```

The most important parts are the interface ID `intf_id` and the `binding_method` settings which will be used by the registry to find implementations of the interface and to determine how to bind to them. For more information see the *Binding* (page 277) section below.

The registry model is accompanied by a database model that allows to store persistently which parts of the system (services, plugins, etc.) are enabled and which resources they have access to.

For every registered implementation an `Implementation` instance and database record are created. Implementations are subdivided into components and resource classes, each with their respective model deriving from `Implementation`. Components stand for the active parts of the system like Service Handlers. They can be enabled or disabled. Resource classes relate to a specific resource wrapper which in turn relate to some specific model derived from `Resource`.

Furthermore, there is the possibility to create, enable and disable relations between components and specific resource instances or resource classes. These relations are used to determine whether a given component has access to a given resource or resource class. They allow to configure the behaviour e.g. of certain services and protect parts of an EOxServer instance from unwanted access.

As the number of registered components is quite large and as there are many interdependencies between them and to resource classes specific Component Managers shall be introduced in order to:

- group them to larger entities which are more easy to handle
- validate the configuration with respect to these interdependencies
- facilitate relation management
- automatically create the needed relations

These managers shall implement the common `ComponentManagerInterface` (page 152).

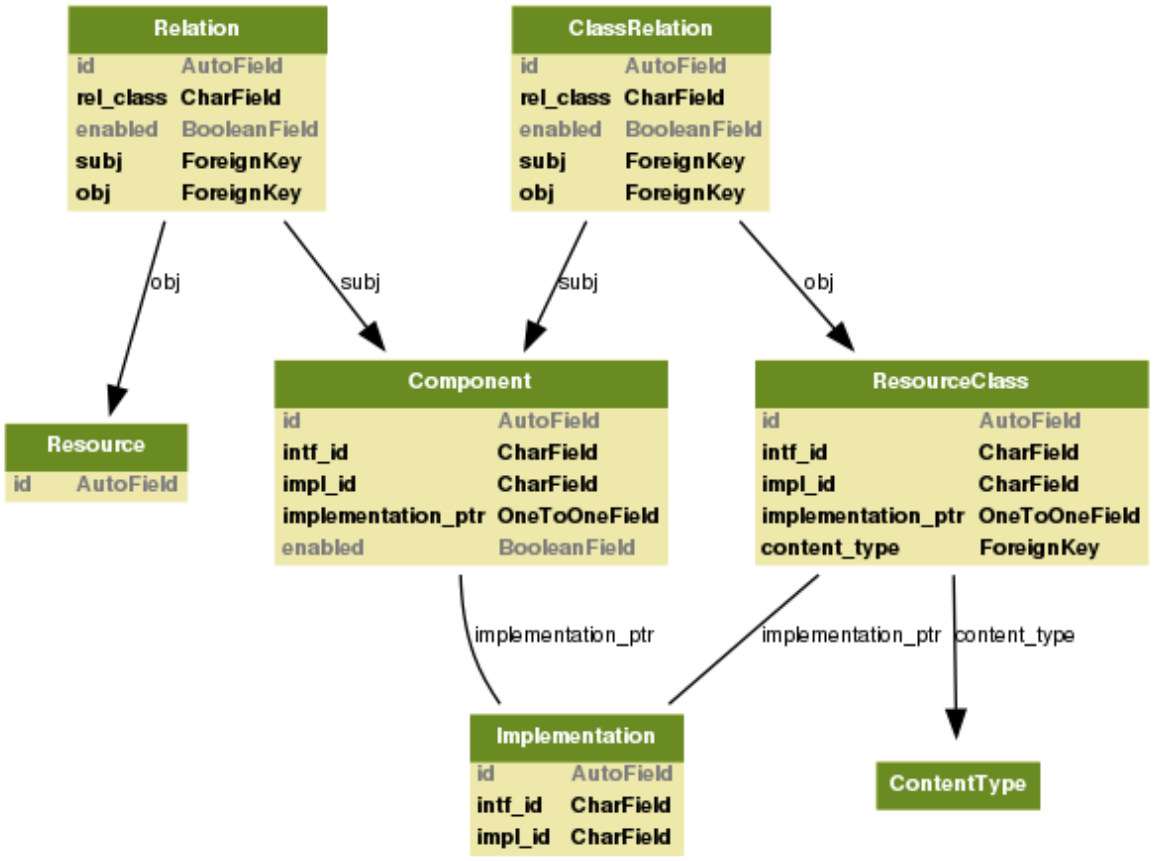


Figure 3.3: *Database Model for the Registry*

Detection The first step in the dynamic binding process provided by the registry is the detection of interfaces and implementations to be registered. For this end the registry loads the modules defined in the configuration files and searches them for descendants of `RegisteredInterface` (page 152) and their implementations. The metadata of the detected interfaces and implementations (the contents of‘‘REGISTRY_CONF‘‘) is ingested into the registry. This metadata is used for binding to the implementations, see the following subsection *Binding* (page 277) for details.

The main EOxServer configuration file `eoxserver.conf` contains options for determining which modules shall be scanned during the detection phase. The user can define single modules and whole directories to be searched for modules there.

Binding The registry provides four binding methods:

- direct binding
- KVP binding
- test binding
- factory binding

Direct binding means that the implementation to bind to is directly referenced by the caller using its implementation ID:

```
from eoxserver.core.system import System

impl = System.getRegistry().bind(
    "somemodule.SomeImplementation"
)
```

Direct binding is available for every implementation. You can also set the `binding_method` in the `REGISTRY_CONF` of an interface to `direct`, meaning that its implementations are reachable only by this method. This is used e.g. for component managers and factories.

The easiest method for parametrized dynamic binding is key-value-pair matching, or KVP binding. It is used if an interface defines `kvp` as its `binding_method`. The interface must then define in its `REGISTRY_CONF` one or more `registry_keys`, the implementations in turn must define `registry_values` for these keys. When looking up a matching implementation, the parameters given with the request are matched against these key-value-pairs. Finally, the registry returns an instance of the matching implementation:

```
from eoxserver.core.system import System

def dispatch(service_name, req):

    service = System.getRegistry().findAndBind(
        intf_id = "services.interfaces.ServiceHandler",
        params = {
            "services.interfaces.service": service_name.lower()
        }
    )

    response = service.handle(req)

    return response
```

This binding method is used e.g. for binding to service, version and operation handlers for OGC Web Services based on the parameters sent with the request.

A more flexible way to determine which implementation to bind to is the test binding method (`"binding_method": "testing"`). In this case, the interface must be derived from `TestingInterface` (page 152). The implementation must provide a `test()` (page 152) method which will be invoked by the registry in order to determine if it is suitable for a given set of parameters. This can be used e.g. to determine which format handler to use for a given dataset:

```
from eoxserver.core.system import System

format = System.getRegistry().findAndBind(
    intf_id = "resources.coverages.formats.FormatInterface",
    params = {
        "filename": filename
    }
)

...
```

The fourth binding method is factory binding (`"binding_method": "factory"`). In this case the registry invokes a factory that returns an instance of the desired implementation. Factories must be implementations of a descendant of `FactoryInterface` (page 152). Implementations and factories are linked together only at runtime, based on the metadata collected during the detection phase. This binding method is used e.g. for binding to instances of a resource wrapper:

```
from eoxserver.core.system import System

resource = System.getRegistry().getFromFactory(
    factory_id = "resources.coverages.wrappers.SomeResourceFactory",
    obj_id = "some_resource_id"
)
```

In order to access other functions of the factory you can bind to it directly. For retrieving all resources that are accessible through a factory you would use code like this:

```
from eoxserver.core.system import System

resource_factory = System.getRegistry().bind(
    "resources.coverages.wrappers.SomeResourceFactory"
)

resources = resource_factory.find()
```

Voting History

Motion To accept RFC 2

Voting Start 2011-07-25

Voting End 2011-09-15

Result +6 for ACCEPTED

Traceability

Requirements N/A

Tickets N/A

RFC 3: OGC Service Extensions

Author Stephan Krause

Created 2011-02-20

Last Edit 2011-02-20

Status IN PREPARATION

Discussion http://www.eoxserver.org/wiki/DiscussionRfc3

<short description of the RFC>

Introduction

<Mandatory. Overview of motivation, addressed problems and proposed solution>

Voting History N/A

Traceability

Requirements N/A

Tickets N/A

RFC 4: Data Packaging

Author Stephan Krause

Created 2011-02-20

Last Edit 2011-02-25

Status IN PREPARATION

Discussion http://www.eoxserver.org/wiki/DiscussionRfc4

<short description of the RFC>

Introduction

<Mandatory. Overview of motivation, addressed problems and proposed solution>

Voting History N/A

Traceability

Requirements N/A

Tickets N/A

RFC 5: Processing Chains

Author Stephan Krause

Created 2011-02-23

Last Edit 2011-03-01

Status IN PREPARATION

Discussion http://www.eoxserver.org/wiki/DiscussionRfc5

<short description of the RFC>

Introduction

<Mandatory. Overview of motivation, addressed problems and proposed solution>

Voting History N/A

Traceability

Requirements N/A

Tickets N/A

RFC 6: Directory Structure

Author Stephan Krause

Created 2011-02-24

Last Edit 2011-09-15

Status ACCEPTED

Discussion http://www.eoxserver.org/wiki/DiscussionRfc6

This RFC proposes a directory structure for the EOxServer distribution as well as EOxServer instances.

Introduction *RFC 1: An Extensible Software Architecture for EOxServer* (page 248) introduces a layered architecture for EOxServer as well as a separation of EOxServer distribution and instances. This RFC lays out a directory structure that is in line with this architecture.

Directory Structure

Distribution

- `core`: contains the modules of the Core
  - `util`: contains utility modules to be used throughout the project
- `services`: contains the modules of the Service Layer
  - `ows`: contains implementations of OGC Web Services
- `processing`: contains the modules of the Processing Layer
  - `processes`: contains processes
- `resources`: contains the modules of the Data Integration Layer
  - `coverages`: contains the modules related to coverage resources
    - `formats`: contains the modules related to coverage formats
  - `vector`: contains the modules related to vector data
    - `formats`: contains the modules related to vector data formats
- `contrib`: contains (links to) third party modules
- `conf`: contains the default configuration

Instance The instance directory contains the three Django project modules:

- `settings.py`
- `manage.py`
- `urls.py`

And the following subdirectories

- `conf`: configuration files
  - `eoxserver.conf`: the central EOxServer configuration
  - `template.map`: template MapFile for OWS requests

- `data`: database files
  - `config.sqlite`: SQLite database

Voting History

Motion To accept RFC 6

Voting Start 2011-07-25

Voting End 2011-09-15

Result +6 for ACCEPTED

Traceability

Requirements N/A

Tickets N/A

Distribution Core

The Core shall act as a "glue" for EOxServer that links the different parts of the software together and provides functionality used throughout the EOxServer project.

It defines the core of the configuration data model which is extended by the layers and implementing modules. The configuration is partly stored in the database and partly in files. Both parts need to be easily modifiable and extensible.

Therefore the Core also includes an administration client that can be used by system operators to edit the part of the configuration stored in the database. The basic functionality of the administrator, the entry view and its extension mechanisms shall be part of the Core.

The Core includes modules for common use, for instance utilities for the handling of spatio-temporal metadata as well as for decoding and encoding of XML documents.

Most importantly, the Core contains the central logic that enables the dynamic extension of system functionality. The layers shall provide interface definitions based on the extension model of the Core that can be implemented by extending modules and plugins. For more details see *RFC 2: Extension Mechanism for EOxServer* (page 270).

Service Layer

The Service Layer contains the OWS request handling logic as well as the implementation of services and service extensions.

It defines a configuration **data model** for OGC Web Services and for their metadata. The model includes:

- service metadata to be published in the GetCapabilities response
- options to enable or disable a specific service or service extension for a given data source
- options to configure the services themselves, e.g. enabling or disabling certain non-mandatory features

The Service Layer provides **views** for public access, namely the central entrance point for OWS requests. It also contains views for the administration client that allow to configure services and service metadata.

The **core handling logic** for OGC Web Services is part of the Service Layer. It implements the behaviour defined by OWS Common and defines a structured approach to request handling that discerns different levels:

- services
- service versions
- service extensions
- service operations

The way services and service extensions interact is described in further detail in *RFC 3: OGC Service Extensions* (page 279).

The Service Layer defines request handler **interfaces** for each of these levels that are **implemented** by modules for:

- WPS
- WCS
  - EO-WCS
  - WCS-T
- WMS
  - EO-WMS
- WFS

Processing Layer

The Processing Layer contains the processing models used internally by EOxServer as well as the data model and the basic handling routines for processes to be published using WPS.

In its **data model** it defines the configuration options and metadata for processes. The model shall also support processing chains as described in further detail in *RFC 5: Processing Chains* (page 280). The Processing Layer publishes administration client **views** to support the configuration of processes and processing chains.

The Processing Layer defines **interfaces** for processes. It also contains implementations of the processes used internally by EOxServer; these include:

- coverage tiling
- coverage mosaicking

Further processes as required e.g. by the O3S Use Cases will be added as plugins based on the data model and interface definitions of the Processing Layer.

Data Integration Layer

The Data Integration Layer shall provide data models for resources as well as an abstraction layer for different data formats and data packaging formats.

Data packaging formats are explained in greater detail in *RFC 4: Data Packaging* (page 279). Roughly speaking, they represent the way data and metadata for an EO product or derived product are packaged. They shall abstract from the actual substructure of the packaging format in directories and files so these resources can be handled transparently by EOxServer.

Its **data model** shall include items common to all types of data as well as individual models for:

- coverages
- vector data
- metadata

Just as the other layers the Data Integration Layer shall publish administration client **views** that support adding, modifying and removal of resources and their respective metadata.

The **interface definitions** of the Data Integration Layer shall provide an abstraction layer for:

- various data formats
- various metadata formats
- various data packaging formats

The modules **implementing** these interfaces shall support:

- coverage data formats supported by:
  - GDAL[34]
  - NEST[35] (optional)
- vector data formats supported by OGR[36]
- metadata formats:
  - EO-GML
  - DIMAP (optional)
  - INSPIRE (optional)
  - GSC-DA (optional)
- data packaging formats:
  - directories
  - ZIP archives
  - TAR archives
  - compressed file formats:
    - ZIP
    - GZIP
    - BZ2

Data Access Layer

The Data Access Layer shall provide transparent access to local and remote data using different backends. It constitutes an abstraction layer for data sources.

Its **data model** therefore provides configuration options for the backends. It contains **views** for the administration client to configure different data sources.

The Data Access Layer is built around the **interface definitions** of backends and data sources stored by them. The following backends need to be **implemented**:

- local backends:
  - file system
  - rasdaman[37] backend
- remote backends:
  - using HTTP/HTTPS
  - using FTP
  - using WCS

Instances

EOxServer instances are Django projects that import different EOxServer modules as Django applications.

Like every Django project they contain a settings file that governs the Django configuration and in addition the most basic parts of EOxServer configuration. Specifically:

[34] http://www.gdal.org
[35] http://www.array.ca/nest
[36] http://www.gdal.org/ogr/
[37] http://www.rasdaman.com

- the connection details for the database containing the EOxServer configuration is defined in the settings file
- the Django `INSTALLED_APPS` setting must be used to define the parts of the EOxServer data model that shall be loaded
- some EOxServer configuration settings that are needed in the startup phase will be appended to the Django settings file

Apart from the settings, every Django project has an "urlconf" that defines which URLs shall point to the different views of the project. For using the full EOxServer functionality there have to be URLs pointing to the Service Layer OWS entrance point and the administration client entrance point defined by the EOxServer core.

Furthermore the instance contains the Django configuration files whose content is defined by the configuration data model of the Core and the layers.

Optionally, the instance directory may include subdirectories for the data (if stored locally) and the database (if using the file-based SpatiaLite spatial database backend).

Finally, in a production setting, it shall contain the modules needed to deploy the instance. The favourite deployment method is WSGI (see **PEP 333**[38]). These must be configured as well to include the path to the instance.

The Django project may or may not contain applications itself, which may or may not use EOxServer functionality. Writing an own application is not necessary to use EOxServer, though; placing links to EOxServer views in the urlconf is sufficient.

Voting History

Moved to ACCEPTED by unanimous consent without a formal vote on July 20th, 2011.

Traceability

Requirements HMA-FO SR_ODA_IF_070, O3S_CAP_001[39], O3S_CAP_013[40], O3S_CAP_014[41], O3S_CAP_017[42], O3S_CAP_100[43], O3S_CAP_150[44], O3S_CAP_200[45], O3S_CAP_220[46], O3S_CAP_240[47], O3S_CAP_260[48], O3S_QUA_004[49]

Tickets N/A

3.3.3 RFC 2: Extension Mechanism for EOxServer

Author Stephan Krause

Created 2011-02-20

Last Edit 2011-09-15

Status ACCEPTED

Discussion http://www.eoxserver.org/wiki/DiscussionRfc2

This RFC proposes an extension mechanism that allows to integrate extension modules and plugins dynamically into the EOxServer distribution and instances.

[38] http://www.python.org/dev/peps/pep-0333
[39] https://o3s.eox.at/requirements/ticket/7
[40] https://o3s.eox.at/requirements/ticket/68
[41] https://o3s.eox.at/requirements/ticket/69
[42] https://o3s.eox.at/requirements/ticket/183
[43] https://o3s.eox.at/requirements/ticket/8
[44] https://o3s.eox.at/requirements/ticket/198
[45] https://o3s.eox.at/requirements/ticket/9
[46] https://o3s.eox.at/requirements/ticket/204
[47] https://o3s.eox.at/requirements/ticket/210
[48] https://o3s.eox.at/requirements/ticket/214
[49] https://o3s.eox.at/requirements/ticket/122

Introduction

RFC 1: An Extensible Software Architecture for EOxServer (page 248) proposes an extensible architecture for EOxServer in order to ensure

- modularity
- extensibility
- flexibility

of the design. It establishes the need for an extension mechanism which acts as a sort of "glue" between different parts of the architecture and enables dynamic binding to these components.

This RFC discusses the extension mechanism in further detail and identifies the architectural principles and components needed to implement it.

The constituent components of the extension mechanism design are interface declarations, the respective implementations and a central registry that contains metadata about interfaces and implementations and enables dynamic binding to the latter ones.

Requirements

RFC 1: An Extensible Software Architecture for EOxServer (page 248) proposes an extension mechanism for EOxServer. It shall assure extensibility by additional modules and plugins and provide functionality to enable dynamic binding to extending modules.

In the layered architecture of RFC 1 the *Distribution Core* (page 267) shall be the place where the central logic that enables the dynamic extension of system functionality resides. The layers shall provide interface definitions based on the extension model of the Core that can be implemented by extending modules and plugins.

Now which extensions are needed and which requirements do they impose on the extension mechansims? Digging deeper we have a look at the four architectural layers of EOxServer and analyze the interfaces and implementations needed by each of them.

The *Service Layer* (page 267) defines a structured approach to OGC Web Service (OWS) request handling that discerns different levels:

- services
- service versions
- service extensions
- service operations

For all of these levels interfaces are defined that are implemented by extending modules for specific OWS and their different versions and extensions.

The *Processing Layer* (page 268) defines interfaces for processes and processing chains (see *RFC 5: Processing Chains* (page 280)). Some of these are used internally and integrated into the distribution, most will be provided by plugins. While the process interface needs to be generic in order to make the implementation of many different processes possible, it must be concise enough to allow binding between processes in a processing chain. So, this must be sustained by the extension mechanism as well.

The *Data Integration Layer* (page 268) shall provide an abstraction layer for different data formats, metadata formats and data packaging formats. This shall be achieved using common interfaces for coverage data, vector data and metadata respectively.

Data and packaging formats are often not known by the system before ingesting a dataset. Thus, some kind of autodetection of formats is necessary. This is provided partly by the underlying libraries such as GDAL[50], but shall also be considered for the design of the extension mechanism: it must be possible to dynamically bind to the right data, metadata and data packaging format based on evaluations of the data. These tests should be implemented by format extensions and supported by the extension mechansim.

[50] http://www.gdal.org

The *Data Access Layer* (page 269) is built around the interface definitions of backends and data sources stored by them.

In addition to modularity and extensibility RFC 1 states that the system shall be

> flexible in the sense that it must be possible to select different combinations of modules to deploy and activate

Modules can be combined to build a specific application. From a user perspective it is essential to be able to activate and deactivate services, service versions and service extensions globally and/or separately for each resource or process. The same applies for other extensible parts of the system such as backends.

The O3S Use Case 2 for instance requires a server setup that consists of:

- local and WCS backends in the Data Access Layer
- a specific combination of coverage, vector data, metadata and packaging formats in the Data Integration Layer
- a feature detection process in the Processing Layer
- WPS and WFS implementations in the Service Layer

All other backends, services and processes shall be disabled.

Summarizing the requirements the extension mechanism shall support:

- extensibility by additional modules and plugins
- dynamic binding
- interface definitions for extensions
- implementations that can be enabled or disabled
  - globally
  - per resource or per process
- modules that can be configured dynamically to build an application
- autodetection of data, metadata and data packaging formats

Extension Mechanism

The basic questions for the design of the extension mechanism are:

- how to declare extensible interfaces
- how to design implementations of these interfaces
- how to advertise them
- how to bind to them

Unlike Java or C++, Python does not have a built-in mechanism to declare interfaces. A method definition always comes with an implementation. With Python 2.6 support for abstract base classes and abstract methods was added, but at the moment it is not an option to use this framework as this would break support for earlier Python versions.

So, two basic design options remain:

- using conventional Python classes and inheritance mechanisms for interfaces and implementations
- customize the interface declaration and implementation creation using Python metaclasses

Whereas the first approach is easier, the second one can provide more control and a clear differentiation between interface declaration and implementation. Both design options are discussed in further detail in the *Interfaces and Implementations* (page 273) section below.

The second major topic is how to find and bind to implementations of an interface if not all implementations are known to the system a priori, as is the case with plugins. Some "glue" is needed that holds the system together and

allows for dynamic binding. In the case of EOxServer this is implemented by a central registry that keeps track of implementations by automatically scanning Python modules in certain directories that are supposed to contain EOxServer extending modules or plugins. For more details on the basics of *Registry* (page 275) see below.

In most cases an instance of EOxServer will not need all the functionality provided by the distribution or plugins installed on the system. Dynamic binding allows for enabling and disabling certain services, processes, formats, backends and plugins in an interactive way using the administration client. In order to assure this required functionality a configuration data model is needed that allows to store information about what parts of the system are activated and what resources they may operate on. See the section *Data Model* (page 275) for further details.

Implementations of interfaces are not isolated objects. They depend on libraries, functionality provided by the EOxServer core and layers and, last but not least, on other interface implementations. In order to assure that the dynamically configurable system is in a consistent state, the interdependencies between implementations need to be properly advertised and stored in the configuration data model.

After this short overview, we will go more in depth in the following sections.

Interfaces and Implementations

As already discussed before there are two design options for interfaces and implementations:

- interfaces and implementations as conventional Python classes that are linked through inheritance
- interfaces as special Python classes that are linked to implementations by a custom mechanism.

Whereas the first approach is straightforward and easy to implement and handle it has also some serious drawbacks. Most importantly it does not allow for a clear separation between interface declaration and implementation. A method declared in the interface always must contain an implementation, and an implementation may change the signature of the methods it implements in any possible way.

What's more, as the implementation inherits (mostly generic) method code from the interface there is no way to validate if it actually defines concrete methods to override the "abstract" ones the interface class provides.

So, there are also good reasons for the second approach although it is more challenging for developers. The approach proposed here allows to customize class generation and inheritance enabling validation at "compile time" (i.e. when classes are created) and runtime (i.e. when instance methods are invoked) as well as separation of interface definition from implementation.

How can this be achieved? The proposed mechanism relies on an interface base class called `Interface` that concrete interface declarations can derive from, implementing code contained in a conventional Python class and a method called `implement()` that generates a special implementation class from the interface declaration and the class containing the implementing code.

Interface Declaration

It has already been said that interface declarations shall derive from a common base class called `Interface`. But that is not the end of the story - one big question remains: how to declare actual methods without implementation? The proposed approach is not to declare methods as such at all, but use classes representing them instead.

For this end three classes are to be defined alongside the `Interface` base class.

- instances of the `Constant` class represent constants defined by the interface
- instances of the `Method` class represent methods
- instances of the `Arg` class represent method arguments; subclasses of `Arg` allow for type validation, e.g. instances of `IntArg` represent integer arguments

Let's have a look at a quick example:

```
from eoxserver.core.interfaces import Interface, Method, Arg

class ServiceInterface(Interface):
    handle = Method(
```

```
        Arg("req")
    )
```

Note: Code examples in this RFC are merely informational. The actual implementation may deviate a little bit from them. A reference documentation will be prepared for the definitive extension mechanism.

This snippet of Python code represents a simple and complete interface declaration. The `ServiceInterface` class will be used in further examples as well. It shows a method definition that declares the following: the method `handle` shall take one argument of arbitrary type named `req` that stands for an OWS request.

As you can see the declaration is a class variable containing an instance of the `Method` class. It is not a method (it does not even have to be callable). It serves two purposes:

- documentation of the interface
- validation of the implementation

Thinking of these two goals, the writer of the code could have been more rigorous and declare an argument like this:

```
handle = Method(
    ObjectArg("req", arg_class=OWSRequest)
)
```

That way it is documented what kind of argument is expected. When defining the implementation it is enforced that it have a method `handle` which takes exactly one argument besides `self`, otherwise an exception will be raised. When invoking an interface of the implementation it can be validated that the argument is of the right type. More on this later under *Validation of Implementations* (page 274). Now let's have a look at implementations.

Implementations

The proposed design of interface implementation intends to hide all the complexity of this process from the developers of implementations. They just have to write an implementing class which is a normal new-style Python class, and wrap it with the `implement()` method of the interface, such as in the following example:

```
from eoxserver.services.owscommon import ServiceInterface

class WxSService(object):

    def handle(self, req):

        # ...

        return response

WxSServiceImplementation = ServiceInterface.implement(WxSService)
```

The call to `implement()` ensures validation of the interface and produces an implementation class that inherits all the code of the implementing class and contains information about the interface. This is only the basic functionality of the interface implementation process: more is to be revealed in the following sections.

Validation of Implementations

The validation of implementations is performed in two ways:

- at class creation time
- at instance method invocation time

Validation at class creation time checks:

- if all methods declared by the interface are implemented
- if the method arguments of the interface and implementation match

Class creation time validation is performed unconditionally.

Instance method invocation time ("runtime") validation is optional. It can be triggered by the `runtime_validation_level` setting. There are three possible values for this option:

- `trust`: no runtime validation
- `warn`: argument types are checked against interface declaration; in case of mismatch a warning is written to the log file
- `fail`: argument types are checked against interface declaration; in case of mismatch an exception is raised

The `runtime_validation_level` option can be set

- globally (in configuration file)
- per interface
- per implementation

where stricter settings override weaker ones.

Note: The `warn` and `fail` levels are intended for use throughout the development process. In a production setting `trust` should be used.

Registry

The Registry is the core component for managing the extension mechanism of EOxServer. It is the central entry point for:

- automated detection of registered interfaces and implementations
- dynamical binding to the implementations
- configuration of components and relations between them

Its functionality shall be discussed in further detail in the following subsections:

- *Data Model* (page 275)
- *Detection* (page 277)
- *Binding* (page 277)

Data Model

The data model for the Extension Mechanism including dynamic binding is implemented primarily by the *Registry* (page 275); for persistent information it relies on the configuration files and the database.

As you'd expect, the Registry data model relies on interfaces and implementations. However, not all of them are registered, but only descendants of `RegisteredInterface` (page 152) and their respective implementations. `RegisteredInterface` (page 152) extends the configuration model for interfaces with information relevant to the registration and dynamic binding processes. This is an example for a valid configuration:

```
from eoxserver.core.registry import RegisteredInterface

class SomeInterface(RegisteredInterface):

    REGISTRY_CONF = {
        "name": "Some Interface",
        "intf_id": "somemodule.SomeInterface",
```

```
        "binding_method": "direct"
    }
```

The most important parts are the interface ID `intf_id` and the `binding_method` settings which will be used by the registry to find implementations of the interface and to determine how to bind to them. For more information see the *Binding* (page 277) section below.

The registry model is accompanied by a database model that allows to store persistently which parts of the system (services, plugins, etc.) are enabled and which resources they have access to.

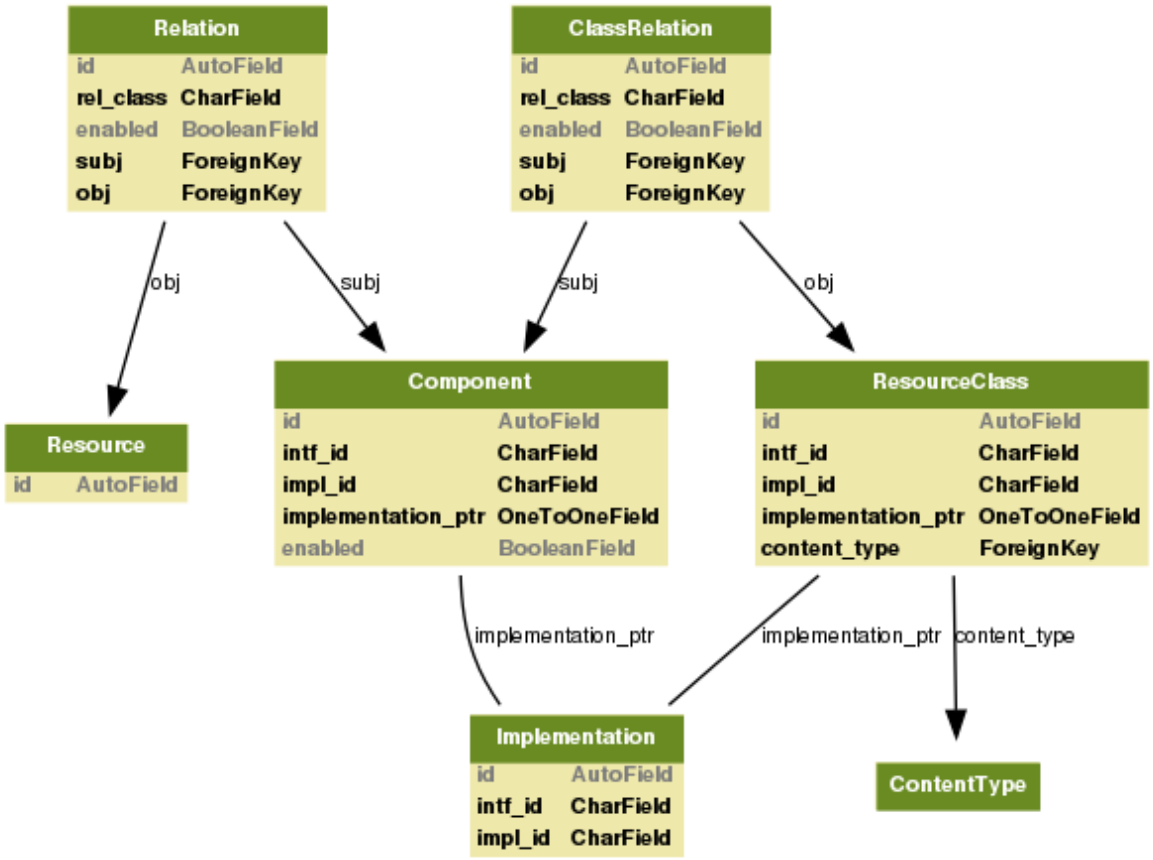


Figure 3.4: *Database Model for the Registry*

For every registered implementation an `Implementation` instance and database record are created. Implementations are subdivided into components and resource classes, each with their respective model deriving from `Implementation`. Components stand for the active parts of the system like Service Handlers. They can be enabled or disabled. Resource classes relate to a specific resource wrapper which in turn relate to some specific model derived from `Resource`.

Furthermore, there is the possibility to create, enable and disable relations between components and specific resource instances or resource classes. These relations are used to determine whether a given component has access to a given resource or resource class. They allow to configure the behaviour e.g. of certain services and protect parts of an EOxServer instance from unwanted access.

As the number of registered components is quite large and as there are many interdependencies between them and to resource classes specific Component Managers shall be introduced in order to:

- group them to larger entities which are more easy to handle
- validate the configuration with respect to these interdependencies
- facilitate relation management

- automatically create the needed relations

These managers shall implement the common `ComponentManagerInterface` (page 152).

Detection

The first step in the dynamic binding process provided by the registry is the detection of interfaces and implementations to be registered. For this end the registry loads the modules defined in the configuration files and searches them for descendants of `RegisteredInterface` (page 152) and their implementations. The metadata of the detected interfaces and implementations (the contents of‘‘REGISTRY_CONF‘‘) is ingested into the registry. This metadata is used for binding to the implementations, see the following subsection *Binding* (page 277) for details.

The main EOxServer configuration file `eoxserver.conf` contains options for determining which modules shall be scanned during the detection phase. The user can define single modules and whole directories to be searched for modules there.

Binding

The registry provides four binding methods:

- direct binding
- KVP binding
- test binding
- factory binding

Direct binding means that the implementation to bind to is directly referenced by the caller using its implementation ID:

```
from eoxserver.core.system import System

impl = System.getRegistry().bind(
    "somemodule.SomeImplementation"
)
```

Direct binding is available for every implementation. You can also set the `binding_method` in the `REGISTRY_CONF` of an interface to `direct`, meaning that its implementations are reachable only by this method. This is used e.g. for component managers and factories.

The easiest method for parametrized dynamic binding is key-value-pair matching, or KVP binding. It is used if an interface defines `kvp` as its `binding_method`. The interface must then define in its `REGISTRY_CONF` one or more `registry_keys`, the implementations in turn must define `registry_values` for these keys. When looking up a matching implementation, the parameters given with the request are matched against these key-value-pairs. Finally, the registry returns an instance of the matching implementation:

```
from eoxserver.core.system import System

def dispatch(service_name, req):

    service = System.getRegistry().findAndBind(
        intf_id = "services.interfaces.ServiceHandler",
        params = {
            "services.interfaces.service": service_name.lower()
        }
    )

    response = service.handle(req)

    return response
```

This binding method is used e.g. for binding to service, version and operation handlers for OGC Web Services based on the parameters sent with the request.

A more flexible way to determine which implementation to bind to is the test binding method (`"binding_method": "testing"`). In this case, the interface must be derived from `TestingInterface` (page 152). The implementation must provide a `test()` (page 152) method which will be invoked by the registry in order to determine if it is suitable for a given set of parameters. This can be used e.g. to determine which format handler to use for a given dataset:

```
from eoxserver.core.system import System

format = System.getRegistry().findAndBind(
    intf_id = "resources.coverages.formats.FormatInterface",
    params = {
        "filename": filename
    }
)

...
```

The fourth binding method is factory binding (`"binding_method": "factory"`). In this case the registry invokes a factory that returns an instance of the desired implementation. Factories must be implementations of a descendant of `FactoryInterface` (page 152). Implementations and factories are linked together only at runtime, based on the metadata collected during the detection phase. This binding method is used e.g. for binding to instances of a resource wrapper:

```
from eoxserver.core.system import System

resource = System.getRegistry().getFromFactory(
    factory_id = "resources.coverages.wrappers.SomeResourceFactory",
    obj_id = "some_resource_id"
)
```

In order to access other functions of the factory you can bind to it directly. For retrieving all resources that are accessible through a factory you would use code like this:

```
from eoxserver.core.system import System

resource_factory = System.getRegistry().bind(
    "resources.coverages.wrappers.SomeResourceFactory"
)

resources = resource_factory.find()
```

Voting History

Motion To accept RFC 2

Voting Start 2011-07-25

Voting End 2011-09-15

Result +6 for ACCEPTED

Traceability

Requirements N/A

Tickets N/A

3.3.4 RFC 3: OGC Service Extensions

Author Stephan Krause

Created 2011-02-20

Last Edit 2011-02-20

Status IN PREPARATION

Discussion http://www.eoxserver.org/wiki/DiscussionRfc3

<short description of the RFC>

Introduction

<Mandatory. Overview of motivation, addressed problems and proposed solution>

Voting History

N/A

Traceability

Requirements N/A

Tickets N/A

3.3.5 RFC 4: Data Packaging

Author Stephan Krause

Created 2011-02-20

Last Edit 2011-02-25

Status IN PREPARATION

Discussion http://www.eoxserver.org/wiki/DiscussionRfc4

<short description of the RFC>

Introduction

<Mandatory. Overview of motivation, addressed problems and proposed solution>

Voting History

N/A

Traceability

Requirements N/A

Tickets N/A

3.3.6 RFC 5: Processing Chains

Author Stephan Krause

Created 2011-02-23

Last Edit 2011-03-01

Status IN PREPARATION

Discussion http://www.eoxserver.org/wiki/DiscussionRfc5

<short description of the RFC>

Introduction

<Mandatory. Overview of motivation, addressed problems and proposed solution>

Voting History

N/A

Traceability

Requirements N/A

Tickets N/A

3.3.7 RFC 6: Directory Structure

Author Stephan Krause

Created 2011-02-24

Last Edit 2011-09-15

Status ACCEPTED

Discussion http://www.eoxserver.org/wiki/DiscussionRfc6

This RFC proposes a directory structure for the EOxServer distribution as well as EOxServer instances.

Introduction

RFC 1: An Extensible Software Architecture for EOxServer (page 248) introduces a layered architecture for EOxServer as well as a separation of EOxServer distribution and instances. This RFC lays out a directory structure that is in line with this architecture.

Directory Structure

Distribution

- `core`: contains the modules of the Core
  - `util`: contains utility modules to be used throughout the project
- `services`: contains the modules of the Service Layer
  - `ows`: contains implementations of OGC Web Services
- `processing`: contains the modules of the Processing Layer

    - `processes`: contains processes
- `resources`: contains the modules of the Data Integration Layer
    - `coverages`: contains the modules related to coverage resources
        * `formats`: contains the modules related to coverage formats
    - `vector`: contains the modules related to vector data
        * `formats`: contains the modules related to vector data formats
- `contrib`: contains (links to) third party modules
- `conf`: contains the default configuration

Instance

The instance directory contains the three Django project modules:

- `settings.py`
- `manage.py`
- `urls.py`

And the following subdirectories

- `conf`: configuration files
    - `eoxserver.conf`: the central EOxServer configuration
    - `template.map`: template MapFile for OWS requests
- `data`: database files
    - `config.sqlite`: SQLite database

Voting History

Motion To accept RFC 6

Voting Start 2011-07-25

Voting End 2011-09-15

Result +6 for ACCEPTED

Traceability

Requirements N/A

Tickets N/A

3.3.8 RFC 7: Release Guidelines

Author Stephan Meißl

Created 2011-05-04

Last Edit $Date$

Status ACCEPTED

Discussion http://eoxserver.org/wiki/DiscussionRfc7

Id Id

Overview

This RFC documents the EOxServer release manager role and the phases of EOxServer's release process.

(Credit: Inspired by the MapServer release guidelines at: http://mapserver.org/development/rfc/ms-rfc-34.html)

The EOxServer Release Manager Role

For every release of EOxServer, the PSC elects a release manager via motion and vote on the dev mailing list.

The overall role of the release manager is to coordinate the efforts of the developers, testers, documentation, and other contributors to lead to a release of the best possible quality within the scheduled timeframe.

The PSC delegates to the release manager the responsibility and authority to make certain final decisions for a release, including:

- Approving or not the release of each beta, release candidate, and final release
- Approving or rejecting non-trivial bug fixes or changes after the feature freeze
- Maintaining the release schedule (timeline) and making changes as required

When in doubt or for tough decisions (e.g. pushing the release date by several weeks) the release manager is free to ask the PSC to vote in support of some decisions, but this is not a requirement for the areas of responsibility above.

The release manager's role also includes the following tasks:

- Setup and maintain a release plan wiki page for each release
- Coordinate with the developers team
- Coordinate with the QA/testers team
- Coordinate with the docs/website team
- Keep track of progress via Trac milestones and ensure tickets are properly targeted
- Organize IRC meetings if needed (including agenda and minutes)
- Tag source code in SVN for each beta, RC, and release
- Branch source code in SVN after the final release (trunk becomes the next dev version)
- Update version in files for each beta/RC/release
- Package source code distribution for each beta/RC/release
- Update appropriate website/download page for each beta/RC/release
- Make announcements on users and announce mailing lists for each release
- Produce and coordinate bugfix releases as needed after the final release

Any of the above tasks can be delegated but they still remain the ultimate responsibility of the release manager.

The EOxServer Release Process

The normal development process of a EOxServer release consists of various phases.

- Development phase

  The development phase usually lasts several months. New features are proposed via RFCs and voted by the EOxServer PSC.

- RFC freeze date

  For each release there is a certain date by which all new feature proposals (RFCs) must have been submitted for review. After this date no features will be accepted anymore for this particular release.

- Feature freeze date / Beta releases

  By this date all features must have been completed and all code has to be integrated. Only non-invasive changes, user interface work and bug fixes are done now. There are usually 3 to 4 betas and a couple of release candidates before the final release.

- Release Candidate

  Ideally, the last beta that is bug free. No changes to the code. Should not require any migration steps apart from the ones required in the betas. If any problems are found and fixed, a new release candidate is issued.

- Final release / Expected release date

  Normally the last release candidate that is issued without any show-stopper bugs.

- Bug fix releases

  No software is perfect. Once a sufficient large or critical number of bugs have been found for a certain release, the release manager releases a new bug fix release a.k.a. third-dot release.

EOxServer Version Numbering

EOxServer's version numbering scheme is very similar to Linux's. For example, a EOxServer version number of 1.2.5 can be decoded as such:

- 1: Major version number.

  The major version number usually changes when significant new features are added or when major architectural changes or backwards incompatibilities are introduced.

- 2: Minor version number.

  Increments in minor version number almost always relate to additions in functionality.

- 5: Revision number.

  Revisions are bug fixes only. No new functionality is provided in revisions.

Voting History

Motion Adopted on 2011-11-16 with +1 from Stephan Meißl, Milan Novacek, Martin Paces

Traceability

Requirements N/A

Tickets N/A

3.3.9 RFC 8: SVN Commit Management

Author Stephan Meißl

Created 2011-05-04

Last Edit 2011-05-18

Status ACCEPTED

Discussion http://www.eoxserver.org/wiki/DiscussionRfc8

Overview

This RFC documents the EOxServer guidelines for SVN commit access and specifies some guidelines for SVN committers.

(Credit: Inspired by the MapServer SVN commit management guidelines at: http://mapserver.org/development/rfc/ms-rfc-7.1.html)

Election to SVN Commit Access

Permission for SVN commit access shall be provided to new developers only if accepted by the EOxServer Project Steering Committee (PSC). A proposal should be written to the PSC for new committers and voted on normally. It is not necessary to write an RFC document for these votes. An e-mail to the dev mailing list is sufficient.

Removal of SVN commit access should be handled by the same procedure.

The new committer should have demonstrated commitment to EOxServer and knowledge of the EOxServer source code and processes to the committee's satisfaction, usually by reporting tickets, submitting patches, and/or actively participating in the various EOxServer forums.

The new committer should also be prepared to support any new feature or changes that he/she commits to the EOxServer source tree in future releases, or to find someone to which to delegate responsibility for them if he/she stops being available to support the portions of code that he/she is responsible for.

All committers should also be a member of the dev mailing list so they can stay informed on policies, technical developments, and release preparation.

Committer Tracking

A list of all project committers will be kept in the main eoxserver directory (called COMMITTERS) listing for each SVN committer:

- Userid: the id that will appear in the SVN logs for this person.
- Full name: the users actual name.
- Email address: A current email address at which the committer can be reached. It may be altered in normal ways to make it harder to auto-harvest.
- A brief indication of areas of responsibility.

SVN Administrator

One member of the PSC will be appointed the SVN Administrator. That person is responsible for giving SVN commit access to folks, updating the COMMITTERS file, and other SVN related management. Initially Stephan Meißl will be the SVN Administrator.

SVN Commit Practices

The following are considered good SVN commit practices for the EOxServer project.

- Use meaningful descriptions for SVN commit log entries.
- Add a ticket reference like "(#1232)" at the end of SVN commit log entries when committing changes related to a ticket in Trac.
- Include changeset revision numbers like "r7622" in tickets when discussing relevant changes to the code-base.
- Changes should not be committed in stable branches without a corresponding ticket. Any change worth pushing into a stable version is worth a Trac ticket.

- Never commit new features to a stable branch: only critical fixes. New features can only go in the main development trunk.
- Only ticket defects should be committed to the code during pre-release code freeze.
- Significant changes to the main development version should be discussed on the dev maling list before making them, and larger changes will require an RFC approved by the PSC.
- Do not create new branches without the approval of the PSC. A Release manager designated under *RFC 7: Release Guidelines* (page 281) is automatically granted permission to create a branch, as defined by their role described in *RFC 7: Release Guidelines* (page 281).
- All source code in SVN should be in Unix text format as opposed to DOS text mode.
- When committing new features or significant changes to existing source code, the committer should take reasonable measures to insure that the source code continues to work.
- Include the standard EOxServer header in every new file and set the following SVN properties:
  - svn propset svn:keywords 'Author Date Id Rev URL' <new_file>
  - svn propset svn:eol-style native <new_file>

Legal

Commiters are the front line gatekeepers to keep the code base clear of improperly contributed code. It is important to the EOxServer users and developers to avoid contributing any code to the project without it being clearly licensed under the project license.

Generally speaking the key issues are that those providing code to be included in the repository understand that the code will be released under the EOxServer License, and that the person providing the code has the right to contribute the code. For the committer themselves understanding about the license is hopefully clear. For other contributors, the committer should verify the understanding unless the committer is very comfortable that the contributor understands the license (for instance frequent contributors).

If the contribution was developed on behalf of an employer (on work time, as part of a work project, etc) then it is important that an appropriate representative of the employer understand that the code will be contributed under the EOxServer License. The arrangement should be cleared with an authorized supervisor/manager, etc.

The code should be developed by the contributor, or the code should be from a source which can be rightfully contributed such as from the public domain, or from an open source project under a compatible license.

All unusual situations need to be discussed and/or documented.

Committers should adhere to the following guidelines, and may be personally legally liable for improperly contributing code to the source repository:

- Make sure the contributor (and possibly employer) is aware of the contribution terms.
- Code coming from a source other than the contributor (such as adapted from another project) should be clearly marked as to the original source, copyright holders, license terms and so forth. This information can be in the file headers, but should also be added to the project licensing file if not exactly matching normal project licensing (eoxserver/COPYING and eoxserver/README).
- Existing copyright headers and license text should never be stripped from a file. If a copyright holder wishes to give up copyright they must do so in writing to the project before copyright messages are removed. If license terms are changed it has to be by agreement (written in email is ok) of the copyright holders.
- When substantial contributions are added to a file (such as substantial patches) the author/contributor should be added to the list of copyright holders for the file.
- If there is uncertainty about whether a change it proper to contribute to the code base, please seek more information from the PSC.

Voting History

Motion Adopted on 2011-05-17 with +1 from Arndt Bonitz, Stephan Krause, Stephan Meißl, Milan Novacek, Martin Paces, Fabian Schindler

Traceability

Requirements N/A

Tickets N/A

3.3.10 RFC 9: SOAP Binding of WCS GetCoverage Response

Author Milan Novacek

Created 2011-05-17

Last Edit 2011-05-30

Status ACCEPTED

Discussion http://www.eoxserver.org/wiki/DiscussionRfc9

Introduction

The current/draft OGC specifications for the SOAP binding for a WCS GetCoverage Response are inconsistent with the SOAP spec if the GetCoverage response includes a binary file. This RFC proposes an update to OGC 09-149r1 to resolve the inconsistencies: Requirements 5 and 6 should be changed to use SOAP MTOM where the entire coverage response comprises the attachment. This coverage attachment is referred to from within a new element 'Coverage' which is also defined as part of this RFC.

Problem Description

In OGC 09-149r1, Requirement 5 mandates that a GetCoverage SOAP response shall be encoded as "SOAP with Attachments" as defined in [W3C Note 11], but using SOAP 1.2 rather than SOAP 1.1. Requirement 6 says, rather imprecisely, that in a GetCoverage response, the SOAP Envelope shall contain one Body element which contains the Coverage as its single element.

For binary attachments to SOAP 1.2 messages, W3C recommends the usage of MTOM instead of SwA (see [1] and [2]). According to the guidance in [1], the SOAP 1.2 MTOM standard requires the use of the xop:Include element to refer to binary attachments. The difficulty arises because the "gml:rangeSet" element, which according to OGC 09-110r is mandated for a GetCoverage response, does not have a provision for using the xop:Include element to refer to an attached file. For this reason one cannot include a reference to an MTOM SOAP attachment in the GetCoverage response.

Proposed Changes to OGC 09-149r1

To resolve the problem, we propose to update two requirements of OGC 09-149r1 as follows:

> **Requirement 5:** A GetCoverage SOAP response **shall** be encoded according to the W3C SOAP 1.2 standard [http://www.w3.org/TR/soap12-part1/] using MTOM [http://www.w3.org/TR/soap12-mtom/].
>
> **Requirement 6:** In a GetCoverage response, the SOAP Body **shall** contain one element, "Coverage" of type "SoapCoverageType", defined in the namespace http://www.opengis.net/wcs/2.0, according to the schema definition in http://www.opengis.net/wcs/2.0/wcsSoapCoverage.xsd.

Schema Location

For discussion purposes of this RFC, the proposed schema *wcsSoapCoverage.xsd* is available in the sandbox [3]. For convenience, *wcsCommon.xsd* in the same directory has been modified to include *wcsSoapCoverage.xsd*.

References

[1] http://www.w3.org/TR/soap12-part0/

[2] http://www.w3.org/TR/soap12-mtom/

[3] sandbox/sandbox_wcs_soap_proxy/schemas/wcs/2.0/wcsSoapCoverage.xsd

Voting History

Motion Adopted on 2011-05-30 with +1 from Martin Paces, Stephan Meißl, Milan Novacek, Stephan Krause, and +0 from Arndt Bonitz

Traceability

Requirements "N/A"

Tickets "N/A"

3.3.11 RFC 10: SOAP Proxy

Author Milan Novacek

Created 2011-05-18

Last Edit 2011-05-30

Status ACCEPTED

Discussion http://www.eoxserver.org/wiki/DiscussionRfc10

Introduction

This RFC proposes the design and implementation of the module *soap_proxy*. Initially *soap_proxy* is for use with WCS services. The intent of *soap_proxy* is to provide a soap processing front end for those WCS services which do not natively accept soap messages. *Soap_proxy* extracts the xml of a request from an incoming SOAP message and invokes mapserver or eoxserver in POST mode with the extracted xml. It then accepts the response from mapserver or eoxserver and repackages it in a SOAP reply.

Description

Soap-proxy should implement OGC 09-149 *Web Coverage Service 2.0 Interface Standard - XML/SOAP Protocol Binding Extension*. See RFC-9 for a proposal to address certain problems with the current revision of this standard (which is OGC 09-149r1).

Initially it is planned that *soap_proxy* supports WCS 2.0. WCS 1.1 is a low priority. The possibility should be investigated to generalize *soap_proxy* to enable support of other protocols such as WPS.

Soap_proxy is implemented as a Web Service using the Axis2/C framework [AXIS], plugged into a standard Apache HTTP server via its mod_axis2 module.

Governance

Source Code Location

The *soap_proxy* code will be located in the subdirectory '*soap_proxy*' at the main level of the eoxserver repository, i.e. at the same level as the eoxserver directory: trunk/*soap_proxy*.

Initial Code Base

A first prototype implementing parts of the functionality has been developed under the O3S project. The source of this prototype will be copied to the *soap_proxy* repository and form the basis for further development.

RFCs and Decision Process

In the early stages, development surrounding of *soap_proxy* not directly affecting eoxserver will be undertaken in a relaxed manner compared to the RFC based decision taking that prevails for eoxserver.

All non trivial changes to the *soap_proxy* core will be announced for discussion on the eoxserver-dev mailing list, but will not undergo the RFC voting process unless there is a direct impact on any actual eoxserver functionality.

Once the transition phase of the integration has been completed, the development of *soap_proxy* will follow the standard RFC based decision taking.

License

Soap_proxy will use either GPL or a MapServer-style license, this is yet TBD.

Wiki, Trac, Tickets

Soap_proxy will use all of the eoxserver support infrastructure.

References

[AXIS] http://axis.apache.org/axis2/c/core/

Voting History

Motion Adopted on 2011-05-30 with +1 Martin Paces, Stephan Meißl, Fabian Schindler, Milan Novacek

Traceability

Requirements "N/A"

Tickets "N/A"

3.3.12 RFC 11: WPS 1.0.0 Interface Prototype

Author Martin Paces

Created 2011-07-20

Last Edit 2011-07-21

Status DRAFT

Discussion http://www.eoxserver.org/wiki/DiscussionRfc11

Introduction

This RFC describes the design and implementation of the OGC WPS 1.0.0 Interface prototype. The WPS (Web Processing Service) interface prototype adds the processing functionality to the EOX-Server and in capable of invocation of both synchronous ans asynchronous processes invoked using either XML or KVP encoding as described in OGC 05-007r7 *OpenGIS Web Processing Service* document.

Description

The implementation extends the set of EOX-Server's OWS service handlers by the WPS specific interface. Namely, it ads following handlers

- WPS service handler
- WPS 1.0.0 version handler
- WPS *GetCapabilities* operation handler
- WPS *DescribeProcess* operation handler
- WPS *Execute operation* handler

The added WPS functionality could be split three (currently separated) logical parts:

- WPS interface and operation logic (subject to this RFC)
- WPS data model and generic process class (loosely based on PyWPS, currently separated from the interface and operation logic)
- WPS process instances – user defined processes, ancestors of the generic service class (completely independent of the EOX-Server, not subject to this RFC)

WPS Interface and Operation Logic

This part implements the actual OWS service handlers and it is tightly coupled with the EOX-Server. It parses and interprets the operation request and generates the operation responses reusing existing parts of the EOX-Server (primarily the XML and KVP request decoders). This interface has access to the installed WPS process instances (implemented as python modules) and it reads their descriptions. In case of the *Execute* operation it fetches the parsed input data to the selected process instance, triggers the actual execution of the process, and generates the status responses and handles output data XML packing and encoding.

In case of a synchronous execution the WPS processes are executed in context of the EOX-Server's OWS request. In case of an asynchronous WPS processes a dedicated OS process is started from the context of the EOX-Server's OWS request.

This part is distributed under the EOX-Server's MapServer-like open source licence.

WPS Data Model and Generic Process Class

This part (not subject to this RFC) is loosely based on the WPS Process API of the :PyWPS: SW. Due to the flaws of the original data model and requirements of the EOX-Server integration the the original :PyWPS: code was substantially modified (practicall rewritten) leaving only traces of the generic (parent) WPS Process class.

The work is based on the stable :PyWPS: version 3.1.0. The reason we have replaced the original data model was that it had several design and implantation flaws (e.g., the way how the multiple input and output data occurrences were handled, bounding box data handling and encoding, the way how input sequences were detected). After first initial correcting attempts we gave up and rewrote the model from scratch. The generic Process class was

modified: (i) due to the the changes made to the data model, (ii) removing unused parts of code (e.g., useless class reinitialization, Grass integration, internationalization), (iii) and finally due to the needs of the EOX-Server integration.

Despite the only fragments of the original :PyWPS:, this code was derived from the :PyWPS: and it is distributed under the terms of the original GPL licence.

WPS Process Instances

The process instances are not subject to this RFC and should be written by the WPS users to provide the desired functionality. The processes are created as separated python modules each containing a single customized subclass of the generic process class. The unique process identifier is the same as the name of python module (file's base name), the rest of the process description is defined by an implementer in the class definition.

We provide set of sample demo process samples covering from basic to most advanced cases. This part is distributed under the terms of PyWPS GPL licence.

Transition to Operation - Issues to be resolved

The existing prototype has still a couple issues to be resolved before operational deployment.

- licence issues - the WPS Process's data model and parent Process's class shall be merged with the WPS Interface and Operation logic and distributed together under the same licence terms
- resource tracker - there should be a resource tracker looking after the used resources, i.e., stored files and executed asynhronous processes. Each of these resources shall be monitored and released (deleted in case of unused files, properly killed in case of "zombie" processes) once is not usefull anymore.

Governance

Source Code Location

WPS Interface Currently the Interface code can be downloaded from the WPS sandbox:

http://eoxserver.org/svn/sandbox/sandbox_wps

WPS - Data Model and Generic Process Class The code derived from the PyWPS (only the parts needed for EOX-Server integration) can be found at:

http://o3s.eox.at/svn/deliverables/developments/wps/server

WPS - Demo Processes The demo services are available at:

https://o3s.eox.at/svn/deliverables/developments/wps/wps_demo_services/

Initial Code Base

A first prototype implementing parts of the functionality has been developed under the O3S project.

RFCs and Decision Process

TBD

License

WPS Interface prototype shall be distributed under the terms of the EOX-Server's MapServer-like licence.

The other parts required by the WPS functionality are available under the terms of the [PyWPS] GPL licence.

Wiki, Trac, Tickets

TBD

References

[PyWPS] http://pywps.wald.intevation.org/

Voting History

N/A

Traceability

Requirements "N/A"

Tickets "N/A"

3.3.13 RFC 12: Backends for the Data Access Layer

Author Stephan Krause

Created 2011-08-31

Last Edit $Date$

Status ACCEPTED

Discussion http://eoxserver.org/wiki/DiscussionRfc12

This RFC proposes the implementation of different backends that provide common interfaces for data stored in different ways. It describes the first version of the Data Access Layer implementation as well as changes to the Data Integration Layer that are caused by the changes to the data model.

Introduction

RFC 1: An Extensible Software Architecture for EOxServer (page 248) introduced the Data Access Layer as an abstraction layer for access to different kinds of data storages. These are most notably:

- data stored on the local file system
- data stored on a remote file system that can be accessed using FTP
- data stored in a rasdaman database

The term *backend* has been coined for the part of the software implementing data access to different storages.

This RFC discusses an architecture for these backends which is based on the extension mechanisms discussed in *RFC 2: Extension Mechanism for EOxServer* (page 270). After the *Requirements* (page 292) section the architecture of the Data Access Layer is presented. It is structured into a section describing the *Data Access Layer Data Model* (page 292) which consists basically of *Storages* (page 293) and *Locations* (page 293).

Furthermore, the necessary changes to the Data Integration Layer are explained. On the one hand these affect the *Data Model* (page 293) which is altered considerably. On the other hand new structures (*Data Sources* (page 294)

and *Data Packages* (page 294)) that provide more flexible solutions for data handling by the Data Integration Layer and the layers that build on it.

Requirements

We may refer here to the *Backends Requirements* (page 251) section as well as the description of the *Data Access Layer* (page 269) in *RFC 1: An Extensible Software Architecture for EOxServer* (page 248). These state the need for different backends to access local and remote data in different ways and thus are the incentive for this RFC and the respective implementation.

Data Access Layer Data Model

The new database model for the Data Access Layer is shown in the figure below:

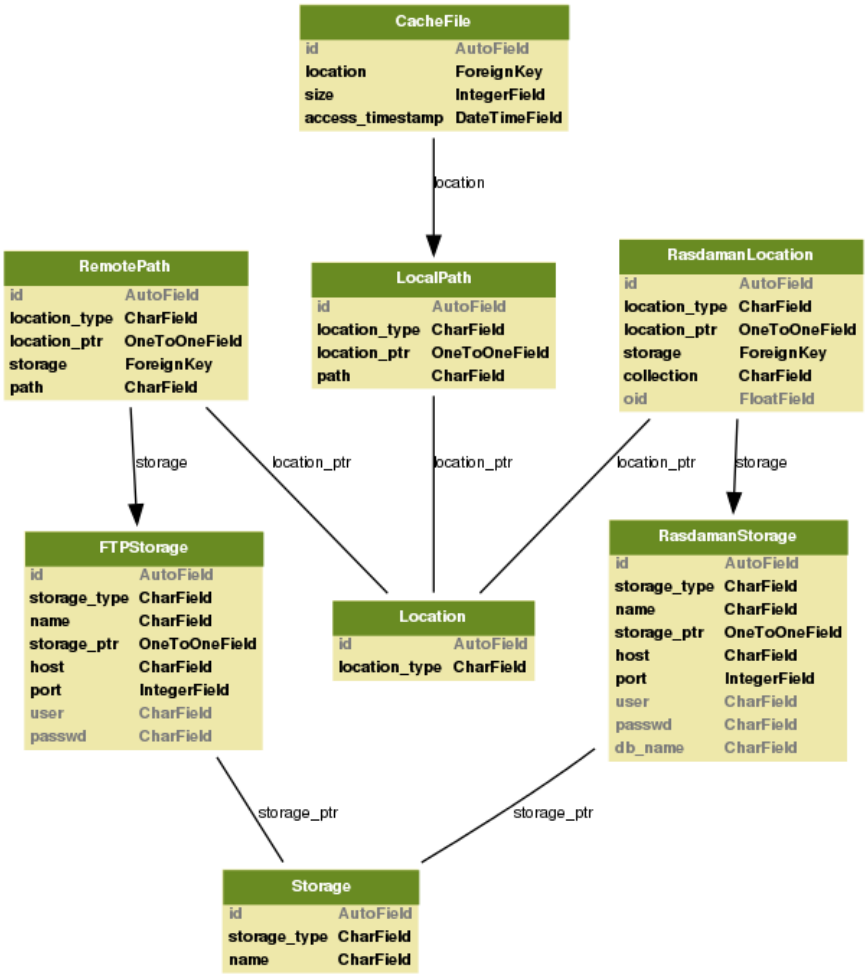


Figure 3.5: Data Access Layer Database Model

The core element of the Data Access Layer data model is the `Location` (page 240). A location designates a piece of data or metadata, actually any object that can be stored in one of the `Storage` (page 240) facilities supported. Each backend defines its own subclasses of `Location` (page 240) and `Storage` (page 240) to represent repositories, databases, directories and objects stored therein.

The database model is embedded in wrappers that add logic to the model and provide common interfaces to access the data and metadata of the objects in the backend. Internally, they make use of the extension mechanism of *RFC2* (page 270) to allow to find and get the right model records and wrappers.

Last but not least, there is a *File Cache* (page 293) for storing files retrieved from remote hosts. The locations of the cache files are stored in the database so EOxServer can keep track of them and implement an intelligent cleanup process.

Storages

The `Storage` (page 240) subclasses represent different types of storage facilities. In the database model, only FTP and rasdaman backends have their own models defined that contain the information how to connect to the server. This is not needed for locally mounted file systems, so the local backend does not have a representation in the database.

The wrapper layer constructed on top of the database model on the other hand knows three classes of storages that provide a common interface to access their data:

- `LocalStorage` (page 239) which implements access to locally mounted file systems
- `FTPStorage` (page 236) which implements access to a remote FTP server
- `RasdamanStorage` (page 241) which implements access to a rasdaman database

Each of these storage classes is associated to a certain type of location.

The common interface for storages allows to retrieve their type and their capabilities. Depending on these capabilities the storage classes also provide methods for getting a local copy of the data and retrieving the size of an object as well as scanning a directory for files. At the moment these three methods are implemented by file-based backends only (`LocalStorage` (page 239) and `FTPStorage` (page 236)).

Locations

Locations represent the points where to access single objects on a storage facility. At the moment three types of locations corresponding to the three storage types are implemented:

- `LocalPath` (page 240) defines a path on the locally mounted file system
- `RemotePath` (page 240) defines a path on a remote server reachable via FTP
- `RasdamanLocation` (page 240) defines a collection (database table) and oid corresponding to a single rasdaman array

Locations share a common interface that is closely related to the storage interface. So, given the storage capabilities, it is possible to fetch a local copy, retrieve the size of an object and scan the location for files. The `LocationWrapper` (page 234) subclasses extend these interfaces to make storage specific location information (e.g. host name for remote storages) accessible.

File Cache

With the `CacheFileWrapper` (page 235) class the Data Access Layer provides a very simple file cache implementation at the moment that serves to cache remote files retrieved via FTP. The cache keeps track of the files it contains using the `CacheFile` (page 240) model in the database.

So far, no synchronization for data access is implemented, i.e. threads that are processing requests have no possibility to lock a cache file in order to prevent it from being removed by another thread or process (e.g. periodical cleanup process). This is foreseen for the future.

Changes to Data Integration Layer Data Model

In order to use the new possibilities brought by the implementation of the Data Access Layer, the Data Integration Layer had to be revised and changed considerably. Up until now there has been a strong link between the type of coverage and the way it was stored. Datasets had to be stored as files in the local file system whereas mosaics were stored in tile indexes. This strong link had to be weakened to allow for new combinations.

The solution is a compromise between flexibility and simplicity. Although one can think of many more combinations, we introduce three classes of so-called `DataPackage` objects. A data package combines a data resource with an accompanying metadata resource. Both resources are referred to by `Location` (page 240) subclass instances. Now the three data package classes are:

- `LocalDataPackage` which combines a local data file with a local metadata file
- `RemoteDataPackage` which combines a remote data file with a remote metadata file (both reachable via FTP); it contains a `CacheFile` (page 240) reference for data in the local cache
- `RasdamanDataPackage` which combines a rasdaman array with a local metadata file

Furthermore, the concept of data directories where to look up datasets automatically had to be revised in order to use the new capabilities of the Data Access Layer. They were replaced by a concept called data sources which includes local and remote repositories. The `DataSource` model combines a local or remote `Location` (page 240) with a search pattern for dataset names. Automatic lookup of rasdaman arrays is not foreseen at the moment.

Like most database objects, data packages and data sources are accessible using wrappers that provide a common interface and add application logic to the data model.

Data Packages

The `DataPackageInterface` (page 199) defines methods for high-level and low-level data access and for metadata extraction from the underlying datasets. It is implemented by wrappers for local, remote and rasdaman data packages (`LocalDataPackageWrapper` (page 192), `RemoteDataPackageWrapper` (page 193) and `RasdamanDataPackageWrapper` (page 192) respectively).

The implementation of the data package wrappers is based on the GDAL[51] library and its Python binding for data access as well as for geospatial metadata extraction. It contains an `open()` (page 191) method that returns a GDAL dataset providing a uniform interface for raster data from different sources and formats. For low-level data access a `getGDALDatasetIdentifier()` (page 191) method is provided which allows to retrieve the correct connection string for GDAL and thus to configure MapServer.

Geospatial metadata is read from the datasets themselves at the moment. Note that this is not possible for rasdaman arrays so far, so automatic detection and ingestion of these is not enabled.

EO Metadata is read from the accompanying metadata file and translated into the internal data model of EOxServer. The existing metadata extraction classes have been revised in order to comply with the extensible architecture presented in *RFC 1* (page 248) and *RFC 2* (page 270).

Data Sources

The wrappers for data sources (`DataSourceWrapper` (page 192)) provide the capability to search a local or remote location for datasets. At the moment only file lookup is implemented whereas automatic rasdaman array lookup has been omitted. This is mostly due to the fact that rasdaman arrays do not contain geospatial metadata and a separate mechanism has to be found to retrieve this vital information.

The wrapper implementations provide a `detect` method that returns a list of `DataPackageWrapper` (page 190) objects with which coverages are initialized (using the geospatial and EO metadata read from the data package).

Ingestion and Synchronization

The `Synchronizer` implementation in `eoxserver.resources.coverages.synchronize` has to be revised according to the changes in the Data Access Layer and Data Integration Layer.

The implementations for containers, i.e. Rectified Stitched Mosaics and Dataset Series, shall retrieve the data sources associated with a coverage and use its `detect` method to obtain the data packages included in it. Rectified or Referenceable Datasets are constructed from these. The interfaces of both should not change.

[51] http://www.gdal.org/

The interface of `RectifiedDatasetSynchronizer` on the other hand will have to change in order to allow for remote files to be ingested. In detail, the `create()` and `update()` methods will not expect a file name any more, but a location wrapper instance (either `LocalPathWrapper` (page 239) or `RemotePathWrapper` (page 236)). These can be generated by a call to the `LocationFactory` like this:

```
from eoxserver.core.system import System

factory = System.getRegistry.bind("backends.factories.LocationFactory")

location = factory.create(
    type = "local",
    path = "<path/to/file>"
)

...
```

Voting History

Motion To accept RFC 12

Voting Start 2011-09-06

Voting End 2011-09-15

Result +5 for ACCEPTED (including 1 +0)

Traceability

Requirements N/A

Tickets N/A

3.3.14 RFC 13: WCS-T 1.1 Interface Prototype

Author Martin Paces

Created 2011-09-13

Last Edit $Date$

Status ACCEPTED

Discussion http://www.eoxserver.org/wiki/DiscussionRfc13

Introduction

This RFC describes the design and implementation of the interface prototype of the *Open Geospatial Consortium* (OGC) Web Coverage Service - Transaction operation extension (WCS-T) [OGC 07-068r4][52] standard. The WCS-T extends the baseline WCS (allowing download of coverages only) by additionally allowing modifications of the stored coverages, namely, it allows adding, deleting, or updating of the coverages' data and their metadata.

WCS Transaction Operation

The WCS-T standard [OGC 07-068r4] defines an additional WCS *transaction* operation to perform modifications of the WCS Coverages. A single *transaction* contains one or more actions to be performed over coverages (coverage actions). The WCS-T standard requires that all WCS-T implementations shall support at least one action per request, multiple actions per request are optional.

The possible coverage actions are:

[52] http://portal.opengeospatial.org/files/?artifact_id=28506

- Add - inserts new coverage and its metadata (required by all WCS-T implementations)
- Delete - removes an existing coverage and its metadata (optional)
- UpdateAll - replace data and metadata of an existing coverage (optional)
- UpdateMetadata - replace metadata of an existing coverage (optional)
- UpdateDataPart - replace data subset of an existing coverage (optional)

The supported optional features (multiple actions per request or optional coverage actions) shall be announced in the *ServiceIdentification* section of the WCS *Capabilities* XML document using the *Profile* XML element (see [OGC 07-068r4] for a detailed list of the applicable URNs).

Although not explicitly mentioned by the WCS-T standard, we assume the *transaction* operation shall be present in the *OperationMetada* section of the WCS *Capabilities*.

The WCS-T standard allows XML encoded requests submitted as HTTP/POST requests. The KVP encoding of HTTP/GET requests is not supported by WCS-T since "the KVP encoding appears impractical without significantly restricting Transaction requests" [OGC 07-068r4]. Further, the [OGC 07-068r4] introduction mentions that the exchanged XML documents shall use the SOAP packaging, however, the examples are presented without the SOAP wrapping leaving this requirement in doubts.

The WCS-T requests can be processed synchronously or asynchronously. In the first case, the request is processed immediately and the transaction response is returned once actions have been processed successfully. In the latter case, the request is validated and accepted by the server returning simple acknowledgement XML document. The request is than processed asynchronously possibly much later than the acknowledgement XML document has been returned to the client. The asynchronous operation is triggered by presence of the *responseHandler* element in the WCS-T request. This element contains an URL where the response document should be uploaded.

All the data passed to the server by the WCS-T requests are in form of URL references. The support for direct data passing via MIME/multi-part encoded requests is not considered by the WCS-T standard.

The format of the ingested coverage data is not considered by the WCS-T standard at all. Neither it can be annotated by the WCS-T request nor by the WCS-T *OperationMetadata*. Thus we assume the format selection is left at discretion of the WCS-T implementation.

The WCS-T standard requires that certain metadata shall be provided by the client. These are geo-transformation, coverage description, and coverage summary. Apart from this mandatory metadata application specific metadata may be added by the implementation.

The WCS-T standard allows clients to submit their request and (created) coverages identifiers. These identifiers do not need to be used by the WCS-T server as they may collide with the identifiers of other requests or coverages, respectively, or simply not follow the naming convention of the particular WCS server. Thus the client provided identifiers are not binding for the WCS server and they rather provide a naming hint. As result of this the WCS-T client shall never rely on the identifiers provided to the WCS-T server but it shall always read the identifier returned by the WCS-T XML response.

EOxServer Implementation

The WCS *transaction* operations is implemented using the service handlers API of EOxServer. Since the WCS-T standard requires the version of the *transaction* operation to be '1.1' (rather than the '1.1.0' version used by other WCS operations) a specific WCS 1.1 version handler must have been employed. The operation itself is then implemented as a request handler.

Since the presence of the WCS-T operation needs to be announced by the WCS *Capabilities* the WCS 1.1.x *getCapabilities* operation request handlers have to be modified. Since the *Capabilities* XML response is generated by the MapServer (external library) the only feasible way to introduce the additional information to the *getCapabilities* XML response is to: i) capture the MapServer's response, ii) modify the XML document, and iii) send the modified XML instead of the MapServer's one.

The *transaction* request or response XML documents do not use the (presumably) required SOAP packaging. We have intentionally refused to follow this requirement in our implementation as the SOAP packing and unpacking is

duty of EOxServer's *SOAP Proxy* component and our own implementation would rather duplicate the functionality implemented elsewhere.

Our implementation, by default, offers the WCS-T core functionality only. All the optional features such as multiple coverage actions per request or the optional coverage actions shall be explicitly enabled by EOxServer's configuration (see following section for details).

Both synchronous and asynchronous modes of operation are available. While the synchronous request are processed within the context of the WCS-T request handler the asynchronous requests are parsed and validated within the context of the WCS-T request but the processing itself is performed by the Asynchronous Task Processing (ATP) subsystem of EOxServer. Namely, the processing task is enqueued to the task queue and than later executed by one of the employed Asynchronous Task Processing Daemons (ATPD). More details about the ATP can be found in [ATP-RFC].

As it was already mentioned, the asynchronous mode of operation is triggered by presence of the *responseHandler* element in the WCS-T request and this element contains an URL where the response document should be uploaded. Our implementation supports following protocols:

- FTP - using the PUT command; username/password FTP authentication is possible
- HTTP - using POST HTTP request; username/password FTP authentication is possible

Secured (SSL or TLS) versions of the protocols are currently not supported.

The username/password required for authentication can be specified directly by the URL

```
scheme://[username:password@]domain[:port]/path
```

In case of FTP, when the paths point to a directory a new file will be created taking the request ID as the base file-name and adding the '.xml' extension. Otherwise a file given by the path will be created or rewritten.

The WCS-T implementation uses always pairs of identifiers (internal and public) for both request and (created) coverage identifiers. The public identifiers are taken from the WCS-T request, provided they do not collide with identifiers in use. In case of not supplied or colliding identifiers the public identifiers are set from the internal ones. The public identifiers are used in the client/server communication or for naming of the newly created coverages. The internal identifiers are exclusively used for naming of the internal server resources (asynchronous tasks, directory and file names, etc.)

Each WCS-T request, internally, gets a *context*, i.e. set of resources assigned to a particular request instance. These resources are: i) an isolated temporary workspace (a directory to store intermediate files deleted automatically once the request is finished), ii) an isolated permanent storage (a directory where the inserted coverages and their metadata is stored) and iii) in case of asynchronous mode of operation ATP task instance. These resources make use of the internal identifiers only.

EOxServer Configuration

The EOxServer's WCS-T implementation need to be configured prior to the operation. The configuration is set in EOxServer's 'eoxserver.conf' file. The WCS-T specific options are grouped together in the 'services.ows.wcst11' section.

The WCS-T options are:

- allow_multiple_actions (False|True) - allow multiple actions per single WCS-T request.
- allowed_optional_action (Delete,UpdateAll,UpdateMetadata,UpdateDataPart) - comma separated list of enabled optional WCS-T coverage action. Set empty if none.
- path_wcst_temp (*path*) - directory to use as temporary workspace
- path_wcst_perm (*path*) - directory to use as permanent workspace

Example:

```
...
# WCS-T 1.1 settings
[services.ows.wcst11]
```

```
# enble disable multiple actions per request
allow_multiple_actions=False

# list enabled optional actions {Delete,UpdateAll,UpdateMetadata,UpdateDataPart}
allowed_optional_actions=Delete,UpdateAll

# temporary storage
path_wcst_temp=/home/test/o3s/sandbox_wcst_instance/wcst_temp

# permanent data storage
path_wcst_perm=/home/test/o3s/sandbox_wcst_instance/wcst_perm
...
```

Coverages, Data and Metadata

The one and only currently supported format of pixel data is GeoTIFF.

All the necessary meta-data required by the EOxServer are extracted from the GeoTIFF annotation and (optionally) from the provided EO meta-data (see section below).

Due to the limitations of the current Coverage Managers' API of the EOxServer the current WCS-T implementation has following restrictions:

- only rectified grid coverages can be ingested;
- `urn:ogc:def:role:WCS:1.1:CoverageDescription` metadata are ignored and even not required as this information cannot be inserted to EOxServer anyway;
- `urn:ogc:def:role:WCS:1.1:CoverageSummary` metadata are ignored as this information cannot be inserted to EOxServer anyway;
- `urn:ogc:def:role:WCS:1.1:GeoreferencingTransform` metadata are ignored as this information is relevant to referenced data only
- `urn:ogc:def:role:WCS:1.1:OtherSource` metadata are ignored as this information cannot be inserted to EOxServer anyway.

WCS-T and Earth Observation Application Profile

In order to be able to ingest additional metadata as defined by the *WCS 2.0 - Earth Observation Application Profile* [EO-WCS] we allow the ingestion of client-defined EO-WCS metadata attached to the ingested pixel data. The EO-WCS XML is passed as coverage OWS Metadata XML element with 'xlink:role="http://www.opengis.net/eop/2.0/EarthObservation"'.

Governance

Source Code Location

http://eoxserver.org/svn/sandbox/sandbox_wcst

RFCs and Decision Process

TBD

License

The WCS-T implementation shall be distributed under the terms of *EOxServer's MapServer-like license* (page 333).

Wiki, Trac, Tickets

TBD

References

[OGC 07-068r4] http://portal.opengeospatial.org/files/?artifact_id=28506

[ATP-RFC] http://eoxserver.org/doc/en/rfc/rfc14.html

[EO-WCS] *TBD*

Voting History

Motion To accept RFC 13

Voting Start 2011-12-15

Voting End 2011-12-22

Result +3 for ACCEPTED

Traceability

Requirements *N/A*

Tickets *N/A*

3.3.15 RFC 14: Asynchronous Task Processing (ATP)

Author Martin Paces

Created 2011-10-25

Last Edit 2011-12-09

Status ACCEPTED

Discussion http://www.eoxserver.org/wiki/DiscussionRfc14

This RFC describes the Asynchronous Task Processing subsystem of the *EOxServer*.

Introduction

The *Asynchronous Task Processing* (ATP) subsystem, as the name suggests, extends the EOxServer functionality by ability to process tasks asynchronously, i.e., on background independently of the default EOxServer's synchronous client requests processing.

Although the ATP is designed primarily to support asynchronous request processing of OGC Web Services such as the Web Coverage Service transaction extension and/or the Web Processing Service, it is not limited to these and other application may use it as well.

The ATP employs the model of a single central task queue and one or more *Asynchronous Task Processing Daemons* (APTD) executing the pending tasks pulled from the task queue. A single ATPD is not restricted to a single

processed task at time and can internally process multiple tasks concurrently, e.g., by employing a pool of worker processes assigned to multiple CPU cores.

The ATP subsystem is implemented as Django application using the DB model as the task queue. The underlying DB storage although it may be seen as suboptimal in terms of the performance and latency it assure tolerance of the subsystem to possible failures or maintenance shut-downs of both EOxServer or APTDs.

The ATP can be shared by multiple application at time as each task has its type (application to which it belongs) and each type of the task has a predefined handler subroutine. The shared nature of the APT subsystem allows fine control over the processing resources, e.g., the number of concurrently running task matching number of available CPU cores.

The ATP is primarily designed for resource demanding longer running tasks (10 seconds and more) which in case of synchronous operation could clog the system or lead to connection time-outs. On contrary, *light* tasks (less than 1 sec.) should preferably be executed synchronously as the extra ATP latency might be unfavourable.

Asynchronous Task Processing

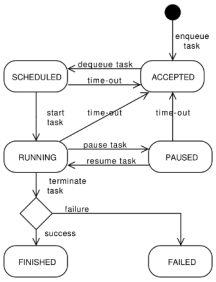


Figure 3.6: *Fig.1: ATP Task State Diagram*

The ATP subsystem is capable of tracking of the tasks during their life cycle depicted by the Task state diagram Fig.1. The task can be in one of the following states:

- ACCEPTED - a new enqueued task waiting to be pulled by the processing daemon
- SCHEDULED - a task pulled (dequeued) by the processing daemon but not yet stared
- RUNNING - a task being processed by the processing daemon
- PAUSED - a task which has been put on hold by the processing daemon and which is waiting to be resumed
- FINISHED - a task which has been finished successfully (terminal state)
- FAILED - a task which has been finished by a failure (terminal state)

When a task becomes identified as staled (by exceeding the type specific time-out) it may be re-enqueued, i.e., the processing shall be terminated, enqueued as a new task again changing its status from one of the non-terminal states (SCHEDULED, RUNNING, PAUSED) to ACCEPTED. This procedure is implemented to avoid abandoned "zombie" tasks left, e.g., by an aborted processing daemon. However, this procedure is repeated only limited times (the count is task type specific, three by default), once the allowed restart's count is exceeded the task is marked as FAILED.

The history of the task's state transition is logged in order to provide information to the system operator.

The finished tasks are kept recorded for ever by default, however, this can changed by a task type (application) specific retention time, which allow automatic removal of out-dated tasks, e.g., one day, week or month after their finish.

To inspect the state of the APT subsystem, a couple of simple DJango html views has been created.

ATP DB Model

The APT Django DB model consists of six classes as depicted in Fig.2.

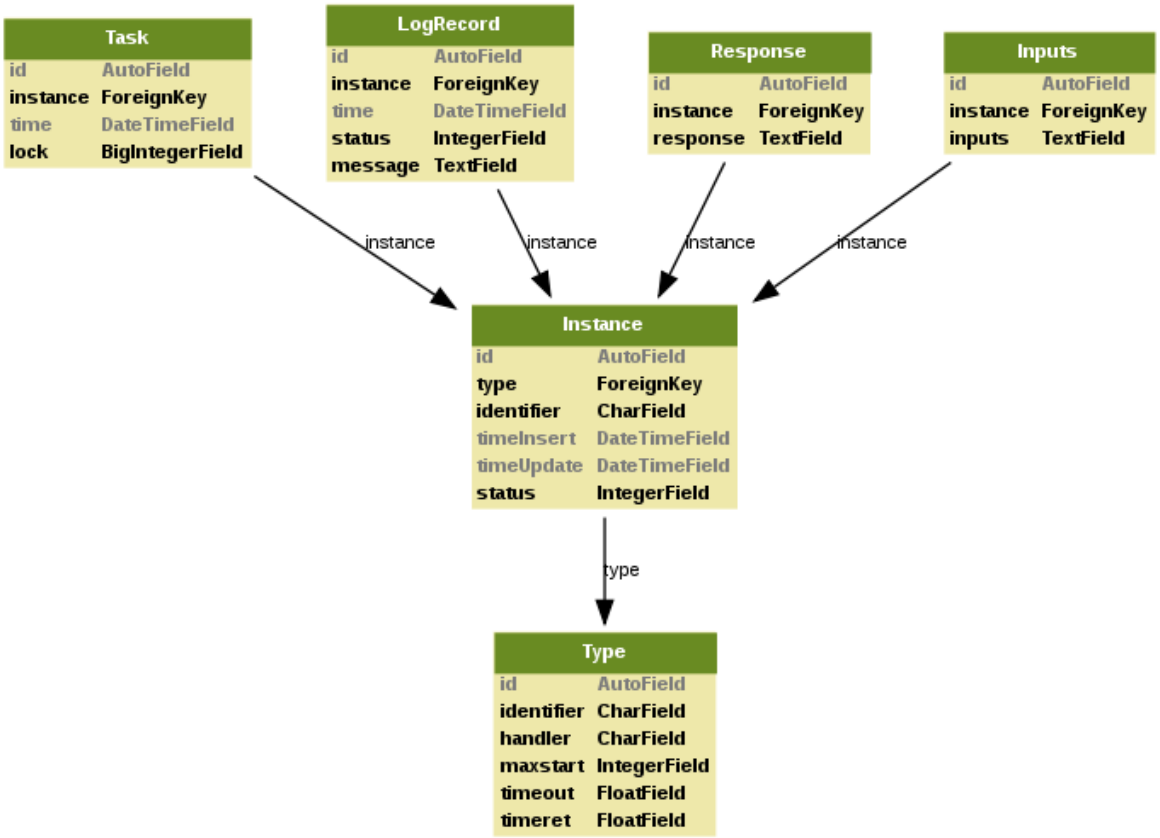


Figure 3.7: *Fig.2: ATP DB model*

- **Type - defining the type of task instance, its unique identifier, task handler (python subroutine),** and the type specific parameters such as maximum unsuccessful attempts to start the task execution, time-out after the which the task is considered to be abandoned and re-enqueued for processing (e.g., due to ATPD failure), retention time to keep the record of the finished task.
- Instance - defining a single task instance, its identifier and current state.
- Inputs - record holding input parameters stored serialized (pickled) Python object
- Response - record holding the optional tasks output (most likely an XML response document or serialized Python object)
- LogRecord - single log entry. The log keeps history of the task's state transition.
- Task - single task queue record. The task table holds the accepted tasks, their enqueuetime, ATPD assignment.

ATP API

The ATP subsystem provides simple API which allows:

- registering of new task type and its parameters (repeated registration updates the parameters)
- removal of unused task types (provided there is no instance of the removed type)
- enqueueing of new task instance and input parameters (implies creation of new task instance)
- dequeueing of enqueued instance (used by APTD)
- removal of finished tasks
- re-enqueueing of a non terminal state task
- changing of the task status
- adding and retrieval of the response (output)

Further the mandatory function prototype to define new handlers is given.

Governance

Source Code Location

http://eoxserver.org/svn/sandbox/sandbox_wcst

RFCs and Decision Process

TBD

License

The APT implementation shall be distributed under the terms of *EOxServer's MapServer-like license* (page 333).

Wiki, Trac, Tickets

TBD

References

Voting History

Motion To accept RFC 14

Voting Start 2011-12-15

Voting End 2011-12-22

Result +4 for ACCEPTED

Traceability

Requirements *N/A*

Tickets *N/A*

3.3.16 RFC 15: Access Control Support

Author Arndt Bonitz

Created 2011-11-14

Last Edit 2011-02-09

Status ACCEPTED

Discussion http://eoxserver.org/wiki/DiscussionRfc15

Overview

This RFC describes access control support for the EOxServer. The following figure gives an overview of the proposed access control implementation and its different components:

The access control implementation relies on the Shibboleth framework[53] and parts of the CHARON framework[54], namely the CHARON Authorisation Service. The components are all released as Open Source. Shibboleth is used for the authentication of users; the CHARON Authentication Service is responsible for making authorisation decisions if a certain request may be performed.

Authentication

Authentication is not handled directly by the EOxServer components, but uses the Shibboleth federated identity management system. In order to do this, two requirements must be met:

- A Shibboleth Identity Provider (IdP) must be available for authentication
- A Shibboleth Service Provider must be installed and configured in an Apache HTTP Server[55] to protect the EOxServer resource.

A user has to authenticate at an IdP in order to perform requests to an EOxServer with access control enabled. The IdP issues a SAML token which will be validated by the SP.

Is the user valid, the SP adds the user attributes by the IdP to the HTTP Header of the original service requests and conveys it to the protected EOxServer instance. The whole process ensures, that only authenticated users can access the EOxServer.

[53] http://shibboleth.internet2.edu/
[54] http://www.enviromatics.net/charon/index.html
[55] http://httpd.apache.org/

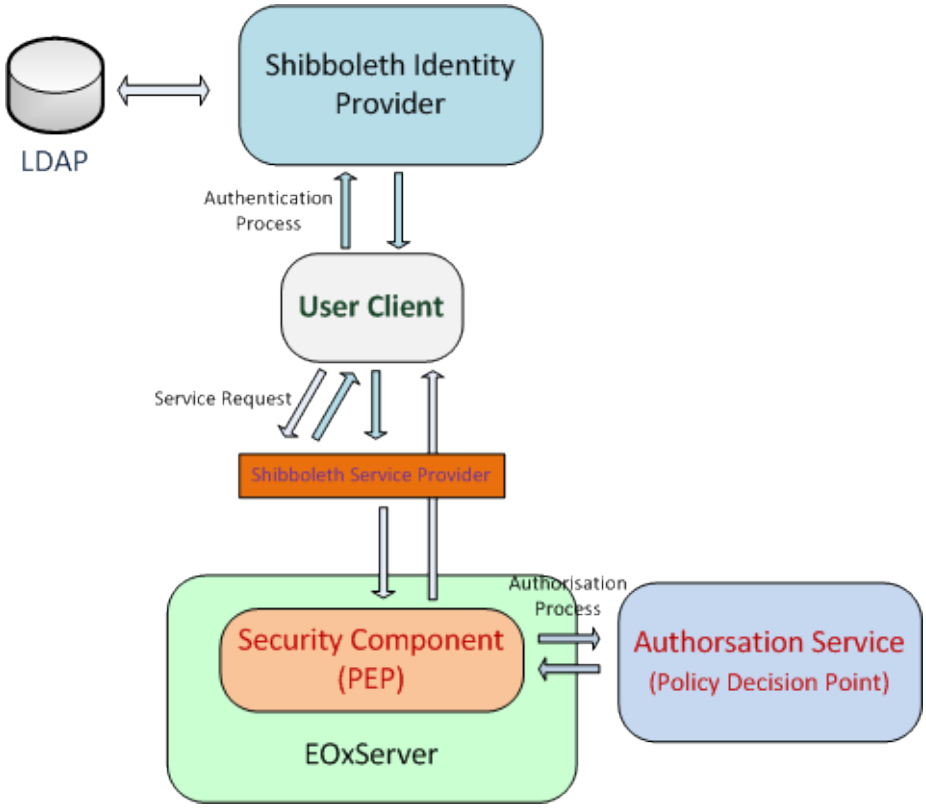


Figure 3.8: *EOxServer Access Control Implementation*

Authorisation

As noted in the previous section, the Shibboleth system provides the underlying service and all asserted user attributes. These attributes can be used to make an decision if a certain user is allowed to perform an operation on the EOxServer. The authorisation decision is not made directly in the EOxServer, but relies on the CHARON Authorisation Service.

The Authorisation Service is responsible for the authorisation of service requests. It makes use of XACML[56], a XML based language for access policies. The Authorisation Service is part of the CHAORN[57] project. The EOxServer security components are only responsible for performing an authorisation decision request on the Authorisation Server and enforcing the authorisation decision.

EOxServer Security Component

The EOxServer security component is located in the package `eoxserver.services.auth.base` in the EOxServer source code directory. The implementation of the `PolicyDecisionPointInterface` for the proposed setup is included in `eoxserver.services.auth.charonpdp.py`, which is a wrapper for the CHARON Authorisation Service client. Every request for authorisation is encoded into a XACML Authorization Query and sent to the Authorisation Service. The decision (permit, deny) of the service is then enforced by the EOxServer.

A first implementation can be found in this EOxServer sandbox[58] and there's also an e-mail discussion[59] about this in the dev mailing list archives.

[56] http://www.oasis-open.org/committees/xacml/#XACML20
[57] http://www.enviromatics.net/charon/index.html
[58] http://eoxserver.org/browser/sandbox/sandbox_security
[59] http://eoxserver.org/pipermail/dev/2011-October/000295.html

Voting History

Motion Adopted on 2011-02-09 with +1 from Arndt Bonitz, Fabian Schindler, Stephan Meißl and +0 from Milan Novacek, Martin Paces

Traceability

Requirements N/A

Tickets N/A

3.3.17 RFC 16: Referenceable Grid Coverages

Authors Stephan Krause, Stephan Meissl, Fabian Schindler

Created 2011-11-24

Last Edit $Date$

Status ACCEPTED

Discussion http://www.eoxserver.org/wiki/DiscussionRfc16

This RFC proposes an implementation for Referenceable Grid Coverages as well as for the WCS 2.0 operations working on them.

The implementation is available in the SVN under http://eoxserver.org/svn/sandbox/sandbox_ref.

Introduction

Referenceable Grid Coverages are coverages whose internal grid structure can be mapped to a coordinate reference system by some general transformation. They differ from rectified grid coverages in that the coordinate transformation is not necessarily affine.

In the context of Earth Observation, raw satellite data can be seen as referenceable grid coverages. They are typically delivered as image files but do not have an affine transformation from the image geometry to a georeferenced coordinate system. Depending on the desired geocoding precision, the referencing transformation can be very complex involving additional data (DEMs) and orbit metadata.

EOxServer shall be able to deliver (subsets of) Earth Observation raw data in its original (referenceable grid) geometry using WCS 2.0 and EO-WCS. Furthermore, it shall implement easily computable approximate referencing algorithms based on ground control points (GCPs) in order to enable coordinate transformations and rectified previews of the original data using WMS.

For the time being, the implementation will focus on SAR image data collected by the ENVISAT-ASAR sensor made available by ESA.

Requirements

The main requirement source for Referenceable Grid Coverage implementation in EOxServer is the ESA O3S project. In the course of this project EOxServer shall be installed in front of a small archive of ENVISAT-ASAR data. In a first step, we will focus on covering the requirements of this use case, adding more generic referenceable grid support in future iterations.

The ENVISAT-ASAR data are available in ENVISAT .N1 original format.

Delivery of the original referenceable grid data shall be supported using WCS 2.0 and EO-WCS. Subsetting shall be supported in pixel coordinates (imageCRS) and in a coordinate reference system. The CRS subsets shall be mapped to pixel coordinates using a simple coordinate transformation based on GCPs.

No support for resampling (`size` and `resolution`) or reprojection (`outputcrs`) parameters is required as these are not applicable to referenceable grid coverages.

At least GeoTIFF shall be supported as output format. GCP and metadata information contained in the .N1 original file shall be preserved.

In order to support (rectified) WMS previews, a simple georeferencing algorithm based on GCPs shall be implemented. This shall be reused to provide rectified versions of referenceable grid coverages using WCS 2.0.

Implementation Details

Input Formats

The implementation for referenceable grid coverages relies on GDAL for input data and metadata (georeferencing information, GCPs). Any format that supports storage of GCPs with the dataset can be used. The two most important formats are the ENVISAT .N1 format and GeoTiff.

Referencing Algorithm and Subsetting

WCS 2.0 allows to define subsets either in the image CRS, i.e. pixel coordinates, or in some geographic or projected coordinate system. For rectified grid coverages geographic coordinates can be easily transformed to pixel coordinates in a straightforward way. This is not the case for referenceable grid coverages, though.

For referenceable grid coverages produced by Earth Observation missions, the "correct" referencing transformation is not known in general. Instead, there are many different algorithms some of them relying on different additional data and metadata (DEMs, orbit information).

For the purposes of the EOxServer Referenceable Grid Coverage implementation, a simple first order interpolation algorithm based on GCPs is used. This algorithm does not use any additional data or metadata. The rationale for this decision is that there is no way to advertise the actual referencing algorithm in WCS or WMS, and therefore the most simple and straightfoward algorithm was used.

Subsets given in georeferenced coordinates are transformed to the image CRS using the inverse transformation algorithm based on GCPs. The implementation uses not only the corner coordinates of the subsetting rectangle but also intermediary points to calculate an envelope and thus to guarantee that the requested extent be included in the result.

Genuine Referenceable Grid Coverage Support in WCS 2.0

Referenceable Grid Coverages in their original geometry are available using the EO-WCS extension of WCS 2.0.

The current implementation supports the `subset` parameter and transforms the given subsets as indicated in the previous subsection. The `size` and `resolution` parameters are not supported as they do only apply to rectified grid coverages.

The `format` parameter options are implemented in the same way as for rectified grid coverages.

The `rangesubset` parameter is foreseen for implementation.

In order to be able to serve referenceable grid data, the original `WCS20GetCoverageHandler` (page 177) was split up into `WCS20GetReferenceableCoverageHandler` (page 177) and `WCS20GetRectifiedCoverageHandler` (page 177). While the latter one still relies on MapServer, the one for referenceable grid data uses the vanilla GDAL Python bindings as well as additional GDAL-based extensions written for the EOxServer project.

Metadata is read from the original dataset and tagged onto the result dataset using the capabilities of the respective GDAL format drivers. Depending on the driver implementation, the way the metadata is stored may be specific to GDAL.

Coverage Metadata Tayloring

The WCS 2.0 standard specifies that the complete referencing transformation be described in the metadata of a referenceable grid coverage. This is a major problem for Earth Observation data as in general there is no predefined transformation; rather there are several different possible algorithms of varying complexity that can be used for georeferencing the image, possibly involving Earth Observation metadata such as orbit information, GCPs and additional data such as DEMs.

Furthermore there is no way to define an algorithm and describe its parameters (e.g. the GCPs) in GML, but only the outcome of the algorithm, i.e. a pixel-by-pixel mapping to geographic coordinates. This would produce a tremendous amount of mostly useless metadata and blow up the XML descriptions of coverage metadata to hundreds of megabytes for typical Earth Observation products.

Therefore the current EOxServer implementation does not deliver any of the `gml:AbstractReferenceableGrid` extensions in its metadata. Instead a non-standard `ReferenceableGrid` element is returned that contains all the elements inherited from `gml:Grid` but no further information. This is only a provisional solution that will be changed as soon as an appropriate way to describe referencing metadata is defined by the WCS 2.0 standard or any of its successors.

Support for Rectified Data in WMS and WCS 2.0

The implementation of the WCS 2.0 (EO-WCS) GetCoverage request as well as the WMS implementation is based on MapServer which supports rectified grid coverages only. It is not possible to use any kind of GCP based referencing algorithm in MapServer directly.

GDAL provides a mechanism to create so-called virtual raster datasets (VRT). These consist of an XML file describing the parameters for transformation, warping and other possible operations on raster data. They can be generated using the GDAL C API and are readable by MapServer (which relies on GDAL as well).

In order to provide referenced versions of referenceable data, EOxServer creates such VRTs on the fly using the EOxServer GDAL extension. The VRT files are deleted after each request.

GDAL Extension

The EOxServer GDAL extension provides a Python binding to some C functions using the GDAL C API that implement utilities for handling referenceable grid coverages. At the moment the Python bindings are implemented using the Python ctypes[60] module.

The `eoxserver.processing.gdal.reftools` module contains functions for

- computing the pixel coordinate envelope from a georeferenced subset
- computing the footprint of a referenceable grid coverage
- creating a rectified GDAL VRT from referenceable grid data

All functions use a simple GCP-based referencing algorithm as indicated above.

The GDAL Extension was made necessary because the standard GDAL Python bindings do not support GCP based coordinate transformations.

Voting History

Motion Adopted on 2012-01-03 with +1 from Arndt Bonitz, Martin Paces, Fabian Schindler, Stephan Meißl and +0 from Milan Novacek

[60] http://docs.python.org/library/ctypes.html

Traceability

Requirements "N/A"

Tickets "N/A"

3.3.18 RFC 17: Configuration of Supported Output Formats and CRSes

Author Stephan Krauses, Martin Pačes

Created 2012-05-08

Last Edit $Date$

Status ACCEPTED

Discussion n/a

This RFC proposes modifications of the EOxServer allowing configuration of

- the supported output formats for WMS and WCS
- the supported CRSes for WMS and WCS

The RFC presents the rationale and proposes data model changes and new global configuration options.

Introduction

The reason for preparation of this RFC is the need to change the way how the supported (file) formats and CRSes (CRS - Coordinate Reference Systems) for raster data are handled by the EOxServer's WCS and WMS services to assure compliance to OGC standards, interoperability and configurability of the services.

In case of WMS, the formats and CRSes shall be listed in the WMS Capabilities.

In case of WCS, the supported formats and CRSes shall be reported by the WCS Capabilities (per service parameters) and in the Coverage Descriptions (per coverage parameters). Compatibility with the WCS 2.0.1 corrigendum and the upcoming WCS 2.0 CRS Extension document shall be assured.

Currently, only the native CRS of a dataset is reported in the metadata and only a small hard-coded set output file format is announced as supported (JPEG2000, HDF4, netCDF and GeoTIFF for WCS). Hence, there is no way to configure these parameters.

Furthermore, the underlying MapServer implementation does not return proper OWS exceptions if an CRS not advertised in the service capabilities or coverage descriptions is requested.

Supported CRSes and Output Formats in OGC Web Services

The table below gives an overview over the support for reporting CRS and output format metadata in different standards implemented by EOxServer.

Table 3.2: Support for CRS and output format metadata

Service and Version	Supported CRS	Supported Formats
WMS 1.1.0	per layer	per service
WMS 1.1.1	per layer	per service
WMS 1.3.0	per layer	per service
WCS 1.1.2	per coverage	per coverage
WCS 2.0.0	n/a	n/a
WCS 2.0.1	per service	per service

All services but the WCS 2.0 CRS extension (listed under WCS 2.0.1) allow for reporting CRSes for each coverage / layer individually; the CRS extension could still be amended, though.

On the other hand, only WCS 1.1.2 allows output format specification on a per coverage basis whereas all others standards allow to report supported formats in the global service metadata only.

The WCS 2.0.1 corrigendum introduces the concept of native CRSes and formats which are reported in the coverage description. The native CRS is the one the domain set uses.

Counterintuitively, the WCS 2.0.1 native file format is not necessarily the same as the file format of the stored data. Since not all source file formats are supported as the output file format (e.g. ENVISAT N1), it is rather the default format delivered when there no specific file format is requested (omitting the `FORMAT` parameter in GetCoverage requests).

Supported Output Formats and WCS 2.0.1 Native Format

As most services (all but WCS 1.1.2, see the table above) allow output format configuration only per service instance, we propose that the list of supported formats shall be kept in the global configuration. This can be most easily done by adding new items to the global configuration file `conf/eoxserver.conf`.

Due to the nature of the data transmitted by these services the configuration should be separate for WMS and WCS.

The EOxServer implementation for WCS 2.0 and EO-WCS requires three parameters to be defined for each supported format:

- the MIME type
- the name of the GDAL driver
- the default file extension

The possible format choices are restricted by the capabilities of the underlying SW components (MapServer and GDAL). The list of allowed formats can be fount at http://www.gdal.org/formats_list.html.

Although the source format (i.e. the actual format of the stored data) could be determined for each coverage individually at runtime it is preferable to store this information in the database for performance reasons.

The actual native format announced by the WCS 2.0.1 compliant coverage description can differ from the source format as not every source format can be used as output format.

The implementation of the native format reporting for WCS 2.0.1 requires that EOxServer knows the mapping from the source to WCS 2.0.1 native format. As this mapping varies depending on the GDAL version, available external libraries or simply on the preference of the instance administrator the actual mapping shall be configurable, i.e., it shall be a configuration item in `conf/eoxserver.conf`.

For all the proposed configuration items reasonable default shall be provided.

Supported CRSes

All services but WCS 2.0.1 support per-coverage or per-layer reporting of CRSes. The WCS 2.0 CRS extension is not yet finished and it is suggested that it, too, should allow for CRS metadata being reported in the coverage description, although this provision is not included in the current draft of the document.

Currently, the EOxServer implementation of WMS and WCS sets the `ows_srs` MapServer parameter to the original CRS of a coverage. Thus the currently only announced CRS is the native CRS of the dataset.

This RFC proposes to introduce global configuration items for WCS and WMS, respectively, allowing definition of CRSes to be reported in addition to the native CRS. These CRSes shall also be used for EO-WMS layers corresponding to DatasetSeries.

In order to report a native CRS for Referenceable Grid Coverages the data model needs to be changed to include the SRID of the GCP projection of ReferenceableDatasets.

Proposed Implementation

Changes to the Data Model

For implementing the native format reporting in WCS 2.0.1, an additional field `gdal_driver_name` on the `LocalDataPackage` and `RemoteDataPackage` model shall be added. For the `RasdamanDataPackage` model, a dedicated database field is not necessary as the GDAL driver is already known because of the nature of the data package. The driver name should be provided by the `DataPackageWrapper` (page 190) implementation.

In order to report the native CRS of Referenceable Datasets, a `srid` field shall be added to the `ReferenceableDatasetRecord` model.

Changes to the Configuration Files

The following new configuration settings are needed for output format handling:

- a list of GDAL formats with MIME types and a flag indicating if the format is writable or read-only
- a list of MIME types to be reported as supported formats in WMS
- a list of MIME types to be reported as supported formats in WCS
- a default format MIME type to be used for native format reporting in WCS 2.0.1
- an optional mapping of source format to for native format reporting in WCS 2.0.1

The list of GDAL formats shall be configured in a CSV-like separate configuration file in `conf/formats.conf`. Each line in the file shall correspond to a given format. The syntax is as follows:

```
<GDAL driver name>,<MIME type>,<either "rw" for writable or "ro" for read-only formats>,<default
```

e.g.:

```
GTiff,image/tiff,rw,.tiff
```

Empty lines shall be ignored as well as any comments started by single # (hash) character and ended by the end of the line.

A default configuration (`default_formats.conf`) and a template (`TEMPLATE_formats.conf`) shall be included in the `eoxserver/conf` directory. The default configuration shall only be used as a fall-back if no `formats.conf` file is available in the instance `conf` directory.

The other configuration settings shall be defined in `conf/eoxserver.conf`:

```
[services.ows.wcs]
supported_formats=<MIME type>[,<MIME type>,...]

[services.ows.wms]
supported_formats=<MIME type>[,<MIME type>,...]

[services.ows.wcs.wcs20]
default_native_format=<MIME type>
source_to_native_format_map=[<src MIME type>,<dst MIME type>[,<src MIME type>,<dst MIME type>,...
```

The following new configuration settings are needed for CRS handling:

- a list of supported CRS IDs (SRIDs) for WMS layers
- a list of supported CRS IDs (SRIDs) for WCS coverages

The respective entries in `conf/eoxserver.conf`:

```
[services.ows.wcs]
supported_crs=<SRID>[,<SRID>,...]

[services.ows.wms]
supported_crs=<SRID>[,<SRID>,...]
```

Default settings shall be defined in `eoxserver/conf/default.conf`.

Module eoxserver.resources.coverages.formats

In order to support output format handling a dedicated module shall be implemented that

- reads the list of GDAL formats from the configuration files
- map GDAL driver names to MIME types and vice versa
- map MIME type (i.e., format) to default file extensions
- map source format to WCS 2.0.1 native format

Changes to the Service Implementations

The WMS and WCS modules need to be altered to use the new global settings in the service and layer / coverage configuration.

The hard-coded format settings in WCS 2.0 (`eoxserver.services.ows.wcs.wcs20.getcov` (page 176) module) shall be removed.

The GDAL driver name obtained from the `DataPackageWrapper` (page 190) implementation shall be translated at runtime to the respective MIME type using the functionality provided by `eoxserver.resources.coverages.formats` (page 196) module (inluding the translation from the source MIME type to the WCS 2.0.1 native MIME type).

Changes to the Administration Tools

The `create_instance` command shall copy the template format configuration file to the `conf` directory of the instance.

The Coverage Managers shall store the GDAL driver name of the native format in the database.

Voting History

Motion To accept RFC 17

Voting Start 2012-05-11

Voting End 2012-05-17

Result +5 for ACCEPTED

Traceability

Requirements N/A

Tickets N/A

3.3.19 RFC 18: Operator Interface Architecture

Author Stephan Krause, Fabian Schindler

Created 2012-05-08

Last Edit $Date$

Status PENDING

Discussion n/a

The new Operator Interface of EOxServer shall become the main entrance point for operators who want to administrate an EOxServer instance. The Web UI design shall focus on usability and support for frequent administration tasks.

The architecture of the Operator Interface shall be modular and extensible in order to accomodate for future extension and facilitate the maintenance of the software.

Introduction

At the moment operators have two possibilities to administrate an EOxServer instance:

- Command Line Tools
- Administration Web Client

The current Administration Client implementation is based on the `django.contrib.admin`[61] package and very tightly coupled with the data model of EOxServer. Whereas this approach has made the development considerably easier it has several severe drawbacks with respect to usability and safety of the system:

- the EOxServer data model is fairly complicated and handling it requires a deep understanding of the EO-WCS standard as well as Django concepts like model inheritance
- certain actions trigger long-running processing tasks on the server side that are so far hidden from the operators
- there is no support for asynchronous requests which would be the preferred method
- error reporting and status monitoring is only minimal
- the current Admin Interface allows to edit database records without checks for consistency; the danger of breaking the system unintentionally is quite high

Therefore a new web-based Operator Interface shall be designed that facilitates the administration tasks. It shall be more usable in the sense that

- the design shall focus on frequent administration tasks rather than the data model
- the interface shall provide guidance for operators
- safety shall be increased by checking the consistency of input data and organizing the operator actions in a way that precludes unintentionally breaking the system
- the operator shall have an overview of the processing tasks going on in the backend

From the software point of view, the design shall focus on

- modularity and extensibility, thus preparing for future extensions of EOxServer and increasing maintainability
- reusing existing administration code like Coverage Managers
- separation of model, view and controller components where model and controller components should be concentrated on the server side and the view on the client side

[61] https://docs.djangoproject.com/en/1.4/ref/contrib/admin/#module-django.contrib.admin

Requirements

The Operator Interface shall support the most frequent tasks for administration. These include:

- registering a Dataset
- handling the Range Types
- creating a Dataset Series
- creating a Stitched Mosaic
- deleting a Dataset, Dataset Series or Stitched Mosaic
- adding a Dataset to a Dataset Series or Stitched Mosaic
- removing a Dataset from a Dataset Series or Stitched Mosaic
- creating / adding / removing a data source to/from a Dataset Series or Stitched Mosaic
- viewing the logs
- enabling / disabling of components
- user management

Basic Concepts

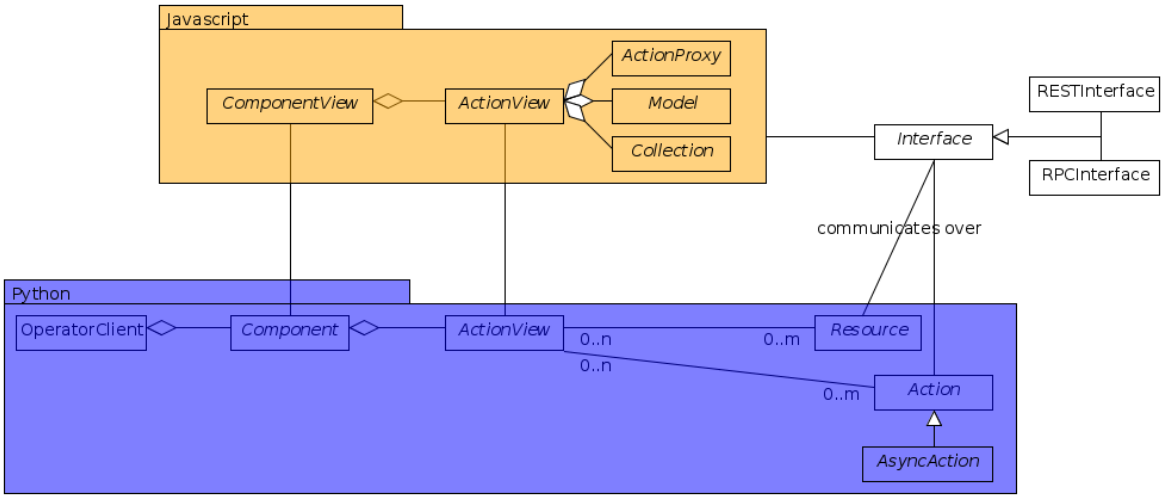


Figure 3.9: *The Operator Interface structure expressed in a UML class diagram.*

The Operator Interface shall be organized in so called Operator Components. Operator Components correspond to groups of related packages and modules of EOxServer or its extensions. The most important components at the moment are `eoxserver.core` and `eoxserver.resources.coverages`.

An Operator Component bundles Actions and Views related to the specific EOxServer component in the backend.

Actions provide an interface for operators to edit the system configuration including the data and metadata stored in the database. Most Actions are related to resources, e.g. coverages or Dataset Series.

In order to make the functionality of these Actions available, the Operator Interface shall include Action Views. Action Views shall group actions and information that are closely related to each other.

Each Operator Component may contain several Action Views. They represent a UI for access to the actions in the backend. Several Actions may be attached to a single Action View, and Actions may appear in several Action Views.

For example, an Action View might show a list of Rectified Datasets with basic metadata which allows to create and delete items. Creation and deletion should each be modeled as Actions on the server side. Another Action View may show the whole information for a single Rectified Dataset and include forms and inputs to edit the metadata.

As far as possible, the Action Views should be composed of reusable Widgets. Widgets consist of HTML and/or JavaScript. The aforementioned list of Rectified Datasets would be a typical example. It could be used also in the Dataset Series View.

The core implementation of the Operator Interface shall provide reusable components to build Widgets of (e.g. lists ...).

The communication between the Action Views and the underlying Actions should be done via specific Interfaces. One REST-based interface shall be implemented whiche shall allow to read data and metadata to be displayed, and one RPC-based interface shall be implemented in order to trigger actions on the server side.

Detailed Concept Description

In this chapter, the introduced concepts will be elaborated in detail.

Layout of the Operator Interface

The entry point to the operator interface shall be a dashboard-like page. It is envisaged to present a tab for each Operator Component; this tab shall contain an overview of the Action Views the Operator Component exhibits.

So, on the client side, each Operator Component should provide:

- A name for the Operator Component that will be shown as caption of the tab
- the overview of the Operator Component, which links to the Action Views
- the Action Views
- the Widgets used in the Action Views
- a widget to be displayed on the entry page dashboard (optional)

Each visual representation of the Operator Interface, namely the entry page dashboard, the Operator Component overview and the Action Views consist of:

- A Django HTML template
- A JavaScript View class
- A python class, entailing arbitrary information and “glue” between the other two parts

Only the third part needs to be adjusted when creating a new visual element, for both the template and the JavaScript class defaults shall help with the usage.

Action Views and Operator Component overviews should fit into the same basic layout; customizable CSS should be used for styling. The design of the entry page design (dashboard) may differ from the design of the sub-pages.

Components and Operator Components

Proposed Operator Components:

- User Management
- Configuration Management
- Action Control Center
- Coverages

Action Views

Proposed Action Views:

- User Management
  - add/delete users
  - edit permissions
- Configuration Management
  - enable and disable components
  - edit configuration settings
- Action Control Center
  - overview over running and completed actions
  - detail views for actions, including status and logs
- Coverages
  - For both Rectified and Referenceable datasets:
    - list view including limited update and delete actions
    - detail view including update and delete actions
    - create view to create a new dataset
  - For both Rectified Stitched Mosaics and Dataset Series
    - list view including limited update and delete actions
    - detail view including update, delete and synchronize actions and a list display of all contained datasets and data sources including actions to insert/remove data sources or datasets
    - create view to create a new dataset
  - list view of Range Types with create, limited update and delete actions
  - detail view of Range Types with update and delete actions and a list display of all included Bands with update actions
  - list view of Nil Values with create, update and delete actions

Actions

The Actions shall be represented by corresponding Python classes on the server side. Actions shall be reusable in the sense that they can also be invoked using a CLI command.

Most Actions are tied to resources like coverages. Resources in that sense should not be confused with database models. In most cases, a resource will be tied to a higher-level object: coverage resources for instance shall be tied to the wrappers defined in `eoxserver.resources.coverages.wrappers` (page 218).

It should be possible to invoke Actions in synchronous and asynchronous mode. For the asynchronous mode, the existing facilities of the *Asynchronous Task Processing* (page 105) (the `eoxserver.resources.processes`) shall be adapted and extended. For this purpose, the `eoxserver.resources.processes.models.LogRecord` shall receive an additional field `level`, which specifies the log level the log record was created with. This allows easy filtering for a minimum log level and e.g: only show errors and warnings raised during a process.

Every Action shall expose methods to

- validate the parameters
- start the Action and return the ID of that action

- stop the Action
- check the status of the Action
- check the log messages issued by the Action (maybe this is better implemented using the Resource mechanism)

On the client side, Actions are wrapped with ActionProxy objects that offer an easy API and abstraction for the remote invocation of the Actions methods. For Asynchronous Action the AsyncActionProxy offers a specialization.

Resources

Resources are an interface to the data stored as models in the database but also custom data sources are possible. When applied to models, a resource allows the create, read, update and delete (CRUD) methods, but this may be restricted per resource for certain models where the modification of data requires a more elaborate handling.

On the client side, Resources are wrapped in Models and Collections, which provide a layer of abstraction and handle the communication with and consume the REST interface offered by the server. A Model is the abstraction of a single dataset and a Collection is a set of models in a certain context.

Both Models and Collections offer certain events, to which the client can react in a suitable manner. This may trigger a synchronization of data with the server or a (re-)rendering of data on the client in an associated view. Additionally, models offer validation, which can be used for example to check if all mandatory fields are set, or inputs are syntactically correct.

Interfaces

The following interfaces will be used to exchange data between the server and the client:

RPC Interface Actions shall be triggered via the RPC Interface. Invocation from the Operator Interface can be synchronous or asynchronous. Incoming requests from the Operator Interface shall be dispatched to the respective Actions using a common mechanism that implements the following workflow:

- validate the parameters conveyed with the request, using the Action interface
- in case they are invalid, return an error code
- in case they are valid, proceed
- queue the Action in the asynchronous processing queue
- return a response that contains the Action ID

Using the Action ID, the Operator Interface can

- check the status of the Action
- view the log messages issued by the Action
- cancel the Action

REST Interface The REST interface shall be used for resource data retrieval and simple modification. Usually a REST interface is tightly bound to a database model and its fields. Thus modification of data via REST should only be possible in simple situations where there is no dependency tp other resources and no other synchronization mechanism necessary.

Where the REST interface is not applicable, the RPC interface shall be used.

Directory Structure

For the server part, the directory structure of the operator interface follows the standard guidelines for Django apps (as created with the *django-admin.py startapp* command):

```
operator/
|-- action.py
|-- common.py
|-- component.py
|-- __init__.py
|-- resource.py
|-- sites.py
|-- static
|   `-- operator
|       |-- actions.js
|       |-- actionviews.js
|       |-- componentviews.js
|       |-- main.js
|       |-- router.js
|       `-- widgets.js
`-- templates
    `-- operator
        |-- base_actionview.html
        |-- base_component.html
        `-- operatorsite.html
```

In the templates directory all Django templates are held. It is encouraged to use the same scheme for all components to be implemented.

The static files are placed in the sub-folder "operator" which serves as a namespaces for javascript module retrieval. All components shall use an additional unique subfolder to avoid collision. For example: "operator/coverages".

Implementation Details

In this chapter, the proposed implementation API of components explained.

Implementing Components

To create a component, one simply shall have to subclass the abstract base class provided by the Operator Interface API. It shall be easily adjustible by using either a custom JavaScript view class or a different django template.

To further improve the handling of components, several default properties within the subclass can be used, like title, name, description or others. Of course default values shall be provided.

Components are registered by the Operator Interface API function `register()`, which shall be sufficient to append it to the visualized components.

Example:

```
import operatorinterface as operator

class MyAComponent(operator.Component):
    dependencies = [SomeOtherComponent]
    name = "ComponentA"
    javascript_class = "operator/component/MyAComponentView"

operator.site.register(MyAComponent)
```

Implementing Action Views

The implementation of action views is very much like the implementation of components and should follow the same rules concerning JavaScript view classes and django templates.

Additionally it shall have two fields named `actions` and `resources`, each is a list of Action or Resource classes.

Example:

```
class MyTestActionView(operator.ActionView):
    actions = [MyTestAction]
    resources = [ResourceA, ResourceB]
    name = "mytestactionview"
    javascript_class = "operator/component/MyTestActionView"
```

Implementing Resources

Implementing Resources should be as easy as implementing actions. As with Actions, Resources are implemented by subclassing the according abstract base class and providing several options. The only mandatory arguments shall be the Django model to be externalized, optional are the permissions required for this resource, maybe means to limit the acces to read-/write-only (maybe coupled to the provided permissions) and the inc-/exclusion of model fields.

Example:

```
class MyResource(ModelResource):
    model = MyModel
    exclude = ( ... )
    include = ( ... )
    permissions = [ ... ]
```

Implementing Actions

To implement a new Action, it shall be enough to inherit from an abstract base class and implement the required methods. Once registered the operator framework shall handle the URL and method registration.

Example:

```
class ProgressAction(BaseAction):
    name = "progressaction"
    permissions = [ ... ]

    def validate(self, params):
        ...

    def start(self):
        ...

    def status(self, obj_id):
        ...

    def stop(self, obj_id):
        ...

    def view_logs(self, obj_id, timeframe=None):
        ...
```

Access Control

The Operator Interface itself, its Resources and its Actions shall only be accessible for authorized users. Also, the Interface shall distinguish between at least two types of users: administrative users and users that only have reading permissions and are not allowed to alter data. The permissions shall be able to be set fine-grained, on a per-action or per-resource basis.

It is proposed to use the Django buil-in auth framework and its integrations in other software frameworks.

Configuration and Registration of Components

On the server side, the Operator Interface is set up similar to the Djangos built-in Admin Interface. To enable the Operator Interface, its app identifier has to be inserted in the *INSTALLED_APPS* list setting and its URLs have to be included in the URLs configuration file.

Also similar to the Admin Interface, the Operator Interface provides an *autodiscover()* function, which sweeps through all *INSTALLED_APPS* directories in search of a *operator.py* module, which shall contain the apps setup of Components, Action Views, Actions and Resources.

Example Component: Coverage Component

This chapter explains an the example component to handle all kinds of interactions concerning coverages, mosaics and dataset series respectively all types of assorted metadata.

Requirements

As described earlier, the interactions shall entail creating/updating/deleting coverages and containers aswell inserting coverages into containers. Additionally users shall also trigger a synchronization process on rectified stitched mosaics and dataset series. As this may well be a time-consuming task, scanning through both the database and the (possibly remote) filesystem, it shall be handled asynchronously and output status messages.

Last but not least, all coverage metadata shall also be handled, including geo-spatial, earth observational and raster specific metadata.

The above requirements can be summarized in the following groups:

- Coverage Handling (also includes geospatial and EO-meta-data as the relation is one-to-one)
- Container Handling (same as above)
- Range Type Handling (as other more tied meta-data is handled in the other sections)

The requirement groups will be implemented as Action Views on the client, using specific widgets to allow interaction.

Server-Side implementation

The identified requirements have several implications on the server side. First off the three Action Views need to be declared to implement the three groups of reqiurements listed above and suited with the needed resources and actions.

Resources For simple access to the internally stored data, a list of Resources need to be defined: one for each coverage/container type, one for range types, bands and nil values and also for data sources.

For asynchronous tasks, also the running tasks and their logs need to be exposed as resources.

Actions The actions derived from the requirements can be summarized in the following list: add coverage to a container, remove a coverage from a container, add a data source to a container, remove a datasource from a container, manually start a synchronization process for a container. The first two actions can likely be handled synchronously as the management overhead is potentially not as high as with the latter three actions. Thus the introduced actions can be split into synchronous and asynchronous actions.

Additionally, for creating/deleting coverages and containers is done by using Actions instead of their Resources, because it involves a higher order of validation and additional tasks to be done which are too complex and unreliable if controlled by the server.

Summary The following classes with their according hierarchical structure has been identified.

Component	Action Views	Resources	Actions
Ref. Coverages Rect. Mosaics Range Types Bands NilValues	Coverage Handling	Rect. Coverages Remove from Container Create Coverage Delete Coverage	Add to Container
Rect. Mosaics Coverages Dataset Series Logs	Coverages Remove Coverage Add Datasource Remove Datasource	Add Coverage	
Container Handling	Synchronize		
Create Container			
Delete Container			
Range Type Handling	Range Types		
Bands NilValues			

Client-Side implementation

From the requirements we allready have designed three Action Views, which will be implemented as Backbone views. Each offered resource from the server will have a Backbone model/collection counterpart communicating with that interface. Similarily each action will have a proxy class on the client side.

Views The hierarchy of the client views can be seen in the following figure.

Models/Collection Each offered resource is encapsulated in a model and collection. The following figure shows the relation of the model/collection layout:

ActionProxies For each Action on the server, an ActionProxy has to be instantiated on the client which handle the communication with the server. For the three Actions that are running asynchronously, a special ActionProxy subclass is used. The following figure shows which actions are handled synchronously and which follow an asynchronous approach.

Technologies Used

On the server side, the Django framework shall be used to provide the basic functionality of the Operator Interface including specifically the URL setup, HTML templating and request dispatching.

To help publishing RESTful resources, the django extension Django REST framework[62] can be used. It provides a rather simple, yet customizeable access to database model. It also supports user authorization as required in the chapter Access Control (page 319). The library is available under the BSD license.

[62] http://django-rest-framework.org/

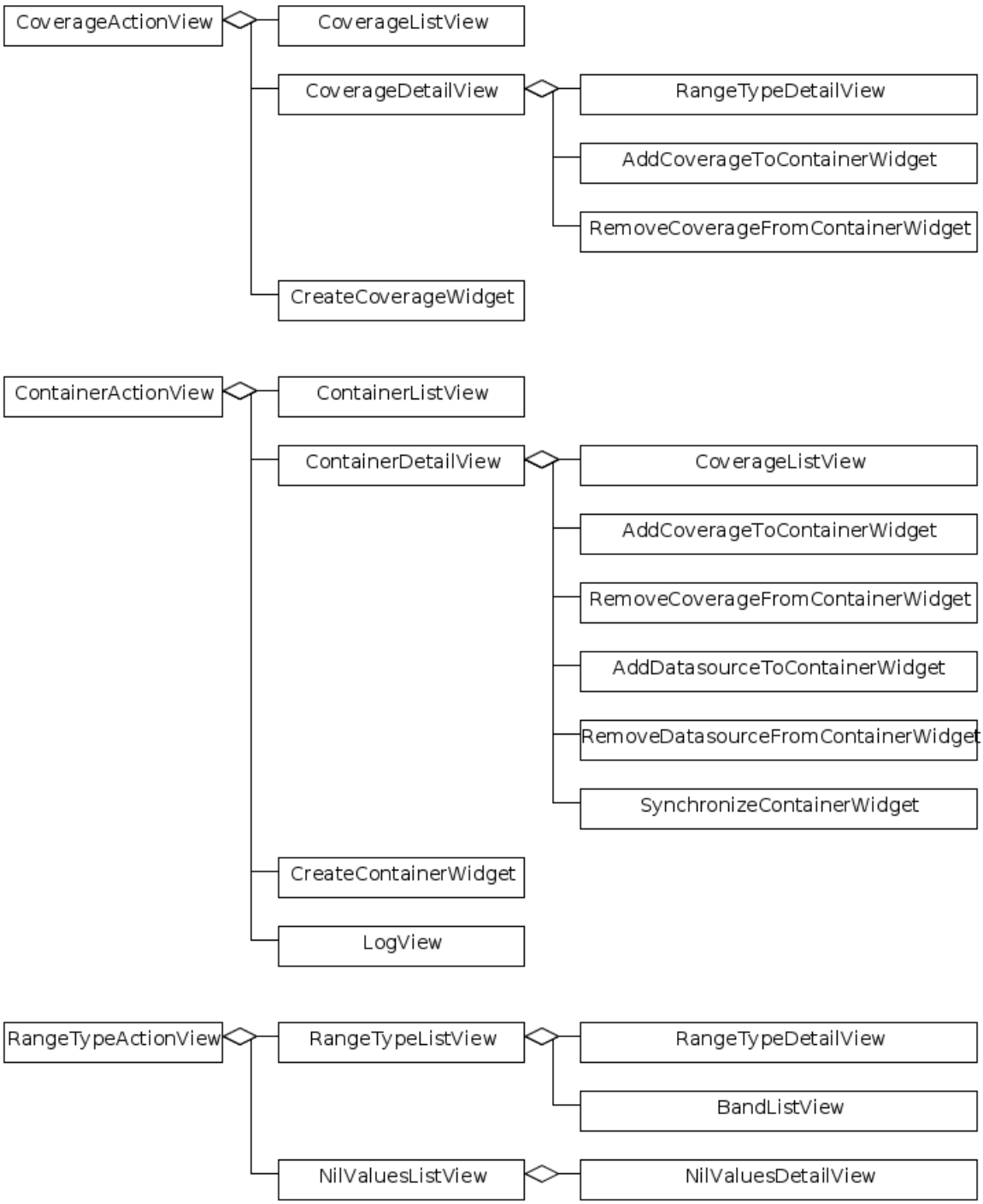


Figure 3.10: *The client views/widget hierarchy.*

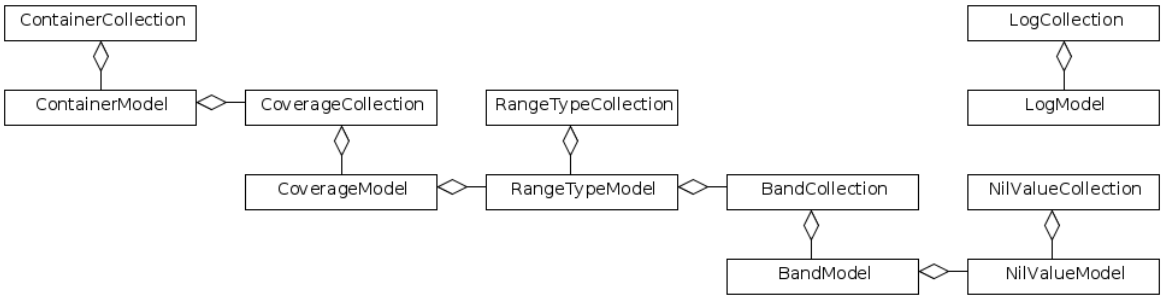


Figure 3.11: *The models/collection hierarchy on the client.*

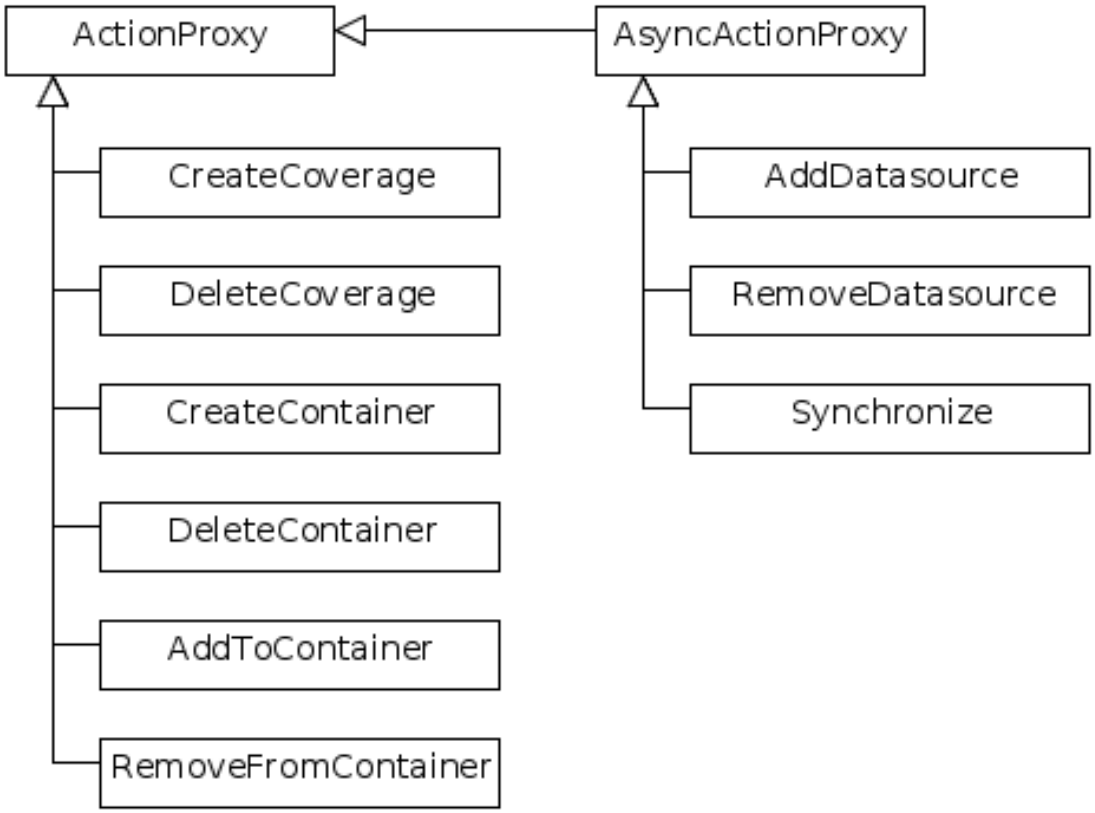


Figure 3.12: *The action proxies used on the client.*

To provide the RPC interface, there are two possibilities. The first is a wrapped setup of the SimpleXMLRPC-Server module[63], which would represent an abstraction of the XML to the actual entailed data and the dispatching of registered functions. As the module is already included in the standard library of recent Python versions, this approach would not impose an additional dependency. A drawback is the missing user authorization, which has to be implemented manually. Also, this method is only suitable for XML-RPC, which is more verbose than its JSON counterpart.

The second option would be to use a django extension framework, e.g rpc4django[64]. This framework eases the setup of RPC enabled functions, provides user authorization an is agnostic to the RPC protocol used (either JSON- or XML-RPC). This library also uses the BSD license.

On the client side, several JavaScript libraries are required. For DOM manipulation and several utility functions jQuery[65] and jQueryUI[66] are used. The libraries are licensed under the GPL and MIT licenses.

As a general utility library and dependency for other module comes Underscore[67]. To implement a working MVC layout, Backbone[68] is suggested. This library also abstracts the use of REST resources. Both libraries are distributed under the MIT license.

For calling RPC functions and parsing the output, the library rpc.js[69] is required. It adheres to either the JSON-RPC or the XML-RPC protocol. The library is dual-licensed under the MIT and the GPL license.

To display larger amounts of objects and to efficiently manipulate them, the SlickGrid[70] and its integration with Backbone, Slickback[71] are used. The two libraries are both licensed under the MIT license.

For easy management of javascript files in conjunction with other resources the requirejs[72] framework is included. It provides means to modularize javascript code and resolve dependencies. The toolset also includes an optimizer which merges and minimizes all modules into a single javascript file with no changes to the client code. The framework is published under both MIT and BSD license.

To avoid incompatibilities and third party server dependencies, all javascript libraries will be served from the EOxServer static files. This implies that for the operator client-side libraries no additional software needs to be installed as EOxServer ships with all requirements.

On the server-side the two packages *rpc4django* and *djangorestframework* need to be installed for the operator to function. As both libraries can be found on the Python Package Index (PyPI) the installation procedure using *pip* is straightforward when both dependencies are added to the EOxServer *setup.py*.

When EOxServer is installed using another technique than *pip* (like using the RPM or Debian packages), the libraries will likely have to be installed manually. For this reason they have to be listed in the dependencies page in the user manual aswell.

Dependency	Cat.	License	Purpose
Django REST Framework	Server	BSD	Expose server data via REST
RPC 4 Django	Server	BSD	Expose server methods via RPC
jQuery	Client	GPL/MIT	DOM Manipulation / AJAX Client
UnderscoreJS	Client	MIT	General Javascript utilities
BackboneJS	Client	MIT	MVC Framework, REST abstraction
json-xml-rpc	Client	GPL/MIT	RPC client
SlickGrid	Client	MIT	Data Grid widget implementation
Slickback	Client	MIT	SlickGrid to Backbone bridge
requirejs	Client	MIT/BSD	Modularization and optimization

[63] http://docs.python.org/library/simplexmlrpcserver.html
[64] http://davidfischer.name/rpc4django/
[65] http://jquery.com/
[66] http://jqueryui.com/
[67] http://underscorejs.org/
[68] http://backbonejs.org/
[69] https://github.com/westonruter/json-xml-rpc
[70] https://github.com/mleibman/SlickGrid
[71] https://github.com/teleological/slickback
[72] http://requirejs.org/

Voting History

N/A

Traceability

Requirements N/A

Tickets http://eoxserver.org/ticket/4

3.3.20 RFC 19: Migrate project repository from svn to git

Author Marko Locher

Created 2013-04-05

Last Edit $Date$

Status ACCEPTED

Discussion n/a

Migrating from Subversion to git and in the process also switch from Trac to github.

(Credit: Inspired by MapServer's RFC 84 at: http://mapserver.org/development/rfc/ms-rfc-84.html)

Introduction

While svn suits our needs as a collaborative source code version management system, it has shortcomings that make it difficult to work with for developpers working on multiple tasks in parallel. Git's easy branching makes it possible to set up branches for individual task, isolating code changes from other branches, thus making the switch from one task to another possible without the risk of loosing or erroneously commiting work-in-progress code. Three-way merging of different branches means that merging code from one branch to another becomes a rapid task, by only having to deal with actual conflicts in the code. Offline committing and access to entire history make working offline possible.

There is already somewhat of a consensus that the migration from svn to git is a good move. Discussion remains as to how this transition should be performed. This RFC outlines the different options available for hosting the official repository, and the different options available for our ticket tracking.

Current investigation has retained two majors options that we could go down with:

- Repository migrated to github, use github provided issue tracking. This option will be referred to as "Github hosting".
- Repository hosted by EOX, current trac instance migrated to hook on the new repository. This option will be referred to as "EOX hosting"

Github hosting

This option consists in moving our entire code+ticket infrastructure to github. The current trac instance becomes nearly read-only, new tickets cannot be created on it. Existing tickets are migrated to github with a script taking a trac postgresql dump (once the migration starts, our trac instance becomes read-only).

Advantages

- Code hosting:
- No need to worry about hosting infrastructure
- Can be up and running with a short delay

- Support for pull requests, allowing external contributions to be rapidly merged into our repository
- Online code editing for quick fixups
- Github visualization tools, for example to check which branches are likely to contain conflicting code sections
- Code and patch commenting make collaboratively working on a given feature very lightweight, i.e. just add your comment on the code line which seems problematic to you
- Documentation contributions highly simplified for one-shot contributions
- Issue tracking:
- Integration of ticket state with commit messages (e.g: "fix mem allocation in mapDraw(), closes issue #1234
- Email replies to ticket notifications
- The free-form label tagging of issues might open up some interesting usages
- Versionned text-base attachments (gists), with commenting

Inconveniences

- Hosting by a private company, which might become an issue if their TOS evolve or if they go out of business. The source code availability is not an issue as is possible to maintain a mirror on any server, and each developer has a checkout of the full source control history. Ticket migration would be an issue, but there are APIs available to extract existing tickets.
- Issue tracker is in some ways less feature full than trac. The only hard coded attributes are the assignee and the milestone. All the other triaging information goes into free formed labels, a la gmail.
- No way to automatically assign a ticket owner given a component
- No support for image attachments, can be referenced by url but must be hosted elsewhere.
- No support for private security tickets
- Administering committer access will be done through github, old credentials do not apply. Git does not support fine-grained commit permissions per directory, there will be a separate repository for the docs to account for the larger number of committers there.

Git Workflows

Stable Branches

This document outlines a workflow for fixing bugs in our stable branches: http://www.net-snmp.org/wiki/index.php/Git_Workflow I believe it is a very good match for our stable release management:

- pick the oldest branch where the fix should be applied
- commit the fix to this oldest branch
- merge the old branch down to all the more recent ones, including master

Release Management

Instead of freezing development during our beta cycle, a new release branch is created once the feature freeze is decided, and our betas, releases and subsequent bugfix releases are tagged off of this branch. Bug fixes are committed to this new stable branch, and merged into master. New features can continue to be added to master during all the beta phase. http://nvie.com/posts/a-successful-git-branching-model/ is an interesting read even if it does not fit our stable release branches exactly.

Upgrade path for svn users

For those users who do not wish to change their workflow and continue with svn commands. This is not the recommended way to work with git, as local or remote changes might end up in having conflicts to resolve, like with svn.

Checkout the project

```
git clone git@github.com:EOX-A/eoxserver
```

Update

```
git pull origin master
```

Commit changes

```
git add [list of files]
git commit -m "Commit message"
git push origin master
```

Fix a bug in a branch, and merge the fix into master

```
git checkout feature-branch
git add [list of files]
git commit -m "Commit message"
git push origin feature-branch
git checkout master
git merge feature-branch
git push origin master
```

Tasks

- import svn to git
- assign github users
- split into sub-projects:
- eoxserver
- autotest
- docs
- soap_proxy
- document release process
- migrate website scripts
- switch trac site to read-only

Voting History

Motion Adopted on 2013-05-15 with +1 from Stephan Meißl, Fabian Schindler, and Martin Paces

Traceability

Requirements N/A

Tickets N/A

3.3.21 RFC Policies

Author Stephan Krause, Stephan Meißl

Date 2011-05-13

This document contains the policies that govern the life cycle of Requests for Comments (RFCs). It may be changed by submitting an RFC for discussion and vote following the provisions of this document.

In this document the terms *shall*, *should* and *may* have a normative meaning, that is well known from software engineering and standards definition:

- *shall*: indicates an absolute requirement to be strictly followed
- *should*: indicates a recommendation
- *may*: indicates an option

Status of RFCs

Every RFC has a status. That status may be one of:

- **IN PREPARATION**: Some text for the RFC has been posted, but that is not the version to be submitted for discussion and voting. An RFC that has this status is still work in progress.
- **PENDING**: The text of the RFC has been submitted for discussion. It may still be altered by the RFC authors in order to reflect the state of the discussion.
- **WITHDRAWN**: The text of the RFC has been withdrawn.
- **VOTING ACTIVE**: The text of the RFC has been frozen and voting is going on.
- **ACCEPTED**: A vote has been held on the RFC and it has been accepted. Implementation has started.
- **REJECTED**: A vote has been held on the RFC and it has been rejected. The RFC is not going to be implemented and the discussion is closed.
- **POSTPONED**: A vote has been held on the RFC and it has been postponed to a later stage of development. The RFC may be reopened any time.
- **OBSOLETE**: A vote has been held on the RFC and it has been declared obsolete. It has been superseded by another RFC or it is not considered applicable any more.

The status IN PREPARATION may be declared by the authors of the RFC. They may move it to PENDING once they consider it ready for discussion and submission to a vote. Any further status changes shall be declared according to the results of the discussion and the voting (see *RFC 0: Project Steering Committee Guidelines* (page 246)).

The following status changes are possible:

- from IN PREPARATION to PENDING, WITHDRAWN
- from PENDING to WITHDRAWN or via VOTING ACTIVE to ACCEPTED, REJECTED, POSTPONED
- from ACCEPTED via VOTING ACTIVE to PENDING, POSTPONED, OBSOLETE
- from POSTPONED to PENDING or via VOTING ACTIVE to ACCEPTED, REJECTED, OBSOLETE

Creation of RFCs

Any one who has write access to the EOxServer SVN may submit an RFC. It shall obey the rules of the *Guidelines for Requests for Comments* (page 328). The initial status of the RFC is IN PREPARATION, lest the authors deem it to be mature for discussion from the start, in which case they may submit it as PENDING. The RFC shall be assigned the next possible consecutive number.

When beginning work on an RFC the authors shall inform the PSC chair.

As long as the RFC is IN PREPARATION or PENDING, only the authors of the RFC shall edit it. Anyone else who wants to contribute to the document shall submit his or her text to the discussion page. The authors may also decide to let him or her become a co-author who has all the rights of an author.

Authors may choose to support their RFC by implementing the needed changes and committing them to a subdirectory of the sandbox directory for review.

Discussion Pages

Any RFC, especially those still IN PREPARATION, shall have a discussion page on the EOxServer Trac Wiki (http://eoxserver.org/wiki). The design and the location of the discussion page is detailed in the *Guidelines for Requests for Comments* (page 328).

The discussion page may include links to preliminary implementations which have been committed to a sandbox subdirectory.

Pending RFCs

PENDING RFCs are submitted for discussion. They may still be edited to reflect the state of the discussion or to correct errors. They should not be altered in a radical manner though, changing the proposed solution completely. In this case the authors may withdraw the RFC and propose another one.

An RFC shall be PENDING for at least two business days (in Austria) till a vote can be held on it (see *RFC 0: Project Steering Committee Guidelines* (page 246)).

Withdrawal of RFCs

The authors may withdraw an RFC at any time as long as it is IN PREPARATION or PENDING. The RFC status will change to WITHDRAWN. The authors may decide to leave the text as is or remove everything except for the basic information as defined in the *Guidelines for Requests for Comments* (page 328).

Voting on RFCs

The voting on RFCs is defined in the first RFC: *RFC 0: Project Steering Committee Guidelines* (page 246).

3.3.22 Guidelines for Requests for Comments

Author Stephan Krause

Date 2011-02-19

Last Edit $Date$

Discussion http://eoxserver.org/wiki/DiscussionRfcTemplate

This document contains instructions for writing RFCs as well as a template for RFCs. Please read it carefully before submitting your own requests.

In this document the terms *shall*, *should* and *may* have a normative meaning that is well known from software engineering and standards definition:

- *shall*: indicates an absolute requirement to be strictly followed
- *should*: indicates a recommended item
- *may*: indicates an optional item

Location of an RFC

The text of an RFC shall be located in the EOxServer SVN Trunk in the directory `docs/en/rfc` under the file name `rfc<number>.rst`. It will be published automatically on the Request For Comments site once the documentation has been built anew.

Discussion Page

Once the RFC status has been moved to PENDING, it is required that the authors create a discussion page for the RFC on the EOxServer Trac Wiki. A *Template for RFC Discussion Pages* (page 331) is included below.

Structure of an RFC

Heading

The page heading shall be in the format "RFC <number>: <title>".

Basic Information

The RFC shall start with a block containing the author(s) of the request, the creation date, the date of the last edit and its status, like in the following example:

> **Author** John Doe
>
> **Created** 2011-02-18
>
> **Last Edit** $Date$
>
> **Status** PENDING
>
> **Discussion** http://eoxserver.org/wiki/DiscussionRfcTemplate

Description of the RFC

The first one or two paragraphs shall contain a short description of the RFC. They should give a high-level overview of the propositions of the request.

Introduction

The first section of the RFC shall be called "Introduction". It should contain a motivation for the RFC, describe the problem(s) the RFC addresses and give an overview of the proposed solution. It should contain forward references to the sections where specific items are discussed further where applicable.

Keep the introduction short and simple! It is not the place to go into the details, this should be done in the sections of the body of the RFC.

Body of the RFC

The body of the RFC starts right after the introduction. It may start with a more in-depth description of the motivation for the RFC and the problems to address if this cannot be discussed exhaustively in the introduction. Following that the proposed solution should be described in detail and as vividly as possible.

Use examples, tables and pictures where appropriate! Use references to external resources, to the documentation, to other RFCs, to the EOxServer Trac or to the source code.

The body of the RFC may be contained in one section or structured in sections, subsections and subsubsections or even further.

Voting History

The penultimate section of the RFC shall be called "Voting History". It shall contain the records of the votes held on subject of the RFC. As long as the RFC is in preparation or pending, the section body shall be "N/A". Example of a voting record:

> **Motion** To accept RFC 1
>
> **Voting Start** 2011-03-01
>
> **Voting End** 2011-03-02
>
> **Result** 3 ACCEPTED, 0 PENDING

Traceability

The last section of the RFC shall be called "Traceability". It shall contain references to the requirements that have motivated the request if applicable. Furthermore, if the request was accepted, it shall contain references to the tickets in the EOxServer Trac system that concern its implementation. Example:

> **Requirements** O3S_CAP_100
>
> **Tickets** #1

Where possible, the requirements and tickets shall be hyperlinked to the respective resources (e.g. requirements document, requirement tracing system, EOxServer Trac).

Template for RFCs

Here is a template you should use for your RFCs. Please replace the items in brackets <> by the appropriate text:

```
.. _rfc_<number>:

RFC <number>: <title>
=====================

:Author: <author name>
:Created: <date when RFC was created: YYYY-MM-DD>
:Last Edit: <date of last edit: YYYY-MM-DD, please use subversion keyword "Date">
:Status: <one of: IN PREPARATION, PENDING, WITHDRWAWN, VOTING ACTIVE,
          ACCEPTED, REJECTED, POSTPONED, OBSOLETE>
:Discussion: <external link to discussion page on EOxServer Trac>

<short description of the RFC>

Introduction
------------

<Mandatory. Overview of motivation, addressed problems and proposed
 solution>

<Section title>
---------------

<Any number of sections may follow.>

<Subsection title>
~~~~~~~~~~~~~~~~~~

<They may have any number of subsections.>

<Subsubsection title>
```

```
^^^^^^^^^^^^^^^^^^^^^

<And even subsubsections.>

Voting History
--------------

<Voting Records or "N/A">

:Motion: <Text of the motion>
:Voting Start: <YYYY-MM-DD>
:Voting End: <YYYY-MM-DD>
:Result: <Result>

Traceability
------------

:Requirements: <links to requirements or "N/A">
:Tickets: <links to tickets or "N/A">
```

Template for RFC Discussion Pages

RFC Discussion pages shall have the URL `http://eoxserver.org/wiki/DiscussionRfc<number>`. They shall be referenced on the page http://eoxserver.org/wiki/RfcDiscussions.

```
= Discussion Page RFC <number>: <title> =

'''RFC <number>:''' [<link>]

== Template Comment ==

<comment text>

''Author: <author name> | Created: <date and time of creation: YYYY-MM-DD HH:MM:SS>''
----

== Discussion ==
```

CHAPTER

FOUR

LICENSE

4.1 EOxServer Open License

EOxServer Open License
Version 1, 8 June 2011

4.2 EOxServer-Soap Proxy Open License

CHAPTER

FIVE

CREDITS

Work on EOxServer has been partly funded by the European Space Agency (ESA)[1] in the frame of the HMA-FO[2] and O3S[3] projects.

[4]

[1] http://www.esa.int/esaMI/ESRIN_SITE/
[2] http://wiki.services.eoportal.org/tiki-index.php?page=HMA-FO
[3] http://wiki.services.eoportal.org/tiki-index.php?page=O3S

INDEX

Symbols

__add__() (eoxserver.core.util.bbox.BBox method), 158
__and__() (eoxserver.core.util.bbox.BBox method), 157
__enter__() (eoxserver.core.util.filetools.TmpFile method), 161
__exit__() (eoxserver.core.util.filetools.TmpFile method), 161
__or__() (eoxserver.core.util.bbox.BBox method), 158
__str__() (eoxserver.core.util.filetools.TmpFile method), 161
__sub__() (eoxserver.core.util.bbox.BBox method), 158
_gerexValDriv (in module eoxserver.resources.coverages.formats), 198
_gerexValMime (in module eoxserver.resources.coverages.formats), 198
_handleException() (eoxserver.services.base.BaseRequestHandler method), 168
_initializeNamespaces() (eoxserver.core.util.xmltools.XMLEncoder method), 167
_makeElement() (eoxserver.core.util.xmltools.XMLEncoder method), 167
_processRequest() (eoxserver.services.base.BaseRequestHandler method), 168

A

access() (eoxserver.backends.cache.CacheFileWrapper method), 235
acquire() (eoxserver.resources.processes.tracker.DummyLock method), 188
acquireID() (eoxserver.resources.coverages.interfaces.ManagerInterface method), 202
addBand() (eoxserver.resources.coverages.rangetype.RangeType method), 217
addCoverage() (eoxserver.resources.coverages.interfaces.ContainerInterface method), 198
addCoverage() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 226
addCoverage() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 224
addLayers() (eoxserver.services.ows.wcs.common.WCSCommonHandler method), 178
append() (eoxserver.core.util.xmltools.XPath method), 166
applyToQuerySet() (eoxserver.core.filters.FilterInterface method), 135
Arg (class in eoxserver.core.interfaces), 140
as_tuple() (eoxserver.core.util.bbox.BBox method), 158
asDict() (eoxserver.resources.coverages.rangetype.Band method), 218
asDict() (eoxserver.resources.coverages.rangetype.NilValue method), 218
asDict() (eoxserver.resources.coverages.rangetype.RangeType method), 217
asInteger() (in module eoxserver.resources.coverages.crss), 190
asProj4Str() (in module eoxserver.resources.coverages.crss), 190
asShortCode() (in module eoxserver.resources.coverages.crss), 190
asURL() (in module eoxserver.resources.coverages.crss), 190
asURN() (in module eoxserver.resources.coverages.crss), 190
AuthorizationResponse (class in eoxserver.services.auth.base), 184
authorize() (eoxserver.services.auth.base.BasePDP method), 184
authorize() (eoxserver.services.auth.base.PolicyDecisionPointInterface method), 184
Autotest, 118
available() (eoxserver.resources.coverages.managers.CoverageIdManager method), 205

B

Band (class in eoxserver.resources.coverages.rangetype), 217
BaseExceptionHandler (class in eoxserver.services.base), 168
BaseManager (class in eoxserver.resources.coverages.managers), 213
BasePDP (class in eoxserver.services.auth.base), 184
BaseRequestHandler (class in eoxserver.services.base), 168

BBox (class in eoxserver.core.util.bbox), 157
bind() (eoxserver.core.registry.Registry method), 150
BindingMethodError, 133
BoolArg (class in eoxserver.core.interfaces), 141
BoundedArea (class in eoxserver.resources.coverages.filters), 194
BoundedAreaExpression (class in eoxserver.resources.coverages.filters), 195

C

CacheConfigReader (class in eoxserver.backends.cache), 235
CacheFile (class in eoxserver.backends.models), 240
CacheFileWrapper (class in eoxserver.backends.cache), 235
capitalize() (in module eoxserver.core.util.multiparttools), 164
check() (eoxserver.resources.coverages.managers.CoverageIdManager method), 205
check_id() (eoxserver.resources.coverages.managers.DatasetSeriesManager method), 212
check_id() (eoxserver.resources.coverages.managers.RectifiedDatasetManager method), 207
check_id() (eoxserver.resources.coverages.managers.RectifiedStitchedMosaicManager method), 210
check_id() (eoxserver.resources.coverages.managers.ReferenceableDatasetManager method), 208
clone() (eoxserver.core.registry.Registry method), 150
CommandFaultTestCase (class in eoxserver.testing.core), 242
CommandTestCase (class in eoxserver.testing.core), 242
Commit Management, 283
ComplexArg (class in eoxserver.core.interfaces), 141
ComponentManagerInterface (class in eoxserver.core.registry), 152
Config (class in eoxserver.core.config), 132
ConfigError, 133
ConfigFile (class in eoxserver.core.config), 132
ConfigReaderInterface (class in eoxserver.core.readers), 142
Configuration, 21
Configuration Options, 98
configure() (eoxserver.services.connectors.FileConnector method), 168
configure() (eoxserver.services.connectors.RasdamanArrayConnector method), 169
configure() (eoxserver.services.connectors.TiledPackageConnector method), 169
configureMapObj() (eoxserver.services.ows.wcs.common.WCSCommonHandler method), 178
configureMapObj() (eoxserver.services.ows.wcs.wcs20.getcap.WCS20GetCapabilitiesHandler method), 176
configureMapObj() (eoxserver.services.ows.wcs.wcs20.getcov.WCS20GetRectifiedCoverageHandler method), 177
configureRequest() (eoxserver.services.mapserver.MapServerOperationHandler method), 171
congigure() (eoxserver.services.mapserver.MapServerDataConnectorInter method), 171
containedIn() (eoxserver.resources.coverages.interfaces.EOCoverageInter method), 201
containedIn() (eoxserver.resources.coverages.wrappers.RectifiedDatasetW method), 219
containedIn() (eoxserver.resources.coverages.wrappers.RectifiedStitchedM method), 224
containedIn() (eoxserver.resources.coverages.wrappers.ReferenceableDat method), 222
ContainedRectifiedDatasetFilter (class in eoxserver.resources.coverages.filters), 196
ContainedReferenceableDatasetFilter (class in eoxserver.resources.coverages.filters), 196
ContainerInterface (class in eoxserver.resources.coverages.interfaces), 198
ContainingTimeIntervalExpression (class in eoxserver.resources.coverages.filters), 194
ContainingTimeIntervalFilter (class in eoxserver.resources.coverages.filters), 195
contains() (eoxserver.resources.coverages.data.DataSourceWrapper method), 192
contains() (eoxserver.resources.coverages.interfaces.ContainerInterface method), 198
contains() (eoxserver.resources.coverages.interfaces.DatasetSeriesInterfac method), 200
contains() (eoxserver.resources.coverages.interfaces.DataSourceInterface method), 200
contains() (eoxserver.resources.coverages.interfaces.EOCoverageInterface method), 201
contains() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrappe method), 227
contains() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrap method), 219
contains() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMos method), 224
contains() (eoxserver.resources.coverages.wrappers.ReferenceableDataset method), 222
copy() (eoxserver.backends.cache.CacheFileWrapper method), 235
CoverageDataInterface (class in eoxserver.resources.coverages.interfaces), 198
CoverageExpressionFactory (class in eoxserver.resources.coverages.filters), 196
CoverageGML10Encoder (class in eoxserver.services.ows.wcs.encoders), 179
CoverageIdManager (class in eoxserver.resources.coverages.managers), 205
CoverageInterface (class in eoxserver.resources.coverages.interfaces), 199
CoverageWrapper (class in eoxserver.resources.coverages.wrappers), 232

create() (eoxserver.backends.cache.CacheFileWrapper class method), 235
create() (eoxserver.core.records.RecordWrapperFactory method), 143
create() (eoxserver.core.records.RecordWrapperFactoryInterface method), 144
create() (eoxserver.core.resources.ResourceFactory method), 153
create() (eoxserver.core.resources.ResourceFactoryInterface method), 155
create() (eoxserver.resources.coverages.interfaces.ManagerInterface method), 203
create() (eoxserver.resources.coverages.managers.BaseManager method), 213
create() (eoxserver.resources.coverages.managers.DatasetSeriesManager method), 212
create() (eoxserver.resources.coverages.managers.RectifiedDatasetManager method), 207
create() (eoxserver.resources.coverages.managers.RectifiedStitchedMosaicManager method), 210
create() (eoxserver.resources.coverages.managers.ReferenceableDatasetManager method), 208
create() (eoxserver.resources.coverages.wrappers.DatasetSeriesFactory method), 230
create() (eoxserver.resources.coverages.wrappers.EOCoverageFactory method), 228
createCoverages() (eoxserver.services.ows.wcs.common.WCSCommonHandler method), 178
createCoverages() (eoxserver.services.ows.wcs.wcs20.descov.WCS20DescribeCoverageHandler method), 175
createCoverages() (eoxserver.services.ows.wcs.wcs20.getcap.WCS20GetCapabilitiesHandler method), 176
createCoverages() (eoxserver.services.ows.wcs.wcs20.getcov.WCS20GetCoverageHandler method), 177
createModel() (eoxserver.core.resources.ResourceInterface method), 155
createModel() (eoxserver.core.resources.ResourceWrapper method), 156
createModel() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 227
createModel() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 219
createModel() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 224
createModel() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
createWCSEOObjects() (eoxserver.services.ows.wcs.wcs20.desceo.WCS20DescribeEOCoverageSetHandler method), 175
Credits, 331
cup (eoxserver.core.util.bbox.BBox attribute), 158

D

Data Access Layer, 254, 269
Data Integration Layer, 254, 268
DatabaseLocationInterface (class in eoxserver.backends.interfaces), 237
DataPackageFactory (class in eoxserver.resources.coverages.data), 190
DataPackageInterface (class in eoxserver.resources.coverages.interfaces), 199
DataPackageWrapper (class in eoxserver.resources.coverages.data), 190
DatasetMetadataFileReader (class in eoxserver.resources.coverages.metadata), 213
DatasetSeriesFactory (class in eoxserver.resources.coverages.wrappers), 230
DatasetSeriesInterface (class in eoxserver.resources.coverages.interfaces), 200
DatasetSeriesManager (class in eoxserver.resources.coverages.managers), 211
DatasetSeriesWrapper (class in eoxserver.resources.coverages.wrappers), 226
DataSourceFactory (class in eoxserver.resources.coverages.data), 191
DataSourceInterface (class in eoxserver.resources.coverages.interfaces), 200
DataSourceWrapper (class in eoxserver.resources.coverages.data), 192
dbLocker() (in module eoxserver.core.util.multiprocessing.processes.tracker), 188
DecimalCodec (class in eoxserver.core.util.decoders), 160
DecoderException, 133
DecoderInterface (class in eoxserver.core.util.decoders), 160
defaultExt (eoxserver.resources.coverages.formats.Format attribute), 197
delete() (eoxserver.core.records.RecordWrapper method), 143
delete() (eoxserver.core.records.RecordWrapperFactory method), 143
delete() (eoxserver.core.records.RecordWrapperFactoryInterface method), 144
delete() (eoxserver.core.records.RecordWrapperInterface method), 145
delete() (eoxserver.core.resources.ResourceFactory method), 153
delete() (eoxserver.core.resources.ResourceFactoryInterface method), 155
delete() (eoxserver.resources.coverages.interfaces.ManagerInterface method), 203
delete() (eoxserver.resources.coverages.managers.DatasetSeriesManager method), 212
delete() (eoxserver.resources.coverages.managers.RectifiedDatasetManag method), 207
delete() (eoxserver.resources.coverages.managers.RectifiedStitchedMosai method), 210

delete() (eoxserver.resources.coverages.managers.ReferenceableDatasetManager method), 208
delete() (eoxserver.resources.coverages.wrappers.DatasetSeriesFactory method), 230
delete() (eoxserver.resources.coverages.wrappers.EOCoverageFactory method), 229
deleteModel() (eoxserver.core.resources.ResourceInterface method), 156
deleteModel() (eoxserver.core.resources.ResourceWrapper method), 156
deleteModel() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 227
deleteModel() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 219
deleteModel() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 224
deleteModel() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
deleteRetiredTasks() (in module eoxserver.resources.processes.tracker), 188
deleteTask() (in module eoxserver.resources.processes.tracker), 188
deleteTaskByIdentifier() (in module eoxserver.resources.processes.tracker), 188
Demonstration, 36
Dependencies, 15
Deployment, 22
dequeueTask() (in module eoxserver.resources.processes.tracker), 187
DescribeCoverage (Demonstration), 38
DescribeCoverage (EO-WCS Request Parameters), 43
DescribeEOCoverageSet (Demonstration), 38
DescribeEOCoverageSet (EO-WCS Request Parameters), 43
detect() (eoxserver.backends.base.LocationWrapper method), 234
detect() (eoxserver.backends.ftp.FTPStorage method), 236
detect() (eoxserver.backends.interfaces.StorageInterface method), 239
detect() (eoxserver.backends.local.LocalStorage method), 239
detect() (eoxserver.backends.rasdaman.RasdamanStorage method), 241
detect() (eoxserver.resources.coverages.data.DataSourceWrapper method), 192
detect() (eoxserver.resources.coverages.interfaces.DataSourceInterface method), 200
DictArg (class in eoxserver.core.interfaces), 141
disableImplementation() (eoxserver.core.registry.Registry method), 150
dispatch() (eoxserver.services.mapserver.MapServerOperationHandler method), 171
DisableDataCollection, 267
DOMElementToXML() (in module eoxserver.core.util.xmltools), 167
DOMtoXML() (in module eoxserver.core.util.xmltools), 167
driver (eoxserver.resources.coverages.formats.Format attribute), 197
dummyHandler() (in module eoxserver.resources.processes.tracker), 186
DummyLocW (class in eoxserver.resources.processes.tracker), 188
dynamic binding, 254

E

enableImplementation() (eoxserver.core.registry.Registry method), 150
encodeBoundedBy() (eoxserver.services.ows.wcs.encoders.CoverageGML method), 179
encodeContents() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEn method), 181
encodeContributingDatasets() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 181
encodeCountDefaultConstraint() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 182
encodeCoverageDescription() (eoxserver.services.ows.wcs.encoders.WCS20Encoder method), 184
encodeCoverageDescription() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 182
encodeCoverageDescriptions() (eoxserver.services.ows.wcs.encoders.WCS20Encoder method), 184
encodeCoverageSummary() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 182
encodeDatasetSeriesDescription() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 182
encodeDatasetSeriesDescriptions() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 182
encodeDatasetSeriesSummary() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 182
encodeDescribeEOCoverageSetOperation() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 182
encodeDomainSet() (eoxserver.services.ows.wcs.encoders.CoverageGML method), 179
encodeEarthObservation() (eoxserver.services.ows.wcs.encoders.EOPEncoder method), 180

encodeEOCoverageSetDescription() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 182
encodeEOMetadata() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 182
encodeEOProfiles() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 183
encodeException() (eoxserver.services.interfaces.ExceptionEncoderInterface method), 170
encodeExtension() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 184
encodeFootprint() (eoxserver.services.ows.wcs.encoders.EOPEncoder method), 181
encodeInvalidRequestException() (eoxserver.services.interfaces.ExceptionEncoderInterface method), 170
encodeLinearRing() (eoxserver.services.ows.wcs.encoders.GMLEncoder method), 181
encodeMetadataProperty() (eoxserver.services.ows.wcs.encoders.EOPEncoder method), 181
encodeMultiPolygon() (eoxserver.services.ows.wcs.encoders.GMLEncoder method), 181
encodeNilValue() (eoxserver.services.ows.wcs.encoders.CoverageGML10Encoder method), 179
encodePolygon() (eoxserver.services.ows.wcs.encoders.GMLEncoder method), 181
encodeRangeSet() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 183
encodeRangeType() (eoxserver.services.ows.wcs.encoders.CoverageGML10Encoder method), 179
encodeRangeTypeField() (eoxserver.services.ows.wcs.encoders.CoverageGML10Encoder method), 180
encodeRectifiedDataset() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 183
encodeRectifiedGrid() (eoxserver.services.ows.wcs.encoders.CoverageGML10Encoder method), 180
encodeRectifiedStitchedMosaic() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 183
encodeReferenceableDataset() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 183
encodeReferenceableGrid() (eoxserver.services.ows.wcs.encoders.CoverageGML10Encoder method), 180
encodeSubsetCoverageDescription() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 183
encodeSubsetDomainSet() (eoxserver.services.ows.wcs.encoders.CoverageGML10Encoder method), 180
encodeSupportedCRSs() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 184
encodeTimePeriod() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 184
encodeVersionNegotiationException() (eoxserver.services.interfaces.ExceptionEncoderInterface method), 170
encodeWGS84BoundingBox() (eoxserver.services.ows.wcs.encoders.WCS20EOAPEncoder method), 184
encodeXLink() (in module eoxserver.resources.processes.tracker), 186
EnvisatDatasetMetadataFormat (class in eoxserver.resources.coverages.metadata), 214
EO-WCS Request Parameters, 42
EOCoverageFactory (class in eoxserver.resources.coverages.wrappers), 228
EOCoverageInterface (class in eoxserver.resources.coverages.interfaces), 200
EOCoverageWrapper (class in eoxserver.resources.coverages.wrappers), 232
EODatasetWrapper (class in eoxserver.resources.coverages.wrappers), 232
EOMetadata (class in eoxserver.resources.coverages.metadata), 214
EOMetadataGML10Encoder (class in eoxserver.resources.coverages.interfaces), 201
EOMetadataInterface (class in eoxserver.resources.coverages.interfaces), 201
EOMetadataReaderInterface (class in eoxserver.resources.coverages.interfaces), 202
EOMetadataWrapper (class in eoxserver.resources.coverages.wrappers), 233
EOOMFormat (class in eoxserver.resources.coverages.metadata), 214
EOPEncoder (class in eoxserver.services.ows.wcs.encoders), 180
EOWCSObjectInterface (class in eoxserver.resources.coverages.interfaces), 202
EOxServer Configuration, 21
EOxServer Dependencies, 15
EOxServer Deployment, 22
EOxServer Installation, 14
EOxServer Instance Creation, 17
EOxServer instances, 269
EOxServer Migration, 33
EOxServer OpenLicense, 331

EOxServer Service Instance Creation, 20
EOxServer Upgrade, 33
EOxServer-SoapProxy Open License, 331
eoxserver.backends.base (module), 234
eoxserver.backends.cache (module), 235
eoxserver.backends.ftp (module), 236
eoxserver.backends.interfaces (module), 237
eoxserver.backends.local (module), 239
eoxserver.backends.models (module), 240
eoxserver.backends.rasdaman (module), 241
eoxserver.core.config (module), 132
eoxserver.core.exceptions (module), 133
eoxserver.core.filters (module), 134
eoxserver.core.interfaces (module), 136
eoxserver.core.readers (module), 142
eoxserver.core.records (module), 143
eoxserver.core.registry (module), 145
eoxserver.core.resources (module), 153
eoxserver.core.startup (module), 157
eoxserver.core.system (module), 157
eoxserver.core.util.bbox (module), 157
eoxserver.core.util.decoders (module), 158
eoxserver.core.util.filetools (module), 161
eoxserver.core.util.geotools (module), 161
eoxserver.core.util.kvptools (module), 161
eoxserver.core.util.kvptools.KVPDecoder (class in eoxserver.core.util.kvptools), 162
eoxserver.core.util.multiparttools (module), 163
eoxserver.core.util.timetools (module), 164
eoxserver.core.util.xmltools (module), 164
eoxserver.core.util.xmltools.XMLDecoder (class in eoxserver.core.util.xmltools), 165
eoxserver.resources.coverages.crss (module), 189
eoxserver.resources.coverages.data (module), 190
eoxserver.resources.coverages.filters (module), 194
eoxserver.resources.coverages.formats (module), 196
eoxserver.resources.coverages.interfaces (module), 198
eoxserver.resources.coverages.managers (module), 205
eoxserver.resources.coverages.metadata (module), 213
eoxserver.resources.coverages.rangetype (module), 216
eoxserver.resources.coverages.wrappers (module), 218
eoxserver.resources.processes.tracker (module), 185
eoxserver.services.auth.base (module), 184
eoxserver.services.base (module), 168
eoxserver.services.connectors (module), 168
eoxserver.services.exceptions (module), 169
eoxserver.services.interfaces (module), 169
eoxserver.services.mapserver (module), 170
eoxserver.services.ogc (module), 172
eoxserver.services.ows.wcs.common (module), 178
eoxserver.services.ows.wcs.encoders (module), 179
eoxserver.services.ows.wcs.wcs20.desccov (module), 175
eoxserver.services.ows.wcs.wcs20.desceo (module), 175
eoxserver.services.ows.wcs.wcs20.getcap (module), 176
eoxserver.services.ows.wcs.wcs20.getcov (module), 176
eoxserver.services.owscommon (module), 172
eoxserver.services.requests (module), 174
eoxserver.services.views (module), 175
eoxserver.testing.core (module), 242
eoxserver.testing.xcomp (module), 242
EOxServerTestCase (class in eoxserver.testing.core), 242
EOxServerTestRunner (class in eoxserver.testing.core), 242
EOxSException, 133
EPSG_AXES_REVERSED (in module eoxserver.resources.coverages.crss), 190
ExceptionEncoderInterface (class in eoxserver.services.interfaces), 170
ExceptionHandlerInterface (class in eoxserver.services.interfaces), 170
execute_command() (eoxserver.testing.core.CommandFaultTestCase method), 242
execute_command() (eoxserver.testing.core.CommandTestCase method), 242
exists() (eoxserver.backends.ftp.FTPStorage method), 236
exists() (eoxserver.core.resources.ResourceFactory method), 153
exists() (eoxserver.core.resources.ResourceFactoryInterface method), 155
exists() (eoxserver.resources.coverages.wrappers.DatasetSeriesFactory method), 230
exists() (eoxserver.resources.coverages.wrappers.EOCoverageFactory method), 229
ext (eoxserver.core.util.bbox.BBox attribute), 158

F

FactoryInterface (class in eoxserver.core.registry), 152
FactoryQueryAmbiguous, 133
FailingDescriptor (class in eoxserver.core.interfaces), 142
FailingWrapper (class in eoxserver.core.interfaces), 142
FIELDS (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper attribute), 226
FIELDS (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapp attribute), 219
FIELDS (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosai attribute), 224
FIELDS (eoxserver.resources.coverages.wrappers.ReferenceableDatasetW attribute), 221
FileConnector (class in eoxserver.services.connectors), 168
FilterExpressionInterface (class in eoxserver.core.filters), 134
FilterInterface (class in eoxserver.core.filters), 135
find() (eoxserver.core.filters.SimpleExpressionFactory method), 136
find() (eoxserver.core.records.RecordWrapperFactory method), 143

find() (eoxserver.core.registry.FactoryInterface method), 152
find() (eoxserver.core.resources.ResourceFactory method), 153
find() (eoxserver.resources.coverages.wrappers.DatasetSeriesFactory method), 231
find() (eoxserver.resources.coverages.wrappers.EOCoverageFactory method), 229
findAndBind() (eoxserver.core.registry.Registry method), 150
findFiles() (in module eoxserver.core.util.filetools), 161
findImplementations() (eoxserver.core.registry.Registry method), 150
FloatArg (class in eoxserver.core.interfaces), 141
FootprintFilter (class in eoxserver.resources.coverages.filters), 196
FootprintIntersectsAreaExpression (class in eoxserver.resources.coverages.filters), 195
FootprintIntersectsAreaFilter (class in eoxserver.resources.coverages.filters), 196
FootprintWithinAreaExpression (class in eoxserver.resources.coverages.filters), 195
FootprintWithinAreaFilter (class in eoxserver.resources.coverages.filters), 196
Format (class in eoxserver.resources.coverages.formats), 197
FormatLoaderStartupHandler (class in eoxserver.resources.coverages.formats), 198
FormatLoaderStartupHandlerImplementation (in module eoxserver.resources.coverages.formats), 198
FormatRegistry (class in eoxserver.resources.coverages.formats), 197
fromInteger() (in module eoxserver.resources.coverages.crss), 190
fromProj4Str() (in module eoxserver.resources.coverages.crss), 190
fromShortCode() (in module eoxserver.resources.coverages.crss), 190
fromURL() (in module eoxserver.resources.coverages.crss), 190
fromURN() (in module eoxserver.resources.coverages.crss), 190
FTPStorage (class in eoxserver.backends.ftp), 236
FTPStorage (class in eoxserver.backends.models), 240

G

gdalconst_to_imagemode() (in module eoxserver.services.mapserver), 172
gdalconst_to_imagemode_string() (in module eoxserver.services.mapserver), 172
GenericEOMetadataInterface (class in eoxserver.resources.coverages.interfaces), 202
GenericMetadataInterface (class in eoxserver.resources.coverages.interfaces), 202
get() (eoxserver.core.config.ConfigFile method), 132
get() (eoxserver.core.filters.SimpleExpressionFactory method), 136
get() (eoxserver.core.records.RecordWrapperFactory method), 144
get() (eoxserver.core.registry.FactoryInterface method), 152
get() (eoxserver.core.resources.ResourceFactory method), 154
get() (eoxserver.resources.coverages.wrappers.DatasetSeriesFactory method), 231
get() (eoxserver.resources.coverages.wrappers.EOCoverageFactory method), 229
get_all_ids() (eoxserver.resources.coverages.managers.DatasetSeriesMana method), 212
get_all_ids() (eoxserver.resources.coverages.managers.RectifiedDatasetM method), 207
get_all_ids() (eoxserver.resources.coverages.managers.RectifiedStitchedM method), 210
get_all_ids() (eoxserver.resources.coverages.managers.ReferenceableData method), 208
getAccessibleLocation() (eoxserver.resources.coverages.data.DataPackageWrapper method), 191
getAccessibleLocation() (eoxserver.resources.coverages.data.LocalDataPackageWrapper method), 192
getAccessibleLocation() (eoxserver.resources.coverages.data.RasdamanDataPackageWra method), 192
getAccessibleLocation() (eoxserver.resources.coverages.data.RemoteDataPackageWrapp method), 193
getAccessibleLocation() (eoxserver.resources.coverages.interfaces.DataPackageInterface method), 200
getAllowedValues() (eoxserver.resources.coverages.rangetype.RangeType method), 217
getAllRangeTypeNames() (in module eoxserver.resources.coverages.rangetype), 216
getAllReservedIds() (eoxserver.resources.coverages.managers.CoverageId method), 205
getAttrField() (eoxserver.core.resources.ResourceInterface method), 156
getAttrField() (eoxserver.core.resources.ResourceWrapper method), 156
getAttrField() (eoxserver.resources.coverages.wrappers.DatasetSeriesWra method), 227
getAttrField() (eoxserver.resources.coverages.wrappers.RectifiedDatasetW method), 219
getAttrField() (eoxserver.resources.coverages.wrappers.RectifiedStitchedI method), 224
getAttrField() (eoxserver.resources.coverages.wrappers.ReferenceableDat method), 222
getAttrNames() (eoxserver.core.resources.ResourceInterface method), 156

getAttrNames() (eoxserver.core.resources.ResourceWrapper method), 156
getAttrNames() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 227
getAttrNames() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 219
getAttrNames() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 224
getAttrNames() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getAttrValue() (eoxserver.core.resources.ResourceInterface method), 156
getAttrValue() (eoxserver.core.resources.ResourceWrapper method), 156
getAttrValue() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 227
getAttrValue() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 219
getAttrValue() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 224
getAttrValue() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getAttrValues() (eoxserver.core.resources.ResourceFactory method), 154
getAttrValues() (eoxserver.core.resources.ResourceFactoryInterface method), 155
getAttrValues() (eoxserver.resources.coverages.wrappers.DatasetSeriesFactory method), 231
getAttrValues() (eoxserver.resources.coverages.wrappers.EOCoverageFactory method), 229
getAxesSwapper() (in module eoxserver.resources.coverages.crss), 189
getBeginTime() (eoxserver.resources.coverages.interfaces.EOMetadataInterface method), 201
getBeginTime() (eoxserver.resources.coverages.metadata.EOMetadata method), 214
getBeginTime() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 227
getBeginTime() (eoxserver.resources.coverages.wrappers.EOMetadataWrapper method), 233
getBeginTime() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 219
getBeginTime() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 224
getBeginTime() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getCacheDir() (eoxserver.backends.cache.CacheConfigReader method), 235
GetCapabilities (Demonstration), 37
GetCapabilities (EO-WCS Request Parameters), 42
getCollection() (eoxserver.backends.rasdaman.RasdamanArrayWrapper method), 241
getConcurringConfigValues() (eoxserver.core.config.Config method), 132
getConfigValue() (eoxserver.core.config.Config method), 132
getContainerCount() (eoxserver.resources.coverages.interfaces.EOCoverageInterface method), 201
getContainerCount() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 219
getContainerCount() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 224
getContainerCount() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getContainers() (eoxserver.resources.coverages.interfaces.EOCoverageInterface method), 201
getContainers() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 219
getContainers() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 224
getContainers() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getContentType() (eoxserver.services.mapserver.MapServerResponse method), 172
GetCoverage (Demonstration), 40
GetCoverage (EO-WCS Request Parameters), 44
getCoverageId() (eoxserver.resources.coverages.interfaces.CoverageInterface method), 199
getCoverageId() (eoxserver.resources.coverages.wrappers.CoverageWrapper method), 232
getCoverageId() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getCoverageId() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getCoverageId() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getCoverageIds() (eoxserver.resources.coverages.managers.CoverageIdManager method), 205
getCoverages() (eoxserver.resources.coverages.data.DataPackageWrapper method), 191
getCoverageSubtype() (eoxserver.resources.coverages.interfaces.CoverageInterface method), 199
getCoverageSubtype() (eoxserver.resources.coverages.wrappers.CoverageWrapper method), 232
getCoverageSubtype() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getCoverageSubtype() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getCoverageSubtype() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getCoverageType() (eoxserver.resources.coverages.managers.CoverageIdManager method), 205
getData() (eoxserver.resources.coverages.interfaces.CoverageInterface method), 199
getData() (eoxserver.resources.coverages.wrappers.CoverageWrapper method), 232
getData() (eoxserver.resources.coverages.wrappers.PackagedDataWrapper method), 234
getData() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getData() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getData() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getData() (eoxserver.resources.coverages.wrappers.TiledDataWrapper

method), 234
getDataDirs() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 227
getDataDirs() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getDatasets() (eoxserver.resources.coverages.interfaces.DatasetSeriesInterface method), 200
getDatasets() (eoxserver.resources.coverages.interfaces.EOCoverageInterface method), 201
getDatasets() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 227
getDatasets() (eoxserver.resources.coverages.wrappers.EOCoverageWrapper method), 232
getDatasets() (eoxserver.resources.coverages.wrappers.EODatasetWrapper method), 232
getDatasets() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getDatasets() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getDatasets() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getDataSources() (eoxserver.resources.coverages.interfaces.ContainerInterface method), 198
getDataStructureType() (eoxserver.resources.coverages.data.LocalDataPackage method), 192
getDataStructureType() (eoxserver.resources.coverages.data.RasdamanDataPackage method), 192
getDataStructureType() (eoxserver.resources.coverages.data.RemoteDataPackage method), 193
getDataStructureType() (eoxserver.resources.coverages.data.TileIndexWrapper method), 193
getDataStructureType() (eoxserver.resources.coverages.interfaces.CoverageDataInterface method), 198
getDataStructureType() (eoxserver.resources.coverages.interfaces.CoverageInterface method), 199
getDataStructureType() (eoxserver.resources.coverages.wrappers.CoverageWrapper method), 232
getDataStructureType() (eoxserver.resources.coverages.wrappers.PackedDataWrapper method), 234
getDataStructureType() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getDataStructureType() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getDataStructureType() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getDataStructureType() (eoxserver.resources.coverages.wrappers.TiledDataWrapper method), 234
getDataType() (eoxserver.resources.coverages.rangetype.RangeType method), 217
getDateTime() (in module eoxserver.core.util.timetools), 164
getDBName() (eoxserver.backends.interfaces.DatabaseLocationInterface method), 237
getDBName() (eoxserver.backends.rasdaman.RasdamanArrayWrapper method), 241
getDefaultConfigValue() (eoxserver.core.config.Config method), 132
getEndTime() (eoxserver.resources.coverages.interfaces.EOMetadataInterface method), 201
getEndTime() (eoxserver.resources.coverages.metadata.EOMetadata method), 214
getEndTime() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 227
getEndTime() (eoxserver.resources.coverages.wrappers.EOMetadataWrapper method), 233
getEndTime() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getEndTime() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getEndTime() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getEOCoverages() (eoxserver.resources.coverages.interfaces.DatasetSeriesInterface method), 200
getEOCoverages() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 227
getEOCoverageSubtype() (eoxserver.resources.coverages.interfaces.EOCoverageInterface method), 201
getEOCoverageSubtype() (eoxserver.resources.coverages.wrappers.EOCoverageWrapper method), 232
getEOCoverageSubtype() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getEOCoverageSubtype() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getEOCoverageSubtype() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getEOGML() (eoxserver.resources.coverages.interfaces.EOWCSObjectInterface method), 202
getEOGML() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 227
getEOGML() (eoxserver.resources.coverages.wrappers.EOMetadataWrapper method), 233
getEOGML() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getEOGML() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getEOGML() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getEOID() (eoxserver.resources.coverages.interfaces.EOMetadataInterface

method), 201
getEOID() (eoxserver.resources.coverages.metadata.EOMetadataFormat method), 214
getEOID() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 227
getEOID() (eoxserver.resources.coverages.wrappers.EOMetadataWrapper method), 233
getEOID() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getEOID() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getEOID() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 222
getEOMetadata() (eoxserver.resources.coverages.interfaces.EOMetadataFormatInterface method), 201
getEOMetadata() (eoxserver.resources.coverages.metadata.XMLEOMetadataFormat method), 216
getEOxSPath() (eoxserver.core.config.Config method), 132
getExpectedType() (eoxserver.core.interfaces.Arg method), 140
getExtent() (eoxserver.resources.coverages.interfaces.RectifiedGridInterface method), 204
getExtent() (eoxserver.resources.coverages.interfaces.ReferenceableGridInterface method), 204
getExtent() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getExtent() (eoxserver.resources.coverages.wrappers.RectifiedGridWrapper method), 233
getExtent() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getExtent() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
getExtent() (eoxserver.resources.coverages.wrappers.ReferenceableGridWrapper method), 234
getFactoryImplementations() (eoxserver.core.registry.Registry method), 150
getFootprint() (eoxserver.resources.coverages.interfaces.EOMetadataFormatInterface method), 201
getFootprint() (eoxserver.resources.coverages.metadata.EOMetadataFormat method), 214
getFootprint() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 227
getFootprint() (eoxserver.resources.coverages.wrappers.EOMetadataWrapper method), 233
getFootprint() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getFootprint() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getFootprint() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
getFormatByMIME() (eoxserver.resources.coverages.formats.FormatRegistry method), 197
getFormatRegistry() (in module eoxserver.resources.coverages.formats), 196
getFormatsAll() (eoxserver.resources.coverages.formats.FormatRegistry method), 197
getFormatsByDriver() (eoxserver.resources.coverages.formats.FormatRegistry method), 197
getFormatsByWCS10Name() (eoxserver.resources.coverages.formats.FormatRegistry method), 197
getFromFactory() (eoxserver.core.registry.Registry method), 150
getGDALDatasetIdentifier() (eoxserver.resources.coverages.data.DataPackageWrapper method), 191
getGDALDatasetIdentifier() (eoxserver.resources.coverages.data.LocalDataPackageWrapper method), 192
getGDALDatasetIdentifier() (eoxserver.resources.coverages.data.RasdamanDataPackageWrapper method), 192
getGDALDatasetIdentifier() (eoxserver.resources.coverages.data.RemoteDataPackageWrapper method), 193
getGDALDatasetIdentifier() (eoxserver.resources.coverages.interfaces.DataPackageInterface method), 200
getGDALInterpretationAsString() (eoxserver.resources.coverages.rangetype.Band method), 218
getHeader() (eoxserver.services.requests.OWSRequest method), 174
getHost() (eoxserver.backends.ftp.RemotePathWrapper method), 236
getHost() (eoxserver.backends.interfaces.DatabaseLocationInterface method), 237
getHost() (eoxserver.backends.interfaces.RemotePathInterface method), 238
getHost() (eoxserver.backends.rasdaman.RasdamanArrayWrapper method), 241
getHTTPServiceURL() (eoxserver.services.owscommon.OWSCommonConfigReader method), 172
getId() (eoxserver.core.resources.ResourceInterface method), 156
getId() (eoxserver.core.resources.ResourceWrapper method), 156
getId() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 228
getId() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getId() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getId() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
getIdentifier() (eoxserver.resources.processes.tracker.TaskStatus method), 186
getIds() (eoxserver.core.resources.ResourceFactory method), 154
getIds() (eoxserver.core.resources.ResourceFactoryInterface method), 155
getIds() (eoxserver.resources.coverages.wrappers.DatasetSeriesFactory

method), 231
getIds() (eoxserver.resources.coverages.wrappers.EOCoverageFactory method), 230
getImagePattern() (eoxserver.resources.coverages.interfaces.RectifiedStitchedMosaicInterface method), 204
getImagePattern() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 228
getImagePattern() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getImplementationIds() (eoxserver.core.registry.Registry method), 151
getImplementationStatus() (eoxserver.core.registry.Registry method), 151
getInfo() (eoxserver.resources.processes.tracker.TaskStatus method), 186
getInstanceConfigValue() (eoxserver.core.config.Config method), 132
getLayerMetadata() (eoxserver.resources.coverages.interfaces.CoverageInterface method), 199
getLayerMetadata() (eoxserver.resources.coverages.wrappers.CoverageWrapper method), 232
getLayerMetadata() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 228
getLayerMetadata() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getLayerMetadata() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getLayerMetadata() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
getLineage() (eoxserver.resources.coverages.interfaces.EOCoverageInterface method), 201
getLineage() (eoxserver.resources.coverages.wrappers.EOCoverageWrapper method), 232
getLineage() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getLineage() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 225
getLineage() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
getLocalCopy() (eoxserver.backends.base.LocationWrapper method), 234
getLocalCopy() (eoxserver.backends.ftp.FTPStorage method), 236
getLocalCopy() (eoxserver.backends.interfaces.LocationInterface method), 238
getLocalCopy() (eoxserver.backends.interfaces.StorageInterface method), 239
getLocalCopy() (eoxserver.backends.local.LocalStorage method), 239
getLocalCopy() (eoxserver.backends.rasdaman.RasdamanStorage method), 241
getLocation() (eoxserver.backends.cache.CacheFileWrapper method), 235
getLocation() (eoxserver.resources.coverages.data.DataPackageWrapper method), 191
getLocation() (eoxserver.resources.coverages.interfaces.DataPackageInterface method), 199
getMapServerLayer() (eoxserver.services.ows.wcs.common.WCSCommonSubsetHandler method), 178
getMapServerLayer() (eoxserver.services.ows.wcs.wcs20.getcap.WCS20GetCapabilitiesHandler method), 176
getMapServerLayer() (eoxserver.services.ows.wcs.wcs20.getcov.WCS20GetCoverageHandler method), 177
getMaxQueueSize() (in module eoxserver.resources.processes.tracker), 189
getMaxSize() (eoxserver.backends.cache.CacheConfigReader method), 235
getMetadataFormat() (eoxserver.resources.coverages.interfaces.GenericMetadataInterface method), 202
getMetadataFormat() (eoxserver.resources.coverages.metadata.EOMetadataWrapper method), 214
getMetadataKeys() (eoxserver.resources.coverages.interfaces.GenericMetadataInterface method), 202
getMetadataKeys() (eoxserver.resources.coverages.interfaces.MetadataFormatInterface method), 203
getMetadataKeys() (eoxserver.resources.coverages.metadata.EnvisatDatasetWrapper method), 214
getMetadataKeys() (eoxserver.resources.coverages.metadata.EOMetadataWrapper method), 214
getMetadataKeys() (eoxserver.resources.coverages.metadata.MetadataFormatWrapper method), 215
getMetadataKeys() (eoxserver.resources.coverages.metadata.XMLMetadataWrapper method), 216
getMetadataLocation() (eoxserver.resources.coverages.data.DataPackageWrapper method), 191
getMetadataLocation() (eoxserver.resources.coverages.interfaces.DataPackageInterface method), 199
getMetadataValues() (eoxserver.resources.coverages.interfaces.GenericMetadataInterface method), 202
getMetadataValues() (eoxserver.resources.coverages.interfaces.MetadataFormatInterface method), 203
getMetadataValues() (eoxserver.resources.coverages.metadata.EOMetadataWrapper method), 214
getMetadataValues() (eoxserver.resources.coverages.metadata.MetadataFormatWrapper method), 215
getMetadataValues() (eoxserver.resources.coverages.metadata.XMLMetadataWrapper method), 216
getMimeType() (in module eoxserver.core.util.multiparttools), 164
getModel() (eoxserver.backends.cache.CacheFileWrapper method), 235
getModel() (eoxserver.core.resources.ResourceInterface method), 155
getModel() (eoxserver.core.resources.ResourceWrapper method), 156
getModel() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 228
getModel() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getModel() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper

method), 225
getModel() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
getModuleDirectories() (eoxserver.core.registry.RegistryConfigReader method), 152
getModules() (eoxserver.core.registry.RegistryConfigReader method), 152
getMSOutputFormatsAll() (in module eoxserver.services.ows.wcs.common), 179
getMSWCS10FormatMD() (in module eoxserver.services.ows.wcs.common), 179
getMSWCSFormatMD() (in module eoxserver.services.ows.wcs.common), 179
getMSWCSSRSMD() (in module eoxserver.services.ows.wcs.common), 179
getMultipartBoundary() (in module eoxserver.core.util.multiparttools), 164
getName() (eoxserver.resources.coverages.interfaces.MetadataFormatInterface method), 203
getName() (eoxserver.resources.coverages.metadata.EOOMFormat method), 214
getName() (eoxserver.resources.coverages.metadata.MetadataFormat method), 215
getName() (eoxserver.resources.coverages.metadata.NativeMetadataFormat method), 215
getNodes() (eoxserver.core.util.xmltools.XPath method), 166
getNodeType() (eoxserver.core.util.xmltools.XPath method), 166
getNumOperands() (eoxserver.core.filters.FilterExpressionInterface method), 134
getNumOperands() (eoxserver.core.filters.SimpleExpression method), 135
getOID() (eoxserver.backends.rasdaman.RasdamanArrayWrapper method), 241
getOperands() (eoxserver.core.filters.FilterExpressionInterface method), 135
getOperands() (eoxserver.core.filters.SimpleExpression method), 135
getOpName() (eoxserver.core.filters.FilterExpressionInterface method), 134
getOpName() (eoxserver.core.filters.SimpleExpression method), 135
getOpSymbol() (eoxserver.core.filters.FilterExpressionInterface method), 134
getOpSymbol() (eoxserver.core.filters.SimpleExpression method), 135
getOrCreate() (eoxserver.core.records.RecordWrapperFactory method), 144
getOrCreate() (eoxserver.core.records.RecordWrapperFactory method), 144
getParams() (eoxserver.core.util.decoders.Decoder method), 160
getParams() (eoxserver.core.util.decoders.DecoderInterface method), 160
getParams() (eoxserver.core.util.kvptools.eoxserver.core.util.kvptools.KV method), 163
getParams() (eoxserver.core.util.xmltools.eoxserver.core.util.xmltools.XM method), 166
getParams() (eoxserver.services.requests.OWSRequest method), 174
getParamType() (eoxserver.core.util.decoders.Decoder method), 160
getParamType() (eoxserver.core.util.decoders.DecoderInterface method), 160
getParamType() (eoxserver.core.util.kvptools.eoxserver.core.util.kvptools method), 163
getParamType() (eoxserver.core.util.xmltools.eoxserver.core.util.xmltools method), 166
getParamType() (eoxserver.services.requests.OWSRequest method), 174
getParamValue() (eoxserver.services.requests.OWSRequest method), 174
getParamValueStrict() (eoxserver.services.requests.OWSRequest method), 174
getPassword() (eoxserver.backends.interfaces.RemotePathInterface method), 238
getPassword() (eoxserver.backends.ftp.RemotePathWrapper method), 237
getPassword() (eoxserver.backends.interfaces.DatabaseLocationInterface method), 237
getPasswordFd() (eoxserver.backends.rasdaman.RasdamanArrayWrapper method), 241
getPath() (eoxserver.backends.ftp.RemotePathWrapper method), 237
getPath() (eoxserver.backends.interfaces.LocalPathInterface method), 237
getPath() (eoxserver.backends.interfaces.RemotePathInterface method), 238
getPath() (eoxserver.backends.local.LocalPathWrapper method), 239
getPort() (eoxserver.backends.ftp.RemotePathWrapper method), 237
getPort() (eoxserver.backends.interfaces.DatabaseLocationInterface method), 237
getPort() (eoxserver.backends.interfaces.RemotePathInterface method), 238
getPort() (eoxserver.backends.rasdaman.RasdamanArrayWrapper method), 241
getProcessedResponse() (eoxserver.services.mapserver.MapServerResponse method), 172
getQueueSize() (in module eoxserver.resources.processes.tracker), 189
getRangeType() (eoxserver.resources.coverages.interfaces.CoverageInterf method), 199
getRangeType() (eoxserver.resources.coverages.wrappers.CoverageWrapp method), 233
getRangeType() (eoxserver.resources.coverages.wrappers.RectifiedDatase method), 220

getRangeType() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 226
getRangeType() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
getRangeType() (in module eoxserver.resources.coverages.rangetype), 216
getRecord() (eoxserver.core.records.RecordWrapper method), 143
getRecord() (eoxserver.core.records.RecordWrapperInterface method), 145
getRecord() (eoxserver.resources.coverages.data.DataPackageWrapper method), 191
getRecord() (eoxserver.resources.coverages.data.DataSourceWrapper method), 192
getRegistryValues() (eoxserver.core.registry.Registry method), 151
getRequestId() (eoxserver.resources.coverages.managers.IdManager method), 205
getReservedIds() (eoxserver.resources.coverages.managers.IdManager method), 205
getResolution() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getResolution() (eoxserver.resources.coverages.wrappers.RectifiedGridWrapper method), 233
getResolution() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 226
getRetentionTime() (eoxserver.backends.cache.CacheConfigReader method), 236
getRuntimeValidationLevel() (eoxserver.core.interfaces.IntfConfigReader method), 141
getShapeFilePath() (eoxserver.resources.coverages.data.TileIndexWrapper method), 193
getShapeFilePath() (eoxserver.resources.coverages.interfaces.RectifiedGridInterface method), 204
getShapeFilePath() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 226
getSignificantFigures() (eoxserver.resources.coverages.rangetype.RangeType method), 217
getSize() (eoxserver.backends.base.LocationWrapper method), 234
getSize() (eoxserver.backends.cache.CacheFileWrapper method), 235
getSize() (eoxserver.backends.ftp.FTPStorage method), 236
getSize() (eoxserver.backends.interfaces.LocationInterface method), 238
getSize() (eoxserver.backends.interfaces.StorageInterface method), 238
getSize() (eoxserver.backends.local.LocalStorage method), 239
getSize() (eoxserver.backends.rasdaman.RasdamanStorage method), 241
getSize() (eoxserver.resources.coverages.interfaces.CoverageInterface method), 199
getSize() (eoxserver.resources.coverages.wrappers.CoverageWrapper method), 222
getSize() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getSize() (eoxserver.resources.coverages.wrappers.RectifiedGridWrapper method), 234
getSize() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 226
getSize() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
getSize() (eoxserver.resources.coverages.wrappers.ReferenceableGridWrapper method), 234
getSourceFormat() (eoxserver.resources.coverages.data.DataPackageWrapper method), 191
getSourceFormat() (eoxserver.resources.coverages.data.RasdamanDataPackage method), 192
getSourceFormat() (eoxserver.resources.coverages.data.TileIndexWrapper method), 193
getSRID() (eoxserver.resources.coverages.interfaces.RectifiedGridInterface method), 204
getSRID() (eoxserver.resources.coverages.interfaces.ReferenceableGridInterface method), 204
getSRID() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 220
getSRID() (eoxserver.resources.coverages.wrappers.RectifiedGridWrapper method), 234
getSRID() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 226
getSRID() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
getSRID() (eoxserver.resources.coverages.wrappers.ReferenceableGridWrapper method), 234
getStatus() (eoxserver.resources.processes.tracker.TaskStatus method), 186
getStatus() (eoxserver.services.mapserver.MapServerResponse method), 172
getStorageCapabilities() (eoxserver.backends.base.LocationWrapper method), 234
getStorageCapabilities() (eoxserver.backends.ftp.FTPStorage method), 236
getStorageCapabilities() (eoxserver.backends.interfaces.LocationInterface method), 237
getStorageCapabilities() (eoxserver.backends.interfaces.StorageInterface method), 238
getStorageCapabilities() (eoxserver.backends.local.LocalStorage method), 239
getStorageCapabilities() (eoxserver.backends.rasdaman.RasdamanStorage method), 241
getStorageDir() (eoxserver.resources.coverages.data.TileIndexWrapper method), 193
getStorageType() (eoxserver.backends.ftp.RemotePathWrapper method), 237
getStorageType() (eoxserver.backends.interfaces.RemotePathInterface method), 238
getSupportedCRS_WCS() (in module eoxserver.resources.coverages.crss), 189
getSupportedCRS_WMS() (in module

eoxserver.resources.coverages.crss), 189
getSupportedFormatsWCS() (eoxserver.resources.coverages.formats.FormatRegistry method), 197
getSupportedFormatsWMS() (eoxserver.resources.coverages.formats.FormatRegistry method), 197
getSystemModules() (eoxserver.core.registry.RegistryConfigReader method), 153
getTaskIdentifier() (in module eoxserver.resources.processes.tracker), 187
getTaskInfo() (in module eoxserver.resources.processes.tracker), 187
getTaskLog() (in module eoxserver.resources.processes.tracker), 188
getTaskResponse() (in module eoxserver.resources.processes.tracker), 188
getTaskStatus() (in module eoxserver.resources.processes.tracker), 187
getTaskStatusByIdentifier() (in module eoxserver.resources.processes.tracker), 187
getType() (eoxserver.backends.ftp.FTPStorage method), 236
getType() (eoxserver.backends.ftp.RemotePathWrapper method), 237
getType() (eoxserver.backends.interfaces.StorageInterface method), 238
getType() (eoxserver.backends.local.LocalPathWrapper method), 239
getType() (eoxserver.backends.local.LocalStorage method), 239
getType() (eoxserver.backends.rasdaman.RasdamanArrayWrapper method), 241
getType() (eoxserver.backends.rasdaman.RasdamanStorage method), 241
getType() (eoxserver.core.records.RecordWrapper method), 143
getType() (eoxserver.core.records.RecordWrapperInterface method), 144
getType() (eoxserver.resources.coverages.data.DataSourceWrapper method), 192
getType() (eoxserver.resources.coverages.data.LocalDataPackageWrapper method), 192
getType() (eoxserver.resources.coverages.data.RasdamanDataPackageWrapper method), 193
getType() (eoxserver.resources.coverages.data.RemoteDataPackageWrapper method), 193
getType() (eoxserver.resources.coverages.data.TileIndexWrapper method), 193
getType() (eoxserver.resources.coverages.interfaces.CoverageInterface method), 199
getType() (eoxserver.resources.coverages.interfaces.DatasetSeriesInterface method), 200
getType() (eoxserver.resources.coverages.managers.CoverageIdManager method), 205
getType() (eoxserver.resources.coverages.wrappers.CoverageWrapper method), 233
getType() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 228
getType() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 221
getType() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 226
getType() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
getUser() (eoxserver.backends.ftp.RemotePathWrapper method), 237
getUser() (eoxserver.backends.interfaces.DatabaseLocationInterface method), 237
getUser() (eoxserver.backends.interfaces.RemotePathInterface method), 238
getUser() (eoxserver.backends.rasdaman.RasdamanArrayWrapper method), 241
getValue() (eoxserver.core.util.decoders.Decoder method), 160
getValue() (eoxserver.core.util.decoders.DecoderInterface method), 160
getValue() (eoxserver.core.util.kvptools.eoxserver.core.util.kvptools.KVPDecoder method), 163
getValue() (eoxserver.core.util.xmltools.eoxserver.core.util.xmltools.XMLDecoder method), 165
getValueStrict() (eoxserver.core.util.decoders.Decoder method), 161
getValueStrict() (eoxserver.core.util.decoders.DecoderInterface method), 160
getValueStrict() (eoxserver.core.util.kvptools.eoxserver.core.util.kvptools.KVPDecoder method), 163
getValueStrict() (eoxserver.core.util.xmltools.eoxserver.core.util.xmltools.XMLDecoder method), 166
getVersion() (eoxserver.services.requests.OWSRequest method), 174
getWGS84Extent() (eoxserver.resources.coverages.interfaces.EOWCSObject method), 202
getWGS84Extent() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 228
getWGS84Extent() (eoxserver.resources.coverages.wrappers.EOMetadataWrapper method), 233
getWGS84Extent() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 221
getWGS84Extent() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 226
getWGS84Extent() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
GMLEncoder (class in eoxserver.services.ows.wcs.encoders), 181

H

handle() (eoxserver.services.base.BaseRequestHandler

method), 168
handle() (eoxserver.services.interfaces.RequestHandlerInterface method), 170
handle() (eoxserver.services.ows.wcs.wcs20.getcov.WCS20GetRefer method), 177
handleException() (eoxserver.services.base.BaseExceptionHandler method), 168
handleException() (eoxserver.services.interfaces.ExceptionHandler method), 170
hasSwappedAxes() (in module eoxserver.resources.coverages.crss), 189

I

IDInUse, 133
implement() (eoxserver.core.interfaces.Interface class method), 139
ImplementationAmbiguous, 133
ImplementationDisabled, 133
ImplementationNotFound, 133
init() (eoxserver.core.system.System class method), 157
initialize() (eoxserver.core.filters.FilterExpressionInterface method), 135
initialize() (eoxserver.core.filters.SimpleExpression method), 135
initialize() (eoxserver.resources.coverages.data.RemoteDataPackageWrapper method), 193
initialize() (eoxserver.resources.coverages.data.TileIndexWrapper method), 193
Installation, 14
Installation on CentOS, 17
Instance Creation, 17, 20
IntArg (class in eoxserver.core.interfaces), 141
Interface (class in eoxserver.core.interfaces), 139
InternalError, 133
IntersectingTimeIntervalExpression (class in eoxserver.resources.coverages.filters), 194
IntersectingTimeIntervalFilter (class in eoxserver.resources.coverages.filters), 195
IntfConfigReader (class in eoxserver.core.interfaces), 141
InvalidAxisLabelException, 169
InvalidExpressionError, 133
InvalidParameterException, 133
InvalidRequestException, 169
InvalidSubsettingException, 169
IpcException, 133
is_automatic() (eoxserver.resources.coverages.managers.RectifiedDatasetManager method), 207
is_automatic() (eoxserver.resources.coverages.managers.ReferenceableDatasetManager method), 208
isAbsolute() (eoxserver.core.util.xmltools.XPath method), 166
isAutomatic() (eoxserver.resources.coverages.wrappers.CoverageWrapper method), 233
isAutomatic() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 221
isAutomatic() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 226
isAutomatic() (eoxserver.resources.coverages.wrappers.ReferenceableDat method), 223
isAvailable() (eoxserver.resources.coverages.managers.CoverageIdManag method), 206
isOptional() (eoxserver.core.interfaces.Arg method), 140
isotime() (in module eoxserver.core.util.timetools), 164
isProjected() (in module eoxserver.resources.coverages.crss), 189
isRangeTypeName() (in module eoxserver.resources.coverages.rangetype), 216
isReserved() (eoxserver.resources.coverages.managers.CoverageIdManag method), 206
isUsed() (eoxserver.resources.coverages.managers.CoverageIdManager method), 206
isValid() (eoxserver.core.interfaces.Arg method), 140
isValidType() (eoxserver.core.interfaces.Arg method), 140
isWriteable (eoxserver.resources.coverages.formats.Format attribute), 197
IterableArg (class in eoxserver.core.interfaces), 141

K

KVPDecoder() (eoxserver.core.util.kvptools.eoxserver.core.util.kvptools.K method), 162
KVPDecoderException, 133
KVPKeyNotFound, 133
KVPKeyOccurrenceError, 133
KVPTypeError, 133
KwArgs (class in eoxserver.core.interfaces), 141

L

License, 331
ListArg (class in eoxserver.core.interfaces), 141
listToXPathExpr() (eoxserver.core.util.xmltools.XPath class method), 166
load() (eoxserver.core.registry.Registry method), 151
LocalDataPackageWrapper (class in eoxserver.resources.coverages.data), 192
LocalPath (class in eoxserver.backends.models), 240
LocalPathInterface (class in eoxserver.backends.interfaces), 237
LocalPathWrapper (class in eoxserver.backends.local), 239
LocalStorage (class in eoxserver.backends.local), 239
Location (class in eoxserver.backends.models), 240
LocationInterface (class in eoxserver.backends.interfaces), 237
LocationWrapper (class in eoxserver.backends.base), 234
LongArg (class in eoxserver.core.interfaces), 141

M

Mailing List, 36
ManagerInterface (class in eoxserver.resources.coverages.interfaces), 202

MapServerDataConnectorInterface (class in eoxserver.services.mapserver), 170
MapServerLayerGeneratorInterface (class in eoxserver.services.mapserver), 171
MapServerOperationHandler (class in eoxserver.services.mapserver), 171
MapServerRequest (class in eoxserver.services.mapserver), 171
MapServerResponse (class in eoxserver.services.mapserver), 171
mapSourceToNativeWCS20() (eoxserver.resources.coverages.formats.FormatRegistry method), 197
matches() (eoxserver.resources.coverages.wrappers.CoverageWrapper method), 233
matches() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 221
matches() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 226
matches() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
MAX_QUEUE_SIZE (in module eoxserver.resources.processes.tracker), 189
MetadataFormat (class in eoxserver.resources.coverages.metadata), 215
MetadataFormatInterface (class in eoxserver.resources.coverages.interfaces), 203
Method (class in eoxserver.core.interfaces), 139
Migration, 33
mimeType (eoxserver.resources.coverages.formats.Format attribute), 197
MissingParameterException, 134
mpPack() (in module eoxserver.core.util.multiparttools), 163
mpUnpack() (in module eoxserver.core.util.multiparttools), 163

N

NativeMetadataFormat (class in eoxserver.resources.coverages.metadata), 215
NativeMetadataFormatEncoder (class in eoxserver.resources.coverages.metadata), 215
NilValue (class in eoxserver.resources.coverages.rangetype), 218
NUM_OPS (eoxserver.core.filters.SimpleExpression attribute), 135

O

ObjectArg (class in eoxserver.core.interfaces), 141
off (eoxserver.core.util.bbox.BBox attribute), 158
OGCExceptionEncoder (class in eoxserver.services.ogc), 172
OGCExceptionHandler (class in eoxserver.services.ogc), 172
OP_NAME (eoxserver.core.filters.SimpleExpression attribute), 135
OP_SYMBOL (eoxserver.core.filters.SimpleExpression attribute), 135
open() (eoxserver.backends.interfaces.LocalPathInterface method), 237
open() (eoxserver.backends.local.LocalPathWrapper method), 239
open() (eoxserver.resources.coverages.data.DataPackageWrapper method), 191
open() (eoxserver.resources.coverages.interfaces.DataPackageInterface method), 199
OperationHandlerInterface (class in eoxserver.services.interfaces), 170
ows() (in module eoxserver.services.views), 175
OWSCommonExceptionEncoder (class in eoxserver.services.owscommon), 172
OWSCommonExceptionHandler (class in eoxserver.services.owscommon), 172
OWSCommonConfigReader (class in eoxserver.services.owscommon), 172
OWSCommonExceptionEncoder (class in eoxserver.services.owscommon), 173
OWSCommonExceptionHandler (class in eoxserver.services.owscommon), 173
OWSCommonHandler (class in eoxserver.services.owscommon), 173
OWSCommonServiceHandler (class in eoxserver.services.owscommon), 173
OWSCommonVersionHandler (class in eoxserver.services.owscommon), 173
OWSRequest (class in eoxserver.services.requests), 174
ox (eoxserver.core.util.bbox.BBox attribute), 158
oy (eoxserver.core.util.bbox.BBox attribute), 158

P

PackagedDataWrapper (class in eoxserver.resources.coverages.wrappers), 234
parse_format_param() (in module eoxserver.services.ows.wcs.common), 179
parseEPSGCode() (in module eoxserver.resources.coverages.crss), 190
pathToModuleName() (in module eoxserver.core.util.filetools), 161
pauseTask() (in module eoxserver.resources.processes.tracker), 187
PolicyDecisionPointInterface (class in eoxserver.services.auth.base), 184
PosArgs (class in eoxserver.core.interfaces), 141
postprocess() (eoxserver.services.ows.wcs.common.WCSCommonHandle method), 178
postprocess() (eoxserver.services.ows.wcs.wcs20.getcap.WCS20GetCapa method), 176

postprocess() (eoxserver.services.ows.wcs.wcs20.getcov.WCS20GetCoverageHandler method), 177
prepareAccess() (eoxserver.resources.coverages.data.DataPackageWrapper method), 191
prepareAccess() (eoxserver.resources.coverages.data.LocalDataPackageWrapper method), 192
prepareAccess() (eoxserver.resources.coverages.data.RasdamanDataPackageWrapper method), 193
prepareAccess() (eoxserver.resources.coverages.data.RemoteDataPackageWrapper method), 193
prepareAccess() (eoxserver.resources.coverages.interfaces.DataPackageInterface method), 200
Processing Chains, 265, 279
Processing Layer, 254, 268
Project Steering Committee (PSC) Guidelines, 246
purge() (eoxserver.backends.cache.CacheFileWrapper method), 235
Python Enhancement Proposals
PEP 333, 270

Q

QueueEmpty (class in eoxserver.resources.processes.tracker), 189
QueueException (class in eoxserver.resources.processes.tracker), 189
QueueFull (class in eoxserver.resources.processes.tracker), 189

R

RangeType (class in eoxserver.resources.coverages.rangetype), 216
RasdamanArrayConnector (class in eoxserver.services.connectors), 169
RasdamanArrayWrapper (class in eoxserver.backends.rasdaman), 241
RasdamanDataPackageWrapper (class in eoxserver.resources.coverages.data), 192
RasdamanLocation (class in eoxserver.backends.models), 240
RasdamanStorage (class in eoxserver.backends.models), 240
RasdamanStorage (class in eoxserver.backends.rasdaman), 241
readEOMetadata() (eoxserver.resources.coverages.data.DataPackageWrapper method), 191
readEOMetadata() (eoxserver.resources.coverages.interfaces.DataPackageInterface method), 200
readEOMetadata() (eoxserver.resources.coverages.interfaces.EOMetadataReaderInterface method), 202
readEOMetadata() (eoxserver.resources.coverages.metadata.DatasetMetadataFileReader method), 214
readEOMetadata() (eoxserver.resources.coverages.metadata.XMLEOMetadataFileReader method), 215
readGeospatialMetadata() (eoxserver.resources.coverages.data.DataPackageWrapper method), 191
readGeospatialMetadata() (eoxserver.resources.coverages.interfaces.DataPackageInterface method), 199
Recommendations for Operational Installation, 26
RecordWrapper (class in eoxserver.core.records), 143
RecordWrapperFactory (class in eoxserver.core.records), 143
RecordWrapperFactoryInterface (class in eoxserver.core.records), 144
RecordWrapperInterface (class in eoxserver.core.records), 144
RectifiedDatasetContainingTimeIntervalFilter (class in eoxserver.resources.coverages.filters), 195
RectifiedDatasetFootprintIntersectsAreaFilter (class in eoxserver.resources.coverages.filters), 196
RectifiedDatasetFootprintWithinAreaFilter (class in eoxserver.resources.coverages.filters), 196
RectifiedDatasetInterface (class in eoxserver.resources.coverages.interfaces), 203
RectifiedDatasetIntersectingTimeIntervalFilter (class in eoxserver.resources.coverages.filters), 195
RectifiedDatasetManager (class in eoxserver.resources.coverages.managers), 206
RectifiedDatasetSpatialSliceFilter (class in eoxserver.resources.coverages.filters), 195
RectifiedDatasetTimeSliceFilter (class in eoxserver.resources.coverages.filters), 195
RectifiedDatasetWrapper (class in eoxserver.resources.coverages.wrappers), 219
RectifiedGridInterface (class in eoxserver.resources.coverages.interfaces), 203
RectifiedGridWrapper (class in eoxserver.resources.coverages.wrappers), 233
RectifiedStitchedMosaicContainingTimeIntervalFilter (class in eoxserver.resources.coverages.filters), 195
RectifiedStitchedMosaicFootprintIntersectsAreaFilter (class in eoxserver.resources.coverages.filters), 196
RectifiedStitchedMosaicFootprintWithinAreaFilter (class in eoxserver.resources.coverages.filters), 196
RectifiedStitchedMosaicInterface (class in eoxserver.resources.coverages.interfaces), 204
RectifiedStitchedMosaicIntersectingTimeIntervalFilter (class in eoxserver.resources.coverages.filters), 195
RectifiedStitchedMosaicManager (class in eoxserver.resources.coverages.managers), 209

RectifiedStitchedMosaicSpatialSliceFilter (class in eoxserver.resources.coverages.filters), 196
RectifiedStitchedMosaicTimeSliceFilter (class in eoxserver.resources.coverages.filters), 195
RectifiedStitchedMosaicWrapper (class in eoxserver.resources.coverages.wrappers), 224
reenqueueTask() (in module eoxserver.resources.processes.tracker), 187
reenqueueZombieTasks() (in module eoxserver.resources.processes.tracker), 188
ReferenceableDatasetContainingTimeIntervalFilter (class in eoxserver.resources.coverages.filters), 195
ReferenceableDatasetFootprintIntersectsAreaFilter (class in eoxserver.resources.coverages.filters), 196
ReferenceableDatasetFootprintWithinAreaFilter (class in eoxserver.resources.coverages.filters), 196
ReferenceableDatasetInterface (class in eoxserver.resources.coverages.interfaces), 204
ReferenceableDatasetIntersectingTimeIntervalFilter (class in eoxserver.resources.coverages.filters), 195
ReferenceableDatasetManager (class in eoxserver.resources.coverages.managers), 208
ReferenceableDatasetSpatialSliceFilter (class in eoxserver.resources.coverages.filters), 196
ReferenceableDatasetTimeSliceFilter (class in eoxserver.resources.coverages.filters), 195
ReferenceableDatasetWrapper (class in eoxserver.resources.coverages.wrappers), 221
ReferenceableGridInterface (class in eoxserver.resources.coverages.interfaces), 204
ReferenceableGridWrapper (class in eoxserver.resources.coverages.wrappers), 234
RegisteredInterface (class in eoxserver.core.registry), 152
registerTaskType() (in module eoxserver.resources.processes.tracker), 185
Registry (class in eoxserver.core.registry), 148
RegistryConfigReader (class in eoxserver.core.registry), 152
Release Guidelines, 281
release() (eoxserver.resources.coverages.managers.CoverageIdManager method), 206
release() (eoxserver.resources.processes.tracker.DummyLock method), 188
releaseID() (eoxserver.resources.coverages.interfaces.ManagerInterface method), 203
RemoteDataPackageWrapper (class in eoxserver.resources.coverages.data), 193
RemotePath (class in eoxserver.backends.models), 240
RemotePathInterface (class in eoxserver.backends.interfaces), 238
RemotePathWrapper (class in eoxserver.backends.ftp), 236
removeCoverage() (eoxserver.resources.coverages.interfaces.ContainerInt method), 198
removeCoverage() (eoxserver.resources.coverages.wrappers.DatasetSeries method), 228
removeCoverage() (eoxserver.resources.coverages.wrappers.RectifiedStitc method), 226
RequestHandlerInterface (class in eoxserver.services.interfaces), 170
reserve() (eoxserver.resources.coverages.managers.CoverageIdManager method), 206
reset() (eoxserver.core.startup.StartupHandlerInterface method), 157
reset() (eoxserver.resources.coverages.formats.FormatLoaderStartupHand method), 198
ResourceFactory (class in eoxserver.core.resources), 153
ResourceFactoryInterface (class in eoxserver.core.resources), 154
ResourceInterface (class in eoxserver.core.resources), 155
resourceMatches() (eoxserver.core.filters.FilterInterface method), 135
ResourceWrapper (class in eoxserver.core.resources), 156
Response (class in eoxserver.services.requests), 174
resumeTask() (in module eoxserver.resources.processes.tracker), 187
reverse() (eoxserver.core.util.xmltools.XPath class method), 167
RFC
  RFC 0, 246
  RFC 1, 248
  RFC 7, 281
  RFC 8, 283
  RFC Guidelines, 328
  RFC Policies, 326

S

save() (eoxserver.core.registry.Registry method), 151
saveModel() (eoxserver.core.resources.ResourceInterface method), 155
saveModel() (eoxserver.core.resources.ResourceWrapper method), 156
saveModel() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrap method), 228
saveModel() (eoxserver.resources.coverages.wrappers.RectifiedDatasetW method), 221
saveModel() (eoxserver.resources.coverages.wrappers.RectifiedStitchedM method), 226

saveModel() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
Service Layer, 254, 267
ServiceHandlerInterface (class in eoxserver.services.interfaces), 170
setAttrs() (eoxserver.backends.ftp.RemotePathWrapper method), 236
setAttrs() (eoxserver.backends.local.LocalPathWrapper method), 239
setAttrs() (eoxserver.backends.rasdaman.RasdamanArrayWrapper method), 241
setAttrs() (eoxserver.core.records.RecordWrapper method), 143
setAttrs() (eoxserver.core.records.RecordWrapperInterface method), 145
setAttrs() (eoxserver.resources.coverages.data.DataPackageWrapper method), 190
setAttrs() (eoxserver.resources.coverages.data.DataSourceWrapper method), 192
setAttrValue() (eoxserver.core.resources.ResourceInterface method), 156
setAttrValue() (eoxserver.core.resources.ResourceWrapper method), 156
setAttrValue() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 228
setAttrValue() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 221
setAttrValue() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 226
setAttrValue() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
setFailure() (eoxserver.resources.processes.tracker.TaskStatus method), 186
setHTTPStatusCodes() (eoxserver.services.owscommon.OWSCommonExceptionHandler method), 173
setModel() (eoxserver.core.resources.ResourceInterface method), 155
setModel() (eoxserver.core.resources.ResourceWrapper method), 156
setModel() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 228
setModel() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 221
setModel() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 226
setModel() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
setMutable() (eoxserver.core.resources.ResourceInterface method), 156
setMutable() (eoxserver.core.resources.ResourceWrapper method), 157
setMutable() (eoxserver.resources.coverages.wrappers.DatasetSeriesWrapper method), 228
setMutable() (eoxserver.resources.coverages.wrappers.RectifiedDatasetWrapper method), 221
setMutable() (eoxserver.resources.coverages.wrappers.RectifiedStitchedMosaicWrapper method), 226
setMutable() (eoxserver.resources.coverages.wrappers.ReferenceableDatasetWrapper method), 223
setParams() (eoxserver.core.util.decoders.Decoder method), 161
setParams() (eoxserver.core.util.decoders.DecoderInterface method), 160
setParams() (eoxserver.core.util.kvptools.eoxserver.core.util.kvptools.KVP method), 162
setParams() (eoxserver.core.util.xmltools.eoxserver.core.util.xmltools.XML method), 165
setPaused() (eoxserver.resources.processes.tracker.TaskStatus method), 186
setRangeType() (in module eoxserver.resources.coverages.rangetype), 216
setRecord() (eoxserver.core.records.RecordWrapper method), 143
setRecord() (eoxserver.core.records.RecordWrapperInterface method), 145
setRunning() (eoxserver.resources.processes.tracker.TaskStatus method), 186
setSchema() (eoxserver.core.util.decoders.Decoder method), 161
setSchema() (eoxserver.core.util.decoders.DecoderInterface method), 160
setSchema() (eoxserver.core.util.kvptools.eoxserver.core.util.kvptools.KVP method), 162
setSchema() (eoxserver.core.util.xmltools.eoxserver.core.util.xmltools.XML method), 165
setServiceParameters() (eoxserver.services.requests.OWSRequest method), 174
setSuccess() (eoxserver.resources.processes.tracker.TaskStatus method), 186
setTaskResponse() (in module eoxserver.resources.processes.tracker), 188
setVersion() (eoxserver.services.requests.OWSRequest method), 174
SimpleExpression (class in eoxserver.core.filters), 135
SimpleExpressionFactory (class in eoxserver.core.filters), 136
size (eoxserver.core.util.bbox.BBox attribute), 158
SliceDatasetWrapper (class in eoxserver.resources.coverages.filters), 194
software architecture
  draft architecture, 252
  layers, 254
  overview, 248
  release 0.1.1, 254
  requirements, 249
SpatialFilter (class in eoxserver.resources.coverages.filters), 195
SpatialSliceExpression (class in eoxserver.resources.coverages.filters), 194
SpatialSubsetWrapper (class in eoxserver.resources.coverages.filters), 195
spiffyResponse() (eoxserver.services.mapserver.MapServerResponse method), 172

startTask() (in module eoxserver.resources.processes.tracker), 187
startup() (eoxserver.core.startup.StartupHandlerInterface method), 157
startup() (eoxserver.resources.coverages.formats.FormatLoaderStartupHandler method), 198
StartupHandlerInterface (class in eoxserver.core.startup), 157
STATUS2TEXT (in module eoxserver.resources.processes.models), 189
stopTaskFailure() (in module eoxserver.resources.processes.tracker), 187
stopTaskSuccess() (in module eoxserver.resources.processes.tracker), 187
stopTaskSuccessIfNotFinished() (in module eoxserver.resources.processes.tracker), 187
Storage (class in eoxserver.backends.models), 240
StorageInterface (class in eoxserver.backends.interfaces), 238
storeResponse() (eoxserver.resources.processes.tracker.TaskStatus method), 187
StrArg (class in eoxserver.core.interfaces), 141
StringArg (class in eoxserver.core.interfaces), 141
SubscriptableArg (class in eoxserver.core.interfaces), 141
Supported CRSs and Their Configuration, 101
Supported Raster File Formats and Their Configuration, 102
sx (eoxserver.core.util.bbox.BBox attribute), 158
sy (eoxserver.core.util.bbox.BBox attribute), 158
sync() (eoxserver.core.records.RecordWrapper method), 143
sync() (eoxserver.core.records.RecordWrapperInterface method), 145
sync() (eoxserver.resources.coverages.data.DataPackageWrapper method), 191
sync() (eoxserver.resources.coverages.data.DataSourceWrapper method), 192
synchronize() (eoxserver.resources.coverages.managers.DatasetSeriesManager method), 211, 212
synchronize() (eoxserver.resources.coverages.managers.RectifiedStitchedMosaicManager method), 209, 210
System (class in eoxserver.core.system), 157

T

TaskStatus (class in eoxserver.resources.processes.tracker), 186
test() (eoxserver.core.registry.TestingInterface method), 152
test() (eoxserver.resources.coverages.metadata.EnvisatDatasetMetadataFormat method), 215
test() (eoxserver.resources.coverages.metadata.EOOMFormat method), 214
test() (eoxserver.resources.coverages.metadata.MetadataFormat method), 215
test() (eoxserver.resources.coverages.metadata.NativeMetadataFormat method), 215
TestingInterface (class in eoxserver.core.registry), 152
TEXT2STATUS (in module eoxserver.resources.processes.models), 189
TiledDataWrapper (class in eoxserver.resources.coverages.wrappers), 234
TiledPackageConnector (class in eoxserver.services.connectors), 169
TileIndexFactory (class in eoxserver.resources.coverages.data), 193
TileIndexInterface (class in eoxserver.resources.coverages.interfaces), 204
TileIndexWrapper (class in eoxserver.resources.coverages.data), 193
TimeInterval (class in eoxserver.resources.coverages.filters), 194
TimeIntervalExpression (class in eoxserver.resources.coverages.filters), 194
TimeSliceExpression (class in eoxserver.resources.coverages.filters), 194
TimeSliceFilter (class in eoxserver.resources.coverages.filters), 195
TmpFile (class in eoxserver.core.util.filetools), 161
TypeMismatch, 134

U

UnicodeArg (class in eoxserver.core.interfaces), 141
UniquenessViolation, 134
UnknownAttribute, 134
UnknownParameterFormatException, 134
unregisterTaskType() (in module eoxserver.resources.processes.tracker), 186
update() (eoxserver.core.records.RecordWrapperFactory method), 144
update() (eoxserver.core.records.RecordWrapperFactoryInterface method), 144
update() (eoxserver.core.resources.ResourceFactory method), 154
update() (eoxserver.core.resources.ResourceFactoryInterface method), 155
update() (eoxserver.resources.coverages.interfaces.ManagerInterface method), 203
update() (eoxserver.resources.coverages.managers.BaseManager method), 213
update() (eoxserver.resources.coverages.managers.DatasetSeriesManager method), 212
update() (eoxserver.resources.coverages.managers.RectifiedDatasetManager method), 207

update() (eoxserver.resources.coverages.managers.RectifiedStitchedMosaicManager method), 211
update() (eoxserver.resources.coverages.managers.ReferenceableDatasetManager method), 208
update() (eoxserver.resources.coverages.wrappers.DatasetSeriesFactory method), 231
update() (eoxserver.resources.coverages.wrappers.EOCoverageFactory method), 230
updateModel() (eoxserver.core.resources.ResourceInterface method), 155
Upgrade, 33
Use Case, 4
UTCOffsetTimeZoneInfo (class in eoxserver.core.util.timetools), 164
ux (eoxserver.core.util.bbox.BBox attribute), 158
uy (eoxserver.core.util.bbox.BBox attribute), 158

V

valDriver() (in module eoxserver.resources.coverages.formats), 198
validate() (eoxserver.backends.cache.CacheConfigReader method), 236
validate() (eoxserver.core.interfaces.IntfConfigReader method), 142
validate() (eoxserver.core.readers.ConfigReaderInterface method), 142
validate() (eoxserver.core.registry.Registry method), 152
validate() (eoxserver.core.registry.RegistryConfigReader method), 153
validate() (eoxserver.services.owscommon.OWSCommonConfigReader method), 173
validateArgs() (eoxserver.core.interfaces.Method method), 140
validateEPSGCode() (in module eoxserver.resources.coverages.crss), 189
validateImplementation() (eoxserver.core.interfaces.Method method), 140
validateParams() (eoxserver.services.ows.wcs.common.WCSCommonHandler method), 179
validateReturnType() (eoxserver.core.interfaces.Method method), 140
validateType() (eoxserver.core.interfaces.Method method), 140
ValidationDescriptor (class in eoxserver.core.interfaces), 142
ValidationWrapper (class in eoxserver.core.interfaces), 142
valMimeType() (in module eoxserver.resources.coverages.formats), 198
VersionHandlerInterface (class in eoxserver.services.interfaces), 170
VersionNegotiationException, 169

W

WarningDescriptor (class in eoxserver.core.interfaces), 142
WarningWrapper (class in eoxserver.core.interfaces), 142
wcs10name (eoxserver.resources.coverages.formats.Format attribute), 197
WCS20CorrigendumGetCoverageHandler (class in eoxserver.services.ows.wcs.wcs20.getcov), 176
WCS20DescribeCoverageHandler (class in eoxserver.services.ows.wcs.wcs20.desccov), 175
WCS20DescribeEOCoverageSetHandler (class in eoxserver.services.ows.wcs.wcs20.desceo), 175
WCS20Encoder (class in eoxserver.services.ows.wcs.encoders), 184
WCS20EOAPEncoder (class in eoxserver.services.ows.wcs.encoders), 181
WCS20GetCapabilitiesHandler (class in eoxserver.services.ows.wcs.wcs20.getcap), 176
WCS20GetCoverageHandler (class in eoxserver.services.ows.wcs.wcs20.getcov), 177
WCS20GetRectifiedCoverageHandler (class in eoxserver.services.ows.wcs.wcs20.getcov), 177
WCS20GetReferenceableCoverageHandler (class in eoxserver.services.ows.wcs.wcs20.getcov), 177
WCSCommonHandler (class in eoxserver.services.ows.wcs.common), 178

X

xmlCompareDOMs() (in module eoxserver.testing.xcomp), 242
xmlCompareFiles() (in module eoxserver.testing.xcomp), 242
xmlCompareStrings() (in module eoxserver.testing.xcomp), 242
XMLDecoder() (eoxserver.core.util.xmltools.eoxserver.core.util.xmltools method), 165
XMLDecoderException, 134
XMLEncoder (class in eoxserver.core.util.xmltools), 167
XMLEncoderException, 134
XMLEOMetadataFileReader (class in eoxserver.resources.coverages.metadata), 215
XMLEOMetadataFormat (class in eoxserver.resources.coverages.metadata), 215
XMLError (class in eoxserver.testing.xcomp), 242

XMLMetadataFormat (class in eoxserver.resources.coverages.metadata), 216
XMLMismatchError (class in eoxserver.testing.xcomp), 242
XMLNodeNotFound, 134
XMLNodeOccurrenceError, 134
XMLParseError (class in eoxserver.testing.xcomp), 242
XMLTypeError, 134
XPath (class in eoxserver.core.util.xmltools), 166
XPathExprToList() (eoxserver.core.util.xmltools.XPath class method), 166