D3.2 End Users' Platform

Document Identification			
Status	Final	**Due Date**	31/10/2018
Version	1.0	**Submission Date**	31/10/2018

Related WP	WP3	**Document Reference**	D3.2
Related Deliverable(s)	D3.1	**Dissemination Level (*)**	PU
Lead Participant	Fabien Castel (ATOS)	**Lead Author**	Fabien Castel (ATOS)
Contributors	Ugo LOPEZ, Jessica BRETAGNE, Matthieu JEAN-JACQUES	**Reviewers**	Tomas Mildorf (Plan4all)
			Marcela Doubkova (PESSL Instruments)

Keywords:

Data Analytics, Big Data, e-Infrastructure, Architecture, Design, EUXDAT

Document Information

List of Contributors	
Name	Partner
Fabien CASTEL	Atos FR
Ugo LOPEZ	Atos FR
Jessica BRETAGNE	Atos FR
Matthieu JEAN-JACQUES	Atos FR

Document History			
Version	Date	Change editors	Changes
0.1	27/09/2018	Fabien Castel (ATOS)	Table of Contents
0.2	03/10/2018	Fabien Castel (ATOS)	Updated table of contents
0.3	12/10/2018	Fabien Castel (ATOS)	Updated table of contents
0.4	17/10/2018	Fabien Castel (ATOS)	Version for partner review
0.5	30/10/2018	Fabien Castel (ATOS)	Version for quality review
0.5	31/10/2018	ATOS	Quality Review
1.0	31/10:2018		FINAL VERSION TO BE SUBMITTED

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable leader	Fabien Castel (ATOS FR)	31/10/2018
Technical manager	Fabien Castel (ATOS FR)	31/10/2018
Quality manager	Susana Palomares (ATOS ES)	31/10/2018
Project Manager	Javier Nieto (ATOS ES)	31/10/2018

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / acronym	Description
API	Application Programming Interface
CGI	Common Gateway Interface
CSW	Catalogue Service for the Web, OGC standard for data catalogue requesting
Dx.y	Deliverable number y belonging to WP x
EC	European Commission
GIS	Geographic Information System
HPC	High Performance Computing
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
LDAP	Lightweight Directory Access Protocol
NDVI	Normalized Difference Vegetation Index
OGC	Open Geospatial Consortium
OLU	Open Land Use
PaaS	Platform as a service
REST	Representational State Transfer
SAR	Synthetic Aperture Radar
SSL	Secure Sockets Layer
URL	Uniform Resource Locator
WFS	Web Feature Service, OGC standard for georeferenced object
WMS	Web Map Service, OGC standard for georeferenced coverage
WP	Work Package
WS	Web Service

1. Executive Summary

This document goes along the first version of the EUXDAT platform, implemented during the first year of the project. It presents the actual status of the ready to use features, by opposition to other documents written in the first year of the project that aims at providing a vision of the expected features of the EUXDAT platform and foresee the work that will be achieved during the whole three years of the project.

This document is focused on the end users' platform components that are deployed and ready to use. These components are deployed on a PaaS layer hosted on a cloud infrastructure. The description of this layer is not part of this document as it is described in the WP4 D4.2 deliverable.

The components described in this document fall into three categories. First, data connectors are described. These components allow users to access specific dataset with easy to use interfaces. Three connectors are presented: one for the European digital elevation model, another one for the "Open Land Use" database and the last one for the Copernicus Sentinel data through Mundi Web Service.

Secondly, components related to the end users' interface are presented. The first set of components implementing an online development environment is described. This environment constitutes the first interface reachable by EUXDAT users to interact with the resources provided by the platform. An overview of the future end users' frontend is also provided.

Lastly, the component related to the management of the EUXDAT users is presented. This component is central to the EUXDAT platform as all other components depend on it to deal with security and authentication matters.

2. Introduction

The EUXDAT End Users platform gathers a set of components that are directly accessed by the end users. The objective of this document is to provide a clear view of the implementation status of this integrated platform at the end of the first year of the project.

2.1 Relation to other project work

This document is the direct follow-up of the D3.1 "Detailed Specification of the End Users' Platform v1 ", which provides specification and design elements use as a basis to develop the platform.

It is also articulated with D4.2 "Infrastructure Platform" and D5.2 "EUXDAT e-Infrastructure" as all together these documents provide insight on the actual status of the overall EUXDAT e-infrastructure:

- D3.2 describes the implementation of the end users' platform
- D4.2 describes the status of the infrastructure platform (for both cloud and HPC resources)
- D5.2 describes the status of the overall e-infrastructure focusing on the user point of view

2.2 Structure of the document

This document is structured in 4 major chapters:

Chapter 3 describes the data connectors deployed on the platform.

Chapter 4 presents the online development environment.

Chapter 5 gives an overview of the future user's frontend.

Chapter 6 describes how users are managed on the platform.

3. Data Connectors

3.1 Digital Elevation Model

The Digital Elevation Model over Europe (EU-DEM) is a digital surface model providing elevation data at 25 meter resolution all over Europe. The data is provided in GeoTIFF in a contiguous dataset divided into 1000x1000 kilometres tiles, resulting in a total of 27 tiles of 40000x40000 pixels. The total size of the dataset is around 47GB.

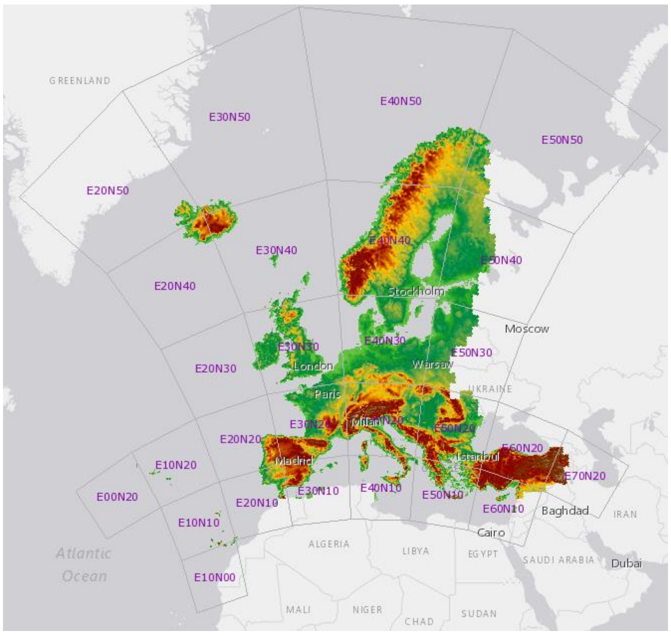

Figure 1: EU-DEM dataset coverage

To make the access to this dataset easier for the EUXDAT platform user, an API was developed on top of this data. The main idea is to abstract the tiling and storage format to the users so that they can access custom defined area in a single request.

This API is coded in Python using the Flask framework [1] and designed with the Swagger [2] tool. It takes as an input the area of interest. The API has two level of reaching for the user:

- A Python level, in this case the users can integrate a Python library to their code and use a high level access function. The data returned is analysis ready (Numpy arrays)

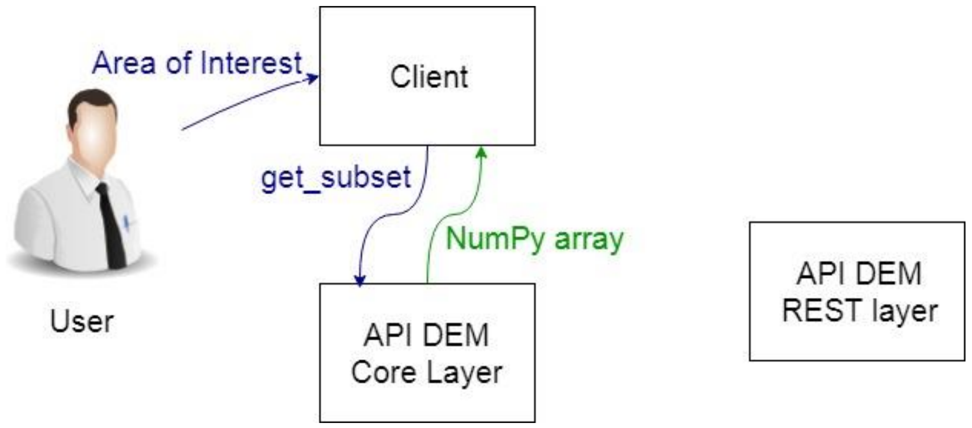

Figure 2: API DEM direct Python access

- An interoperable REST level, in this case the users can reach the REST HTTP API from any technical stack and receive JSON array data.

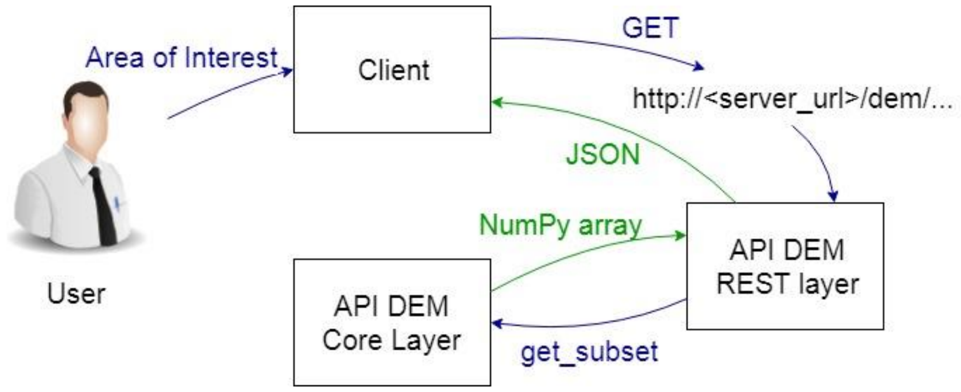

Figure 3: API DEM REST access through HTTP

The main added value provided by using this API the users do not have to deal with the tiling format of the data, this complexity becomes transparent. To access it, the data is enriched with metadata describing the spatial coverage of each tile. This metadata is stored in a PostgreSQL system. By using this information, the core layer can determine which tiles have to be accessed for a given area of interest.

Once the targeted tiles are determined, the core layer uses GDAL (through its Python library) to subset each tile, i.e. retrieve only the area required, and to merge each tile into a single array that is returned to the user. As GDAL is already integrated with Numpy it is straightforward to return Numpy arrays to the user.

3.2 Open Land Use database

A PostgreSQL/PostGIS database storing the Open Land Use data is installed on the EUXDAT platform. This database can be reached through OGC web services using the Mapnik [3] and Mapserver tools.

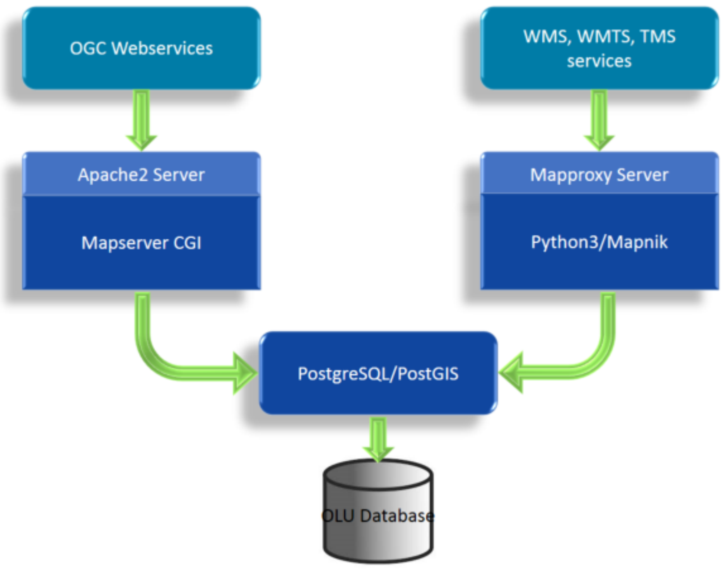

Figure 4: Open Land Use database high level architecture

Mapnik is deployed in the cluster. It is installed as a Python 3 package in a container that runs a MapProxy [4] server. It enables serving OGC web services (WMS, WMTS and TMS) and can be reached from outside using a specific URL. Higher level web services might be built on top of this service and be available later during the project.

On the other hand, Mapserver is installed in another container along with an Apache2 server. The Mapserver "Common Gateway Interface" can be reached from outside at a specific URL. The Mapserver CGI can answer to OGC requests.

Here is an example of a request:

http://<server_url>/mapserver/cgibin/mapserv?map=/var/www/html/mapserverdemo/itasca.map&SERVICE=WMS&VERSION=1.1.1&REQUEST=GetCapabilities

3.3 Sentinel data access through Mundi

"Mundi Web Services" is a Copernicus Data and Information Access Services (DIAS), meaning a service providing an easy way to access Copernicus data and information.

EUXDAT targets at the moment three collections provided by Mundi:

- Sentinel-1 L1A and L1B, providing all-weather, day and night radar imagery for land and ocean services,
- Sentinel-2 L1C or L2A, providing high-resolution optical imagery for land services,
- Sentinel-3, providing high-accuracy optical, radar and altimetry data for marine and land services.

Other satellite missions (Landsat for instance) will be targeted in the future. The provided files accessible by Mundi Services can be in three different formats: GeoTIFF for Sentinel-1, JPEG2000 for Sentinel-2 and NetCDF for Sentinel-3.

Sentinel products are accessed through a data catalogue, exposing CSW and OpenSearch APIs. A complete set of metadata is retrieved from these APIs describing the products (data level, spatial and temporal coverage…) as well as the path on the Mundi object storage to reach the actual data. Data is then accessed using the S3 command line tool, the AWS SDK or higher level libraries.

An API that enables users reaching the Mundi services in an easier way is implemented on the EUXDAT platform side. A REST API is exposed to the users with simple routes and settable parameters. This API is coded in Python using the Flask framework [1] and designed with the Swagger [2] tool. It is divided in two modules (see Figure 5):

- A Flask front API defining URL format and the needed parameters to reach a targeted product. This part also has the function to describe and set documentation for the Swagger API, allowing users to directly test the entire API without any notebook or frontend.
- A backend module parsing and formatting user's parameters to send it to Mundi Services. This part of the API is also in charge of translating the XML response from Mundi Services to a JSON response. It then downloads a product and stores it in a local folder for a personal use.

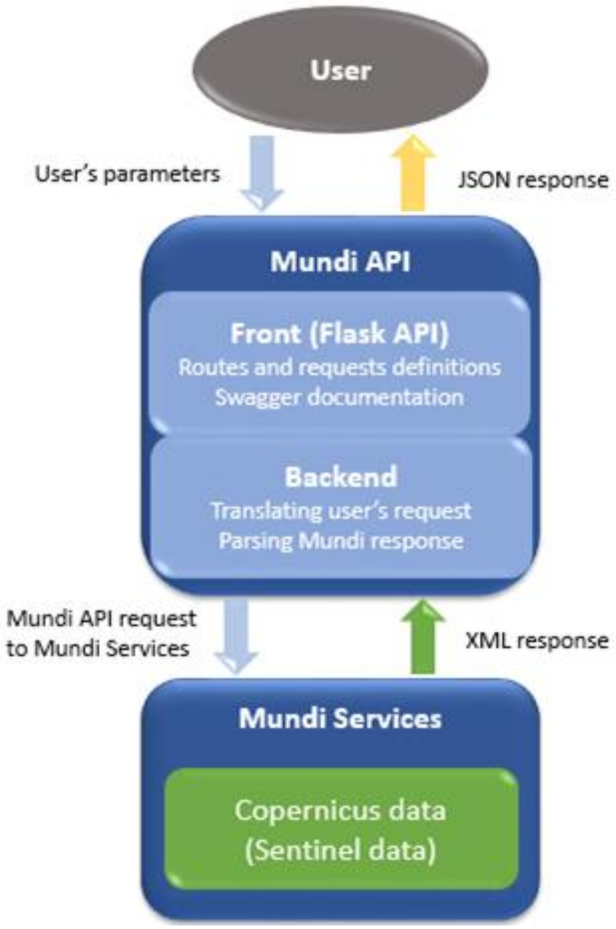

Figure 5: Mundi API in the EXUDAT platform

4. Development environment

EUXDAT implements a solution to provide a secure, pre-configured and individual in-cluster development environment to each developer user, based on the Jupyter Notebook [5] and Jupyter Hub [6] technologies from the open source Jupyter Project.

4.1 The development environment: Jupyter Notebook.

The Jupyter Notebook is a server that allows interactive scripting and code execution for numerous languages (Shell, Python, R ...). The user connects to the server with a web browser which will execute a JavaScript web client with a user-friendly interface that connects to the Notebook server. The server having access to the resources of the system where it's installed can executes the code transmitted by the client. Scripts are stored as special resources called notebooks, allowing for edition, commenting, step-by-step execution and result presentation. The main interest of this system is to provide to a user a functional pre-configured development environment and an access to cluster resources.

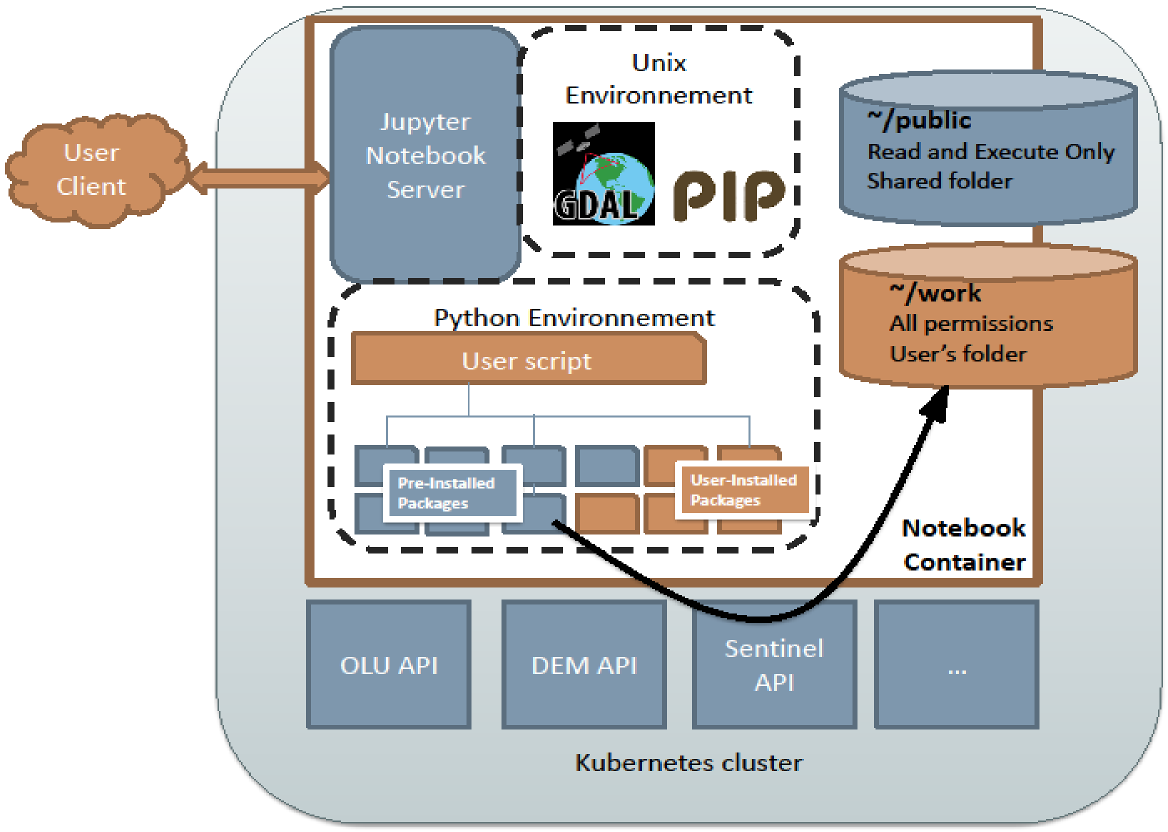

Figure 6: Jupyter Environment Deployment in the Kubernetes cluster

As depicted in Figure 6, the development environment is a Notebook server running in a containerized UNIX system with a Python development environment and a set of tools (GDAL, Pip). The user has an access to script examples in his public folder, and can store permanently its resources in his/her work folder. The system being run in the EUXDAT cluster can access resources like the Sentinel API that allows searching Sentinel 1 and 2 products and download them into the user workspace. Custom python libraries simplifying this process are already installed and their use illustrated by scripts available in the "public" folder.

The system executing a user Notebook server is containerized: it is a Docker image running into a Kubernetes pod inside the EUXDAT cluster. The image running a user server contains a Python environment with pre-installed Python libraries and a UNIX terminal with limited access rights, allowing the use of satellite images processing tools like GDAL, as well as the customization of the development environment through the addition of Python library with the command tool Pip.

The Docker image is updated as the users express new needs for their development environment. A great number of programming language interpreters are available and can be added to the environment development over user demand.

Modifications performed by the user in his/her Jupyter container are lost each time the container stops. In order to save his/her data, mount points are made between the container filesystem and the cluster NFS filesystem. A "Public" folder contains notebooks that are example scripts accessing cluster resources (OLU connector, DEM API, Sentinel API...) or illustrate the provided tools (NDVI calculation with GDAL for instance). The user has "Read and Execute only" rights on this folder as it is intended to be shared by all users. A "Work" folder, belonging to the user is intended to store the resources it uses as well as the scripts it writes.

The Notebook container runs inside the Kubernetes cluster and can thus access resources through the network that are not exposed to the outside world. The user has limited rights over the container system to limit the risk of harmful behaviour, this is ensured by the fact that the Jupyter Notebook server process is not running as "root" user but as a "Jovyan" user, thus giving little privilege escalation possibility to the user. The Notebook pod has no access to the environment variables usually given to pods that allow contacting the Kubernetes API to mitigate the risk of cluster exploitation.

Different execution kernels can be installed as plugin of Jupyter allowing users to develop their code using various languages, frameworks and libraries. Targeted languages and additional libraries are listed in the following table:

Table 1: Jupyter main kernels

Kernel Name	Coding Language	Language Version	Additional libraries
Apache Toree - PySpark	Python	3.6.5	GeoPySpark Cf Python 3 kernel libraries (common environment)
Apache Toree - Scala	Scala	2.11.8	GeoTrellis, GeoTrellis-ETL, GeoMesa
Apache Toree - SparkR	R	3.4.1	SparkR Cf R kernel libraries (common environment)
Python 3	Python	3.6.5	rasterio, netCDF4, NumPy, SciPy, pandas

4.2 The multi-user solution: Jupyter Hub over Kubernetes

Each developer has its own dedicated Jupyter Notebook server, with 4GB memory and 2 CPUs resource limits. The Jupyter Notebooks are spawned on-demand from the user and are shut down after a day of inactivity to spare cluster resources. The Jupyter Hub is the tool that allows multi-user support, authentication and Notebook server spawning through Kubernetes API.

The Jupyter Hub runs into a Pod corresponding to a Replication Controller on the Kubernetes cluster. Any traffic over SSL directed to https://platfom.euxdat.eu/hub is offloaded by the Nginx/Ingress controller and routed to the JupyterHub Pod. This pod will ensure user login, single-user Notebook spawning and user redirection to its Notebook.

The Jupyter Hub pod contains two main elements: the multi-user Hub (written using the Tornado web framework and asynchronous networking library) and a configurable http proxy (node-http-proxy). The proxy forwards by default all requests to the hub. The hub handles login and spawns single-user Notebooks on demand. The hub configures proxy to forward requests with user URL prefixes and valid token to the single-user notebook servers.

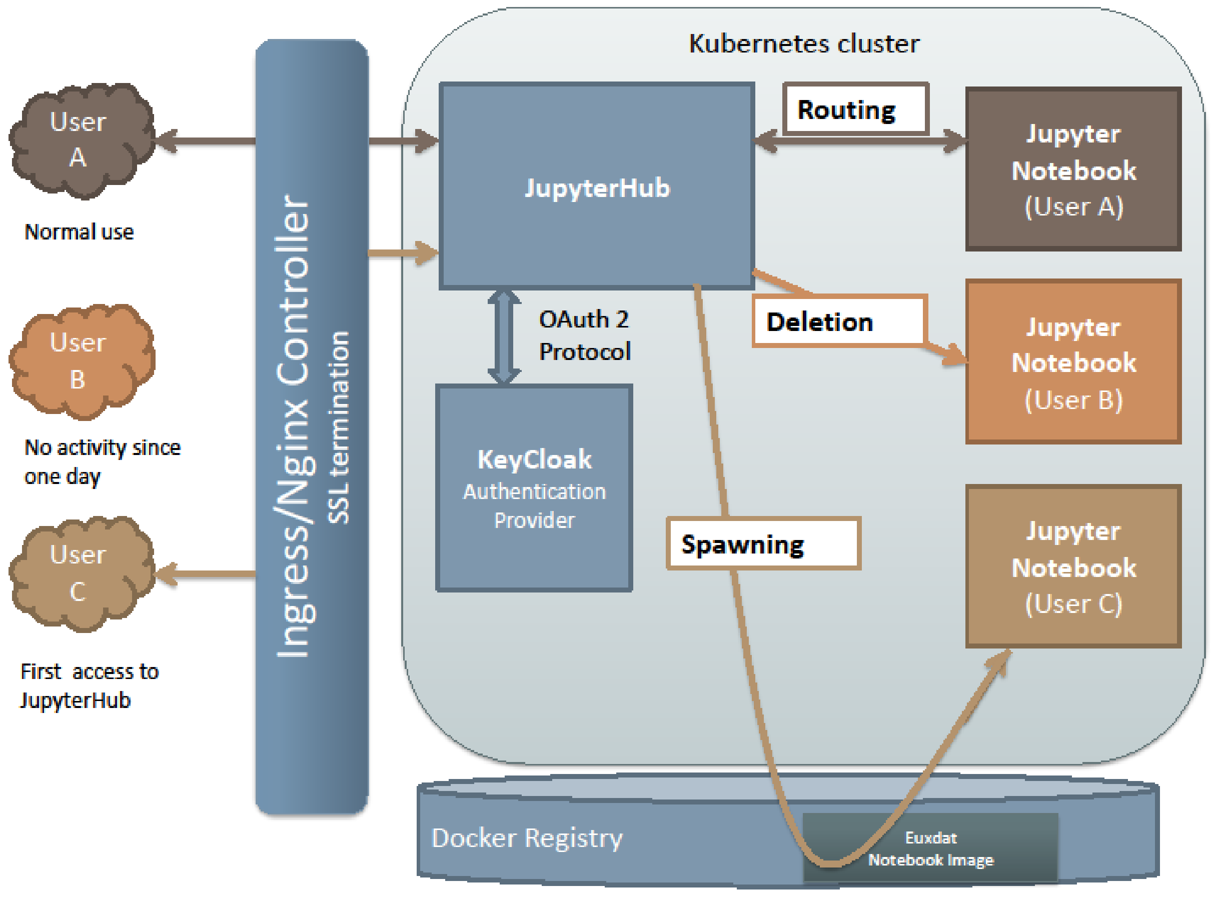

Figure 7: Jupyter Hub deployment structure

The Jupyter Hub Pod is the component ensuring the authentication of the user against the EUXDAT identity provider (Keycloak). It implements the main functions of Notebook server management: spawn a Notebook from the cluster Docker registry image when a user connects to the hub and doesn't have an active Notebook, route the requests of the user to its existing Notebook, and delete unused Notebooks after a certain time of inactivity to free cluster resources.

The authentication and spawning modules are off-the-shelf Hub plugins provided by the Jupyter project team, as well as the service that shut down idle Notebooks after a certain time of inactivity.

Authentication through OAuth 2 is ensured by the OAuthenticator plugin. It is configured to work as an OAuth2/OpenID client dedicated to the JupyterHub registered in identity provider of the EUXDAT cluster. When logging in to JupyterHub, the user is redirected to the login page of the EUXDAT identity provider (see User Management section), which sends an authorization token to the JupyterHub. The Hub never has direct knowledge of a user's credentials.

The spawning process, which consists in sending a pod creation request to the Kubernetes API, is made by the KubeSpawner plugin from the Jupyter Project. The KubeSpawner loads from the Pod environment variables the configuration, identification and certificates necessary to contact the Kubernetes API. These environment variables are provided by Kubernetes to the JupyterHub Pod as it is given a Kubernetes Service Account, i.e. it has been afforded rights to watch Kubernetes events, create and delete pods in a certain Kubernetes Namespace dedicated to the Notebooks pods.

For each user, the Notebook pod is configured to mount two volumes to the Notebook, a "public" folder shared with all users, where users have no modification rights, and a "work" folder where the user has all permissions. This volume corresponds to a folder with a unique name created by Kubernetes on the cluster NFS filesystem the first time a user launches its Notebook. The permissions on this folder are modified each time a Notebook spawns to correspond in the cluster filesystem to the "Jovyan" user id and group id. The image run into the Notebook pod is downloaded from the cluster Docker registry.

The Jupyter Hub UI is written with Jinja [7], a webpage templating framework for Python, and the templates are customized to correspond to our needs.

5. Users' frontend

Vue.js is the framework which will be used for developing the main features of the frontend. Vue.js is a progressive framework which uses a MVVM pattern with focus on the view-model, connecting view and model with two-way reactive data-binding (Figure 8).

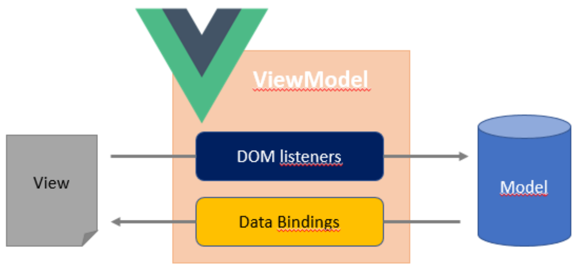

Figure 8: Vue MVVM pattern.

One of the most important features are the componets, the Vue application can be considered as a tree of self contained reusable components (Figure 9). A component is a Vue instance with predefined options which extents a HTML element with encapsulated reusable code. The component can be encapsulated as a single file (.vue).

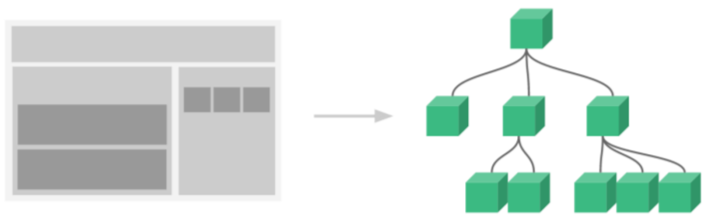

Figure 9: Vue components.

Source: https://vuejs.org/images/components.png

One of the main components which are integrated in the frontend application is the VueLayers component. VueLayers brings the powerfull OpenLayers API to Vue framework and will be used for the map viewer. This component is still in a testing phase and potentially needs to be extended or replaced for the OpenLayers native library to fulfill all the map requirements.

Vue also offers core plugins for specific features such as the routing one. For the frontend, the Vue-Router plugin is used. This plugin provides dynamic routing matching with params and query params as well as nested or named routes.

6. User Management

User management in the EUXDAT platform is ensured by reusing an off-the-shelf component deployed on the cluster, called Keycloak. Keycloak is an open source Identity and Access Management solution providing a simple Single-Sign-On solution. Users authenticate with Keycloak rather than doing so on each individual EUXDAT component. This means that the components don't have to deal with login forms, authenticating users, and storing users. Once logged-in to Keycloak, users don't have to login again to access a different application.

Keycloak is able to connect to various sources of authorized users (LDAP, Active Directory, RDBMS "Social login"...). In the EUXDAT configuration it is configured to store users in a dedicated PostgreSQL database pod saving its content on the cluster filesystem. New users are added by a Keycloak administrator through the management console or through a dedicated API. User accounts are initialized with a random password that the user will change at its first connection. The admin has no access to user passwords by design. The Keycloak server can be connected to an SMTP server to automatize the registration and password reset processes.

EUXDAT components are connected to Keycloak through the standard OAuth 2 protocol. OAuth 2 clients such as the Jupyter Hub component are registered by an administrator over the Keycloak management interface or API, and are given a client ID and secret, corresponding to an origin URL and a call-back URL.

When trying to authenticate a user, the client authenticates itself to Keycloak with its Id and Secret, the origin URL and the call-back address given by the client is verified against Keycloak registered information. When client identity is ensured, the user is redirected to a login page. When user identity is ensured Keycloak calls back the client with an authorization and refresh token. Authorization token can be used by the client to get additional information about the user. Refresh token is used to generate a new token for the user without it having to go through the login process.

7. Conclusions

This document has presented all the components deployed in the first version of the EUXDAT end users' platform.

This first version will serve as a basis for the developments of the pilots, i.e. the applications and APIs that the EUXDAT end users will access. These pilot applications will use the provided data access APIs to generate added value data that will then be serve through higher level APIs and specific frontend applications.

On top of the provided APIs and next to the development environment, a frontend component will be implemented providing high level and easier to use feature than the development environment, in particular for users having low technical background.

The set of data connectors will be enriched in the platform V2 to access more datasets required by the pilots: UAV hyperspectral data, meteorological data, land thematic data like soil maps and hydrology, other satellite imagery data like Landsat…

8. References

[1] The Flask framework official web site, http://flask.pocoo.org/, retrieved 16/10/2018

[2] The Swagger project official web site, https://swagger.io/, retrieved 16/10/2018

[3] The Mapnik project official web site, https://mapnik.org/, retrieved 16/10/2018

[4] The MapProxy project official web site, https://mapproxy.org/, retrieved 16/10/2018

[5] The Jupyter project official web page, http://jupyter.org/, retrieved 16/10/2018

[6] The JupyterHub sub-project source code repository, https://github.com/jupyterhub/jupyterhub, retrieved 16/10/2018

[7] Jinja, a template engine for Python, http://jinja.pocoo.org/, retrieved 16/10/2018