View Server - Operator Guide

EOX IT Services GmbH

Jul 06, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

This guide details the operation of a View Server and all of its components.

Since the View Server is a Docker based software and all of its components are distributed and executed in context of Docker images and containers, basic knowledge of Docker and Docker Swarm is a prerequisite.

1.1 Components

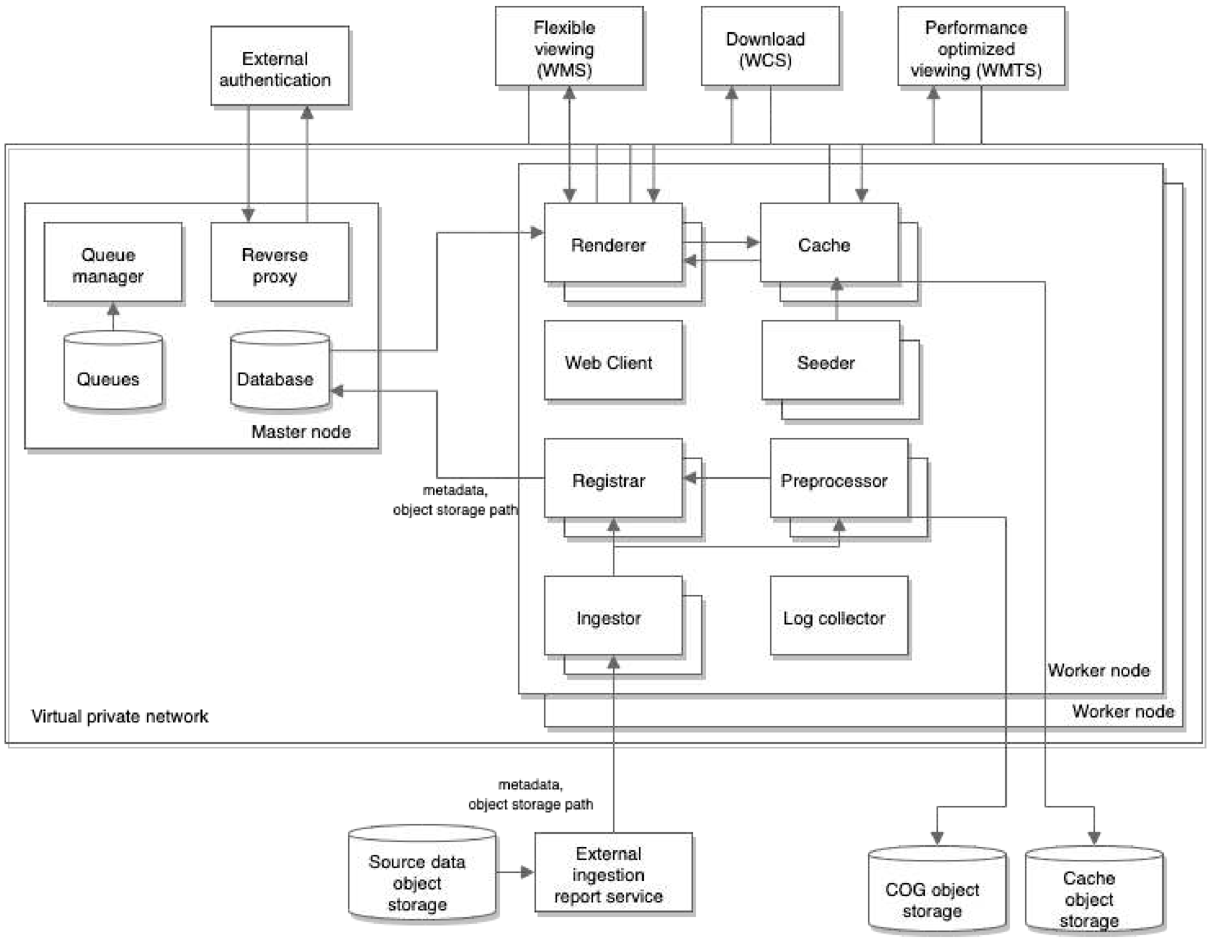


Figure 1.1.1: *View Server Architecture*

The View Server consists of the following service components (with their respective Docker image in parenthesis):

- Reverse proxy (traefik)
- Web Client (client)

- Cache (cache)
- Renderer (core)
- Registrar (core)
- Seeder (cache)
- Preprocessor (preprocessor)
- Ingestor (ingestor)
- Database (postgis)
- Queue Manager (redis)
- Log collector (fluentd)
- Kibana (kibana)
- Elasticsearch (elasticsearch)
- Shibboleth SP3 (shibauth)

These services are bundled and managed together in a Docker Swarm via Docker Compose configuration files.

1.2 Docker Images

The software is distributed as Docker images, which can be instantiated and run in their intended role. Some images are hosted on docker hub, the official and default repository for Docker images. Other images reside on an EOX hosted registry. Images from the official registry are only identified via their name, whereas images from the EOX registry conventionally use the full URL, including the domain name. Below is a list of the used images:

- mdillon/postgis:10
- redis
- traefik
- elasticsearch
- kibana
- atmoz/sftp
- registry.gitlab.eox.at/esa/prism/vs/fluentd
- registry.gitlab.eox.at/esa/prism/vs/pvs_core
- registry.gitlab.eox.at/esa/prism/vs/pvs_cache
- registry.gitlab.eox.at/esa/prism/vs/pvs_preprocessor
- registry.gitlab.eox.at/esa/prism/vs/pvs_client
- registry.gitlab.eox.at/esa/prism/vs/pvs_ingestor
- registry.gitlab.eox.at/esa/prism/vs/pvs_shibauth

1.3 Configuration Files

The following configuration files impact the behavior of the View Server:

- *index.html*: This is the main file to configure the client. In this file, the viewing layer, search and download endpoints are configured. Usually this is associated with additional backdrop and overlay layers.
- *preprocessor.yml*: This file configures the preprocessing steps.
- *mapcache.xml*: This file defines the input sources of the cache and its published layers.
- *init-db.sh*: This file sets up the registrar and renderer side of the VS.

1.4 Initialization and Setup

In order to help with the initial setup of a VS, the `pvs_starter` package described in the section *Initialization* allows to quickly establish the required structure of configuration files.

The section *Setup* describes how to deploy a Docker Swarm stack using the configuration files generated in the initialization step.

CHAPTER TWO

INITIALIZATION

In order to set up an instance of the View Server (VS), the separate `pvs_starter` utility is recommended.

2.1 Running the Initialization

The `pvs_starter` utility is distributed as a Python package and easily installed via `pip`.

```
pip3 install pvs_starter git+git@gitlab.eox.at:esa/prism/pvs_starter.git
```

Now a new VS instance can be set up like this:

```
python3 -m pvs_starter.cli config.yaml out/ -f
```

This takes the initialization configuration `config.yaml` to generate the required structure of a new VS instance in the `out/` directory.

2.2 Configuration of the Initialization

The important part of the initialization is the configuration. The file is structured in YAML as detailed below. It contains the following sections:

2.2.1 `database`

Here, access details and credentials of the database are stored. It defines the internal database name, user, and password that will be created when the stack is deployed. Note that there is no `host` setting, as this will be handled automatically within the Docker Swarm.

```
database:
  name: vs_db
  user: vs_user
  password: Go-J_eOUvj2k
```

2.2.2 django_admin

This section deals with the setup of a Django admin account. This is used to later access the admin panel to inspect the registered data.

```
django_admin:
  user: admin
  mail: office@eox.at
  password: jvLwv_20x-69
```

2.2.3 preprocessor

Here, the preprocessing can be configured in detail.

2.2.4 products

This section defines `product_type` related information. The two most important settings here are the `type_extractor` and `level_extractor` structures which specify how the product type and product level should be extracted from the metadata. For this, an XPath (or multiple) can be specified to retrieve that information.

The `types` section defines the available `product_types` and which `browse` and `mask` types are to be generated.

```
products:
  type_extractor:
    xpath:
    namespace_map:
  level_extractor:
    xpath:
    namespace_map:
  types:
    PL00:
      default_browse: TRUE_COLOR
      browses:
        TRUE_COLOR:
          red:
            expression: red
            range: [1000, 15000]
            nodata: 0
          green:
            expression: green
            range: [1000, 15000]
            nodata: 0
          blue:
            expression: blue
            range: [1000, 15000]
            nodata: 0
        FALSE_COLOR:
          red:
            expression: nir
            range: [1000, 15000]
            nodata: 0
          green:
            expression: red
            range: [1000, 15000]
            nodata: 0
          blue:
```

(continues on next page)

(continued from previous page)

```
            expression: green
            range: [1000, 15000]
            nodata: 0
        NDVI:
          grey:
            expression: (nir-red)/(nir+red)
            range: [-1, 1]
      masks:
        validity:
          validity: true
```

2.2.5 collections

In the `collections` section, the collections are set up and it is defined which products based on `product_type` and `product_level` will be inserted into them. The `product_types` must list types defined in the `products` section.

```
collections:
  COLLECTION:
    product_types:
      - PL00
    product_levels:
      - Level_1
      - Level_3
```

2.2.6 storages

Here, the three relevant storages can be configured: the `source`, `preprocessed`, and `cache` storages.

The `source` storage defines the location from which the original files will be downloaded to be preprocessed. Preprocessed images and metadata will then be uploaded to the `preprocessed` storage. The cache service will cache images on the `cache` storage.

Each storage definition uses the same structure and can target various types of storages, such as OpenStack swift.

These storage definitions will be used in the appropriate sections.

```
storages:
  source:
    auth_type: keystone
    auth_url:
    version: 3
    username:
    password:
    tenant_name:
    tenant_id:
    region_name:
    container:
  preprocessed:
    auth_type: keystone
    auth_url:
    version: 3
    username:
    password:
    tenant_name:
```

(continues on next page)

(continued from previous page)

```
    tenant_id:
    region_name:
    container:
  cache:
    type: swift
    auth_type: keystone
    auth_url: https://auth.cloud.ovh.net/v3/
    auth_url_short: https://auth.cloud.ovh.net/
    version: 3
    username:
    password:
    tenant_name:
    tenant_id:
    region_name:
    container:
```

2.2.7 cache

This section defines the exposed services, and how the layers shall be cached internally.

```
cache:
  metadata:
    title: PRISM Data Access Service (PASS) developed by EOX
    abstract: PRISM Data Access Service (PASS) developed by EOX
    url: https://vhr18.pvs.prism.eox.at/cache/ows
    keyword: view service
    accessconstraints: UNKNOWN
    fees: UNKNOWN
    contactname: Stephan Meissl
    contactphone: Please contact via mail.
    contactfacsimile: None
    contactorganization: EOX IT Services GmbH
    contactcity: Vienna
    contactstateorprovince: Vienna
    contactpostcode: 1090
    contactcountry: Austria
    contactelectronicmailaddress: office@eox.at
    contactposition: CTO
    providername: EOX
    providerurl: https://eox.at
    inspire_profile: true
    inspire_metadataurl: TBD
    defaultlanguage: eng
    language: eng
  services:
    wms:
      enabled: true
    wmts:
      enabled: true
  connection_timeout: 10
  timeout: 120
  expires: 3600
  key: /{tileset}/{grid}/{dim}/{z}/{x}/{y}.{ext}
  tilesets:
    VHR_IMAGE_2018__TRUE_COLOR:
```

(continues on next page)

(continued from previous page)

```
      title: VHR Image 2018 True Color
      abstract: VHR Image 2018 True Color
    VHR_IMAGE_2018__FALSE_COLOR:
      title: VHR Image 2018 False Color
      abstract: VHR Image 2018 False Color
    VHR_IMAGE_2018__NDVI:
      title: VHR Image 2018 NDVI
      abstract: VHR Image 2018 NDVI
      style: earth
    VHR_IMAGE_2018_Level_1__TRUE_COLOR:
      title: VHR Image 2018 Level 1 True Color
      abstract: VHR Image 2018 Level 1 True Color
    VHR_IMAGE_2018_Level_1__FALSE_COLOR:
      title: VHR Image 2018 Level 1 False Color
      abstract: VHR Image 2018 Level 1 False Color
    VHR_IMAGE_2018_Level_1__NDVI:
      title: VHR Image 2018 Level 1 NDVI
      abstract: VHR Image 2018 Level 1 NDVI
      style: earth
    VHR_IMAGE_2018_Level_1__TRUE_COLOR:
      title: VHR Image 2018 Level 3 True Color
      abstract: VHR Image 2018 Level 3 True Color
    VHR_IMAGE_2018_Level_1__FALSE_COLOR:
      title: VHR Image 2018 Level 3 False Color
      abstract: VHR Image 2018 Level 3 False Color
    VHR_IMAGE_2018_Level_1__NDVI:
      title: VHR Image 2018 Level 3 NDVI
      abstract: VHR Image 2018 Level 3 NDVI
      style: earth
```

Once the initialization is finished the next step is to deploy the Docker Swarm stack as described in the section *Setup*.

CHAPTER THREE

SETUP

In this chapter the setup of a new VS stack is detailed. Before this step can be done, the configuration and environment files need to be present. These files can be added manually or be created as described in the *Initialization* step.

3.1 Docker

In order to deploy the Docker Swarm stack to the target machine, Docker and its facilities need to be installed. This step depends on the systems architecture. On a Debian based system this may look like this:

```
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common

curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -

# add the apt repository
sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/debian \
    $(lsb_release -cs) \
    stable"

# fetch the package index and install Docker
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

3.2 Docker Swarm

Now that Docker is installed, the machine can either create a new swarm or join an existing one.

To create a new Swarm, the following command is used:

```
docker swarm init
```

Alternatively, an existing Swarm can be joined. The easiest way to do this, is to obtain a `join-token`. On an existing Swarm manager (where a Swarm was initialized or already joined as manager) run this command:

```
docker swarm join-token worker
```

This prints out a command that can be run on a machine to join the swarm:

```
docker swarm join --token <obtained token>
```

It is possible to dedicate certain workers for example to contribute to ingestion exclusively, while others can take care only for rendering. This setup has benefits, when a mixed setup of nodes with different parameters is available.

In order to set a node for example as *external*, to contribute in rendering only, one can simply run:

```
docker node update --label-add type=external <node-id>
```

Additionally, it is necessary to modify *placement* parameter in the docker compose file.

```
renderer:
  deploy:
    placement:
      constraints:
        - node.labels.type == external
```

Additional information for swarm management can be obtained in the official documentation of the project.

3.3 Docker Image Retrieval

Before the Docker images can be used, they have to be retrieved first. Currently, all images used in VS that are not off-the-shelf are hosted on the public `registry.gitlab.eox.at` registry. For production deployment, tagged releases in format `release-<major.minor.patch>` should be used.

To pull the relevant images:

```
docker pull registry.gitlab.eox.at/esa/prism/vs/pvs_core:release-x.x.x
docker pull registry.gitlab.eox.at/esa/prism/vs/pvs_cache:release-x.x.x
docker pull registry.gitlab.eox.at/esa/prism/vs/pvs_preprocessor:release-x.x.x
docker pull registry.gitlab.eox.at/esa/prism/vs/pvs_client:release-x.x.x
docker pull registry.gitlab.eox.at/esa/prism/vs/fluentd:release-x.x.x
docker pull registry.gitlab.eox.at/esa/prism/vs/pvs_ingestor:release-x.x.x
docker pull registry.gitlab.eox.at/esa/prism/vs/pvs_sftp:release-x.x.x
docker pull registry.gitlab.eox.at/esa/prism/vs/pvs_shibauth:release-x.x.x
```

3.4 Logging

For production, the docker images in the compose files use the default logging driver. Therefore we configure the default logging driver for the docker daemon to be fluent by createing the file `/etc/docker/daemon.json` with the following content:

```
{
    "log-driver": "fluentd"
}
```

For development, we don't want to redirect all of the docker logging output, so the respective compose files for dev configure the logging driver for each container.

3.5 Stack Deployment

Before the stack deployment step, some environment variables and configurations -which are considered sensitive- should be created beforehand, this can done following the `Sensitive variables` steps that are included in the next *Configuration* section.

Now that a Docker Swarm is established and docker secrets and configs are created, it is time to deploy the VS as a stack. This is done using the created Docker Compose configuration files. In order to enhance the re-usability, these files are split into multiple parts to be used for both development and final service deployment.

For a development deployment one would do (replace `name` with the actual service identifier:

```
docker stack deploy -c docker-compose.<name>.yml -c docker-compose.<name>.dev.yml
↪<stack-name>
```

This command actually performs a variety of tasks. First off, it obtains any missing images, such as the image for the reverse proxy, the database, or the redis key-value store.

When all relevant images are pulled from their respective repository the services of the stack are initialized. In the default setting, each service is represented by a single container of its respective service type. When starting for the first time, the startup procedure takes some time, as everything needs to be initialized. This includes the creation of the database, user, required tables, and the Django instance.

That process can be supervised using the `docker service ls` command, which lists all available services and their respective status.

Continue to the next section *Configuration* to read how a running VS stack can be configured.

CHAPTER

FOUR

CONFIGURATION

This chapter details how a running VS stack can be configured. And what steps are necessary to deploy the configuration.

In order for these configuration changes to be picked up by a running VS stack and to take effect some steps need to be performed. These steps are either a "re-deploy" of the running stack or a complete re-creation of it.

4.1 Stack Re-deploy

As will be further described, for some configurations it is sufficient to "re-deploy" the stack which automatically re-starts any service with changed configuration. This is done re-using the stack deployment command:

```
docker stack deploy -c docker-compose.<name>.yml -c docker-compose.<name>.dev.yml
↪<stack-name>
```

Warning: When calling the `docker stack deploy` command, it is vital to use the command with the same files and name the stack was originally created.

4.2 Stack Re-creation

In some cases a stack re-redeploy is not enough, as the configuration was used for a materialized instance which needs to be reverted. The easiest way to do this is to delete the volume in question. If, for example, the renderer/registrar configuration was updated, the `instance-data` volume needs to be re-created.

First, the stack needs to be shut down. This is done using the following command:

```
docker stack rm <stack-name>
```

When that command has completed (it is advisable to wait for some time until all containers have actually stopped) the next step is to delete the `instance-data` volume:

```
docker volume rm <stack-name>_instance-data
```

Note: It is possible that this command fails, with the error message that the volume is still in use. In this case, it is advisable to wait for a minute and to try the deletion again.

Now that the volume was deleted, the stack can be re-deployed as described above, which will trigger the automatic re-creation and initialization of the volume. For the `instance-data`, it means that the instance will be re-created and all database models with it.

4.3 Docker Compose Settings

These configurations are altering the behavior of the stack itself and its contained services. A complete reference of the configuration file structure can be found in the Docker Compose documentation.

4.4 Environment Variables

These variables are passed to their respective containers environment and change the behavior of certain functionality. They can be declared in the Docker Compose configuration file directly, but typically they are bundled by field of interest and then placed into `.env` files and then passed to the containers. So for example, there will be a `<stack-name>_obs.env` file to store the access parameters for the object storage. All those files are placed in the `env/` directory in the instances directory.

Environment variables and `.env` files are passed to the services via the `docker-compose.yml` directives. The following example shows how to pass `.env` files and direct environment variables:

```yaml
services:
  # ....
  registrar:
    env_file:
      - env/stack.env
      - env/stack_db.env
      - env/stack_obs.env
    environment:
      INSTANCE_ID: "prism-view-server_registrar"
      INSTALL_DIR: "/var/www/pvs/dev/"
      INIT_SCRIPTS: "/configure.sh /init-db.sh /initialized.sh"
      STARTUP_SCRIPTS: "/wait-initialized.sh"
      WAIT_SERVICES: "redis:6379 database:5432"
      OS_PASSWORD_FILE: "/run/secrets/OS_PASSWORD"
    # ...
```

4.4.1 .env Files

The following `.env` files are typically used:

- `<stack-name>.env`: The general `.env` file used for all services
- `<stack-name>_db.env`: The database access credentials, for all services interacting with the database.
- `<stack-name>_django.env`: This env files defines the credentials for the django admin user to be used with the admin GUI.
- `<stack-name>_obs.env`: This contains access parameters for the object storage(s).

4.4.2 Groups of Environment Variables

GDAL Environment Variables

This group of environment variables controls the intricacies of GDAL. They control how GDAL interacts with its supported files. As GDAL supports a variety of formats and backend access, most of the full list of env variables are not applicable and only a handful are actually relevant for the VS.

- `GDAL_DISABLE_READDIR_ON_OPEN` - Especially when using an Object Storage backend with a very large number of files, it is vital to activate this setting (`=TRUE`) in order to suppress to read the whole directory contents which is very slow for some OBS backends.

- CPL_VSIL_CURL_ALLOWED_EXTENSIONS - This limits the file extensions to disable the lookup of so called sidecar files which are not used for VS. By default this value is used: =.TIF,.tif,.xml.

OpenStack Swift Environment Variables

These variables define the access coordinates and credentials for the OpenStack Swift Object storage backend.

This set of variables define the credentials for the object storage to place the preprocessed results:

- ST_AUTH_VERSION
- OS_AUTH_URL_SHORT
- OS_AUTH_URL
- OS_USERNAME
- OS_PASSWORD
- OS_TENANT_NAME
- OS_TENANT_ID
- OS_REGION_NAME
- OS_USER_DOMAIN_NAME

This set of variables define the credentials for the object storage to retrieve the original product files:

- OS_USERNAME_DOWNLOAD
- OS_PASSWORD_DOWNLOAD
- OS_TENANT_NAME_DOWNLOAD
- OS_TENANT_ID_DOWNLOAD
- OS_REGION_NAME_DOWNLOAD
- OS_AUTH_URL_DOWNLOAD
- ST_AUTH_VERSION_DOWNLOAD
- OS_USER_DOMAIN_NAME_DOWNLOAD

VS Environment Variables

These environment variables are used by the VS itself to configure various parts.

Note: These variables are used during the initial stack setup. When these variables are changed, they will not be reflected unless the instance volume is re-created.

- COLLECTION - This defines the main collections name. This is used in various parts of the VS and serves as the layer base name.
- UPLOAD_CONTAINER - This controls the bucket name where the preprocessed images are uploaded to.
- DJANGO_USER, DJANGO_MAIL, DJANGO_PASSWORD - The Django admin user account credentials to use the Admin GUI.
- REPORTING_DIR - This sets the directory to write the reports of the registered products to.

Note: These variables are used during the initial stack setup. When these variables are changed, they will not be reflected unless the database volume is re-created.

These are the internal access credentials for the database:

- POSTGRES_USER
- POSTGRES_PASSWORD
- POSTGRES_DB
- DB
- DB_USER
- DB_PW
- DB_HOST
- DB_PORT
- DB_NAME

4.5 Configuration Files

Such files are passed to the containers in a similar way as environment variables, but usually contain more settings at once and are placed at a specific path in the container at runtime.

Configuration files are passed into the containers using the `configs` section of the `docker-compose.yaml` file. The following example shows how such a configuration file is defined and the used in a service:

```
# ...
configs:
  my-config:
    file: ./config/example.cfg
# ...
services:
  myservice:
    # ...
    configs:
    - source: my-config
      target: /example.cfg
```

The following configuration files are used throughout the VS:

4.5.1 `<stack-name>_init-db.sh`

This shell script file's purpose is to set up the EOxServer instance used by both the renderer and registrar.

Some browsetype functions that can be used for elevation rasters are:

`hillshade(band)`

- range 0 - 255
- nodata 0

`aspect(band)`

- range 0 - 360
- nodata -9999

`slopeshade(band)`

- range 0 - 255
- nodata -9999

`contours(band, 0, 30)`

- range 0 - 500
- nodata - 9999

Example:

```
  python3 manage.py browsetype create "DEM" "elevation" \
    --grey "gray" \
    --grey-range -100 4000 \
    --grey-nodata 0 \
    --traceback

python3 manage.py browsetype create "DEM" "hillshade" \
    --grey "hillshade(gray)" \
    --grey-range 0 255 \
    --grey-nodata 0 \
    --traceback

python3 manage.py browsetype create "DEM" "aspect" \
    --grey "aspect(gray)" \
    --grey-range 0 360 \
    --grey-nodata -9999 \
    --traceback

python3 manage.py browsetype create "DEM" "slope" \
    --grey "slopeshade(gray)" \
    --grey-range 0 255 \
    --grey-nodata -9999 \
    --traceback

python3 manage.py browsetype create "DEM" "contours" \
    --grey "contours(gray, 0, 30)" \
    --grey-range 0 500 \
    --grey-nodata -9999 \
    --traceback
```

4.5.2 `<stack-name>_index-dev.html/<stack-name>_index-ops.html`

The clients main HTML page, containing various client settings. The `dev` one is used for development only, whereas the `ops` one is used for operational deployment.

4.5.3 `<stack-name>_mapcache-dev.xml`/`<stack-name>_mapcache-ops.xml`

The configuration file for MapCache, the software powering the cache service. Similarly to the client configuration files, the `dev` and `ops` files used for development and operational usage respectively. Further documentation can be found at the official site.

4.5.4 `<stack-name>_preprocessor-config.yaml`

The configuration for the proprocessing service to use to process to be ingested files.

The files are using YAML as a format and are structured in the following fashion:

source/target

> Here, the source file storage and the target file storage are configured. This can either be a local directory or an OpenStack Swift object storage. If Swift is used for source, download container can be left unset. In that case, container can be inferred from the given path in format <bucket>/<object-name>.

workdir

> The workdir can be configured, to determine where the intermediate files are placed. This can be convenient for debugging and development.

keep_temp

> This boolean decides if the temporary directory for the preprocessing will be cleaned up after being finished. Also, convenient for development.

metadata_glob

> This file glob is used to determine the main metadata file to extract the product type from. This file will be searched in the downloaded package.

glob_case

> If all globs will be used in a case-sensitive way.

type_extractor

> This setting configures how the product type is extracted from the previously extracted metadata. In the `xpath` setting one or more XPath expressions can supplied to fetch the product type. Each XPath will be tried until one is found that produces a result. These results can then be mapped using the `map` dictionary.

level_extractor

> This section works very similar to the `type_extractor` but only for the product level. The product level is currently not used.

preprocessing

> This is the actual preprocessing configuration setting. It is split in defaults and product type specific settings. The defaults are applied where there is no setting supplied for that specific type. The product type is the one extracted earlier.
>
> defaults
>
> > This section allows to configure any one of the available steps. Each step configuration can be overridden in a specific product type configuration.
> >
> > The available steps are as follows:
> >
> > custom_preprocessor
> >
> > > A custom python function to be called.
> > >
> > > path

The Python module path to the function to call.

args

A list of arguments to pass to the function.

kwargs

A dictionary of keyword arguments to pass to the function.

subdatasets

What subdatasets to extract and how to name them.

subdataset_types

Mapping of subdataset identifier to output filename postfix for subdatasets to be extracted for each data file.

georeference

How the extracted files shall be georeferenced.

geotransforms

A list of georeference methods with options to try.

type

The type of georeferencing to apply. One of `gcp`, `rpc`, `corner`, `world`.

options

Additional options for the georeferencing. Depends on the type of georeferencing.

order

The polynomial order to use for GCP related georeferencing.

projection

The projection to use for ungeoreferenced images.

rpc_file_template

The file glob template to use to find the RPC file. Template parameters are {filename}, {fileroot}, and {extension}.

warp_options

Warp options. See https://gdal.org/python/osgeo.gdal-module.html#WarpOptions for details

corner_names

The metadata field name including the corner names. Tuple of four: bottom-left, bottom-right, top-left and top-right

orbit_direction_name

The metadata field name containing the orbit direction

force_north_up

Circumvents the naming of corner names and assumes a north-up orientation of the image.

tps

Whether to use TPS transformation instead of GCP polynomials.

calc

Calculate derived data using formulas.

formulas

A list of formulas to use to calculate derived data. Each has the following fields

inputs

A map of characters in the range of A-Z to respective inputs. Each has the following properties

glob

The input file glob

band

The input file band index (1-based)

data_type

The GDAL data type name for the output

formula

The formula to apply. See https://gdal.org/programs/gdal_calc.html#cmdoption-calc for details.

output_postfix

The postfix to apply for the filename of the created file.

nodata_value

The nodata value to be used.

stack_bands

Concatenate bands and arrange them in a single file.

group_by

A regex to group the input datasets, if consisting of multiple file. The first regex group is used for the grouping.

sort_by

A regex to select a portion of the filename to be used for sorting. The first regex group is used.

order

The order of the extracted item used in 'sort_by'. When the value extracted by `sort_by` is missing, then that file will be dropped.

output

Final adjustments to generate an output file. Add overviews, reproject to a common projection, etc.

options

Options to be passed to *gdal.Warp*. See https://gdal.org/python/osgeo.gdal-module.html#WarpOptions for details.

custom_preprocessor

A custom python function to be called.

path

The Python module path to the function to call.

args

A list of arguments to pass to the function.

kwargs

A dictionary of keyword arguments to pass to the function.

types

This mapping of product type identifier to step configuration allows to define specific step settings, even overriding the values from the defaults.

4.6 Sensitive variables

Since environment variables include credentials that are considered sensitive, avoiding their exposure inside `.env` files would be the right practice. In order to manage transmitting sensitive data securely into the respective containers, docker secrets with the values of these variables should be created. Currently, four variables have to be saved as docker secrets before deploying the swarm: `OS_PASSWORD`, `OS_PASSWORD_DOWNLOAD`, `DJANGO_PASSWORD` and `DJANGO_SECRET_KEY`.

Following docker secret for traefik basic authentication needs to be created too: `BASIC_AUTH_USERS_APIAUTH` - used for admin access to kibana and traefik. Access to the services for alternative clients not supporting main Shibboleth authentication entrypoints is configured by creating a local file BASIC_AUTH_USERS inside the cloned repository folder.

The secret and the pass file should both be text files containing a list of username:hashedpassword (MD5, SHA1, BCrypt) pairs.

Additionally, the configuration of the `sftp` image contains sensitive information, and therefore, is created using docker configs.

An example of creating configurations for sftp image using the following command :

```
printf "<user>:<password>:<UID>:<GID>" | docker config create sftp-users-<name> -
```

An example of creating `OS_PASSWORD` as secret using the following command :

```
printf "<password_value>" | docker secret create OS_PASSWORD -
```

An example of creating `BASIC_AUTH_USERS_APIAUTH` secret:

```
htpasswd -nb user1 3vYxfRqUx4H2ar3fsEOR95M30eNJne >> auth_list.txt
htpasswd -nb user2 YyuN9bYRvBUUU6COx7itWw5qyyARus >> auth_list.txt
docker secret create BASIC_AUTH_USERS_APIAUTH auth_list.txt
```

For configuration of the `shibauth` service, please consult a separate chapter *Access*.

The next section *Service Management* describes how an operator interacts with a deployed VS stack.

CHAPTER

FIVE

SERVICE MANAGEMENT

This section shows how a deployed VS stack can and should be interacted with.

5.1 Scaling

Scaling is a handy tool to ensure stable performance, even when dealing with higher usage on any service. For example, the preprocessor and registrar can be scaled to a higher replica count to enable a better throughput when ingesting data into the VS.

The following command scales the `renderer` service to 5 replicas:

```
docker service scale <stack-name>_renderer=5
```

A service can also be scaled to zero replicas, effectively disabling the service.

Warning: The `redis` and `database` should never be scaled (their replica count should remain 1) as this can lead to service disruptions and corrupted data.

5.2 Updating Images

Updating the service software is done using previously established tools. To update the service in question, it needs to be scaled to zero replicas. Then the new image can be pulled, and the service can be scaled back to its original value. This forces the start of the service from the newly fetched image. Another option to keep the service running during the upgrade procedure is to sequentially restart the individual instances of the services after pulling a newer image using a command:

```
docker service update --force <stack-name>_<service-name>
```

5.3 Updating configurations or environment files

Updating the service configurations or environment files used can not be done just by rescaling the impacted services to 0 and rerunning. The whole stack needs to be shut down using the command:

```
docker stack rm <stack-name>
```

A new deployment of the stack will use the updated configuration. The above mentioned process necessarily involves a certain service downtime between shutting down of the stack and new deployment.

5.4 Inspecting reports

Once a product is registered, a xml report containing wcs and wms getcapabilities of the registered product is generated and can be accessed by connecting to the *SFTP* service via the sftp protocol. In order to log into the logging folders through port 2222 (for `vhr18`, `emg` and `dem` have 2223 and 2224 respectively) on the hosting ip (e.g. localhost if you are running the dev stack) The following command can be used:

```
sftp -P 2222 <username>@<host>
```

this will direct the user into */home/<username>/data* sftp mounted directory which contains the 2 logging directories : *to/panda* and *from/fepd*

Note: The mounted directory that the user is directed into is *`/home/user`*, where *user* is the username, hence when changing the username in the *.conf* file, the *sftp* mounted volumes path in *docker-compose.<collection>.yml* must be changed respectively.

5.5 Inspecting logs in development

All service components are running inside docker containers and it is therefore possible to inspect the logs for anomalies via standard docker logs calls redirected for example to less command to allow paging through them.

```
docker logs <container-name>  2>&1 | less
```

In case that only one instance of a service is running on one node, the <container-name> can be returned by fetching the available containers of a service on that node with a command

```
docker logs $(docker ps -qf "name=<stack-name>_<service-name>")  2>&1 | less
```

It is possible to show logs of all containers belonging to a service from a master node, utilizing *docker service logs* command, but the resulting listing does not enforce sorting by time. Although logs of each task appear in the order they were inserted, logs of all tasks are outputted interleaved. To quickly check latest time-sorted logs from the service, sorting the entries by timestamp column, do:

```
docker service logs <stack-name>_<service-name> -t 2>&1 | sort -k 1 2>&1 | tail -n
↪<number-of-last-lines> 2>&1 | less
```

The docker service logs is intended as a quick way to view the latest log entries of all tasks of a service, but should not be used as a main way to collect these logs. For that, on production setup, an additional EFK (Elasticsearch, Fluentd, Kibana) stack is deployed.

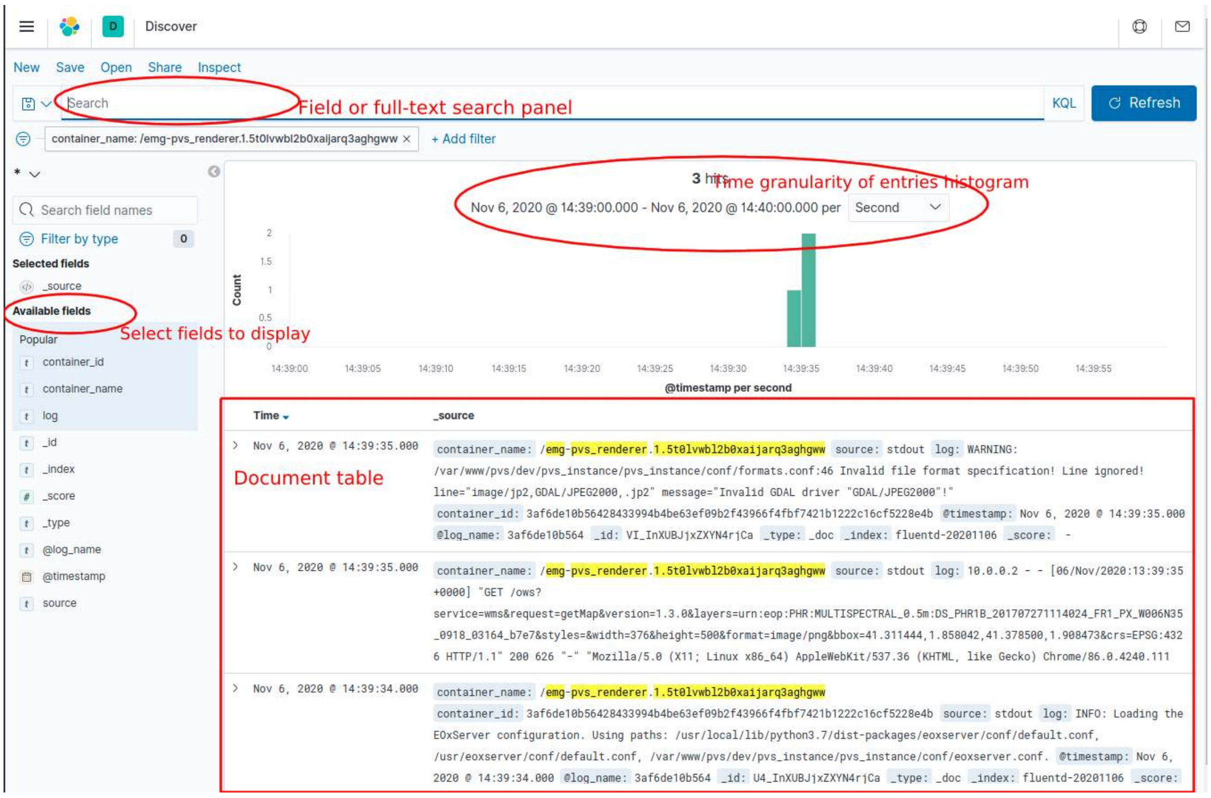
5.6 Inspecting logs in production

Fluentd is configured as main logging driver of the Docker daemon on Virtual machine level. Therefore for other services to run, Fluentd service must be running too. To access the logs, interactive and multi-purpose Kibana interface is available and exposed externally by traefik.

For simple listing of the filtered time-sorted logs as an equivalent to *docker service logs* command, a basic `Discover` app can be used. The main panel to interact with the logs is the `Search` bar, allowing filtered field-data and free-text searches, modyfing time range etc. The individual log results will then appear in the `Document table` panel in the bottom of the page.

For specific help with `Discover` panel, please consult Kibana official documentation

In order to select any other option from the Kibana toolkit, click the horizontal lines selection on the top left and pick a tool.



```
> Nov 6, 2020 @ 14:39:35.000   container_name: /emg-pvs_renderer.1.5t0lvwbl2b0xaijarq3aghgww source: stdout log: WARNING:
                               /var/www/pvs/dev/pvs_instance/pvs_instance/conf/formats.conf:46 Invalid file format specification! Line ignored!
                               line="image/jp2,GDAL/JPEG2000,.jp2" message="Invalid GDAL driver "GDAL/JPEG2000"!"
                               container_id: 3af6de10b56428433994b4be63ef09b2f43966f4fbf7421b1222c16cf5228e4b @timestamp: Nov 6, 2020 @ 14:39:35.000
                               @log_name: 3af6de10b564 _id: VI_InXUBJjxZXYN4rjCa _type: _doc _index: fluentd-20201106 _score: -

> Nov 6, 2020 @ 14:39:35.000   container_name: /emg-pvs_renderer.1.5t0lvwbl2b0xaijarq3aghgww source: stdout log: 10.0.0.2 - - [06/Nov/2020:13:39:35
                               +0000] "GET /ows?
                               service=wms&request=getMap&version=1.3.0&layers=urn:eop:PHR:MULTISPECTRAL_0.5m:DS_PHR1B_201707271114024_FR1_PX_W006N35
                               _0918_03164_b7e7&styles=&width=376&height=500&format=image/png&bbox=41.311444,1.858042,41.378500,1.908473&crs=EPSG:432
                               6 HTTP/1.1" 200 626 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4240.111

> Nov 6, 2020 @ 14:39:34.000   container_name: /emg-pvs_renderer.1.5t0lvwbl2b0xaijarq3aghgww
                               container_id: 3af6de10b56428433994b4be63ef09b2f43966f4fbf7421b1222c16cf5228e4b source: stdout log: INFO: Loading the
                               EOxServer configuration. Using paths: /usr/local/lib/python3.7/dist-packages/eoxserver/conf/default.conf,
                               /usr/eoxserver/conf/default.conf, /var/www/pvs/dev/pvs_instance/pvs_instance/conf/eoxserver.conf. @timestamp: Nov 6,
                               2020 @ 14:39:34.000 @log_name: 3af6de10b564 _id: U4_InXUBJjxZXYN4rjCa _type: _doc _index: fluentd-20201106 _score:
```

Figure 5.6.1: *Kibana discover panel*

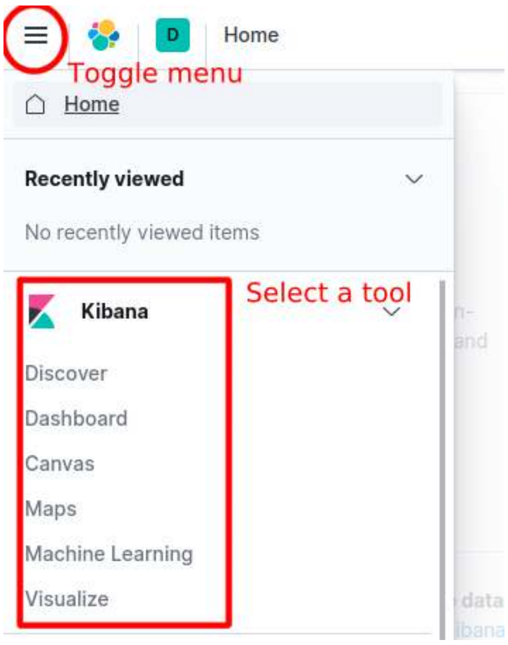


Figure 5.6.2: *Kibana menu*

Kibana also allows to aggregate log data based on a search query in two modes of operation: `Bucketing` and `Metrics` being applied on all buckets.

These aggregations then are used in `Visualisations` with various chart modes like vertical bar chart, horizontal line chart. Using saved searches improves the performance of the charts due to limiting the results list.

5.7 Increasing logging level

In default state, all components are configured to behave in production logging setup, where the amount of information contained in the logs is reduced. Different components contain different ways to increase the reported logging level for debugging purposes.

In order to increase logging level of **EOXServer** component, and therefore of each service, which depends on it, a *DEBUG* configuration option contained in file **$INSTALL_DIR/pvs_instance/settings.py** needs to be set to **True**. This setting needs to be applied on each node, where there is a running a service for which the *DEBUG* logging should be enabled, as it is stored in the respective docker volume <stack-name>_instance-data, which is created per node.

A restart of respective service for the change to be applied is also necessary. In order to change the DEBUG settings on an example of a renderer, do

```
docker exec -it $(docker ps -qf "name=<stack-name>_renderer") bash
cd ${INSTALL_DIR}/pvs_instance
sed -i 's/DEBUG = False/DEBUG = True/g' settings.py
```

In order to increase logging level of registrar and preprocessor services to *DEBUG*, the respective Python commands need to be run with an optional parameter **–debug**.

Ingestor service by default logs its messages in DEBUG mode.

The cache services internally uses a Mapcache software, which usually incorporates an Apache 2 HTTP Server. Due to that, logging level is shared throughout the whole service and is based on Apache *.conf* file, which is stored in $APACHE_CONF environment variable. To change the logging level, edit this file, by setting a **LogLevel debug** and then gracefully restart the Apache component (this way, the cache service itself will not restart and renew default configuration).

```
docker exec -it $(docker ps -qf "name=<stack-name>_cache") bash
sed -i 's/<\/VirtualHost>/    LogLevel debug\n<\/VirtualHost>/g' $APACHE_CONF
apachectl -k graceful
```

5.8 Cleaning up

Current configuration of the services does not have any log rotation set up, which means that service logs can grow significantly over time, if left not maintained and set on verbose logging levels. In order to delete logs older than 7 days from a single node, a following command can be run

```
journalctl --vacuum-time=7d
```

Additionally in order to delete older logs from docker containers present on a node, keeping only a certain number of newest rows, a following command can be run.

```
truncate -s <number rows to keep> $(docker inspect -f '{{.LogPath}}' $container 2> /
↪dev/null)
```

The final section *Data Ingestion* explains how to get data into the VS.

5.9 Database backup

The database can be backed up with the script below. The *STACK* and *BACKUP_PATH* variables can be changed depending on the stack and desired path of backup files

```
#!/bin/bash

# Variables to be changed
STACK="dem"
BACKUP_PATH="/path/to/backup/storage"

# Script variables
FILE_NAME="$(date +'%Y%m%d').sql.gz"
DB_SERVICE=""$STACK"_database"
DB_CONTAINER="$(docker ps -l -q -f name=^/$DB_SERVICE)"

echo "Backing up $STACK stack"
echo "Backup path: $BACKUP_PATH"
echo "Backup file: $FILE_NAME"
echo "Backup service: $DB_SERVICE"
echo "DB container id: $DB_CONTAINER"

echo "Backing up to /$FILE_NAME"
docker exec $DB_CONTAINER sh -c "pg_dump -U "$STACK"_user -d "$STACK"_db -f c > /
↪$FILE_NAME"
echo "Copying to $BACKUP_PATH"
docker cp $DB_CONTAINER:/$FILE_NAME $BACKUP_PATH
echo "Cleaning up"
docker exec $DB_CONTAINER sh -c "rm /$FILE_NAME"
```

To restore from a backed up file run the below script. Here the *STACK*, *DATE* and *BACKUP_PATH* can be changed. Note: Date for last backup must be in YYYYMMDD format

```
#!/bin/bash

# Variables to be changed
STACK="dem"
DATE="20210722"
BACKUP_PATH="/path/to/backups"

# Script variables
BACKUP_FILE="$BACKUP_PATH/$DATE.sql.gz"
UNCOMPRESSED_FILE="$BACKUP_PATH/$DATE.sql"
DB_SERVICE=""$STACK"_database"
DB_CONTAINER="$(docker ps -q -f name=$DB_SERVICE)"

echo "Restoring $STACK stack"
echo "Backup file: $BACKUP_FILE"
echo "Backup service: $DB_SERVICE"
echo "DB container id: $DB_CONTAINER"

echo "Unpacking $BACKUP_FILE"
gunzip $BACKUP_FILE
echo "Copying unpacked file"
docker cp $UNCOMPRESSED_FILE $DB_CONTAINER:/
echo "Restoring database"
docker exec $DB_CONTAINER sh -c "psql -U "$STACK"_user -d "$STACK"_db < /$DATE.sql"
```

(continues on next page)

(continued from previous page)

```
echo "Cleaning up"
docker exec $DB_CONTAINER sh -c "rm /$DATE.sql"
rm $UNCOMPRESSED_FILE
```

CHAPTER

SIX

DATA INGESTION

This section details the data ingestion and later management in the VS.

6.1 Redis Queues

The central synchronization component in the VS is the `redis` key-value store. It provides various queues, which the services are listening to. For operators it provides a high-level interface through which data products can be registered and managed.

Via the Redis, the ingestion can be triggered and observed. In order to eventually start the preprocessing of a product, its path on the configured object storage has to be pushed onto the `preprocess_queue`, as will be explained in detail in this chapter.

As the Redis store is not publicly accessible from outside of the stack. So to interact with it, the operator has to run a command from one of the services. Conveniently, the service running Redis also has the `redis-cli` tool installed that lets users interact with the store.

When doing one off commands, it is maybe more convenient to execute it on a running service. For this, the `docker ps` command can be used to select the identifier of the running docker container of the redis service.

```
container_id=$(docker ps -qf "name=<stack-name>_redis")
```

With this identifier, a command can be issued:

```
docker exec -it $container_id redis-cli ...
```

When performing more than one command, it can be simpler to open a shell on the service instead:

```
docker exec -it $container_id bash
```

As the container ID may change (for example when the replica is restarted) it is better to retrieve it for every command instead of relying on a variable:

```
docker exec -it $(docker ps -qf "name=<stack-name>_redis")
```

For the sake of brevity, the subsequent commands in this chapter can be used with either of the above techniques and will just print the final commands that are run inside the redis container.

Note: For the VS, only the `List` and `Set` Redis data types are really used.

`Sets` are an unordered collection of string elements. In the VS it is used to denote that an element is part of a particular group, e.g: being preprocessed, or having failed registration.

`Lists` are used as a task queue. It is possible to add items to either end of the queue, but by convention items are pushed on the "left" and popped from the "right" end of the list resulting in a first-in-first-out (FIFO) queue. It is entirely possible to push elements to the "right" end as-well, and an operator may want to do so in order to add an element to be processed as soon as possible instead of waiting before all other elements before it are processed.

The full list of available commands can be found for both Lists and Sets.

For a more concrete example the following command executes a `redis-cli lpush` command to add a new path of an object to preprocess on the `preprocess_queue`:

```
redis-cli lpush preprocess_queue "data25/OA/PL00/1.0/00/urn:eop:DOVE:MULTISPECTRAL_
↪4m:20180811_081455_1054_3be7/0001/PL00_DOV_MS_L3A_20180811T081455_20180811T081455_
↪TOU_1234_3be7.DIMA.tar"
```

Usually, with a preprocessor service running and no other items in the `preprocess_queue` this value will be immediately popped from the list and processed. For the sake of demonstration this command would print the contents of the `preprocess_queue`:

```
$ redis-cli lrange preprocess_queue 0 -1
data25/OA/PL00/1.0/00/urn:eop:DOVE:MULTISPECTRAL_4m:20180811_081455_1054_3be7/0001/
↪PL00_DOV_MS_L3A_20180811T081455_20180811T081455_TOU_1234_3be7.DIMA.tar
```

Now that the product is being preprocessed, it should be visible in the `preprocessing_set`. As the name indicates, this is using the `Set` datatype, thus requiring the `SMEMBERS` subcommand to list:

```
$ redis-cli smembers preprocessing_set 0 -1
data25/OA/PL00/1.0/00/urn:eop:DOVE:MULTISPECTRAL_4m:20180811_081455_1054_3be7/0001/
↪PL00_DOV_MS_L3A_20180811T081455_20180811T081455_TOU_1234_3be7.DIMA.tar
```

Once the preprocessing of the product is finished, the preprocessor will remove the currently worked on path from the `preprocessing_set` and add it either to the `preprocess-success_set` or the `preprocess-failure_set` depending on whether the processing succeeded or not. They can be inspected using the same `SMEMBERS` subcommand with one of set names as a parameter.

Additionally, upon success, the preprocessor places the same product path on the `register_queue`, where it can be inspected with the following command.

```
$ redis-cli lrange register_queue 0 -1
/data25/OA/PL00/1.0/00/urn:eop:DOVE:MULTISPECTRAL_4m:20180811_081455_1054_3be7/0001/
↪PL00_DOV_MS_L3A_20180811T081455_20180811T081455_TOU_1234_3be7.DIMA.tar
```

If an operator wants to trigger only the re-registration of a product without preprocessing the product path needs to be pushed to this queue:

```
redis-cli lpush register_queue "/data25/OA/PL00/1.0/00/urn:eop:DOVE:MULTISPECTRAL_
↪4m:20180811_081455_1054_3be7/0001/PL00_DOV_MS_L3A_20180811T081455_20180811T081455_
↪TOU_1234_3be7.DIMA.tar"
```

Very similar to the preprocessing, during the registration the product path is added to the `registering_set`, afterwards the path is placed to either the `register-success_set` or `register-failure_set`. Again, these queues or sets can be inspected by the `LRANGE` or `SMEMBERS` subcommands respectively.

6.1.1 Ingestor and sftp

Triggering preprocessing and registration via pushing to the redis queues is very convenient for single ingestion campaigns, but not optimal for continuous ingestion of new products from “live” sources. `Ingestor` service, together optionally with `sftp` service allow data ingestion to be initiated by external means.

`Ingestor` can work in two modes:

- Default: Exposing a simple / endpoint, and listening for `POST` requests containing `data` with either a Browse Report XML, Browse Report or a string with path to the object storage with product to be ingested. It then parses this information and internally puts it into configured redis queue (preprocess or register).

- Alternative: Listening for newly added Browse Report or Availability Report files on a configured path on a file system via `inotify`.

These Browse Report files need to be in an agreed XML schema to be correctly handled. `Sftp` service enables a secure access to a configured folder via sftp, while this folder can be mounted to other vs services. This way, `Ingestor` can listen for newly created files by the sftp access. If the filedaemon alternative mode should be used, `INOTIFY_WATCH_DIR` environment variable needs to be set and a `command` used in the docker-compose.<stack>.ops.yml for `ingestor` service needs to be set to `python3 filedaemon.py`:

```
ingestor:
  environment:
    REDIS_PREPROCESS_MD_QUEUE_KEY: "preprocess_queue" # to override md_queue (json)␣
↪and instead use (string)
  command:
    ["python3", "/filedaemon.py"]
```

6.2 Direct Data Management

Sometimes it is necessary to directly interact with the preprocessor or registrar. The following section shows what tasks on the preprocessor and registrar can be accomplished.

Warning: This approach is not recommended for everyday use, as it circumvents the Redis sets to track what products have been registered and where the registration failed.

6.2.1 Preprocessing

In this section all command examples are assumed to be run from within a running preprocessor container. To open a shell on a preprocessor, the following command can be used.

```
docker exec -it $(docker ps -qf "name=<stack-name>_preprocessor") bash
```

The preprocessor can be used in two modes. The first (and default mode when used as a service) is to be run as a daemon: it listens to the Redis queue for new items, which will be preprocessed one by one. The second mode is to run the preprocessor in a "one-off" mode: instead of pulling an item from the queue, it is passed as a command line argument, which is then processed normally.

```
preprocess \
    --config-file /preprocessor_config.yml \
    --validate \
    --use-dir /tmp \
    data25/OA/PL00/1.0/00/urn:eop:DOVE:MULTISPECTRAL_4m:20180811_081455_1054_3be7/
↪0001/PL00_DOV_MS_L3A_20180811T081455_20180811T081455_TOU_1234_3be7.DIMA.tar
```

In order to preprocess a ngEO Ingest Browse Report, an additonal `--browse-report` parameter needs to be added:

```
preprocess \
    --config-file /preprocessor_config.yml \
    --browse-report \
    --use-dir /tmp \
    browse_report_test1.json
```

In this "one-off" mode, the item will not be placed in the resulting set (`preprocessing_set`, `preprocess-success_set`, and `preprocess-failure_set`).

6.2.2 Registration Handling

For all intents and purposes in this section it is assumed, that the operator is logged into a shell on the `registrar` service. This can be achieved via the following command (assuming at least one registrar replica is running):

```
```

s of the shared registrar/renderer database can be managed using the registrars instance `manage.py` script. For brevity, the following bash alias is assumed:

```
alias manage.py='python3 /var/www/pvs/dev/pvs_instance/manage.py'
```

A collection is a grouping of earth observation products, accessible as a single entity via various service endpoints. Depending on the configuration, multiple collections are created when the service is set up. They can be listed using the `collection list` command.

New collections can be created using the `collection create` command. This can refer to a `Collection Type`, which will restrict the collection in terms of insertable products: only products of an allowed `Product Type` can be added. Detailed information about the available Collection management commands can be found in the CLI documentation.

Collections can be deleted, without affecting the contained products.

Warning: As some other services have fixed configurations and depend on specific collections, deleting said collections without a replacement can lead to configuration inconsistencies and ultimately service disruptions.

In certain scenarios it may be useful to add specific products to or exclude them from a collection. For this, the Product identifier needs to be known. To find out the Product identifier, either a query of the existing collection via OpenSearch or the CLI command `id list` can be used.

When the identifier is obtained, the following management command inserts a product into a collection:

```
manage.py collection insert <collection-id> <product-id>
```

Multiple products can be inserted in one pass by providing more than one identifier.

The reverse command excludes a product from a collection:

```
manage.py collection exclude <collection-id> <product-id>
```

Again, multiple products can be excluded in a single call.

6.2.3 Product Handling

Registration Products can be registered using the EOxServer CLI tools as well.

```
manage.py product register \
    --metadata-file data25 /OA/PL00/1.0/00/urn:eop:DOVE:MULTISPECTRAL_
↪4m:20180811_081455_1054_3be7/0001/PL00_DOV_MS_L3A_20180811T081455_
↪20180811T081455_TOU_1234_3be7.DIMA.tar/metadata.xml \
    --print-identifier \
    --type PL00
```

The identifier of the newly registered product is printed to the console and can be used to put it into a collection. Additionally, it is necessary to add a coverage to it, which can be registered like:

```
manage.py coverage register \
    -d data25 /OA/PL00/1.0/00/urn:eop:DOVE:MULTISPECTRAL_4m:20180811_081455_1054_
↪3be7/0001/PL00_DOV_MS_L3A_20180811T081455_20180811T081455_TOU_1234_3be7.DIMA.
↪tar/some.tif \
```

(continues on next page)

(continued from previous page)

```
    -m data25 /OA/PL00/1.0/00/urn:eop:DOVE:MULTISPECTRAL_4m:20180811_081455_1054_
↪3be7/0001/PL00_DOV_MS_L3A_20180811T081455_20180811T081455_TOU_1234_3be7.DIMA.
↪tar/metadata.xml \
    --identifier "${product_id}_coverage" \
    --type RGBNir
```

Deregistration Products and coverages need to be derigestered when no longer in use. A product can be deregistered using its identifier:

```
manage.py product deregister "${product_id}"
```

The contained coverage must also be deregistered manually:

```
manage.py coverage deregister "${product_id}_coverage"
```

6.2.4 Preprocessing vs registration

The preprocessing step aims to ensure that cloud optimized GeoTIFF (COG) files are created in order to significantly speed up the viewing of large amount of data in lower zoom levels. There are several cases, where such preprocessing is not necessary or wanted.

- If data are already in COGs and in favorable projection, which will be presented to the user for most of the times, direct registration should be used. This means, paths to individual products will be pushed directly to the register-queue.
- Also for cases, where preprocessing step would take too much time, direct registration allowing access to the metadata and catalog functions, while justifying slower rendering times can be preferred.

6.2.5 Monitoring ingestion

Monitoring ingestion can be done on production system easily via Kibana using its query language KQL. Kibana in *Discover* mode shows time histogram of individual entries, which makes it easy to visually infer the ingestion progress in time. These queries can be saved for later use and more importantly to set up alerts and statistics on these saved queries.

In order to watch for successful registrations or preprocessing campaigns, simply search for

```
"<stack-name>_registrar" AND "Successfully"
```

Example of such a query, filtering data for one day into the past from now:

```
https://kibana.pdas.prism.eox.at/app/discover#/?_g=(filters:!(),
↪refreshInterval:(pause:!t,value:0),time:(from:now-1d,to:now))&_a=(columns:!(log,
↪container_name),filters:!(),index:'57007c50-f270-11ea-8728-ab85b3e61ad6',
↪interval:auto,query:(language:kuery,query:'"emg-pdas_registrar"%20AND%20
↪"Successfully"'),sort:!())
```

stack-name, *kibana-url* and *elasticsearch-index-id* needs to be substituted with valid values.

For failures in preprocessing following search query can be used:

```
"<stack-name>_preprocessor" AND "ERROR" AND NOT "Target.replace"
```

Preprocessor and registrar by default run in mode, where they skip already registered/preprocessed products. This KQL query does not list errors like “file is already in target storage”.

For checking of the status of individual product ingestion (for example to find out why it failed), it can be searched for its *path* and then list *surrounding documents* and filter them by *docker container name*. An example query would be:

```
"emg-pdas_registrar" AND "data26/0000171398/PL00_DOV_MS_L3A_20190313T075450_
↪20190313T075450_PLA_000000_D5E9.DIR.tar"
```

Then click on an arrow on left border of the individual log message (document) to display more details -> *View surrounding documents* link appears, which lists other logs close in time to this one (default 5 before and 5 after).

It is also advisable to filter the logs per container (showing only logs from that registrar/preprocessor container, that has selected surrounding documents).

Querying for *ingestor* logs allows to see if while using the ingestor push ingestion mode, the XML was parsed correctly.

```
"<stack-name>_ingestor"
```

Next chapter *Access* describes used authorization and authentication concepts and lines out how the external access to individual components and service as such is configured.

CHAPTER

SEVEN

ACCESS

This chapter describes general concepts of how external access to each component is provided and how authentication and authorization layer based on Shibboleth SP3. is configured.

7.1 General overview

Each individual docker **stack** has its own internal network `intnet` where services can communicate between each other. This network is not exposed to the public and provides most of the necessary communication. Additionally external user access to some services (client, renderer, cache) is provided via external network `extnet` and reverse-proxy (traefik) with load balancer.

These services can have a set of authentication and authorization rules applied both on traefik level and Shibboleth SP level. Kibana and Traefik dashboard are also accessible externally, but through a different set of default credentials.

7.2 Routing with traefik

`Reverse-proxy` service in base stack provides central access endpoint to the VS. It exposes ports 80 and 443 for HTTP and HTTPS access in the host mode. Configuration of the reverse-proxy is done on three places.

First two are static and dynamic configuration files `traefik.yml` and `traefik-dynamic.yml`. Static configuration sets up connections to providers and define the entrypoints that Traefik will listen to. Dynamic configuration defines how the requests are handled. This configuration can change and is seamlessly hot-reloaded, without any request interruption or connection loss. Third part are docker `labels` on individual services which Traefik provides access to, for which an update requires removing and re-creating the stack.

For example following configuration snippet enables access to certain paths of `renderer` service under a given hostname. It also sets externally set basic authentication and other rules via `@file` identifier, which references global configurations from `traefik-dynamic.yml`.

```
renderer:
  deploy:
    labels:
      # router for basic auth based access (https)
      - "traefik.http.routers.vhr18-renderer.rule=Host(`vhr18.pdas.prism.eox.at`) &&␣
↪PathPrefix(`/ows`, `/opensearch`, `/admin`)"
      - "traefik.http.routers.vhr18-renderer.middlewares=auth@file,compress@file,
↪cors@file"
      - "traefik.http.routers.vhr18-renderer.tls=true"
      - "traefik.http.routers.vhr18-renderer.tls.certresolver=default"
      - "traefik.http.routers.vhr18-renderer.entrypoints=https"
      # general rules
      - "traefik.http.services.vhr18-renderer.loadbalancer.sticky=false"
      - "traefik.http.services.vhr18-renderer.loadbalancer.server.port=80"
```

(continues on next page)

(continued from previous page)

```
      - "traefik.docker.network=vhr18-extnet"
      - "traefik.docker.lbswarm=true"
      - "traefik.enable=true"
```

An example of such auth@file configuration from `traefik-dynamic.yml` would be following snippet, where `BASIC_AUTH_USERS_AUTH` is referencing a docker secret configured earlier:

```
http:
  middlewares:
    auth:
      basicAuth:
        realm: "PRISM View Server (PVS)"
        usersFile: "/run/secrets/BASIC_AUTH_USERS_AUTH"
```

Updating of *usersFile* content needs a restart of reverse-proxy service afterwards. Unsecured HTTP access is configured to be redirected to the HTTPS endpoint. Inside the swarm among the services, only HTTP is used internally.

7.3 Authentication and Authorization

Authentication of access to external parts of VS is made up of two options:

- Traefik provided basic authentication - configured as `file@auth` and `file@apiAuth`

Here, access on such endpoint requires basic authentication credentials (username, password) to be inserted, while web browsers are usually prompted for input. After inserting valid credentials, access is granted.

- Shibboleth Service Provider 3 + Apache 2 instance, to which requests are forwarded by Traefik ForwardAuth middleware.

Middleware delegates the authentication to Shibboleth. If Shibboleth response code is 2XX, access is granted and the original request is performed. Otherwise, the error response from Shibboleth is returned.

In order to authenticate with Shibboleth, a user must log in with valid credentials on the side of Identity Provider (IdP), if doing so, the IdP informs the SP about successful login, accompanied by relevant user attributes and a session is created for the user. SP then saves the information about a created session into a cookie and based on user attributes can authorize access to the services. If the user was already logged in, he is automatically offered the requested resource.

Currently setting individual authorization rules on a `Collection` (docker stack) and `Service` (docker service) level is possible. It is yet not clearly possible to separate viewing and download functionality, as both of these parts are handled by `renderer` service.

7.4 Configuration

For correct configuration of Shibboleth SP3 on a new stack, several steps need to be done. Most of these configurations are usually done in the *Initialization* step using `pvs_starter` tool. Still, it is advised to check following steps, understand them and change if necessary. Briefly summarized, SP and IdP need to exchange metadata and certificates to trust each other, SP needs to know which attributes the IdP will be sending about the logged-in user and respective access-control rules are configured based on those attributes. Most of the configurations are done via docker configs defined in the docker compose files.

- Create a pair of key, certificate using attached Shibboleth utility `config/shibboleth/keygen.sh` in the cloned `vs` repository and save them as respective docker secrets.

```
SP_URL="https://emg.pass.copernicus.eu" # service initial access point made␣
↪accessible by traefik
./config/shibboleth/keygen.sh -h $SPURL -y 20 -e https://$SP_URL/shibboleth -n sp-
↪signing -f
docker secret create <stack-name>_SHIB_CERT sp-signing-cert.pem
docker secret create <stack-name>_SHIB_KEY sp-signing-key.pem
```

- Get IDP metadata and save it as a docker config. Also save the entityID of the IdP for further use in filling the `shibboleth2.xml` template.

```
docker config create idp-metadata idp-metadata-received.xml
```

- Configure Apache ServerName used inside the `shibauth` service by modifying `APACHE_SERVERNAME` environment variable of corresponding `shibauth` service in `docker-compose.<stack>.ops.yml`. This URL should resolve to the actual service URL.
- Configure SP and IdP EntityIDs used inside the `shibauth` service by modifying `SPEntityID` and `IDPEntityID` environment variables of corresponding `shibauth` service in `docker-compose.<stack>.ops.yml`. `SPEntityID` can be chosen in any way, IDPEntityID should be extracted from received IDP metadata.
- Deploy your shibauth service and exchange your SP metadata with the IdP provider and have them register your SP. Necessary metadata can be downloaded from url `<service-url>/Shibboleth.sso/Metadata`.
- Get information about attributes provided by IdP and update `config/shibboleth/attribute-map.xml` by adding individual entries mapping `name` provided by IdP to `id` used by SP internally. Example configuration:

```
<Attributes xmlns="urn:mace:shibboleth:2.0:attribute-map" xmlns:xsi="http://www.w3.
↪org/2001/XMLSchema-instance">
  <Attribute name="urn:mace:dir:attribute-def:signed-terms" id="signed_terms_and_
↪conditions"/>
  <Attribute name="urn:mace:dir:attribute-def:primary-group" id="user_group_primary"/>
</Attributes>
```

- Create custom access rules based on these attributes and map these access controls to different internal Apache routes to which Traefik ForwardAuth middleware will point. Access rules are created in `config/shibboleth/<stack-name>-ac.xml` and `config/shibboleth/<stack-name>-ac-cache.xml`.

Example of external Access control rules configuration:

```
<AccessControl type="edu.internet2.middleware.shibboleth.sp.provider.XMLAccessControl
↪">
  <AND>
    <RuleRegex require="signed_terms_and_conditions">.+</RuleRegex>
    <Rule require="user_group_primary">
     Privileged_Access Public_Access
    </Rule>
  </AND>
</AccessControl>
```

- Check configured link between Apache configuration for `shibauth` service, access rules, Traefik ForwardAuth middleware and per-service Traefik labels. Following simplified examples show the links in more detail:

`APACHE_SERVERNAME` environment variable needs to be set and same as the hostname, that Traefik will be serving as main entry point. Part of docker compose of shibauth service in `docker-compose.emg.ops.yml`:

```
services:
  shibauth:
```

(continues on next page)

(continued from previous page)

```
    environment:
      APACHE_SERVERNAME: "https://emg.pass.copernicus.eu:443"
    deploy:
      labels:
        - "traefik.http.routers.shibauth.rule=Host(`emg.pass.copernicus.eu`) &&␣
↪PathPrefix(`/Shibboleth.sso`)"
        ...
```

Relevant Apache configuration in `config/shibboleth/shib-apache.conf`, enabling Shibboleth authentication and authorization of the renderer service on the `/secure` endpoint.

```
# Internally redirected to here. Rewrite for proper relaystate in shib
<Location /secure>
  RewriteEngine On
  RewriteCond %{HTTP:X-Forwarded-Uri} ^(.*)$ [NC]
  RewriteRule ^.*$ %1 [PT]
</Location>
<LocationMatch "^/(admin|ows|opensearch)">
  RewriteEngine On
  AuthType shibboleth
  ShibRequestSetting requireSession 1
  Require shib-plugin /etc/shibboleth/pass-ac.xml
  RewriteRule ^.*$ - [R=200]
</LocationMatch>
```

Part of Traefik ForwardAuth middleware configuration from `docker-compose.emg.ops.yml`, defining the internal address pointing to the `shibauth-emg` service and `/secure` endpoint in it:

```
renderer:
  deploy:
    labels:
      - "traefik.http.middlewares.emg-renderer-shib-fa.forwardauth.address=http://
↪shibauth-emg/secure"
      - "traefik.http.routers.emg-renderer-shib.middlewares=emg-renderer-shib-fa,
↪compress@file,cors@file"
```