View Server - Operator Guide

EOX IT Services GmbH

Jul 31, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

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

View Server (VS) is a Docker-based software and all of its components are distributed and executed in the context of Docker images, containers, and Helm charts. Basic knowledge of Docker and either Docker Swarm or Helm and Kubernetes is a prerequisite.

1.1 Components

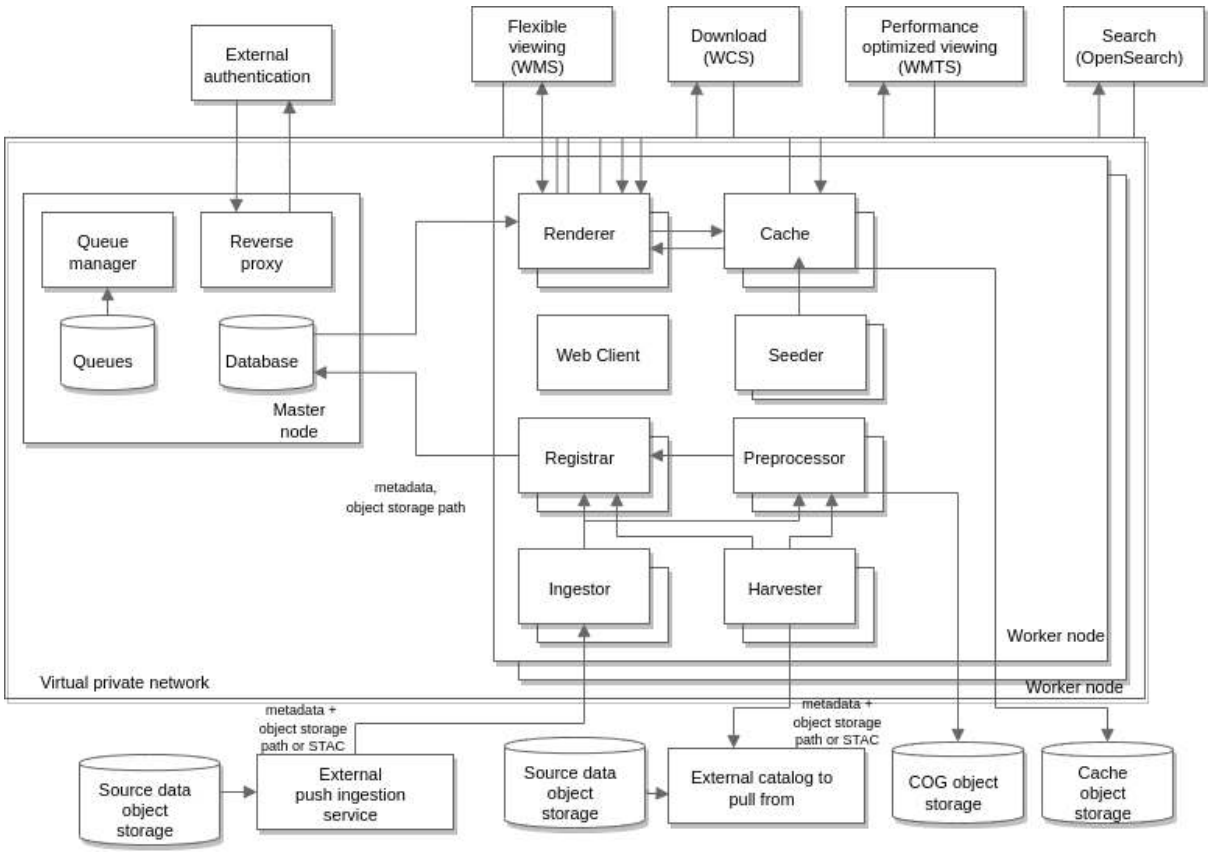

Figure 1.1.1: *View Server Architecture*

Individual components of VS are 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.

VS software brings together all the components in their latest versions in the default *vs-deployment* helm chart which can be found in EOX public charts repository or when needed to be cloned locally from EOX Gitlab repository.

VS consists of the following service components (with their respective Docker image in parentheses):

- Web Client (registry.gitlab.eox.at/vs/vs/client)
- Cache (registry.gitlab.eox.at/vs/vs/cache)
- Renderer (registry.gitlab.eox.at/vs/vs/core)
- Registrar (registry.gitlab.eox.at/vs/vs/core)
- Seeder (registry.gitlab.eox.at/vs/vs/seeder)
- Preprocessor v2 (registry.gitlab.eox.at/vs/preprocessor)
- Preprocessor with Mapchete (registry.gitlab.eox.at/vs/vs/preprocessor)
- Ingestor (registry.gitlab.eox.at/vs/ingestor)
- Harvester (registry.gitlab.eox.at/vs/vs/harvester)
- Scheduler (registry.gitlab.eox.at/vs/vs/scheduler)
- Database (postgis/postgis)
- Queue Manager (redis)

Additional services deployed with View Server in different contexts are:

- Reverse proxy (traefik)
- Log collector (registry.gitlab.eox.at/esa/prism/fluentd)
- Kibana (kibana)
- Elasticsearch (elasticsearch)
- Shibboleth SP3 (registry.gitlab.eox.at/esa/prism/shibauth) - only in PRISM
- SFTP (registry.gitlab.eox.at/esa/prism/sftp) - only in PRISM
- Dem-App (registry.gitlab.eox.at/esa/prism/dem-app) - only for DEM collections in PRISM
- Terrain-server (geodata/cesium-terrain-server) - only for DEM collections in PRISM

These components are expected to be managed together in a Docker Swarm via Docker Compose configuration files and in Kubernetes as Helm releases with configuration Values.

1.2 Initialization and Setup

The section *Operating on Kubernetes* describes how to customize the configuration file *values.yaml* and deploy VS on Kubernetes.

The section *Initialization Swarm* describes additional steps necessary to deploy a Docker Swarm stack using the configuration file *values.yaml*.

You can choose not to read through the setup and configuration and rather focus on a cookbook of steps for creating a complete configuration for a single data collection in section *Create a new collection step by step*

To search for concrete management/operations task after having already setup a running instance, no matter the platform, please continue directly to a condensed chapter listing the individual commands: *Operations and management*.

1.3 Migration Guide

View Server adheres to Semantic Versioning in format MAJOR.MINOR.PATCH and upgrading it by a PATCH version increase (bugfixes) is possible without special considerations by usual deployment upgrade methods (depends if on Docker Swarm or Kubernetes).

For instructions regarding MAJOR or MINOR version increase, please refer to the View Server breaking changes and any extra project specific Migration guides.

CHAPTER

TWO

OPERATING ON KUBERNETES

VS software is shipped as a set of Helm Charts, where each component has a meaningful default set of configuration values. These set values need to be created for each deployment of VS.

The important part of the initialization is the configuration. The `values.yaml` file is structured in YAML as detailed below. It can contain sections for each component, as well as `global` accessible by all individual components.

Full deployment configuration schema enabling strict validation of configuration against it will be released soon.

The following section contains just an extract of available keys and example values for the vs-deployment chart. To find out all possible configurations, please refer to the *Helm configuration reference*.

To go straight to creating your first earth observation data collection in VS, follow section *Create a new collection step by step*.

2.1 Global configurations

Values under `global` key contain mainly parameters that more than 1 component need to set up their behavior. Examples are: Database configuration, collections, product types and layers.

2.1.1 database and django

```
global:
  env:
    DJANGO_MAIL: office@eox.at
    DJANGO_PASSWORD: 7xtMd62&bY#I
    DJANGO_USER: vs_admin
    DB_NAME: vs_db
    DB_PORT: "5432"
    DB_PW:  Go-J_eOUvj2k
    DB_USER: vs_user
```

2.1.2 collections

In the `collections` section, the collections are set up and it is defined which products based on `product_type` will be inserted into them. The `product_types` must list types defined in the `product_types` section, `coverage_types` allowed for a product_type must be a subset of those configured for the whole collection. More information about main EOxServer models.

```
global:
  collections:
    COLLECTION:
```

(continues on next page)

(continued from previous page)

```
      product_types:
        - PL00
      coverage_types:
        - int16_grayscale
```

2.1.3 productTypes

This section defines `product_type` related information. It is a list of possible product types where each entity defines `filters` to register new products into correct `product_type` matching against STAC Item properties (metadata), as well as which `browses` renderings will be generated and `coverages` configuring the mapping of different `named STAC assets` to `coverage_types`, `defaultBrowse` selects one of the existing `browses` and sets it as a default rendering. `Collections` key specifies the names of multiple collections that will a new product be ingested into.

`masks` registration is not yet fully implemented in View Server 2.

More information about main EOxServer models.

```
global:
  productTypes:
    - name: HRA_MS4_1C
      defaultBrowse: TRUE_COLOR
      filter:
        product_type: HRA_MS4_1C
      collections:
        - Deimos-HRA_MS4_1C
      coverages:
        RGBNir:
          assets:
            - ms
        Pan:
          assets:
            - pan
      masks:
        - name: validity
          validity: true
      browses:
        TRUE_COLOR:
          asset: browse # optional name of asset to register as Browse
          red:
            expression: red
            range: [0, 1000]
            nodata: 0
          green:
            expression: green
            range: [0, 1000]
            nodata: 0
          blue:
            expression: blue
            range: [0, 1000]
            nodata: 0
        FALSE_COLOR:
          red:
            expression: nir
            range: [0, 1800]
            nodata: 0
```

(continues on next page)

(continued from previous page)

```
          green:
            expression: red
            range: [0, 1000]
            nodata: 0
          blue:
            expression: green
            range: [0, 1000]
            nodata: 0
        PAN:
          grey:
            expression: pan
            range: [0, 1600]
            nodata: 0
        NDVI:
          grey:
            expression: (nir-red)/(nir+red)
            range: [-1, 1]
```

2.1.4 coverageTypes

Allows to define a new `coverage_type` when not contained in the list of predefined ones. By default View Server contains the following `coverage_types`:

- Sentinel 2 data - `coverage_type` named S2_RGBNir:
- Other common coverage types

More information about main EOxServer models.

```
global:
  coverageTypes:
    - data_type: "Uint16"
      name: "BGR"
      bands:
        - definition: "http://www.opengis.net/def/property/OGC/0/Radiance"
          description: "Blue Channel"
          gdal_interpretation: "BlueBand"
          identifier: "blue"
          name: "blue"
          nil_values:
            - reason: "http://www.opengis.net/def/nil/OGC/0/unknown"
              value: 0
          uom: "W.m-2.Sr-1"
          significant_figures: 5
          allowed_value_ranges:
            -
              - 0
              - 65535
        - definition: "http://www.opengis.net/def/property/OGC/0/Radiance"
          description: "Red Channel"
          gdal_interpretation: "RedBand"
          identifier: "red"
          name: "red"
          nil_values:
            - reason: "http://www.opengis.net/def/nil/OGC/0/unknown"
              value: 0
```

(continues on next page)

(continued from previous page)

```
          uom: "W.m-2.Sr-1"
          significant_figures: 5
          allowed_value_ranges:
            -
              - 0
              - 65535
        - definition: "http://www.opengis.net/def/property/OGC/0/Radiance"
          description: "Green Channel"
          gdal_interpretation: "GreenBand"
          identifier: "green"
          name: "green"
          nil_values:
            - reason: "http://www.opengis.net/def/nil/OGC/0/unknown"
              value: 0
          uom: "W.m-2.Sr-1"
          significant_figures: 5
          allowed_value_ranges:
            -
              - 0
              - 65535
```

2.1.5 storage

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

The `source` storage defines the locations from which the original files will be downloaded to be preprocessed. Preprocessed images and metadata will then be uploaded to the `data` storage, which is also used by `registrar` during registration. 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`, `s3` or `local`.

These storage definitions will be used in the appropriate sections.

```
global:
  source:
    type: swift
    username:
    password:
    project_name:
    project_id:
    region_name:
    auth_url:
    user_domain_name:
    user_domain_id:
    project_domain_name:
    project_domain_id:
  data:
    public:
      type: swift
      ...
  cache:
    type: swift
    ...
```

2.1.6 layers

This section defines how the layers shall be cached and their configuration in the client.

There is a difference between the concept of `parentLayers` and `subLayers`.

If `layer.parentLayer` value is equal to `layer.id`, all of its properties and values are considered as a full layer for client configurations.

If `layer.parentLayer` and `layer.id` are not equal, a new cache tileset is created with the given `id` and `grids` definitions. In the client, such `subLayer` is represented only as a Display option of a parentLayer.

The `subLayer` definitions correspond to the defined `browses` from `product_type` values. Each WMTS `subLayer` tileset created in the cache references the WMS layer of a collection in the renderer with the same name. The `subLayer.id` should therefore be composed in the following manner: `collection.name__browse.name`. The two underscores is a default separator and a configurable value.

Full configuration schema of client - search for layers.

```yaml
global:
  layers:
    - id: VHR_IMAGE_2018_Level_1
      title: VHR IMAGE 2018 Level 1
      displayColor: "#eb3700"
      parentLayer: VHR_IMAGE_2018_Level_1
      maxZoom: 18
      visible: false
      grids: &defaultGridOptions
        - name: WGS84
          zoom: 16
      search: &defaultSearch
        parameters:
          - type: "eo:cloudCover"
            title: "Cloud Coverage in percent"
            name: "Cloud Coverage"
            max: 100
            min: 0
            range: true
          - type: "geo:uid"
            title: "Product ID"
            privileged: true
  - id: VHR_IMAGE_2018_Level_1__TRUE_COLOR
    title: VHR Image 2018 Level 1 True color
    parentLayer: VHR_IMAGE_2018_Level_1
    grids: *defaultGridOptions
  - id: VHR_IMAGE_2018_Level_1__NDVI
    title: VHR Image 2018 Level 1 NDVI
    parentLayer: VHR_IMAGE_2018_Level_1
    style: earth
    grids: *defaultGridOptions
```

2.1.7 overlayLayers

This section defines `overlayLayers` definitions in client and cache. The following example configures a pre-seeded full coverage mosaic layer with limited European extent served as an overlay. Full configuration schema of client - search for overlayLayers.

```
global:
  overlayLayers:
    - id: VHR_IMAGE_2018_Level_3__outlines
      title: VHR Image 2018 Level_3 outlines
      description: "WMS rendering of Level 3 product footprints for current time␣
↪range."
    - id: VHR_IMAGE_2018_Level_3__masked_validity__Full
      title: VHR Image 2018 Level 3 True Color with masked validity Full Coverage
      protocol: WMTS
      urls: baseUrlsWMTS
      synchronizeTime: false
      source: "VHR_IMAGE_2018_Level_3__masked_validity"
      description: "<p>Pre-seeded Full coverage mosaic layer of VHR_IMAGE_2018 Level␣
↪3 products with their validity masks applied to masked out the final True Color␣
↪rendering. Products composing the rendered tiles were sorted by time, placing␣
↪newest products on top.</p><p>This mosaic does not have any search or time␣
↪dimension functionality enabled."
      grids: &defaultFullGridOptions
        - name: WGS84
          zoom: 16
          restricted_extent: "-24.7 27.5 45 71.3"
```

2.1.8 ingress

Global definition of Kubernetes ingress controller which many services can take their URL access patterns from. Optional.

```
global:
  ingress:
    enabled: true
    hosts:
      - host: collection.remoteurl.com
    tls:
      - hosts:
          - collection.remoteurl.com
        secretName: secret
```

2.2 Component specific

2.2.1 preprocessor-v2

Here, the preprocessing can be configured in detail. Example of a preprocessing configuration with defaults and a special configuration for a single `product_type`:

```
preprocessor-v2:
  config:
    type_extractor:
      xpath:
```

(continues on next page)

(continued from previous page)

```
        - /gsc:report/gsc:opt_metadata/gml:metaDataProperty/
↪gsc:EarthObservationMetaData/eop:productType/text()
        - /gsc:report/gsc:sar_metadata/gml:metaDataProperty/
↪gsc:EarthObservationMetaData/eop:productType/text()
    level_extractor:
      xpath: ''
    metadata_glob: "*GSC*.xml"
    stac_output: true
    preprocessing:
      defaults:
        stac_item_structure:
          statistics:
            compute_statistics: true
            stats_approx: 2
          assets:
            pan: &cog_stac_asset
              description: 'Product image converted into a COG'
              title: 'Preprocessed image'
              media_type: 'image/tiff; application=geotiff; profile=cloud-optimized'
              roles:
                - data
              globs:
                - '*.tif'
            ms: *cog_stac_asset
            gsc_metadata: &gsc_metadata_stac_asset
              globs:
                - '*.xml'
              description: 'GSC metadata file from source archive'
              title: 'GSC Metadata file'
              media_type: 'application/xml'
              roles:
                - metadata
        move_files: true
        nested: true
        output:
          options: &default_output_options
            format: COG
            dstSRS: 'EPSG:4326'
            dstNodata: 0
            multithread: True
            warpMemoryLimit: 3000
            creationOptions:
              - BLOCKSIZE=512
              - COMPRESS=DEFLATE
              - NUM_THREADS=8
              - BIGTIFF=YES
              - OVERVIEWS=AUTO
              - PREDICTOR=YES
      types:
        SKY_CBU_3A:
          data_file_globs:
            - "*analytic_clip.tif"
            - "*analytic.tif"
            - "*panchromatic_clip.tif"
            - "*panchromatic.tif"
          output:
```

(continues on next page)

(continued from previous page)

```
            group_by: "(.*)"
            options: *default_output_options
          stac_item_structure:
            statistics:
              compute_statistics: true
              stats_approx: 2
              force_histogram_min_value: 2
            assets:
              pan:
                <<: *cog_stac_asset
                globs:
                  - '*_panchromatic*'
              ms:
                <<: *cog_stac_asset
                globs:
                  - '*_analytic*'
              gsc_metadata: *gsc_metadata_stac_asset
```

2.2.2 client

This section contains other configurations to the client other than layer definitions. Those are referenced under the `layers` key. Full configuration schema of client.

```
client:
  config:
    eoxserverDownloadEnabled: true
    leftPanelTabIndex: 0
    timeDomain:
      - "2010-01-01T00:00:00Z"
      - today
    displayTimeDomain:
      - "2017-01-01T00:00:00Z"
      - "2019-12-31T23:59:59Z"
    selectedTimeDomain:
      - "2018-08-01T00:00:00Z"
      - "2018-08-31T23:59:59Z"
    maxZoom: 17
    displayInterval: P1096D
```

2.2.3 registrar

This section defines registrar-specific configurations, for setting up specific registration routes:

```
registrar:
  config:
    routes:
      collections:
        path: registrar.route.stac.CollectionRoute
        queue: register-collections
        backends:
          - path: registrar.backend.eoxserver.CollectionBackend
          - path: registrar_pycsw.backend.CollectionBackend
```

2.2.4 harvester

This section configures the `harvester` service, filtering capabilities or to which queue should it push the harvested results.

```
harvester:
  config:
    redis:
      host: redis # docker swarm only, otherwise do not override default
    harvesters:
      Deimos-HRA_MS4_1C:
        filter:
          eq:
            - property: "oads:product_type"
            - HRA_MS4_1C
        resource:
          type: OADS
          oads:
            url: https://tpm-ds.eo.esa.int/oads/meta/Kompsat2/index/
            use_oads_ext: true
        output: queue
        queue: register_queue
        postprocessors:
          - type: builtin
            process: static
            kwargs:
              values:
                properties:
                  collection: Deimos-HRA_MS4_1C
```

2.2.5 preprocessor

This section configures the `preprocessor` - Mapchete enabled preprocessor. Each config in `configs` targets a specific set of products based on metadata value *collection*.

```
preprocessor:
  replicaCount: 1
  limits:
    cpu: 2
    memory: 6Gi
  requests:
    cpu: 0.1
    memory: 1Gi
  config:
    filesystems:
      s3:
        type: s3
        s3:
          access_key_id: access
          secret_access_key: key
          region: eu

    processors:
      p1:
        type: local
        local:
```

(continues on next page)

(continued from previous page)

```
        process: preprocessor.processes.local.browse_to_geotiff
  paths:
    output_path: s3://
  collections:
    SPOT6-7:
      filesystems:
        target: s3
      data:
        - input:
            type: http
            http:
              asset_map:
                - key: band_1
                  band: browse
          output:
            path: output_path
            asset: data
            processors:
              - p1
```

2.3 Deploying using Helm

It is generally expected that a user will deploy the VS helm chart into the Kubernetes cluster. This can be done via the Flux Helm Operator, which takes care of the installation of helm charts as well as applying subsequent changes to the configuration automatically.

However, it is also possible to deploy manually using the *helm* command line tool directly:

```
helm install -f values.yaml vs chart-location
```

This will install VS with the configuration specified in `values.yaml`. To apply changes to `values.yaml`, the following command can be used:

```
helm upgrade -f values.yaml vs chart-location
```

Finally *vs* can also be uninstalled using the following command:

```
helm uninstall vs
```

2.4 Helm configuration reference

In this section variables for a helm deployment will be outlined starting with the main values file:

```
global:
  env:
  storage:
    data: {}
    source: {}
    cache:
      type: local
  collections: {}
  productTypes: []
  defaultLayer:
```

(continues on next page)

(continued from previous page)

```
  layers: []
  overlayLayers: []
  coverageTypes: []
  metadata: {}
database: {}
redis: {}
client: {}
cache: {}
renderer: {}
registrar: {}
harvester: {}
scheduler: {}
seeder: {}
preprocessor: {}
```

2.4.1 Global Configuration

Environment variables - `env`

Environment variables noted in other sections are added to this object as `key:value` pairs e.g.

```
global:
  env:
    GDAL_PAM_ENABLED: "NO"
```

Any environment variable added in `global.env` will get passed to `each` service of VS.

Note: Following `global.env` variables are mandatory to be set this way for docker swarm deployment in the `values.yaml`. These variables have their default values set for k8s deployment exclusively.

Any set of `values.yaml` for docker swarm deployments should include the following values:

```
global:
  ingress:
    tls: false
  env:
    DB_HOST: "database"
    RENDERER_HOST: "database"
```

Storage configuration - `storage`

The storage section handles all data storage-related configuration

`data` key, value mapping in form `name:{config}` for registrar and preprocessor services. There may exist multiple `name:{config}` mappings

for swift storage:

`type: swift`

`username` - service username

`password` - service password

`project_name` - name of project

`project_id` - id of project

`region_name` - name of region

`auth_url` - authentication url

`auth_url_short` - short version of `auth_url`

`auth_version` - authentication version, defaults to 3

`user_domain_name` - user domain name

`streaming` - if streaming version of /vsi file accessor is used

for s3 storage:

`type: s3`

`bucket` - name of S3 bucket

`endpoint_url` - url endpoint

`access_key_id` - access key identifier

`secret_access_key` - secret access key

`public` - default "false"

`region_name` - aws s3 region

`validate_bucket_name` - if bucket name should be validated, defaults to true

`streaming` - if streaming version of /vsi file accessor is used

for local storage:

`type: local`

`root_directory` - directory with data (must be accessible inside containers of services that access it)

for http storage:

`type: http`

`endpoint_url` - url endpoint

`streaming` - if streaming version of /vsi file accessor is used

`source` optional data source for the preprocessor. Configuration parameters same as `data`

`cache` configuration for the data source of the cache. Configuration parameters same as `data`. Can be `type:local`. In this case a local sqlite3 database is created.

Data Collections - `collections`

`name:{config}` pairs where the name of the collection is mapped to the product and coverage types

`product_types` - list of product types for the collection

`coverage_types` - list of coverage types for the collection

Product types - `productTypes`

List of product type objects with the following configs:

`name` - product type name

`defaultBrowse` - name of the default browse type

`coverages` - mapping of coverage names to assets

- `assets` - list of assets

`browses` - mapping of browse types to definitions

`collections` - collections to which the product type belongs to

`masks` - masks to which the product type belongs to

Layers - `layers`

Full configuration schema of client - search for layers.

Overlay layers - `overlayLayers`

Full configuration schema of client - search for overlayLayers.

Coverage Types - `coverageTypes`

List of coverage types to add to the backend.

`bands` - list of band definitions

- `definition` - ogc link to band definition
- `description` - description of band
- `identifier` - identifier of band
- `name` - name of band
- `nil_values` - list of NAN values
  - `reason` - ogc reason
  - `value` - what value is considered NAN
- `uom` - unit of measure
- `wavelength` - wavelength

`data_type` - type of data

`name` - name of the band

Service Metadata - `metadata`

Metadata values used by services.

`title` - title of the service

`header` - client header

`abstract` - abstract of the service

`url` - override service url - if not set, then announced links in Capabilities documents will depend on the used hostname of the request

`keywords` - list of keywords

`accessConstraints` - access constraints

`fees` - fees

`contactName` - name of contact person

`contactPhone` - phone of contact person

`contactFacsimile` - facsimile of contact person

`contactOrganization` - contact person organization

`contactCity` - city of contact person

`contactStateOrProvince` - state or province of contact person

`contactPostcode` - postcode of contact

`contactCountry` - country of contact

`contactElectronicMailAddress` - contact email

`contactPosition` - contact position

`providerName` - name of provider

`providerUrl` - url of provider

`inspireProfile` - inspire profile

`inspireMetadataUrl` - inspire metadata url

`defaultLanguage` - default language of service

`language` - language of service

2.4.2 Database configuration - `database`

Database configuration. See https://artifacthub.io/packages/helm/bitnami/postgresql for a comprehensive guide.

2.4.3 Redis configuration - `redis`

Redis configuration. See https://artifacthub.io/packages/helm/bitnami/redis for comprehensive configuration.

2.4.4 Common service configuration

Here is a list of common configurations across services.

`replicaCount` - number of pods to spawn

`nameOverride` - override the short name

`fullNameOverride` - override the full name

`image` - image mapping

- `repository` - repository of image
- `pullPolicy` - pull policy
- `tag` - tag. If unset will default to latest

`service` - service mapping. Available only for forwarded services

- `type` - type of network service
- `port` - port to forward

`resources` - resource mapping

- `limits` - resource limits
  - `cpu`
  - `memory`
- `requests` - request resources
  - `cpu`
  - `memory`

`affinity` - affinity configuration

`livenessProbe` - liveness tests

`ingress` - ingress trigger

`global` - global settings

All non-global configuration relevant to the services is located in the `config` section for each service e.g.

```yaml
cache:
  config:
    # cache configuration values go here

client:
  config:
    # client configuration values go here
```

Client configuration - `client`

Full configuration schema of client.

Cache configuration - `cache`

`wmsEnabled` - wms enable switch

`wmtsEnabled` - wmts enable switch

`connectionTimeout` - timeout in seconds for connection

`timeout` - timeout for upstream connection

`expires` - tile expiry in number of seconds

`key` - cache path scheme with keys

Renderer configuration - `renderer`

Currently accepts no additional custom configuration.

Registrar configuration - `registrar`

`disableDefaultRoute` - disables default route for eoxserver if true

`eoxserverInstanceBasePath` - the default backend instance path

`eoxserverInstanceName` - the default backend instance name

`defaultQueue` - the name of the queue that the registrar listens on - default "register"

`defaultSuccessQueue` - queue that the registrar sends successfully registered items to

`defaultErrorQueue` - queue that the registrar sends failed items to

`defaultReplace` - if set to true, replaces existing items during registration, default "true"

`defaultBackends` - list of backend definitions

`defaultHandlers` - list of handler definitions

`routes` - mapping of custom routes.

```
routes:
  <route-name>:
    path: <import-path>
    queue: <queue>
    backends:
      - path: <backend-import-path>
        kwargs: <backend-keyword-arguments>
```

Example configuration: https://gitlab.eox.at/vs/core/-/blob/main/registrar/config-sample.yaml

Harvester configuration - `harvester`

One-to-one mapping of the original configuration. More info: https://gitlab.eox.at/vs/harvester/-/blob/main/src/harvester/config-schema.json

Scheduler configuration - `scheduler`

One-to-one mapping of the original configuration. More info: https://gitlab.eox.at/vs/scheduler/-/blob/main/config-sample.yaml

Preprocessor configuration - `preprocessor`

One-to-one mapping of the original configuration. More info: https://gitlab.eox.at/vs/vs/-/blob/main/preprocessor/src/preprocessor/config-schema.json

Preprocessor-v2 configuration - `preprocessor-v2`

One-to-one mapping of the original configuration. More info: https://gitlab.eox.at/vs/preprocessor/-/blob/main/preprocessor/config-schema.yaml

Seeder configuration - `seeder`

`minzoom` - minimum zoom from which to seed layers

`maxzoom` - maximum zoom to which to seed layers

`collection_grids` - dictionary of mappings `collection:grids` if only selected grids for a certain collection should to be seeded

Ingestor configuration - `ingestor`

Currently accepts no additional custom configuration.

2.5 Scaling

For Kubernetes deployments, advanced scaling configurations are available.

2.5.1 Renderer

The renderer uses a fixed number of 8 workers for each replica. By default, the replicas have the following resource settings:

```
Limits:
  cpu:     1500m
  memory:  6Gi
Requests:
  cpu:      500m
  memory:   512Mi
```

This can be customized by setting the following helm values:

```
renderer:
  resources:
    requests:
      cpu: 1
    [...]
```

2.5.2 Scaling

The default replica count is set to 1 which can be customized by this helm value:

```
renderer:
  replicaCount: 2
```

Alternatively, horizontal autoscaling is supported based on the CPU metric. If enabled, the default minimum and maximum value for the replicas are 1 and 3 respectively. It can be further customized using the following helm values:

```
renderer:
  hpa:
    enabled: true
    minReplicas: 1
    maxReplicas: 3
```

Note that the horizontal auto-scaler uses a target CPU utilization of 100%, which refers to 100% of the required CPU resources.

2.6 Service management

This subchapter documents k8s specific management steps.

2.6.1 Running commands in VS services

For administration, it can be necessary to run commands directly in one of the services that make up VS.

Most VS services correspond to a `deployment` in Kubernetes, so they can be accessed like this:

```
kubectl exec -it deployment/vs-preprocessor -- bash
```

However stateful components such as *redis* or *postgres* map to a *statefulsets* in Kubernetes and can be accessed using the following command:

```
kubectl exec -n demo-eocat-multiple -it statefulset/vs-redis-master -- bash
```

Note that the command given above launches a shell inside the container. If only one command needs to be run inside the services, the command can be directly given instead of *bash*.

2.6.2 Purge database

Database structure and models are created as a **first step** of deployment of View Server and is afterward **not** updated if the used values change.

> **Warning:** WARNING: The following step deletes all added contents of the database - **ALL products have to be then re-registered**!

To clean the database to enable recreating it from scratch when values changed do:

```
kubectl exec -it deployment/vs-registrar -- bash -c 'python3 $INSTANCE_DIR/manage.py␣
↪flushdb'
helm uninstall name
helm install name --values ... # triggers database structure recreate
```

For platform-agnostic *management and operations* steps, visit chapter *Operations and management*.

Or continue to the section *Data Ingestion* to see how data can be ingested to a VS.

CHAPTER

THREE

SETTING UP DOCKER SWARM

3.1 Prerequisites

3.1.1 Docker

Installation of VS on Docker Swarm requires only Docker Engine installed. It has been successfully tested with version 19 an 20.

3.1.2 Operating System

VS should be deployable on any reasonably new Unix based system and has been successfully tested to be deployed via Docker Swarm on following systems:

- Red Hat Enterprise Linux 7.9
- Red Hat Enterprise Linux 8.6
- Ubuntu 18.04
- Ubuntu 20.04

3.1.3 Python and helm

For configuration files generation, `Python 3, Helm` are essential but do not need to be installed on the target deployment system.

3.2 Initialization Swarm

In order to set up an instance of the View Server (VS) for Docker Swarm, the separate `vs_starter` utility is recommended. Minimum `Python` version to be used is `3.8`.

Its objective is to create a set of static docker compose configuration files using the rendered helm templates, which in turn use a set of previously created `values.yaml` files. See *Operating on Kubernetes* for more information about `values.yaml`, how to create them and meaning of individual values.

First ensure that you have Helm software installed in order to generate the helm templates - instructions can be found on the install page.

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

```
pip3 install git+https://gitlab.eox.at/vs/vs-starter.git
```

Configuration files for a new VS Swarm collection named `test` and deployment `staging` with additional set of PRISM specific configuration values can be created by:

```
# render helm templates from two sets of values.yaml files (generic ones and␣
↪deployment specific).
helm template test-staging <chart-location> --output-dir ./rendered-template --values␣
↪./test/values.yaml --values ./test/values.staging.yaml

# convert the templates content to docker compose swarm deployment files
vs_starter rendered-template/vs/templates <OUTPUT_PATH> --slug test --environment␣
↪staging -o $PWD/test/docker-compose.shared.yml -o $PWD/test/docker-compose.instance.
↪yml
```

Following `environments` are supported: `dev`, `staging`, `ops` in the templates.

Name of the collection - `--slug` parameter should be containing only lowercase letters and underscores.

`-o` optional multiple parameters stand for absolute paths to additional override templates to be rendered together with the default one from `vs-starter`.

For more detailed usage guide for `vs-starter`, continue to vs-starter README and vs-starter sample templates.

Once the initialization is finished the next step is to deploy the Docker Swarm stack.

3.3 Setup Docker Swarm

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 Swarm* step.

3.3.1 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
```

3.3.2 Docker Swarm setup

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 --advertise-address <ip>
```

where `ip` will be the IP of master node, under which it will be reachable by the worker nodes. If only a single node setup (dev) is created, then *–advertise-address* is not needed.

Alternatively, an existing Swarm can be joined by a worker. 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. Note that default `vs-starter` templates do not consider any external/internal label placement restrictions.

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

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

3.3.3 Optional Logging setup

For `staging` and `ops` environments, the services in the sample compose files reference the fluentd logging driver, no manual change is necessary.

Another possible way is to configure the default logging driver on the docker daemon level to be fluent by creating the file `/etc/docker/daemon.json` with the following content:

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

and afterwards restarting the docker daemon via

```
systemctl restart docker
```

For `dev` environment, compose files for configure the `json` logging driver for each service.

3.4 Configuration

VS services configuration before deployment is done in two places. First are the `values.yaml` and the second can optionally be the `docker-compose templates` used by vs-starter.

3.4.1 Applying changes

After the operator makes any changes to any of the `values.yaml`, then for them to be applied, the combination of `helm template` and `vs-starter` commands needs to be performed.

If changes only in `docker-compose` templates are done, then running just `vs-starter` is enough.

It is expected that any necessary changes to the service configurations by the operator are done on the level of `docker-compose templates`, rather than on the level of rendered docker-compose configuration yamls to allow re-running the `vs-starter` and `helm template` commands in the future for the same collection (for example to apply image version upgrades).

The following command then automatically restarts all services of a stack with changed configuration files. This is done by re-using the stack rm and stack deploy commands - see *stack redeployment*.

3.4.2 Types of per-stack defined docker-compose files

The previous command expects the usage of three sets of `docker-compose files` per stack. They are documented in more detail in the following description which should serve as a suggestion of possible deployment, not a must.

`docker-compose.yml`

It is included by default in `vs-starter` and will always be rendered - Current version of the template. The operator does not have a way to change this file in another way than a Merge request to the original repository. It is not expected changes to this template will be necessary as overrides are possible by other docker-compose files.

`docker-compose.shared.yml`

More specific overrides file, which hold "generic" rules for a concrete platform where VS will be deployed like:

- routing rules utilizing the `base` stack via traefik rules as docker labels
- special configuration depending on `dev/staging/ops` environments

This file should be reused or modified by the operator during initial configuration creation and adjusted to the needs of that set of deployments. It is expected that it will be created just once for a certain platform and shared between different stacks for different collections. Sample docker-compose.shared.yml.

`docker-compose.instance.yml`

Collection-specific overrides file, which holds special rules for one single stack, therefore hosting configuration which differs between individual collections. Sample docker-compose.instance.yml. It is expected that it will be created once for every collection that needs specific compose overrides.

For more information to sample compose templates refer to vs-starter sample templates README.

3.4.3 Docker Compose Settings

This section describes general docker-compose concepts used in VS vs-starter templates.

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.

3.4.4 Environment Variables

All necessary files are created from templates when `vs_starter` the tool is used on rendered helm templates.

These variables are passed to their respective container's 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 can be a `db.env` file to store database access details. This file will be already prefilled from the template rendering step.

All those files are placed in the `config/` 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:

```
services:
  # ....
  registrar:
    env_file:
      - ./config/db.env
      - ./config/django.env
    environment:
      INSTANCE_ID: "view-server_registrar"
      # ...
    # ...
```

`.env` Files

The following `.env` files are typically used:

- `db.env`: The database access credentials, for all services interacting with the database.
- `django.env`: This env file defines the credentials for the Django admin user to be used with the admin GUI.

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 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`) 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 is unset, as it may cause unintended behavior with Zarr files. It is recommended to set this to a comma-separated string of file suffixes of expected file formats to be included in the VS instance (`.TIF,.tif,.xml`)
- `GDAL_PAM_ENABLED` - Set to NO to prevent unintended writing to persistent auxiliary metadata.
- `GDAL_HTTP_TIMEOUT` - Set in preprocessor to 30

- `GDAL_HTTP_MAX_RETRY` - Set in preprocessor to 16
- `GDAL_HTTP_RETRY_DELAY` - Set in preprocessor to 5
- `CPL_VSIL_GZIP_WRITE_PROPERTIES` - Set in preprocessor to NO. This is done to ensure no writing happens to recursive GZIP archives

Django 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.

- `DJANGO_USER`, `DJANGO_MAIL`, `DJANGO_PASSWORD` secret - The Django admin user account credentials to use the Admin GUI.

Other Environment Variables

- `DEBUG` - when set to `true`, on most services turns on debug (verbose) logging

Renderer environment variables

- `ENABLE_HTTP_ACCESS` - when set to `true` enables HTTP proxy interface of the renderer for direct data HTTP access by requests with range headers

3.4.5 Configuration Files

Such files are passed to the containers in a similar way as the 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 used in service although direct editing of generated files from vs-starter is not expected. Update the configurations always through the values before the `helm template` step if possible. The following contents are left as a backward-compatible reference.

```
# ...
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:

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 with pre-set variables that can be used for elevation rasters are:

`hillshade(band, var('zfactor', 5), 111120, var('azimuth', 315), var('altitude', 45), var('alg', 'Horn'))`

- range 0 - 255
- nodata 0

`aspect(band, False, False, var('alg', 'Horn'))`

- range 0 - 360
- nodata -9999

`slopeshade(gray, 111120)`

- range 0 - 255
- nodata -9999

`contours(gray, var('offset', 0), var('interval', 30))`

- range 0 - 360
- nodata - 9999

`roughness(gray)`

- range 0 - 20
- nodata - 9999

`tri(gray)`

- range 0 - 8
- nodata - 9999

Example:

```
python3 manage.py browsetype create "DEM_Product_COP-DEM_GLO-30-DTED"  "aspect" \
    --grey "aspect(gray, False, False, var('alg', 'Horn'))" \
    --grey-range 0 360 \
    --grey-nodata -9999

python3 manage.py browsetype create "DEM_Product_COP-DEM_GLO-30-DTED"  "contours" \
    --grey "contours(gray, var('offset', 0), var('interval', 30))" \
    --grey-range 0 360 \
    --grey-nodata -9999

python3 manage.py browsetype create "DEM_Product_COP-DEM_GLO-30-DTED"  "hillshade" \
    --grey "hillshade(gray, var('zfactor', 5), 111120, var('azimuth', 315), var(
↪'altitude', 45), var('alg', 'Horn'))" \
    --grey-range 0 255 \
    --grey-nodata 0

python3 manage.py browsetype create "DEM_Product_COP-DEM_GLO-30-DTED"  "roughness" \
    --grey "roughness(gray)" \
    --grey-range 0 20 \
    --grey-nodata -9999

python3 manage.py browsetype create "DEM_Product_COP-DEM_GLO-30-DTED"  "slope" \
```

(continues on next page)

(continued from previous page)

```
    --grey "slopeshade(gray, 111120)" \
    --grey-range 0 50 \
    --grey-nodata -9999

python3 manage.py browsetype create "DEM_Product_COP-DEM_GLO-30-DTED"  "tri" \
    --grey "tri(gray)" \
    --grey-range 0 8 \
    --grey-nodata -9999
```

To use the pansharpening function the True Color browse type would look like this:

`pansharpen(pan, red, green, blue, nir)[0]`

- range 0 - 2000
- nodata 0

Example:

```
python3 manage.py browsetype create "NAO_BUN_1A"  "TRUE_COLOR_PANSHARPENED" \
    --red "pansharpen(pan, red, green, blue, nir)[0]" \
    --green "pansharpen(pan, red, green, blue, nir)[1]" \
    --blue "pansharpen(pan, red, green, blue, nir)[2]" \
    --red-range 1 1000 \
    --green-range 1 1000 \
    --blue-range 1 1000 \
    --red-nodata 0 \
    --green-nodata 0 \
    --blue-nodata 0 \
    --traceback
```

3.4.6 Sensitive variables

Since environment variables include credentials that are considered sensitive, avoiding their exposure inside `.env` files would be the right practice. To manage to transmit sensitive data securely into the respective containers, docker secrets with the values of these variables should be created.

Currently, basic configuration templates reference two variables that have to be saved as docker secrets before deploying the swarm: `DJANGO_PASSWORD` and `DJANGO_SECRET_KEY`.

An example of creating `DJANGO_PASSWORD` as secret on the manager node using the following command :

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

3.4.7 PRISM specific secrets

The following docker secret for traefik basic authentication needs to be created: `BASIC_AUTH_USERS_APIAUTH` - used for admin access to kibana and traefik. Access to the services for alternative clients not supporting main Shibboleth authentication entry points 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:hashed-password (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 an sftp image using the following command :

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

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 in Docker Swarm*.

3.4.8 Stack Deployment

Before the stack deployment step, some environment variables and configurations which are considered sensitive (SECRETS) should be created beforehand, refer to *Sensitive variables* 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.

The deployment of created stack compose files should be performed in following order:

1) base stack - with updated extnet networks for each <slug> collection
2) logging stack (references logging-extnet network from base)
3) -x) individual {slug} collections (references extnet network managed by base stack)

logging and base stacks

```
docker stack deploy -c docker-compose.base.yml base
docker stack deploy -c docker-compose.logging.yml logging
```

stack redeployment

For a redeployment of a single stack one would do following:

```
# only if was previously running
docker stack rm <stack-name>
collection_env="test-staging" && docker stack deploy -c "$collection_env"/docker-
↪compose.yml -c "$collection_env"/docker-compose.shared.yml -c "$collection_env"/
↪docker-compose.instance.yml <stack-name>
```

(replace test-staging with the actual *slug-env* identifier and assuming that the vs-starter did output the templates to "$collection_env")

These commands performs a set of tasks. First off, it obtains all necessary docker images.

When all relevant images are pulled from their respective repository the services of the stack are initialized. 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.

If a service is not starting or is stuck in 0/x state, inspect its logs or status via

```
docker service ps --no-trunc <service_name>
docker service logs <service_name>
```

The above mentioned process necessarily involves a certain service downtime between possible shutting down of the stack and new deployment.

3.5 Service Management

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

3.5.1 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 manual 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.

Another way how to preset number of replicas is already on the level of `values.yaml` via *values.service_name.replicaCount* key.

3.5.2 Updating Services or configuration files

Updating the VS software is done using previously established tools - `helm template` and `vs-starter` - see *Initialization Swarm*. Image version is located in *docker-compose templates* and is prefilled automatically during *helm template* configuration step. The version values will come from the latest version of the *vs-deployment charts*, so to get the latest version simply *git pull* the latest *vs-deployment* chart.

If the operator wishes to keep a VS on a lower version than the current latest, it is possible to use a certain fixed version of the *vs-deployment* chart rather than the latest.

Each stack that needs to be updated needs to be torn down and redeployed - see *stack redeployment*.

3.5.3 Running commands in VS services

For administration, it can be necessary to run commands directly in one of the services.

This subchapter documents docker swarm-specific management steps.

To exectute a command in a redis container with in a given stack, use:

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

For platform agnostic *management and operations* steps, visit chapter *Operations and management*.

3.5.4 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 fewer commands to allow paging through them.

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

In case only one instance of a service is running on one node, the <container-name> can be returned by fetching the available containers of 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 the 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 are intended as a quick way to view the latest log entries of all tasks of 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.

3.5.5 Inspecting logs from the logging stack

Fluentd is configured as the main logging driver for production VS deployment. To access the logs, an interactive and multi-purpose Kibana interface is available and exposed externally by traefik.

Kibana is usually available under a hostname defined by the operator in logging compose file - the default value is `kibana.<service-url>`.

For a simple listing of the filtered time-sorted logs as an equivalent to the *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, modifying time ranges, etc. The individual log results will then appear in the `Document table` panel at the bottom of the page.

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

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

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

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

3.5.6 Kibana useful queries

Successfully registered or replaced products during registration

```
fulltext search: "stack_registrar" and ("Successfully registered Product" or
↪"Successfully replaced Product")
```

Failed registrations

```
add custom filter exception: exists and use fulltext search for: "stack_registrar"␣
↪and not "fluent"
```

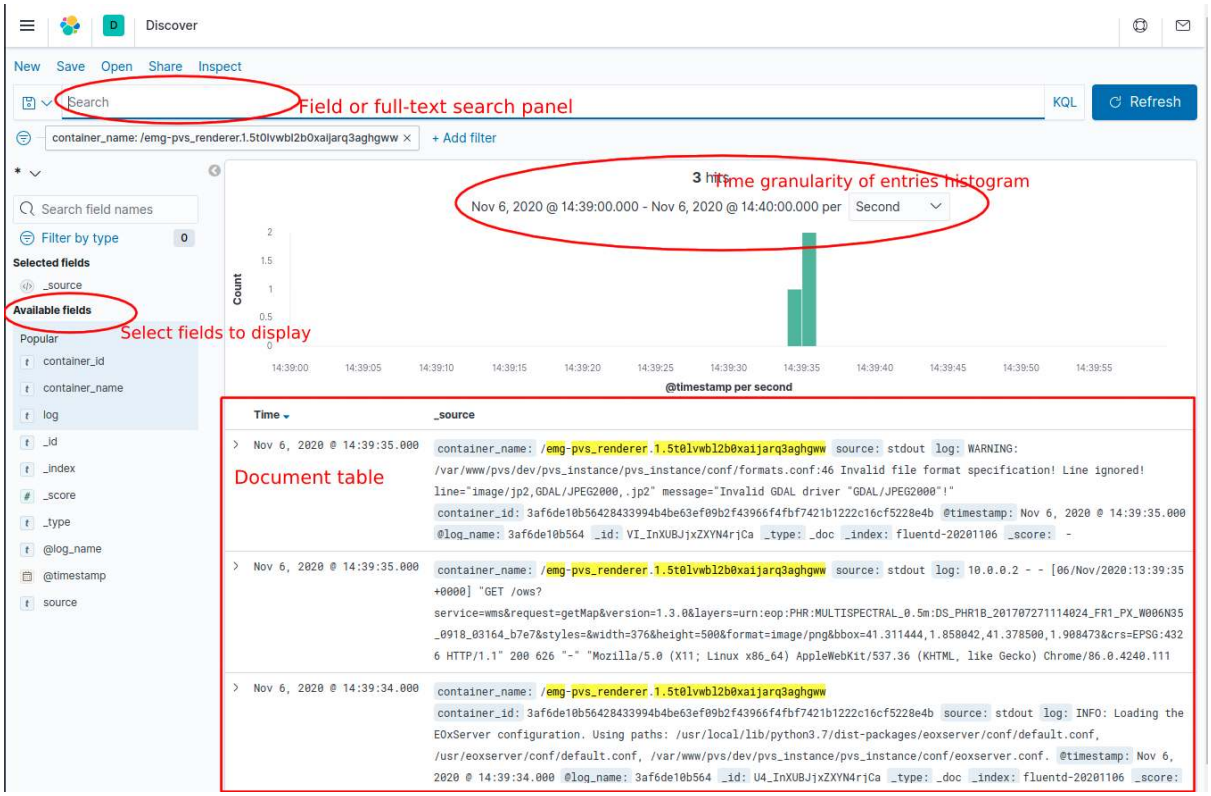

Figure 3.5.1: *Kibana discover panel*

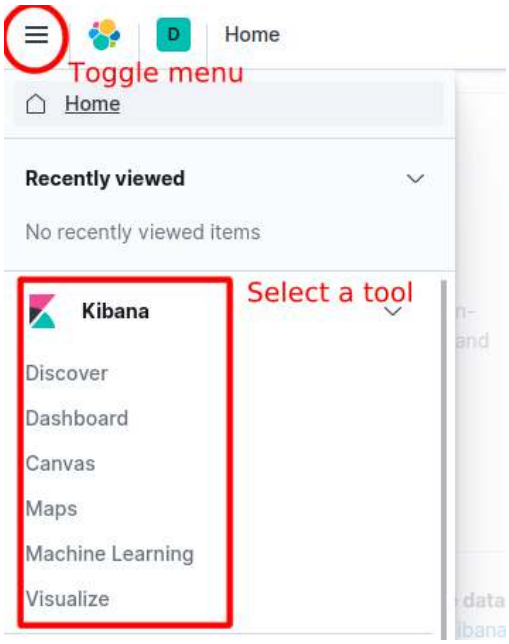

Figure 3.5.2: *Kibana menu*

WMTS + WMS requests for a stack

```
fulltext search: "stack" and ("renderer" or "cache") and "ows"
```

3.5.7 Increasing logging level

In the default state, all components are configured to behave in a production logging setup, where the amount of information contained in the logs is reduced.

All View Server services (except Redis and database) support setting a *DEBUG=True* environment variable increasing servicev verbosity by a rolling update of containers environment in the following way:

```
docker service update --env-add DEBUG=true <stack-name>_<service-name>
```

To put the services back to default production logging state, run:

```
docker service update --env-add DEBUG=false <stack-name>_<service-name>
```

If Operator is inspecting a concrete issue or failure status codes, then setting DEBUG mode of renderer or cache might be necessary.

If map tiles are not visible, consult network tab and get the failing requests.

Errors of cache service are usually reported for 4xx/5xx status code requests in the X-Mapcache-Error response header as a value. Increasing DEBUG logging is usually not needed.

Renderer service does usually not give any indication of the reason of failures unless DEBUG logging is enabled. On DEBUG ows:ExceptionReport XML with last error from the application is returned to the user as a response to the problematic operation or malformed request.

Preprocessor, registrar and harvester can give our more information on setting DEBUG logging.

3.5.8 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"
```

(continues on next page)

(continued from previous page)

```
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: The date for the 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"
echo "Cleaning up"
docker exec $DB_CONTAINER sh -c "rm /$DATE.sql"
rm $UNCOMPRESSED_FILE
```

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

3.5.9 Purging VS from machine

To completely remove VS and all its data from a node, where it was running before, you should do the following steps:

- manually tear down all running VS stacks - *logging, base and all individual collections* via `docker stack rm <stack-name>`
- manually remove all persistent docker volumes via `docker volume rm <volume-name>`. If you are not sure which ones were created by VS, check all the individual docker-compose files used for deploying the stacks and find the key `volumes`.
- manually remove all docker images which contribute to VS by `docker image rm <image-name>`.

If you are not sure which images or volumes contributed to VS, check all the individual `docker-compose` files used for deploying the stacks and find the key `volumes` or `service.image`.

3.5.10 Inspecting reports - PRISM specific

Once a product is registered, an 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. To log into the folders through port 2222 for `vhr18` or other respective ports 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.

3.6 Access in Docker Swarm

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

3.6.1 General overview

Each docker **stack** has its own internal network `intnet` where services can communicate with 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, DEM: terrain-server, dem-app) is provided via external network `extnet` and reverse-proxy (traefik) with a load balancer.

These services can have a set of authentication and authorization rules applied both on the traefik level and Shibboleth SP level.

3.6.2 Routing with traefik

`Reverse-proxy` service in the base stack provides a 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 in three places.

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

For example, the following configuration snippet enables access to certain paths of the `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"
```

(continues on next page)

(continued from previous page)

```
      - "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"
      - "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 afterward. Unsecured HTTP access is configured to be redirected to the HTTPS endpoint. Inside the swarm among the services, only HTTP is used internally.

3.6.3 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's response code is 2XX, access is granted and the original request is performed. Otherwise, the error response from Shibboleth is returned.

To authenticate with Shibboleth, a user must log in with valid credentials on the side of the 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 possible to separate viewing and download functionality, as both of these parts are handled by the `renderer` service.

3.6.4 Configuration

For the 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 Swarm* step using `vs_starter` tool. Still, it is advised to check the following steps, understand them and change them 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, certificates using the 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>
```

(continues on next page)

(continued from previous page)

```
  </AND>
</AccessControl>
```

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

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

```
services:
  shibauth:
    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"
```

CHAPTER

FOUR

CREATE A NEW COLLECTION STEP BY STEP

The following tutorial will guide you through the process of creating and updating a set of configurations (and database) for an imaginary new dataset - layer. Wherever possible it links to other parts of the documentation for further reference.

During the tutorial, elements of the EOxServer Data Model are used and should be understood although by defining the values, direct interaction with the EOxServer database models are usually not required:

- Collection + CollectionType
- Product + ProductType
- Coverage + CoverageType
- Browse + BrowseType
- Storage + StorageAuth

4.1 Examples of public configurations

The following public View Server configuration values examples can be used as a further reference:

- EOEPCA Demo Helm Chart
- VS testing values

4.2 Analyze the data

First, analyze the earth observation data that you will provide by View Server. Below you will find the different types of information necessary to create the configurations:

4.2.1 Data format

For ideal viewing performance of the data, images should be formatted as Cloud Optimized GeoTIFF (COG) or should at least have **internal overviews** and **internal tiling**. If the data fulfill any of those two points, proceed to the next point.

If internal overviews are not present even in the case of large EO Data files, rendering a 1x1 pixel image will cause the whole image file to be read, which will negatively impact the rendering performance.

An important attribute of the raster data is their **data type** (UInt16, Int32, and others). Although View Server will generally be able to read any data type that GDAL can read, having this information is necessary for further steps.

To convert the data to COG it is suggested to:

- either manually use GDAL tools before ingesting the data to View Server (In this case the component pre-processor is not necessary)

- or configure and use the *Preprocessor configuration - preprocessor* and reference the preprocessed data instead.

4.2.2 Metadata format

It is also important to check the format of metadata files (sidecar files) next to the raster data. View Server uses SpatioTemporal Asset Catalog (STAC) items internally as a metadata format, both for storage and for messaging between components.

In an ideal case, the STAC items describing the data and metadata are already generated and should be used.

Having STAC items generated is not a prerequisite for all data. View Server will understand some other metadata formats during ingestion. More on that later.

4.2.3 Data storage

The next step is to clarify where the raster data and metadata are stored.

This does not have to be on the same infrastructure where View Server is going to be deployed. Having the data closer to the deployment (at the same cloud provider for example) should significantly speed up data access.

View Server supports the following access and storage protocols (the *fsspec* library should enable further extensions):

- s3
- OpenStack Swift
- local path
- http

4.2.4 Bands and rendering

Depending on the type of data (optical, radar, other) and the number of bands in the raster data, different types of rendering can be configured. In this step, it should be clarified how many bands are there in the raster data and which wavelengths (or general type of information) each band has. The knowledge of band structure will influence possible definitions of types of rendering further on.

4.2.5 Groups of products

Products can or should be grouped or separated using a shared metadata property.

An example of ideal separation would be Processing levels:

- Level 1 should not be visualized together in one layer with Level 3 products
- SAR products: Single Look Complex (SLC) and Ground Range Detected (GRD) should be separated

4.3 Generate configurations

Let's continue to create View Server configuration values based on the knowledge about the products that have been gathered.

To find out all possible configurations for any of the values configuration keys please refer to the *Helm configuration reference*.

As a foundation for a new set of values, the default vs-deployment config values can be used as an empty template to be filled.

Warning: Database structure and models are created as a **first step** of deployment of View Server and is afterward **not** updated if the used values change (there are no database migrations performed between deploys).

Therefore you have to create a consistent working configuration, it might be an iterative process involving deleting the persistent database storage between each redeploy of updated values if the changes involve database model changes. Refer to *Purge database* for the how-to.

Further steps in this cookbook will contain a note if the configuration is used in the database structure or not.

4.3.1 Coverage Types

Changes involve database structure: YES

The first concept to focus on during values creation is `coverage_types`. The objective of this step is to:

- either map the raster data type and band order to the existing `coverage_type` definition
- or alternatively, define a new `coverage_type`

The possible values and meaning of the `coverage_type` are described in *Global.coverageTypes*.

If there already is an existing `coverage_type` with the same type of bands, just in a different order, (near-infrared band of data is for example not a last band, as in `RGBNir coverage_type`, but first), then for the sake of clarity, it is always better to create a new `coverage_type` although it is not strictly necessary, as for the rendering step, the band order (which band corresponds to which RGB color) can be changed.

Note: Pay attention to the following keys when defining a new `coverage_type`:

- coverageTypes[i].name - needed for collections definition
- coverageTypes[i].bands[i].identifier - needed for browses definition

---

4.3.2 Product Types

Changes involve database structure:

- YES for the `global.productTypes` key and all its values except for following: `filter`, `coverages`.

The second, even more, important step, is to create `productType` definitions. Each `productType` represents an EOxServer `Product Type` model and some of its links to other models:

- `BrowseType` EOxServer model - specifies renderings (one to many) via `browses` key - refer to *Browse Types* for guidance on how to fill this key
- Which `data assets` will map to which EOxServer `Coverage Type` model - `coverages` key. There can be multiple `data assets` named `STAC Item entries` for multiple `coverages`.
- To which `collections` will the product from the `product_type` be added. One product can be added to multiple collections if the `product_type` is allowed for those collections. Refer to *Collections*.

The possible values and meaning of the `product_type` are described in *Global.productTypes*.

4.3.3 Browse Types

Changes involve database structure: YES

The third important step is to define `browses` (rendering) definitions for each `productType`. Each `browses` entry represents an EOxServer `Browse Type` model, therefore adding an available *WMS Layer* to the renderer service.

Multiple simple band expressions and pre-made functions can be used in the `band.expression` value. Full list of usable functions.

The band specifications inside the `expression` (red, pan, gray) need to match those defined in the selected `coverage_type` and correspond to the meaning of the raster data itself. The names of the color specification in browse_type name (red, green, blue, grey) are to be used as-is and reference the stretching into RGB (or grayscale) spectrum of the WMS output image.

If *browse.asset* key has a value with a name of a STAC asset, this asset will be used to as a *Browse* Model. This is a way to attempt to register an asset without a *'data' role*. It is preferred for cases, when a viewing ready Browse has been already pregenerated rather than trying to fit it to a *Coverage* model. The Browse behaves slightly differently than Coverages - for example does not allow WCS to be used with it, but at the same time does not need exact georeferencing of image, just that the footprint is extracted correctly in the original STAC item.

Some examples of configured expressions are:

1) percentile rendering of 2-98% of precomputed histogram stretched to 1-256 with configured defaults if individual STAC Item does not have computed statistics contained in metadata. It also additionally masks our pixels in range 1-10 as extra no data.

```
TRUE_COLOR:
  red:
    expression: "interpolate(red, percentile(red, var('percmin', 2), 1),␣
↪percentile(red, var('percmax', 98), 10), 1, 256, var('clip', True),[var('nodata_
↪start',1),var('nodata_end',10)])"
    range:
      - 1
      - 256
```

2) pansharpening operation on the source RGBNir Pan coverages

```
TRUE_COLOR_PANSHARPEN:
  red:
    expression: pansharpen(pan, red, green, blue, nir)[0]
    range: [0, 1000]
    nodata: 0
  green:
    expression: pansharpen(pan, red, green, blue, nir)[1]
    range: [0, 1000]
    nodata: 0
  blue:
    expression: pansharpen(pan, red, green, blue, nir)[2]
    range: [0, 1000]
    nodata: 0
```

3) hillshade rendering of DEM height data in EPSG:4326 with some parameters of the formula specified as "rendering variables" - allowing the WMS client to specify values

```
hillshade:
  grey:
    expression: hillshade(gray, var('zfactor', 5), 111120, var('azimuth', 315), var(
↪'altitude', 45), var('alg', 'Horn'))
```

(continues on next page)

(continued from previous page)

```
    range: [0, 255]
    nodata: 0
```

4) Default unnamed browse type with 0-255 color range on 4 bands mapped to STAC Item Asset with name *browse*.

```
"":
  asset: browse
```

4.3.4 Collections

Changes involve database structure: YES

The fourth step is to define all `collections` grouping the Products. For each collection, it is necessary to add their allowed `product_types` and `coverage_types`.

Example configuration for creating three `collections`: Level_1, Level_3 and a shared one:

```
collections:
  VHR_IMAGE_2018:
    product_types:
      - DOV_MS_L1A
      - DOV_MS_L3A
    coverage_types:
      - RGBNir
  VHR_IMAGE_2018_Level_1:
    product_types:
      - DOV_MS_L1A
    coverage_types:
      - RGBNir
  VHR_IMAGE_2018_Level_3:
    product_types:
      - DOV_MS_L3A
    coverage_types:
      - RGBNir
```

The part of `productType` values corresponding to the above added `collections` key could be for example:

```
productTypes:
  - name: DOV_MS_L1A
    collections:
      - VHR_IMAGE_2018
      - VHR_IMAGE_2018_Level_1
  - name: DOV_MS_L3A
    collections:
      - VHR_IMAGE_2018
      - VHR_IMAGE_2018_Level_3
```

The possible values and meaning of the `collections` are described in *Global.collections*.

4.3.5 Displaying data

Changes involve database structure: NO

The fifth step influences how the layers can be displayed via the `client` service and which tilesets will be exposed by the `cache` service.

The possible values and meaning of the `layers` and `overlayLayers` are described in *Global.layers* and *Global.overlayLayers*.

4.3.6 External access

Changes involve database structure: NO

The sixth step is to define external access to View Server components. If the values are going to be deployed on Kubernetes, it is possible to use View Server's ingress configuration - refer to *ingress*.

If there is already an external setup configured in the system (external ingress, traefik, etc.), the View Server ingress configurations should be completely disabled by:

```
ingress:
  tls: false
```

4.3.7 How to get the data in

Changes involve database structure: NO

Storage

The seventh step in the workflow is to see where the data are located for the View Server to correctly reference them and ingest them to get the information about Product data and metadata into the database.

Possible values and meaning of the `storage` are described in *storage*.

For successful ingestion, at least the `data` key (location of data) needs to be filled according to the used protocol to access the data on the storage.

There are currently three ways how to ingest data into View Server and they might require further configuration.

Optionally `preprocessor` can be used to convert data format beforehand. Refer to *Preprocessor configuration schema*.

Local storage

> **Warning:** If the files to be ingested are on a local storage, the storage folder(s) need to be mounted into the containers of services, which need access to them. For direct registration without preprocessing, the services would be registrar and renderer.
>
> The mounting needs to be configured on the level of `helm release` or `docker-compose templates`. Each node (master or worker) which will possibly host that service needs to have access to the data folder as well.

Example docker compose configuration mounting a folder */data/test1* ` into renderer container path */data* ` is following

```
renderer:
  volumes:
    - type: bind
      source: /data/test1
      target: /data
```

Global data storage configuration in `values.yaml` for using this folder would look like:

```
global:
  storage:
    data:
      directory-data:
        type: "directory"
        root_directory: "/data/"
```

Ingestion

1) Direct ingestion of STAC Item JSON strings to `redis register_queue`.

This process is suitable if the STAC items of Products already exist and for one-off ingestion campaigns - collections that do not require any regular updates or additions.

No special configuration except for `storage.data` key is necessary.

2) Using the harvester service for a *pulling* approach

If you configured the harvester, it will harvest new or updated data from various endpoints and protocols and convert the metadata and data to STAC internally and then push it to other components (preprocessor, registrar).

Harvester-specific configuration is required. Refer to *Harvester configuration schema*.

3) Ingestor for legacy Browse Reports - *pushing* approach

Ingestor-specific configuration is required. Refer to *Ingestor configuration schema*.

Optionally refer to *Data Ingestion* chapter for more information.

4.3.8 Global env

The last important step is to modify the `global.env` key which lists all environment variables and their values that all services have access to.

It specifies database and Django passwords, which should be changed as well.

Refer to *Global configurations* for more information.

4.3.9 Individual service configurations

Additionally, most View Server services are configurable using their keys in the values. Refer to *Individual service configurations* for more information.

CHAPTER

FIVE

OPERATIONS AND MANAGEMENT

This chapter lists usual management/operations commands and procedures with references to other parts of the documentation where the concepts are generally described in more detail.

5.1 Generate VS configurations for 1 stack

Two commands to generate a set of docker-compose and other configuration files for one stack of VS for docker swarm.

```
# render helm templates from two sets of values.yaml files (generic ones and␣
↪deployment-specific).
helm template test-staging <chart-location> --output-dir ./rendered-template --values␣
↪./test/values.yaml --values ./test/values.staging.yaml
# convert the templates content to docker-compose swarm deployment files
vs_starter rendered-template/vs/templates <OUTPUT_PATH> --slug test --environment␣
↪staging -o $PWD/test/docker-compose.shared.yml -o $PWD/test/docker-compose.instance.
↪yml
```

For more information, see *Initialization Swarm*.

5.2 Starting/Stopping the View Server

```
# stop stack if was previously running
docker stack rm <stack-name>
# deploy stack anew
collection_env="test-staging" && docker stack deploy -c "$collection_env"/docker-
↪compose.yml -c "$collection_env"/docker-compose.shared.yml -c "$collection_env"/
↪docker-compose.instance.yml <stack-name>
```

For more information, see *stack redeployment*.

5.3 Starting/Stopping individual services

Starting or stopping services is done by setting the number of running container replicas to 0. For more information, see *Service Management* or *Scaling*.

Restarting a running service (or updating its environment variables) can be done via the following command:

```
# restart service
docker service update --force <stack_service>
# update environment variable of service, changing the service logging to DEBUG mode
docker service update --env-add DEBUG=true <stack_service>
```

5.4 Deleting VS volumes and images

To delete individual docker volume or image in case of obsolete collections or stacks, when the respective collection stack has already been stopped.

```
# delete docker volume
docker volume rm <stack_volume>
# delete the docker image
docker image rm -f <image:tag>
# command to delete all unused docker images and volumes from a node
docker system prune -f --all --volumes
```

5.5 Harvesting new collection

To initiate harvesting of a certain harvester configuration under name <harvester-name>, two different ways are possible:

```
# inside harvester container
harvester harvest --config-file /config.yaml <harvester-name>
# OR on the main node via redis queue
docker exec -it $(docker ps -qf "name=<stack_redis>") redis-cli lpush harvester_queue
↪'{"name":"<harvester-name>"}'
```

For more information, see *Harvesting*.

5.6 Register a STAC item

To manually register a single STAC item file, two different ways are possible:

```
# inside harvester container
registrar --config-file /config.yaml register items "$(cat json-file-containing-stac-
↪item)"
# OR on the main node via redis queue
docker exec -it $(docker ps -qf "name=<stack_redis>") redis-cli lpush register_queue "
↪$(cat json-file-containing-stac-item)"
```

5.7 Preprocess a STAC item

To manually preprocess a single STAC item, two different ways are possible:

```
# inside preprocessor container
preprocessor preprocess --config-file /config.yaml "$(cat json-file-containing-stac-
↪item)"
# OR on the main node via redis queue
docker exec -it $(docker ps -qf "name=<stack_redis>") redis-cli lpush preprocess_
↪queue "$(cat json-file-containing-stac-item)"
```

5.7.1 Stop ongoing ingestion by deleting all redis queues

```
# delete all used ingestion queues
docker exec -i $(docker ps -qf "name=^$<stack>_redis") redis-cli del preprocess_queue␣
↪register_queue harvester_queue
```

5.8 Verify ingestion into the database

All commands in this subsection are done in `renderer` or `registrar` container.

5.8.1 Checking product/collection status

```
python3 $INSTANCE_DIR/manage.py id check <id>
```

<id> can be either a collection name or product identifier and yields possible outputs:

```
{"event": "The identifier 'urn:eop:SUPERVIEW-2:MULTISPECTRAL_2m:SV-2_20210913_
↪L1B0000147080_1012101725100008-MUX_5532' is already in use by a 'Product'."}
{"event": "The identifier 'VHR_IMAGE_2021' is already in use by a 'Collection'."}
{"event": "The identifier 'test' is currently not in use."}
```

5.8.2 List products

List all existing `products`, optionally constrained by being part of `<collection-name>`.

```
python3 $INSTANCE_DIR/manage.py id list -c <collection-name> --suppress-type
```

5.8.3 Get total sum of products

Check number of `products` currently ingested:

```
python3 $INSTANCE_DIR/manage.py shell -c 'from eoxserver.resources.coverages.models␣
↪import Product;print(Product.objects.count());'
```

5.8.4 Count products being part of a collection or product type

Check number of `products` currently ingested and being part of `ProductType` with name `<product-type-name>`.

```
python3 $INSTANCE_DIR/manage.py shell -c 'from eoxserver.resources.coverages.models␣
↪import ProductType;pt=ProductType.objects.get(name="<product-type-name>);print(pt.
↪products.count())'
```

Count products being part of `collection`.

```
python3 $INSTANCE_DIR/manage.py shell -c 'from eoxserver.resources.coverages.models␣
↪import Collection;c=Collection.objects.get(identifier="Emergency");print(c.products.
↪count())'
```

All the collections are available via OpenSearch interface openly on `/opensearch` endpoint. The example call to get the number of products in a `<collection-name>` as an ATOM response would be: `<service-url>/opensearch/collections/<collection-name>/atom/?count=0`

5.9 Vacuum database tables

After the ingestion campaign of a collection finishes, it is suggested to manually vacuum a database by the following command inside a `database` container

```
vacuumdb -f -d $DB_NAME -U $DB_USER -h $DB_HOST --analyze
```

5.9.1 Export all metadata, data and browse paths

To get an exported list of all data and metadata files referenced by all registered products, use the following set of Python commands in the Django instance shell (registrar or renderer container).

```
python3 $INSTANCE_DIR/manage.py shell
```

```
from eoxserver.resources.coverages.models import (Product, Coverage, MetaDataItem,␣
↪ArrayDataItem, Browse)
# print all metadata paths and all files to a file
with open("/registered_paths.txt", "w") as ff:
    prods = Product.objects.all()
    for prod in prods:
        covs = Coverage.objects.filter(parent_product_id=prod.id)
        metadata_items = MetaDataItem.objects.filter(eo_object_id = prod.id)
        browses = Browse.objects.filter(product_id=prod.id)
        for cov in covs:
            data_items = ArrayDataItem.objects.filter(coverage_id = cov.id)
            for it in data_items:
                print(it.location, file=ff)
        for md in metadata_items:
            print(md.location, file=ff)
        for b in browses:
            print(b.location, file=ff)
```

The file will be saved to the path */registered_paths.txt* inside the container.

5.9.2 Deregister a product

To completely deregister a single product with a given identifier, you can do:

```
python3 $INSTANCE_DIR/manage.py product deregister <identifier>
```

To deregister all products from a certain collection a combination of two commands can be used:

```
for item in $(python3 $INSTANCE_DIR/manage.py id list -c <collection-name> -s); do␣
↪python3 manage.py product deregister $item; done
```

To completely remove a collection:

```
python3 $INSTANCE_DIR/manage.py collection delete <collection-name>
```

5.10 Adding a new configuration to running stack

If operator wants to add a new collection, product type, coverage type or browse type - instance of any of the database models, then a few manual steps are necessary after generating the new configurations.

At the moment there is no auto-update (create/delete/update) of the database structure on change/update of values.

It is necessary to invoke the relevant parts of the *init_db.sh* script containing the *python3 manage.py* CLI commands to manually add the models either renderer or registrar container.

Example of adding of a single collection *SAR_IMP_1P* with a default RGBA rendering and a product type with the same name would be running following:

```
python3 manage.py producttype create "SAR_IMP_1P" \
      --coverage-type "RGBA"
python3 manage.py browsetype create "SAR_IMP_1P"  \
      --red "red" \
      --green "green" \
      --blue "blue" \
      --red-range 0 255 \
      --green-range 0 255 \
      --blue-range 0 255 \
      --red-nodata 0 \
      --green-nodata 0 \
      --blue-nodata 0 \
      --alpha "alpha" \
      --alpha-range 0 255
 python3 manage.py collectiontype create "SAR_IMP_1P_type" \
      --coverage-type "RGBA" \
      --product-type "SAR_IMP_1P"
 python3 manage.py collection create "SAR_IMP_1P" \
      --type "SAR_IMP_1P_type"
```

5.11 Troubleshooting – accessing logs

see more detailed instructions at *Inspecting logs in development*.

5.12 Restart services/memory clean

To clean up used memory by external services as a one-off operation, do the following:

```
for stack in 'bs1' 'bs2'
do
    for service in 'renderer' 'cache'
    do
          docker service update --force "$stack"_"$service"
    done
done
```

5.13 Cache seeding operations

If `seeder` and `registrar` services are configured to be linked via input and output queue, the successful registration will automatically trigger seeding.

To manually trigger the seeding of a certain product, do the following command in *seeder* container:

```
# seed a single product for a set of pre-configured layers
python3 seeder/seeder.py --mode standard --product-to-seed <product-id> --config-file␣
↪/seeder-config.yaml -v 4
```

optionally add *–leave_existing* parameter to not delete existing tiles which, only fills the missing ones.

For seeding a set of configured layers for a whole collection, export the list of products and perform seeding of them one by one. Bulk-seeding of a whole layer is not available in seeder yet.

```
# export all products from all collections for a single stack to a text file on the␣
↪node
docker exec -it $(docker ps -qf "name=^<stack>_registrar") bash -c 'python3 $INSTANCE_
↪DIR/manage.py id list --suppress-type' > export-product-<stack>.txt
# stream a list of products into the seed command in a single seeder container
while read product_id;do docker exec -i $(docker ps -qf "name=^<stack>_seeder")␣
↪python3 seeder/seeder.py --config-file /seeder-config.yaml --product-to-seed "
↪$product_id";done<export-product-<stack>.txt
```

CHAPTER

SIX

DATA INGESTION

This section details the data ingestion and 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.

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.

Please see *Running commands in VS services* or *Running commands in VS services* to learn how commands can be run in Kubernetes and Docker Swarm respectively.

Note: For the VS, only the `List` Redis data type is used.

`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 to add an element to be processed as soon as possible instead of waiting before all other elements before it is processed.

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

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

```
redis-cli lpush register_queue '{"type": "Feature", "stac_version": "1.0.0", "id":
↪"urn:eop:DOVE:MULTISPECTRAL_4m:20180811_081455_1054_3be7", "properties": {"product_
↪type": "DOV_MS_L3A", "start_datetime": "2018-08-11T08:14:55Z", "end_datetime":
↪"2018-08-11T08:14:55Z", "datetime": "2022-10-27T11:53:45Z"}, "geometry": null,
↪"links": [], "assets": {"gsc_metadata": {"href": "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/GSC#CR#ESA#VHR_IMAGE_2018
↪#20190706#165304.xml", "type": "application/xml", "title": "GSC Metadata file",
↪"description": "GSC metadata file from source archive", "roles": ["metadata"]}, "ms
↪": {"href": "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/IMG_
↪PL00_DOV_MS_L3A_20180811T081455_20180811T081455_TOU_1234_3be7_R1C1.TIF", "type":
↪"image/tiff; application=geotiff; profile=cloud-optimized", "title": "Preprocessed␣
↪image", "description": "Product image converted into a COG", "raster:bands": [{
↪"nodata": 0.0, "data_type": "uint16"}], "eo:bands": [{"name": "band1", "common_name
↪": "band1"}], "roles": ["data"]}}, "stac_extensions": []}'
```

STAC Item assets should either contain the full URL or full path to the asset with a protocol prefix (eg. https://, s3://) or if a *storage.container*, *storage.endpoint_url* or *storage.root_directory* is configured, then it can also contain only the *object name*.

Example asset paths:

s3 storage.bucket defined:

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/GSC#CR#ESA#VHR_IMAGE_2018#20190706#165304.xml

s3 storage.bucket not defined:

s3://test-data/OA/PL00/1.0/00/urn:eop:DOVE:MULTISPECTRAL_4m:20180811_081455_1054_3be7/0001/PL00_DOV_MS_L3A

6.1.1 Harvesting

The following command executes a `redis-cli lpush` command to trigger a new harvesting operation on a "Kompsat2" harvester

```
redis-cli lpush harvester_queue '{"name":"Kompsat2"}'
```

To check a certain queue, do the following:

```
$ redis-cli lrange harvester_queue 0 -1
{'name':'Kompsat2'}
```

For configuration details, see *Harvester configuration - harvester*.

6.2 Direct Data Management

Sometimes it is necessary to interact with the services for ingestion directly.

Harvester, registrar, and preprocessor services 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 a Redis queue for new items, which will be triggered one by one. The second mode is to run the service 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.

In this section, all command examples are assumed to be run from within a running preprocessor container.

```
preprocess \
    --config-file /config.yml \
    {STAC_ITEM_JSON_STRING}
```

For all intents and purposes in this section, it is assumed, that the operator is logged into a shell on the `registrar` service.

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

```
alias manage.py='python3 $INSTANCE_DIR/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.

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 on the console and can be used to put it into a collection. Additionally, it is necessary to add 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 \
    -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
```

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

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

The preprocessing step aims to ensure that cloud-optimized GeoTIFF (COG) files are created to significantly speed up the viewing of a large volume 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 most of the time, 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 allows access to the metadata and catalog functions, while justifying slower rendering times can be preferred.

Monitoring ingestion can be done on the production system easily via Kibana using its query language KQL. Kibana in *Discover* mode shows a 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.

To watch for successful registrations or preprocessing campaigns, simply search for

```
event: is one of Successfully registered Product, Successfully replaced Product
```

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 registration, a query would look like this:

```
container_name: *registrar
exception: exists
```

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

```
"urn:eop:PNE:PMS__3_0.3m:ACQ_PNEO4_02520705839470"
```

Then click on an arrow on the 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).

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 by external sources via the `push` approach.

`Ingestor` service, together optionally with `sftp` service allows data ingestion to be initiated by an external trigger.

`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 the path to the object storage with a product to be ingested. It then parses this information and internally pushes it into configured Redis queue.
- Alternative: Listening for newly added Browse Report or Availability Report files on a configured path on a file system via `inotify`.

The Browse Report files need to be in an agreed XML schema to be correctly handled.

`Sftp` service enables 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:
  command:
    ["python3", "/filedaemon.py"]
```