
SCPlatform Documentation

Release 0.0.1

SCC Lab

Dec 23, 2021

Contents

1	Introduction	1
2	Digital Hub Architecture	3
2.1	Logical View	4
2.2	Key Concepts and Characteristics	6
3	Service Hub	9
3.1	Overview	9
4	Data Hub	47
4.1	Overview	47
4.2	Data Wrangling	47
4.3	Data Management	53
4.4	Data Storages	76
4.5	Data Processing and Analysis	79
4.6	Data Interfaces	83
4.7	Data Access	107
5	IoT Layer	113
5.1	ThingsBoard	113
6	Visualization, Dashboards, and Data Applications	115
6.1	Grafana	115
6.2	Cyclotron	120
7	Community Layer	155
7.1	Service Portal	155
7.2	Data Portal	156
7.3	Gamification Engine	157
8	Platform Installation	161
8.1	Prerequisites	161
8.2	Install core components	161
8.3	Install platform components	163

CHAPTER 1

Introduction

Digital Hub represents a open source techonology platform integrating various software components for managing, elaborating, and exposing data and services in a standardized and aligned manner. It is built upon a wide range of Open Source projects and open standards.

Digital Hub Architecture

The goal of the Digital Hub platform is to address the following requirements:

- Possibility to integrate and manage data originating from **heterogeneous** data sources. This possibility refers to both **alphanumeric** data and **geographical** data; data coming both from standardized and **interoperable** sources (eg, Web services, open data formats) and from **legacy** and non-standardized sources (eg, RDBMS, local files, proprietary systems), but also data from sensors and devices in the Internet of Things (IoT) domain.
- Offer tools and solutions to enrich data with **metadata** and semantic **relationships**, enabling the creation of complex aggregations and data traceability. The platform aims at guaranteeing, in particular, the alignment and compatibility of the data with respect to the domain model (e.g, mmaster data), and reuse the reference models and interchange protocols. Some examples concerning the public administration domain refer to
 - ANPR (National Registry of Resident Population)
 - Standardized ontologies and datasets of AGID DAF
- Possibility to elicit and catalogue appropriately enriched and annotated data.
- Provide tools and components necessary for **analysis**, **processing**, **aggregation**, and **visualization** of the information present in the platform. Doing these activities, it is necessary to maintain the right balance between the efficiency and performance of the solution on the one hand and the need to replicate the data within the platform.
- **Expose** information and integrated functionalities using **standard** and **open** protocols. This requirement concerns both data and services specific to the domains of interest of the project and those exposed by the platform itself. Use of open standards is fundamental for the creation of value-added services and applications, for the reuse of services in different contexts, and to guarantee the decoupling of Digital Hub compared to the applications built above.
- Ensure appropriate level of security and access control in the management and display of data and services. In particular,
 - Ensure that different actors (organizations, public bodies) who manage data through Digital Hub do not interfere with one another as if they were managing isolated instances of the platform (**mutli-tenancy**).
 - Ensure that access to data and services is protected with open security protocols and the state of the art.
 - Manage access to data of end users by the operator and third parties in a clear and safe manner, respecting national and international policies and regulations (eg GDPR).

Furthermore, non-functional requirements concerning the Digital Hub platform refer to

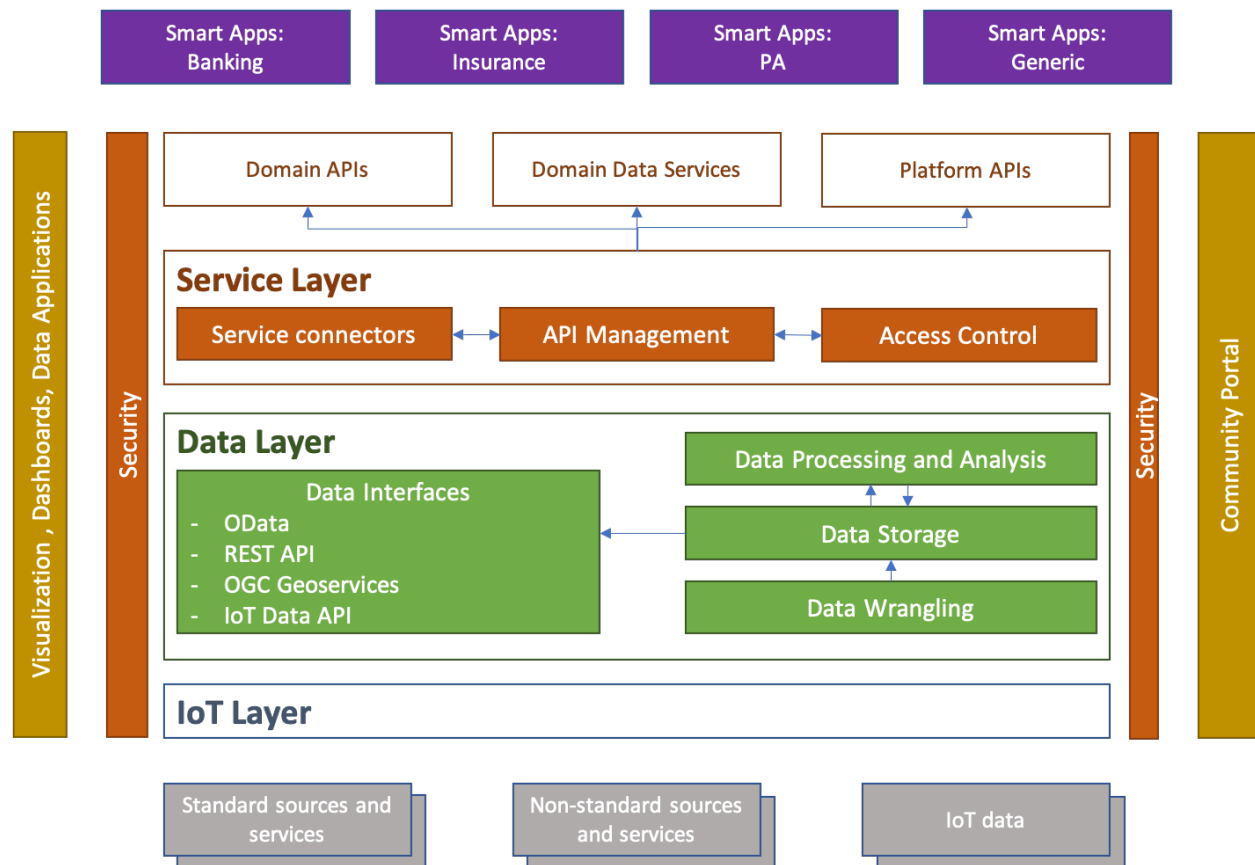
- Be an **open platform** to allow access to the artifacts exposed by the platform and to allow everyone to interact with the platform.
- Permit extensibility of the platform with the new innovative tools and solutions, therefore **open innovation**.
- Enable re-use of the platform and its components through an **open source** paradigm.

2.1 Logical View

The logical architecture of the DigitalHub platform is represented in the following diagram. In this architecture the fundamental elements are organized in two layers, in particular Service Layer (Service Hub) and Data Layer (Data Hub). These two parts of the platform are supported by the IoT layer (which facilitates the integration of data from “Internet of Things”: sensors, devices, etc), from data **visualization and consumption tools** (dashboards, data apps), and **community portal** that delivers the data and services to the community (developers, consumers).

The data and services that the platform “consumes” can therefore characterize the different and heterogeneous sources. This includes IoT data, standardized services and data, non-interoperable systems (eg, databases, files, FTP, ...).

Data and services processed, surveyed, and enriched by the plan are exposed to applications using open and standard data protocols and models. In this way, the platform enables integration not only with local Smart Apps, but also with external reference systems, such as [DAF](#).



2.1.1 Data Hub

Data Layer consists of a set of tools that are necessary for the extraction, processing, aggregation, census, and representation of data in an open and standardized manner. In particular, these tools are structured as:

- **Data Wrangling.** By this name we mean the tools for extracting, preparing, processing, and harmonizing data taken from non-interoperable and / or non-standard sources. Here processing can be activated in “pull” mode (periodically accessing external sources such as relational DBs, FTP, files, etc) or “push” (subscribing to data streams as queues or data streams).
- **Data Storage.** Data storage tools could be used to preserve data from non-interoperable sources that cannot be queried “online” (for performance reasons, technological limits, etc.) and therefore must be replicated within the Data Hub. This storage can be interpreted as a repository of mass data, Data Lake, with formats that are not directly usable (Big Data, multimedia, etc.), or as “online” storage where data can be easily interrogated quickly and effectively.
- **Data Processing and Analysis.** Tools of this Group are used to provide the advanced processing capabilities of raw data taken from interoperable and non-interoperable sources. This analysis can also be applied to “big data”, with the complex logic created for a specific domain, specific application, and specific scenario. The result of this processing can be used directly or transferred to the data storage for future processing and queries.
- **Data Interfaces.** These tools aim to display data using standard data protocols and models. As for the protocols, they following ones are considered
 - **OData:** an Internet-based open protocol for sharing, searching, and querying based on XML and JSON data formats. OData facilitates access and interrogation of alpha-numeric table data coming, for example, from relational DBs.
 - **REST API:** data display through REST services for advanced operations and optimized for particular and effective uses.
 - **OGC Geoservices:** represent geographic information through standard Web services for data, maps, and data processing.
 - **SensorThings API:** display of data collected through the IoT layer from different sensors and devices in standard mode for interrogation.

As for the data models, the objective is to use a set of reference models that have been defined by different bodies in specific domains, such as public administration, banking, insurance, mobility, etc. The data aligned with these models constitute the master data of the platform: the certified, validated, and compatible data with external systems. Examples of data models of this type refer to

- Ontologies and data models proposed by the Agency for Digital Italy (AGID)
- Data models of mobility and public transport (eg, GTFS)
- Data model behind interoperability services and bank payments (PSD2).
- Citizens’ personal data (ANPR)
- Models for interoperable geographic datasets (INSPIRE)

The result of application of the Data Hub tools is exposed by the platform through standard interfaces to Service Hub and also to external tools for visualization or for use by the community side. The latter, in particular, occurs through the data catalog which is used to expose the Open, Semi-open, and also Closed Data to third parties in a protected manner.

2.1.2 Service Hub

Service Hub offers the tools to publish, manage, and govern the services (API) exposed by the domain and / or by the components of the platform in a protected and safe way. For this the tools are organized in

- **API Management.** In this category there are tools for documenting APIs, configuring access to them, defining exposure modes and protocols, publishing them, and monitoring them. The published services are displayed in standard and open mode, to facilitate access and use. The API management functionality follows the interoperability model defined by the three-year plan for ICT in the PA and concerns the definition of the service level, version traceability and service life cycle, documentation, access control management, audit, analytics, etc.
- **Access Control.** The access control system allows to manage various types of users (developers and end users), their access through various channels (eg, social networks), association of permissions and authorizations to the platform, management of organizations that use platforms, protocols of access to services, etc. The access control system is based on the reference security protocols for API access, such as [OAuth2.0](#).
- **Service Connectors.** Exposure of some services and functionalities very often must face the problem of managing services that are not compatible with Open Services standards. This concerns, in particular, access to legacy systems, services protected with “weak” or non-standard protocols, redundant or non-interoperable data. Service connectors offer the possibility to mediate access to these services without significant effort. These connectors can be added with enterprise integration tools (eg, flow brokers) or as independent applications. The open services displayed and managed by Service Hub are registered through an open catalog of services and can be directly used by the various applications.

2.2 Key Concepts and Characteristics

To describe the various components of the platform, it is important to define some common features that each component must have to be an integral part of the same platform and its deployment and use model. These features refer to the interoperability model, organizational and multi-tenancy model, security, access control, and code distribution model.

2.2.1 Interoperability and Protocols

Use of standard protocols at all possible levels of the platform is a fundamental requirement to guarantee extensibility and to avoid “vendor lock-in” of a solution that must be generic, customizable, and easy to integrate.

As for IoT data, the platform is based on various communication standards. In particular,

- REST and [MQTT](#) APIs for generic solutions in the field of heterogeneous sensors and devices, where communication takes place via the Internet connection.
- [LoraWAN](#) protocol for urban scale networks.
- [NB-IoT](#) protocol for networks based on infrastructure of mobile operators.

Regarding data exposure, the platform includes:

- Use of the [OData](#) protocol for the display of generic data (tabular, relational) that standardizes a query language and is based on REST http as transport protocol.
- REST protocol for exposure of data in specific cases, where the use of protocols as OData is not sufficient or is not effective.
- [OGC](#) services (WMS, WFS, etc) for displaying geographical data, display layers, etc.
- [SensorThings API](#): protocol for display of IoT data that standardizes the sensor model, surveys, data history, etc.

As for the display of services, the platform is based on

- **‘Open API Initiative < <https://www.openapis.org/>>’**: the model for standardizing documentation, interface, and data model for REST and JSON-RPC services.

For displaying the data catalogs, the platform follows the de-made standard for Open Data, created by the [CKAN](#) platform.

For generic reference data models or specific domains, the platform intends to use the solutions proposed, such as

- [DAF Ontologies](#) - data models proposed for AgID's three-year plan.
- [Payment Service Directive \(PSD2\)](#) API model for interoperable banking services.

2.2.2 Multitenancy and Organizational Model

Multi-tenancy is a necessary requirement for exposure and use of the platform in Software-as-a-Service mode when multiple organizations and users use the components and features of the platform without having access to the data and configurations of others. In this way Digital Hub can be supplied as an easily scalable Platform-as-a-Service on Cloud.

To meet this requirement, every component of the platform must

- Provide data model to represent different customers (tenants), their data / processes / configurations independently and in isolation. In this way, each client works with the platform as if it were a “standalone” platform.
- Provide the security layer that, following the model above, binds the access of the different customers exclusively to its data guaranteeing the complete isolation of the tenants.
- Provide support to create and manage tenants in a programmable way through API.

The organizational model for the platform provided as SaaS requires that the user operating the components at a cross access to the different components within the scope of his tenant. In addition, multiple users can use within a tenant, with different roles for different components as well. The role model that the platform must support could therefore include:

- Grouping of users in organizations, where an organization has access to one or more tenants of the platform.
- Possibility to associate different roles to the users of an organization.
- Different role models for different components of the platform depending on the specific features of a software component.

Implementation of these requirements also provides access to components in Single Sign-On mode, when the user has a transversal identity at the platform level. All components must have a centralized and standardized way to recover the identity of the user and his roles. Since access to features, data, and services exposed by the components occurs not only through the Web management applications of the different components, but also through APIs, it is important that access control is based on a standard protocol. For this role the platform adopts the [OAuth2.0](#) protocols (for access authorization) and [OpenID Connect](#) (for identity recovery).

Realization of roles within multi-tenant components is based on a model of roles that are contextualised to the concept of tenant (space) and allow to form a hierarchy of tenants. So a role is represented with

- context: the settings it belongs to (eg, API Management context)
- space: the tenant within a specific context (eg, tenant “MyOrg” in the API Management context)
- role: the specific role for the context (eg, “API Publisher” role in the API Management context).

2.2.3 Deployment Model

The platform is implemented as a set of software components (Open Source projects adopted or components implemented ad-hoc) that are brought together in order to solve common problems and to address the specified requirements. Except some core elements, the components are independent and may be used in isolation, even through the real power

of the platform is in chaining them together to address various scenarios (e.g., elaboration and visualization of IoT data, integration of legacy sources, data alignment, service exposure, etc).

A shared set of the core components upon which the others rely deal with the user and tenant management, authentication and access control (also with Single Sign-On support). All the other components have a loose integration with those in order to achieve

- Single Sign-On for the components having web-based UI
- Role and tenant identifications for the user operating the components
- Multi-tenant data / instance isolation based on the user roles

Depending on the internal structure and implementation of the component, this integration may be achieved as

- by the component itself interacting with the core components in order to extract the roles and tenants and to map those onto internal multi-tenant representation, when the component supports multi-tenancy out of the box.
- through the component API in order to create users and tenants, when the component supports multi-tenancy out of the box.
- through instantiating isolated instances of the components when the multi-tenancy is notnatively supported.

The recommended platform deployment is Cloud-native: it is based on the [Docker](#) containers of components and their orchestration using [Kubernetes](#) as an orchestration engine. It is, however, possible to perform deployment manually on premises. The details about the platform installation can be found [here](#).

3.1 Overview

3.1.1 Authentication and Authorization Controller (AAC)

This module exposes the OAuth2.0 protocol functionality for user authentication and for the authorization of resource access (e.g., APIs). The module allows for managing user registration and for authenticating user using social accounts, in particular Google+ and Facebook. Besides, this component manages the role assignment to users used in the resource access authorization.

Data Model

Users

In AAC the user management may be federated with the other Identity Providers. Apart from the built-in user management, where the user is identified by the registration email and authenticated by password, it is possible to authenticate with external standard-based IdPs, e.g., OAuth2.0, SAML, etc.

Each user is, therefore, associated with the account type used at the authentication. The users managed internally, have `internal` account type, the users authenticated with Google, have `google` type, etc. The mechanism for defining different IdP depends on the specific protocol; all the supported IdPs (authentication authorities) should be declared in authority mapping file (`it/smartcommunitylab/aac/manager/authorities.xml`), where for each IdP it is necessary to specify

- identifying attributes (used to uniquely identify the user in this IdP)
- list of supported attributes specific to IdP. Some of the attributes may be annotated in order to extract common values (e.g., name and surname)
- mapping between the IdPs to automatically map the same user across IdPs using common attribute values (e.g., internal email and Google email).

Common user information is captured by the basic user profile, while account information associated with the user is represented with the account profile data (see., Profile API). The standard ways to capture these attributes is represented with the OpenID Connect Claims extracted from the user profile, account, and role information.

Roles

In AAC the roles are contextualized in order to support various multitenancy scenarios. The general idea is that the roles are associated to the role *spaces* uniquely identified by their namespace and managed by the space *owners*. A user may have different roles in different spaces and the authorization control for individual organizations, components, and deployment may be performed within the corresponding space. More specifically, each role is represented as a tuple with

- `context`, defining the “parent” space of roles (if any)
- `space`, defining the specific role space value. Together with the context form the unique role namespace
- `role`, defining the specific role value for the space.

In this way, the spaces may be hierarchically structured (e.g., departments of an organization may have their own role space within the specific organization space).

Syntactically, the role is therefore represented in the following form: `<context>/<space>:<role>`. To represent the owner of the space the role therefore should have the following signature: `<context>/<space>:ROLE_PROVIDER`.

The owner of the space may perform the following functionalities:

- associate/remove users to/from the arbitrary roles in the owned spaces (including other owners).
- create new child spaces within the owned ones. This is achieved through creation of the corresponding owner roles for the child space being created: `<parentcontext>/<parentspace>/<childspace>:ROLE_PROVIDER`.

The operation of user role management and space management may be performed either via API or through the AAC console.

Some of the role spaces may be pre-configured for the AAC. These include:

- `apimanager`: Role space for the API Manager tenants.
- `authorization`: Role space for managing the Authorization API grant tenants.
- `components`: Role space for managing tenants of various platform components.

User Claims and Claim Management

Both for the OpenID connect User Info endpoint and for the JWT access tokens, AAC supports a set of claims representing the user data. The set of returned claims depend on the scopes requested by the client app by default or during the authorization phase. To obtain the access to these claims, the user will be requested an explicit approval.

AAC manages the following standard OIDC claims:

- `sub` (scopes openid, profile, profile.basicprofile.me)
- `username` (scopes openid, profile, profile.basicprofile.me)
- `preferred_username` (scopes openid, profile, profile.basicprofile.me)
- `given_name` (scopes openid, profile, profile.basicprofile.me)
- `family_name` (scopes openid, profile, profile.basicprofile.me)

- email (scopes openid, profile, email, profile.basicprofile.me)

Additionally, AAC supports the following custom claims

- accounts: represent the associated user accounts (requires profile.accountprofile.me scope).
- authorities: list of the roles associated to the user (requires user.roles.me scope).
- groups: list of role contexts the user has a role (requires user.roles.me scope)
- resource_access: roles associated to various role contexts (requires user.roles.me scope).

** Claim Customization **

When a client app requires specific custom claims to be presented, it is possible for the client app to customize the representation of the user claims (but not the predefined ones like sub, aud, etc.). The mapping is defined in the client app configuration **Roles&Claims** section as a JavaScript function that should provide the customized set of claims given the pre-defined ones.

In this way, a client app may map predefined claim values to the expected ones; reduce the number of claims or add new ones.

** Roles Disambiguation **

Frequently, different multitenant application does not allow the user to belong to more than one tenant. AAC, however, does not have this restriction and the user may have roles in different contexts associated to that application. To avoid the customization of these applications to deal with the users associated to multiple tenants, AAC allows for the role disambiguation during the authorization step. That is, after authenticating the user, AAC will ask (together with the request for the scope approval) the user to select a single tenant for the required role context.

To configure this behavior, the client app should list the contexts, for which the user should be asked to select (**Roles&Claims** configuration of the client app). For example, if the client app is associated with the context components/grafana, the context should be specified in the **Unique Role Spaces** list. During the authorization request, if the user is associated with more than one space within that context, he/she will be asked to select a single value to proceed.

User Guide

Using AAC for Authentication

This is a scenario where AAC is used as an Identity Provider in a federated authentication environment.

To enable this scenario, AAC exposes OAuth2.0 protocol. Specifically, it is possible to use *OAuth2.0 Implicit Flow* as follows.

Register Client App

Create Client App with AAC developer console (/aac/). To do this

- Login with authorized account (see access configuration above);
- Click *New App* and specify the corresponding client name
- In the *Settings* tab check *Server-side* and *Browser access* and select the identity providers to be used for user authentication (e.g., google, internal, or facebook). Specify also a list of allowed redirect addresses (comma separated).
- In the *Permissions* tab select *Basic profile service* and check *profile.basicprofile.me* and *profile.accountprofile.me* scopes. These scopes are necessary to obtain the information of the currently signed user using AAC API.

If API Manager is engaged, create a new API Manager application and subscribe the AAC API.

Activate Implicit Flow Authorization

This flow is suitable for the scenarios, where the client application (e.g., client part of a Web app) makes the authentication and then direct access to the API without passing through its own Web server backend. This allows for generating only a token for a short time period, so the next time the API access is required, the authentication should be performed again. In a nutshell, the flow is realized as follows:

- The client app, when there is a need for the token, emits an authorization request to AAC in a browser window. `https://myaac.instance.com/aac/eauth/authorize`. The request accepts the following set of parameters
 - `client_id`: the `client_id` obtained in developer console Indicates the client that is making the request. The value passed in this parameter must exactly match the value in the console.
 - `response_type` with value `token`, which determines if the OAuth 2.0 endpoint returns a token.
 - `redirect_uri`: URL to which the AAC will redirect upon user authentication and authorization. The value of this parameter must exactly match one of the values registered in the APIs Console (including the `http` or `https` schemes, case , and trailing `'/'`).
 - `scope`: space-delimited set of permissions the application requests Indicates the access your application is requesting.
- AAC redirects the user to the authentication page, where the user selects one of the identity providers and performs the sign in.
- Once authenticated, AAC asks the user whether the permissions for the requested operations may be granted.
- If the user accepts, the browser is redirected to the specified redirect URL, attaching the token data in the url hash part:

```
http://www.example.com#access_token=025a90d4-d4dd-4d90-8354-779415c0c6d8&token_type=Bearer&expires_in=3600
```

- Use the obtained token to obtain user data using the AAC API:

```
GET /aac/basicprofile/me HTTPS/1.1
Host: aacserver.com
Accept: application/json
Authorization: Bearer 025a90d4-d4dd-4d90-8354-779415c0c6d8
```

The result of the invocation describes basic user properties (e.g., `userId`) that can be used to uniquely identify the user.

Using AAC for Securing Resources and Services

In this scenario the goal is to restrict access to the protected resources (e.g., an API endpoint). Also in this case the scenario relies on the use of OAuth2 protocol. The two cases are considered here:

- The protected resources deal with user-related data or operation. In this case (according to OAuth2.0), the access to the API should be accompanied with the `Authorization` header that contains the access token obtained via Implicit Flow (or via Authorization Code Flow).
- The protected resource does not deal with user-related data or operation and is not performed client-side. In this case, the access to the API should also be accompanied with the `Authorization` header that contains the access token obtained via OAuth2.0 client credentials flow.

The protected resource will use the OAuth2.0 token and dedicated AAC endpoints to ensure that the token is valid and (in case of user-related data) to verify user identity. If the token is not provided or it is not valid, the protected resource should return 401 (Unauthorized) error to the caller.

Generating Client Credentials Flow Token

In case the access to the non user-resource is performed, it is possible to use access token obtained through OAuth2 client credentials flow. In this flow, the resulting token is associated to an client application only.

The simplest way to obtain such token is through the AAC development console: on the *Overview* page of the client app use the *Get client credentials flow token* link to generate the access token. Note that the token is not expiring and therefore may be reused.

Alternatively, the token may be obtained through the AAC OAuth2.0 token endpoint call:

```
POST /aac/oauth/token HTTPS/1.1
Host: aacserver.com
Accept: application/json
client_id=23123121sdsdfasdf3242&
client_secret=3rwrwsdgs4sergfdsgfsaf&
grant_type=client_credentials
```

The following parameters should be passed:

- grant_type: value client_credentials
- client_id: client app ID
- client_secret: client secret

A successful response is returned as a JSON object, similar to the following:

```
{
  "access_token": "025a90d4-d4dd-4d90-8354-779415c0c6d8",
  "token_type": "bearer",
  "expires_in": 38937,
  "scope": "profile.basicprofile.all"
}
```

Finally, if the API Manager is used, the token may be obtained directly from the API Manager console.

AAC API

The Swagger UI for the AAC API is available at <http://localhost:8080/aac/swagger-ui.html>.

Profile API

To obtain the basic user data the following call should be performed (scope profile.basicprofile.me):

```
GET /aac/basicprofile/me HTTPS/1.1
Host: aacserver.com
Accept: application/json
Authorization: Bearer <token-value>
```

If the token is valid, this returns the user data:

```
{
  "name": "Mario",
  "surname": "Rossi",
  "userId": "6789",
  "username": "mario@gmail.com"
}
```

To obtain the account-related data (e.g., the Identity Provider-specific attributes), the following call should be performed (scope `profile.accountprofile.me`):

```
GET /aac/accountprofile/me HTTPS/1.1
Host: aacserver.com
Accept: application/json
Authorization: Bearer <token-value>
```

If the token is valid, this returns the user data, e.g.,

```
{
  "name": "Mario",
  "surname": "Rossi",
  "username": "rossi@gmail.com",
  "userId": "1",
  "accounts": {
    "google": {
      "it.smartcommunitylab.aac.surname": "Rossi",
      "it.smartcommunitylab.aac.username": "rossi@gmail.com",
      "it.smartcommunitylab.aac.givenname": "Mario",
      "email": "rossi@gmail.com"
    }
  }
}
```

Token API

ion API (ITEF RFC 7662). The call should be provided with the Client Credentials (e.g., as BasicAuth, with `client_id:client_secret`)

```
POST /aac/token_introspection?token=<token-value> HTTPS/1.1
Host: aacserver.com
Accept: application/json
Authorization: Basic <client-credentials>
```

The request returns a JSON token with standard claims regarding the user, the client, and the token. The returned claims represent the standard token claims (see ITEF RFC 7662 for details) as well as custom AAC claims (prefixed with `aac_`).

```
{
  "active": true,
  "client_id": "7b4f9b2a-71f6-412d-93e6-030c14910083",
  "scope": "profile.basicprofile.me profile.accountprofile.me openid",
  "username": "admin@carbon.super",
  "token_type": "Bearer",
  "sub": "8",
  "iss": "https://aac.example.com",
  "aud": "7b4f9b2a-71f6-412d-93e6-030c14910083",
  "exp": 123456789,
  "iat": 123450000,
  "nbf": 123456788,
  "aac_user_d": "8",
  "aac_grantType": "implicit",
  "aac_applicationToken": false,
  "aac_am_tenant": "tenant1.com"
}
```

Role API

The role API allows for the checking the role of the specific users. More details can be found on the Swagger documentation.

OpenID API

AAC provides support for some of the OpenID Connect functionalities. The OIDC metadata information is available at the `/ .well-known/openid-configuration` endpoint, where the supported features and relevant endpoints are captured.

In particular, the OpenID userinfo endpoint allows for getting the standard user info claims (scopes `profile`, `email`). The response is provided in the form of JSON object or JWT token.

```
GET /aac/userinfo HTTPS/1.1
Host: aacserver.com
Accept: application/json
Authorization: Bearer <token-value>
```

The data provided represents the subset of standard OpenID claims.

```
{
  "sub": "123456789",
  "name": "Mario Rossi",
  "preferred_username": "rossi@mario.com",
  "given_name": "Mario",
  "family_name": "Rossi",
  "email": "rossi@mario.com",
}
```

Configuration and Installation

Installation Requirements

- RDBMS (tested on MySQL 5.5+)
- Java 1.8+
- MongoDB 3.4+ (optional, needed for Authorization Module)
- Apache Maven (3.0+)

AAC configuration properties are configured through the corresponding Maven profile settings. By default, local profile is activated and the execution environment is configured in the `src/main/resources/application-local.yml` file. The example configuration for the local execution is defined in `application-local.yml.example`.

DB Configuration

DB Configuration is performed via the JDBC driver configuration, e.g. for MySQL:

```
jdbc:
  dialect: org.hibernate.dialect.MySQLDialect
  driver: com.mysql.jdbc.Driver
  url: jdbc:mysql://localhost:3306/aac
  user: ac
  password: ac
```

The schema and the user with the privileges should have been already configured. **IMPORTANT!** The default configuration already contains MySQL driver version 5.1.40. In case a newer version of the DB is used, update the driver version in `pom.xml` accordingly.

AAC Endpoint Configuration

The way the AAC is exposed is configured via the following properties:

- `server.host` (or similar Spring Boot settings). Defaults to `localhost:8080`
- `server.contextPath`. Defaults to `/aac`
- `application.url`. The reverse proxy endpoint for the AAC needed, e.g., for the authentication callback.

Security Configuration

The following security properties **MUST** be configured for the production environment:

- `admin.password`: the password for the AAC administrator user.
- `admin.contexts`: default role contexts managed by the platform. Defaults to `apimanager, authorization, components`
- `admin.contextSpaces`: default role spaces associated to the admin. Defaults to `apimanager/carbon.super` (used by WSO2 API Manager as a default tenant)
- `security.rememberme.key`: the secret key used to sign the remember me cookie.

For the user registration and the corresponding communications (email verification, password reset, etc), the SMTP email server should be configured, e.g.:

```
mail:
  username: info@example.com
  password: somepassword
  host: smtp.example.com
  port: 465
  protocol: smtps
```

Configuring Identity Providers

By default, AAC relies on the internally registered users. However, the integration with the external identity provider is supported. Out of the box, the support for the OAuth2.0 providers can be achieved through the configuration files. Specifically, to configure an OAuth2.0 identity provider it is necessary to specify the OAuth2.0 client configuration in the properties, e.g. for Facebook,

```
oauth-providers:
  providers:
    - provider: facebook
```

(continues on next page)

(continued from previous page)

```

client:
  clientId: YOUR_FACEBOOK_CLIENT_ID
  clientSecret: YOUR_FACEBOOK_CLIENT_SECRET
  accessTokenUri: https://graph.facebook.com/oauth/access_token
  userAuthorizationUri: https://www.facebook.com/dialog/oauth
  preEstablishedRedirectUri: ${application.url}/auth/facebook-oauth/callback
  useCurrentUri: false
  tokenName: oauth_token
  authenticationScheme: query
  clientAuthenticationScheme: form
  scope:
    - openid
    - email
    - profile

```

Besides, it is necessary to describe the provider and the possible identity mappings in `src/main/resources/it/smartcommunitylab/aac/manager/authorities.xml` file. The configuration amounts to defining the properties to extract, mapping name and surname properties, defining the unique identity attribute for the provider, and relations to the other providers' attributes (e.g., via email). For example,

```

<authorityMapping name="google" url="google" public="true" useParams="true">
  <attributes alias="it.smartcommunitylab.aac.givenname">given_name</attributes>
  <attributes alias="it.smartcommunitylab.aac.surname">family_name</attributes>
  <attributes alias="it.smartcommunitylab.aac.username">email</attributes>
  <attributes>name</attributes>
  <attributes>link</attributes>
  <identifyingAttributes>email</identifyingAttributes>
</authorityMapping>

```

defines the Google authentication attributes. Specifically, the `email` attribute is used to uniquely identify Google users.

Authorization Module

AAC allows for integration of the authorization module, where it is possible to configure the access rights for the user at the level of data. To enable the authorization module, the corresponding `authorization` Maven profile should be activated.

Logging Configuration

The logging settings may be configured via standard Spring Boot properties, e.g., for the log level

logging:

level: ROOT: INFO

The project relies on the Logback configuration (see `src/main/resources/logback.xml`). The default configuration requires the log folder path defined with `aac.log.folder` property. (if the property is not set, application will use default value: `WORKING_DIRECTORY/logs`).

OpenID Configuration

AAC provide a basic implementation of the OpenID protocol. The implementation is based on the [MitreID](#) project.

To configure the issuer, it is necessary to specify the OpenID issuer URL:

```
openid:  
  issuer: http://localhost:8080/aac
```

OpenID extension requires RSA keys for JWT signature. The project ships with the pre-packaged generated key. The key **MUST** be replaced with your specific value in production environment. To generate new key please follow the instructions available [here](#).

The resulting key should be placed in the resources (i.e., src/main/resources).

The OpenId metadata is available at /.well-known/openid-configuration.

Execution

To execute from command line, use maven Spring Boot task:

```
mvn -Plocal spring-boot:run
```

In case you run the tool from the IDE, add the profile configuration to the VM parameters:

```
-Dspring.profiles.active=local
```

To enable the authorization module, add the corresponding profile to the profile list (comma-separated)

Once started, the AAC tool UI is available at <http://localhost:8080/aac>.

3.1.2 Organization Manager

Organization Manager (*Org-Manager* for short) is a tool to centralize tenant and user management for various services. It requires the [AAC](#) identity provider to work.

Org-Manager is structured as follows:

- **Client** - The *Organization Management Console (OMC)* is the front-end console to perform all operations offered by the service.
- **Connectors** - Each connector takes care of reflecting operations issued by the back-end to its corresponding component.
- **Model** - The model connectors are based on. Functions as interface between back-end and connectors.
- **Server** - The back-end, whose APIs are used by the front-end, issues commands to the connectors to apply tenant management-related operations on all components.

Installation

Org-Manager is available here: <https://github.com/smartcommunitylab/AAC-Org>

This section will guide you through the steps necessary for running it: installing the server, configuring the connectors and installing the client.

Org-Manager can also run inside a Docker container. Refer to [Running with Docker](#) for instructions.

Server

Before you configure Org-Manager, you need to install the **AAC identity provider** and create an app for it.

Configuring the AAC identity provider

Org-Manager requires the [AAC](#) identity provider to work. The repository explains how to install and configure it.

Once AAC is running, create a new app for Org-Manager by accessing the **Client Apps** menu and clicking on **NEW APP**.

In the **Settings** tab, under **redirect Web server URLs**, add the redirect URLs for server and client. If you're running them on *localhost*, for example, add both of the following (assuming the ports are 7979 and 4200):

```
http://localhost:7979/login
http://localhost:4200/login
```

To run the server within a Docker container, you need to add a third URL with the port (for example 7878) Docker will expose the service through:

```
http://localhost:7878/login
```

For more information on running the server inside a Docker container, see the [Running with Docker](#) section.

For **Grant types**, check `Implicit` and `Client credentials`. For **Enabled identity providers**, check `internal`.

In the **API Access** tab, grant all permissions under `Basic profile service` and under `Role Management Service` and save the app.

Finally, all users that will be administrators of Org-Manager, as well as all organization owners, need the following role: `apimanager/carbon.super:profilemanager`.

Additionally, administrators will need this one too: `organizations:ROLE_PROVIDER`.

To create the `apimanager/carbon.super` space, access the **Space Owners** menu, choose `apimanager` as **Parent Space** and click on **NEW USER**. Insert the **Username**, insert `carbon.super` under **New spaces** and click **ADD**. Click **UPDATE** to create this space.

Now that the space has been created, all users who will be administrators of Org-Manager, or owners of an organization, need the `profilemanager` role within this space.

Access the **User Roles** menu, pick `apimanager/carbon.super` as **Role Context**, and then, for each user, click **NEW USER**, insert the **Username**, insert `profilemanager` as **New role**, click **ADD** and then **UPDATE**.

Assign the `organizations:ROLE_PROVIDER` role to other administrator users in the same way as you did with the `apimanager/carbon.super:profilemanager` role.

Setting up the server

The `application.yml` file contains various properties used by the server. The following is a brief explanation of the main properties and what values should be given to them. While properties in YAML are defined by indentation, this document will translate that indentation with a dot-separated notation, to keep the explanation shorter.

Properties appear with the following format:

```
<property_name>: ${<environment_variable_name>:<default_value>}
```

When the server is run, the value for the property is taken from the indicated environment variable (set by Docker), but, if the environment variable cannot be found (for example when not running with Docker), it uses the default value instead.

For example, the property for the port of the service appears as follows:

```
server:
  port: ${OMC_SERVER_PORT:7979}
```

If you are not running the server inside a Docker container, and want to use a different port, just change the 7979 part. For more information on running the server inside a Docker container, see the [Running with Docker](#) section.

- `server.port` – The port the server is running at. Sample value: 7979
- `server.servlet.session.cookie.name` – Name of the session cookie, used to avoid conflicts with other applications that use the name `JSESSIONID` and may be running on the same host. Sample value: `ORGMANAGERSESSIONID`
- `spring.datasource.url` – Database server for the Org-Manager server. The format may vary depending on the database type. A typical format can look like this: `jdbc:<database type>://<host>:<port>/<database name>`. Sample value: `jdbc:postgresql://localhost:5432/orgmanager`
- `spring.datasource.username` – Name of the user in the database
- `spring.datasource.password` – Password of the user in the database
- `spring.datasource.driver-class-name` – Driver for the database. Sample value: `org.postgresql.Driver`
- `spring.jpa.database-platform` – Dialect for the database. Sample value: `org.hibernate.dialect.PostgreSQLDialect`

There may be more properties under `spring` related to setting up the database.

- `security.oauth2.client.clientId` – Client ID for Org-Manager in the identity provider.
- `security.oauth2.client.clientSecret` – Client secret for Org-Manager in the identity provider.
- `security.oauth2.client.accessTokenUri` – URI for obtaining the access token
- `security.oauth2.client.userAuthorizationUri` – URI to obtain authorization by the identity provider
- `security.oauth2.client.tokenInfoUri` – URI to inspect the contents of the token
- `security.oauth2.client.tokenName` – Name used by the identity provider to identify the access token
- `security.oauth2.client.userIdField` – Name used by the identity provider for the field of the token that contains the ID of the user
- `security.oauth2.client.organizationManagementScope` – Identifier for the organization management scope, which grants administrator privileges
- `security.oauth2.client.organizationManagementContext` – The AAC context within which component contexts are nested. Having the `ROLE_PROVIDER` role within this context also grants administrator privileges.
- `security.oauth2.resource.userInfoUri` - scope for basic profile information
- `aac.uri`: AAC host
- `aac.apis.manageRolesUri` - AAC API end-point for managing user roles
- `aac.apis.userProfilesUri` - AAC API end-point for retrieving profile information, used to associate user name with ID
- `aac.apis.currentUserRolesApi` - AAC API end-point for retrieving the authenticated user's roles
- `aac.apis.currentUserProfileApi` - AAC API end-point for retrieving the authenticated user's profile

Connectors

All connectors take their configuration parameters from the server, which retrieves them from a single file, `application-components.yml`.

When running the server with Docker, this file is replaced with a different one; see the [Running with Docker](#) section for more information.

Different connectors have different properties, so configuration for each connector is explained in its correspondent section.

Two properties are however required for all connectors:

- `componentId` - Identifies the component.
- `implementation` - Full name of the connector class that reflects tenant/user management operations on the component. The following value corresponds to a dummy class that causes no changes, to be used if the component does not need an external class for this purpose, or to simply disable a connector:

```
it.smartcommunitylab.orgmanager.componentsmodel.DefaultComponentImpl
```

WSO2 Connector

The WSO2 connector may be used for different WSO2 products, such as *API Manager* or *Data Services Server*.

In order to provide the necessary infrastructure for allowing WSO2 products to interact with Org-Manager, it is necessary to include in **repository/components/dropins** the *jar*'s of the following submodules:

```
<module>apim.custom.user.store</module>
<module>apim.custom.user.store.stub</module>
```

This extension is done in order to permit the admin account to create, update, delete users and assign/revoke roles within specific tenants and extends the existing `UserStoreManagerService admin`.

The configuration steps are the following: - Build **orgmanager-wso2connector** project with Maven.

- Copy **apim.custom.user.store-0.0.1.jar** from the project *orgmanager-wso2connector/apim.custom.user.store* to the WSO2 directory **repository/components/dropins**
- Copy **apim.custom.user.store.stub-0.0.1.jar** from the project *orgmanager-wso2connector/apim.custom.user.store.stub* to the WSO2 directory **repository/components/dropins**

As a result, the new admin stub can be accessed from the following end-point: `https://$APIM_URL/services/CustomUserStoreManagerService`

After putting the *jar*'s in the proper folder, you should update the connector's configuration in `application-components.yml` accordingly for APIM and DSS components.

For example, for API Manager:

- `name`: Name of the component, only needed for display.
- `componentId`: ID of the component, should be `apimanager`
- `scope`: Scope of the component, should be `components/apimanager`
- `format`: Regular expression for the tenants' domains. Should be `^([a-z0-9]+(-[a-z0-9]+)*\.[a-z]{2,})$`
- `implementation`: Full class name of the class implementing the component. The class designed for API Manager is `it.smartcommunitylab.apimconnector.APIMConnector`; alternatively, the value

`it.smartcommunitylab.orgmanager.componentsmodel.DefaultComponentImpl` may be used to disable the API Manager connector.

- `roles`: Comma-separated list of roles that may be assigned to users via OMC. It should consist of `ROLE_PUBLISHER` and `ROLE_SUBSCRIBER`, as they are the 2 currently supported roles.
- `host`: URI where NiFi is hosted.
- `usermgmtEndpoint`: User service stub, should be `/services/CustomUserStoreManagerService`
- `usermgmtPassword`: Password for user management, default is `admin`
- `multitenancyEndpoint`: Multi-tenancy service stub, should be `/services/TenantMgtAdminService`
- `multitenancyPassword`: Password for tenant management, should be `admin`

NiFi Connector

This section explains how multitenancy works in NiFi and how to configure the connector so that tenancy operations issued by the server are performed in NiFi.

If you're not interested in how multi-tenancy is represented in NiFi, skip to the [Certificates](#) section to create the necessary certificates.

If you don't need to create certificates and only need to configure the connector, skip to the [Configuration](#) section.

Multi-tenancy in NiFi

The idea of multi-tenancy in NiFi is that **process groups** represent tenants and have policies defined for them, listing which **users** or **user groups** are allowed to view or alter them. User groups are equivalent to teams, so if permission to view a process group is given to a user group, all users belonging to it can view it.

Users will still be able to see other teams' process groups on the flow, but they will only appear as rectangles that they can neither interact with or view details of. The only information they can see about them is the amount of data they are processing.

Certificates

Executing tenant and user management operations in a secured NiFi instance requires specific authorizations, so Org-Manager needs to act with the permissions granted to the administrator user.

Since *OpenID Connect* is used to secure NiFi, we have to authenticate by providing the administrator's SSL certificate and configuring NiFi to recognize it.

This section describes how to do this and is heavily based on a very useful and detailed [article](#) by Matt Clarke from the Hortonworks Community.

Two pieces of software are needed for this process:

- [Keytool](#): comes bundled with Java's JRE, so you should find it in your Java installation folder, usually in `C:\Program Files\Java\jre1.8.0_191\bin`, depending on your version.
- [OpenSSL](#)

The steps described in this section are written for Windows' *Command Prompt*, so the syntax for paths and the like may vary depending on your OS. Remember to quote paths if they contain spaces.

Step 1: Creating a Certificate Authority (CA)

The first thing to do is creating a *Certificate Authority (CA)*. This CA will sign the administrator's certificate, stating that it can be trusted.

Change directory to the `bin` subfolder of your OpenSSL installation (for example, `cd C:\OpenSSL\openssl-1.0.2j\bin`). If you don't, you will need to replace `openssl` in each command with the **path to the openssl.exe file**.

1. Creating the CA's private key

This will create the private key for your CA and place it in the `C:\certs` folder. If you omit the path and just write `myCA.key`, it will be in the **same folder as openssl.exe**. You will be asked to choose a password.

```
openssl genrsa -aes128 -out C:\certs\myCA.key 4096
```

2. Creating a pem certificate

This command creates the CA's certificate. You will be asked to provide the password you chose in [1.1](#). You will then have to fill the CA's profile (country, organization name, etc.): the data you insert in this step is not important for our purposes, but it might be preferable to pick something that will help you recognize this certificate.

1095 is the validity (in days) of the certificate, feel free to change it as you see fit.

```
openssl req -x509 -new -key C:\certs\myCA.key -days 1095 -out C:\certs\myCA.pem
```

3. Converting from pem to der

Converting the certificate into **der** format is necessary for the next step, performed by Keytool.

```
openssl x509 -outform der -in C:\certs\myCA.pem -out C:\certs\myCA.der
```

Step 2: Creating NiFi's Truststore

A truststore lists which certificates can be trusted. It is necessary to add the CA's certificate to this truststore, otherwise the CA's signature on the administrator's certificate will be meaningless.

Change directory to where **keytool.exe** is located (probably something like `C:\Program Files\Java\jre1.8.0_191\bin`), or replace `keytool` with the **path to the keytool.exe file**.

1. Creating the truststore

This will create the truststore and include the CA's certificate in it, meaning it can be trusted. You will be asked to choose a password for the truststore. It will then show you the CA's certificate and ask you to confirm it can be trusted by typing the word *yes* in your system's language.

```
keytool -import -keystore C:\certs\truststore.jks -file C:\certs\myCA.der -alias myCA
```

2. Configuring NiFi to use the new truststore

Open the `nifi.properties` file (it can be found inside the `conf` subfolder of your NiFi installation) and edit the following fields. In newer NiFi versions, the `needClientAuth` field may not be present, in which case you can omit it. The password for the `truststorePasswd` field is the one you chose in [2.1](#).

```
nifi.security.truststore=C:/certs/truststore.jks
nifi.security.truststoreType=JKS
nifi.security.truststorePasswd=MyTruststorePassword
nifi.security.needClientAuth=true
```

Step 3: Generating a keystore for the NiFi server

This is not strictly related to our tenant-providing process; however, when running a secured instance of NiFi, it is necessary for it to have a keystore, and for the browser that accesses NiFi to trust it. Since we just created a CA, we can use it to create NiFi's keystore.

1. Generate a keystore for the NiFi server

Change directory to the path to **keytool.exe** (for example `cd C:\Program Files\Java\jre1.8.0_191\bin`), or replace **keytool** with the **path to keytool.exe**.

The following command will generate the keystore. It will ask you to choose a password for the keystore, and then to fill the profile of the certificate, similarly to what happened when generating the CA. When asked for the full name (it should be the first thing asked after password confirmation), insert your domain's name. If you're doing this on localhost, simply type *localhost*.

Finally, it will ask you to choose a password for the private key. If you just hit enter, it will use the same password as the keystore's.

```
keytool -genkey -alias nifiserver -keyalg RSA -keystore _
↪C:\certs\nifiserver.jks -keysize 2048
```

2. Generating a certificate sign request

This command will generate a certificate with a request to sign it. It may ask for both the passwords you chose in [3.1](#): first the keystore's password and then the private key's password. If they are the same, it will only ask once.

```
keytool -certreq -alias nifiserver -keystore C:\certs\nifiserver.jks -
↪file C:\certs\nifiserver.csr
```

3. Signing the NiFi server's certificate

Once again, change directory to OpenSSL's bin subfolder, or replace **openssl** accordingly.

This command will have the CA sign your NiFi server's certificate to state that it can be trusted. It will ask for the password you chose in [1.1](#).

```
openssl x509 -sha256 -req -in C:\certs\nifiserver.csr -CA C:\certs\myCA.
↪pem -CAkey C:\certs\myCA.key -CAcreateserial -out C:\certs\nifiserver.
↪crt -days 730
```

4. Import the CA's public key into the keystore

Switch back to Keytool's folder, or replace **keytool** accordingly.

This command will include the CA's public key into your keystore, so that it may be used to verify your certificate's validity. It will ask for the keystore's password, which you chose in [3.1](#). You will have to confirm that the certificate can be trusted by typing *yes* in your system's language.

```
keytool -import -keystore C:\certs\nifiserver.jks -file C:\certs\myCA.pem
```

5. Import the signed NiFi server's certificate into the keystore

This command will import the certificate you signed in 3.3 into the keystore. It will ask for the two passwords you chose in 3.1: first the keystore's password and then the private key's password, or just one of them if they are the same.

```
keytool -import -trustcacerts -alias nifiserver -file C:\certs\nifiserver.  
↪ crt -keystore C:\certs\nifiserver.jks
```

6. Configuring NiFi to use the new keystore

Open the *nifi.properties* file (from the *conf* subfolder of your NiFi installation) and edit the following properties, using the two passwords chosen in 3.1:

```
nifi.security.keystore=C:/certs/nifiserver.jks  
nifi.security.keystoreType=JKS  
nifi.security.keystorePassword=MyKeystorePassword  
nifi.security.keyPassword=MyPrivateKeyPassword
```

7. Adding the CA to your browser

When accessing NiFi, your browser will likely state that the connection cannot be trusted. This is because, even though the NiFi server's certificate is signed by your CA, your browser does not know your CA.

It may offer you to add an exception, but at this point you might as well add the CA you created to the list of trusted CAs.

For example, to do it in Mozilla Firefox:

Settings > Options > Privacy and security > Show certificates (on the right, near the bottom, in the *Certificates* section) > **Authorities** tab > **Import** > open your *myCA.pem* file and check both boxes.

You might need to restart your browser. Afterwards, you should be able to access NiFi. If it still says the connection cannot be trusted, you might have inserted the wrong name in 3.1, and have to repeat steps 3.1 through 3.5.

Step 4: Make the CA sign the administrator's certificate

By having the administrator's certificate signed by the CA, it will be recognized as valid by NiFi, since it trusts the CA. Change directory back to the *bin* subfolder of your OpenSSL installation, or replace *openssl* with the **path to the openssl.exe** file.

1. Generating the administrator's certificate's private key

Same command as when you created the CA's private key. It will ask you to choose a password.

```
openssl genrsa -aes128 -out C:\certs\admin.key 2048
```

2. Generating a certificate sign request

Like in step 3.2, this command will generate a certificate with a request to sign it. You will be asked to provide the password to the private key you just created. It will then ask you to fill the profile of the certificate, similarly to what you did with the CA.

It is now important to provide the name of the administrator (for example in **Common Name**, or **Email Address**), as it will be used by NiFi to associate this certificate to the admin user account (see 4.5) for more information). The other fields are not very meaningful, but again, try to pick something that will help you recognize the certificate.

Also note that it will ask you for a **challenge password** and an **optional company name**. The challenge password is very rarely used by some CAs when requesting to revoke a certificate. Both fields can safely be left blank.

```
openssl req -new -key C:\certs\admin.key -out C:\certs\admin.csr
```

3. Signing the administrator's certificate

You can now have the CA sign your administrator's certificate. It will ask for the password you created in 1.1.

```
openssl x509 -req -in C:\certs\admin.csr -CA C:\certs\myCA.pem -CAkey C:\certs\myCA.key -CAcreateserial -out C:\certs\admin.crt -days 730
```

4. Converting from crt to p12

This command will convert the signed certificate into **p12** format. It will ask you to provide the password you chose in 4.1, and then it will ask you to choose an export password, needed to extract the certificate from the p12 file.

```
openssl pkcs12 -export -out C:\certs\admin.p12 -inkey C:\certs\admin.key -in C:\certs\admin.crt -certfile C:\certs\myCA.pem -certpbe PBE-SHA1-3DES -name "admin"
```

5. Configure NiFi to find the administrator's name

Finally, you have to uncomment two lines from the *nifi.properties* file (inside the *conf* subfolder of your NiFi installation) and give them proper values.

They are regular expressions used by NiFi to find the administrator's name inside the certificate you created in 4.2. Configuring these two lines incorrectly can lead to 403 errors.

The field names are: *EMAILADDRESS* (Email address), *CN* (Common Name), *OU* (Organizational Unit Name), *O* (Organization Name), *L* (Locality Name), *ST* (State or Province Name), *C* (Country Code).

This particular configuration will take the administrator's name from the *Common Name* field, but you can alter it depending on how you filled the profile during 4.1.

```
nifi.security.identity.mapping.pattern.dn=^(EMAILADDRESS=(.*?), )?CN=(.*?), OU=(.*?), O=(.*?), L=(.*?), ST=(.*?), C=(.*?)$  
nifi.security.identity.mapping.value.dn=$3
```

Configuration

Many of the fields represent NiFi API end-points and have fixed values: although unlikely, there is a chance that they may change in newer versions of NiFi. If you suspect this has happened, you should be able to find the new end-point in the [official documentation](#).

- `name`: Name of the component, only needed for display.
- `componentId`: ID of the component, should be `nifi`
- `scope`: Scope of the component, should be `components/nifi`
- `implementation`: Full class name of the class implementing the component. The class designed for NiFi is `it.smartcommunitylab.nificonnector.NiFiConnector`; alternatively, the value `it.smartcommunitylab.orgmanager.componentsmodel.DefaultComponentImpl` may be used to disable the NiFi connector.

- **roles:** Comma-separated list of roles that may be assigned to users via OMC. It should consist of all roles listed in the `readRoles` field plus all roles listed in the `writeRoles` field.
- **host:** URI where NiFi is hosted.
- **listUsersApi:** NiFi API end-point for listing users. Should be `/nifi-api/tenants/users`, unless a new version of NiFi changes it into something different.
- **createUserApi:** NiFi API end-point for creating a user. Should be `/nifi-api/tenants/users`
- **deleteUserApi:** NiFi API end-point for deleting a user. Should be `/nifi-api/tenants/users/`
- **listUserGroupsApi:** NiFi API end-point for listing user groups. Should be `/nifi-api/tenants/user-groups`
- **createUserGroupApi:** NiFi API end-point for creating a user group. Should be `/nifi-api/tenants/user-groups`
- **updateUserGroupApi:** NiFi API end-point for updating a user group. Should be `/nifi-api/tenants/user-groups/`
- **deleteUserGroupApi:** NiFi API end-point for deleting a user group. Should be `/nifi-api/tenants/user-groups/`
- **getPolicyApi:** NiFi API end-point to retrieve a policy. Should be `/nifi-api/policies/`
- **createPolicyApi:** NiFi API end-point to create a policy. Should be `/nifi-api/policies`
- **updatePolicyApi:** NiFi API end-point to update a policy. Should be `/nifi-api/policies/`
- **listProcessGroupsApi:** NiFi API end-point to list process groups. Should be `/nifi-api/process-groups/`
- **getProcessGroupApi:** NiFi API end-point to retrieve a process group. Should be `/nifi-api/process-groups/`
- **createProcessGroupApi:** NiFi API end-point to create a process group. Should be `/nifi-api/process-groups/`
- **deleteProcessGroupApi:** NiFi API end-point to delete a process group. Should be `/nifi-api/process-groups/`
- **accessApi:** NiFi API end-point to retrieve the status of the current access. Should be `/nifi-api/access`
- **keystorePath:** Absolute path to the certificate. Following the example in the [Certificates](#) section, it would have the value `C:/certs/admin.p12`, determined in [4.4](#).
- **keystoreType:** Type of the certificate. In the example, it would have the value `PKCS12`.
- **keystoreExportPassword:** The password for the certificate. In the example, it would have the value chosen in [4.4](#).
- **truststorePath:** Absolute path to the truststore. In the example, it would be `C:/certs/truststore.jks`, determined in [2.1](#).
- **truststoreType:** Type of the truststore. In the example, it would have the value `JKS`.
- **truststorePassword:** Password of the truststore. In the example, it would have the value chosen in [2.1](#).
- **adminName:** Name of the administrator user.
- **ownerRole:** Role used by AAC to indicate ownership. Should be `ROLE_PROVIDER`. Will have both read and write permissions on process groups.
- **readRoles:** Names of the roles which will have read-only permissions on process groups. While multiple roles may be listed, separated by a comma, they would all be equivalent, so listing 1 role only is advisable.

The `roles` field should contain all roles listed in this field and all roles listed in the `writeRoles` field, or consistency issues may arise.

- `writeRoles`: Names of roles which will have both read and write permissions (just like the owner role) on process groups. While multiple roles may be listed, separated by a comma, they would all be equivalent, so listing 1 role only is advisable. The `roles` field should contain all roles listed in this field and all roles listed in the `readRoles` field, or consistency issues may arise.

Running with Docker

The server contains some default configuration that, when running with Docker, cannot be changed without recompiling the whole project.

To avoid this, two files are necessary for Docker to override this default configuration.

The **first** one is an `env` file: when Docker runs the container, it will use this file to create several environment variables that the server will read to configure itself.

The `env` file to alter is `/dockerfiles/orgmanager-config.env`, which contains a sample configuration.

Variables appear with the `<NAME>: <value>` format. The uppercase part matches its name as described in the [Setting up the server](#) section, while on the right is the value to assign.

Make sure the `OMC_SECURITY_OAUTH2_CLIENTID` and `OMC_SECURITY_OAUTH2_CLIENTSECRET` variables respectively contain the client ID and secret generated by AAC for the server. In addition, replace `host:port` addresses for the Postgres database and AAC appropriately.

The **second** file will contain the configuration for the components, such as API Manager or Apache NiFi. Unlike the previous file, which creates environment variables for the server to retrieve values from, this one simply replaces a default configuration file.

The file must be in `yml` format and its structure is identical to the default `application-component.yml` file.

A sample configuration is present in `dockerfiles/application-components.yml`.

Replace `host:port` values with the addresses of the services.

Once you have configured these two files and Docker is running, open a console and change directory to the root folder (AAC-`Org`) of the project and execute this command to build a Docker image:

```
docker build -t orgmanager .
```

This command will take some time to compile the whole project and will create an image named `orgmanager`. If you wish to name it something else, simply replace `orgmanager` with the name you wish to use.

Note that the final dot of the command, separated by a space, is important: without it, an error will be returned.

All that remains is to run the container using this image. The following command will run the server inside a Docker container, mounting the two configuration files described earlier.

```
docker run --env-file dockerfiles/orgmanager-config.env -v <absolute_path_to_project>/  
↪dockerfiles/application-components.yml:/tmp/server/target/config/application-  
↪components.yml -p 7878:7979 -t orgmanager
```

Note that you need to replace `<absolute_path_to_project>` with the full path to this project. If you're running it on Windows, the command would look similar to this:

```
docker run --env-file dockerfiles/orgmanager-config.env -v //c/Eclipse/Workspace/AAC-  
↪Org/dockerfiles/application-components.yml:/tmp/server/target/config/application-  
↪components.yml -p 7878:7979 -t orgmanager
```


If you have configured the NiFi connector to run, you need to provide the *Certificates* you created for it:

```
docker run --env-file dockerfiles/orgmanager-config.env -v <absolute_path_to_
↪certificates_folder>:/certs -v <absolute_path_to_project>/dockerfiles/application-
↪components.yml:/tmp/server/target/config/application-components.yml -p 7878:7979 -t_
↪orgmanager
```

The command with the NiFi certificates might look like the following:

```
docker run --env-file dockerfiles/orgmanager-config.env -v //c/Eclipse/Workspace/AAC-
↪Org/dockerfiles/certs:/certs -v //c/Eclipse/Workspace/AAC-Org/dockerfiles/
↪application-components.yml:/tmp/server/target/config/application-components.yml -p_
↪7878:7979 -t orgmanager
```

Client

Angular is required to run the *Organization Management Console*, the front-end for Org-Manager.

Install *Node.js*, then open a console and run the commands `npm install` to install app dependencies and `npm install -g @angular/cli` to install Angular CLI.

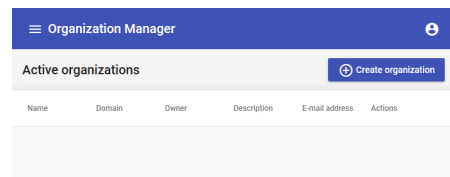
Before running the client, some parameters must be configured in the `src/environments/environment.ts` file:

- `aacUrl`: AAC address. If you're hosting it on *localhost* through port *8080*, it should be `http://localhost:8080/aac/`.
- `aacClientId`: Client ID for Org-Manager on AAC.
- `redirectUrl`: URL where the client is hosted. Default value is `http://localhost:4200/`.
- `scope`: Scopes required to enable authentication via AAC. The value should remain `profile.basicprofile.me,user.roles.me`.
- `locUrl`: Root for the server's APIs. If you're hosting Org-Manager on *localhost* through port *7979*, it should be `http://localhost:7979/api/`.

You can now run the client by opening a console, changing directory to the `client` subfolder of the project and executing `ng serve`.

User Guide

After signing in, you will be brought to the list of active organizations. Unless you have administrator rights, you will only see organizations that you are part of. Administrators also have access to a **Create organization** button in the top-right corner, to create new organizations.



Upon clicking it, a form will pop up, with required fields marked by an asterisk. The domain field is not required and will be generated automatically when left blank, based on the organization's name.

The owner's e-mail address, name and surname must match the equivalent fields of a user registered in the identity provider, AAC.

Create new organization

Please provide this information

Name of the organization *

MyOrganization

Domain

Description *

A test organization.

Owner's e-mail address *

jsmith@example.com

Owner's name *

John

Smith

Web address

example.com

Logo URL

example.com/images/logo.png

Phone numbers

12345 X

67890 X

Add number (hit space to confirm)

Tags

test X

Add tag (hit space to confirm)

OK

Cancel

Once you click **OK**, the page will refresh and, if creation was successful, the new organization will appear on the list. If something went wrong, an error will be displayed instead.

Organization Manager					
Active organizations					Create organization
Name	Domain	Owner	Description	E-mail address	Actions
MyOrganization	myorganization	John Smith	A test organization.	jsmith@example.com	Details Disable

Next to the organization's basic information are the **Details** and **Disable** actions. **Disable** will only be available if you have administrator rights.

If you click on **Details**, you will be brought to a page containing the organization's information. If you're registered as owner of the organization, or if you have administrator's rights, the **Edit organization information** button allows you to change some fields.

Only the following information can be changed: owner's data, description, web address, logo, phone numbers and tags.

Update MyOrganization information

Name of the organization
MyOrganization

Domain
myorganization

Owner's e-mail address *
jsmith@example.com

Owner's name * John Owner's surname * Smith

Description *
Updated organization.

Web address
example.com

Logo URL
example.com/images/logo.png

Phone numbers
12345 X 67890 X

Add number (hit space to confirm)

Tags
test X updated X

Add tag (hit space to confirm)

☒ Active

OK Cancel

The **Configuration** tab is only available to owners of the organization and administrators and displays all tenants of the organization, grouped by the components they are assigned to. Administrators may add tenants by clicking on the **Manage components** button.

A menu will pop up listing all available components. When you click on a component, it will expand, and you will be able to create new tenants by clicking on **Add tenant** and typing the tenant's domain.

List of components

API Manager Add tenants ^

Tenants
test.com Delete

Add tenant

Apache NiFi Add tenants ^

Tenants
sampletenant Delete

Tenants
anothertenant Delete

Add tenant

Data Services Server Add tenants v

Cyclotron Add tenants v

Superset Add tenants v

Geoserver Add tenants v

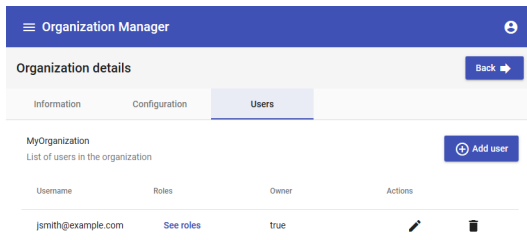
OK Cancel

Once you are done creating tenants, click **OK**. The page will refresh and components that have tenants will appear. By clicking on a component, you can see its tenants.

Renaming and deleting tenants works similarly: click on **Manage components**, expand a component and rename its tenants or delete them with the **Delete** button.

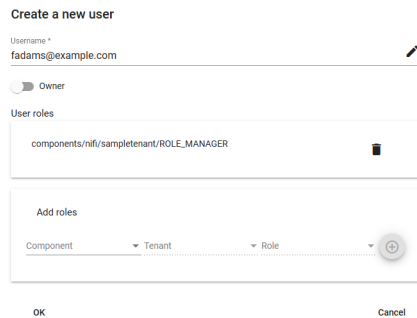
The **Users** tab can be viewed by owners of the organization and administrators and lists the organization's members. It allows adding new members to the organization, assigning or revoking roles and removing users from the organization.

If the organization has just been created, the only user will be its owner.



You can add more users by clicking on the *Add user* button. A small menu will appear, where you will be able to insert the user name and assign roles to the new user.

Administrators may decide if the new user is to be registered as owner of the organization, by toggling the *Owner* switch.



After you click **OK**, the page will refresh and the new user will appear.

You can remove users from the organization by clicking on the *trashcan* icon.

To assign (or revoke) roles, click on the *pencil* button. A menu similar to the previous one will appear.

Only administrators may delete organizations. If you want to delete an organization, it must first be disabled.

Go to the list of organizations (click **Back** in the top right, or click on the menu button in the top left, next to *Organization Manager*, and pick **Active Organizations**), click **Disable** on the organization you want to delete and confirm. The organization will disappear from the list, but has not been deleted yet. Click on the menu button in the top left and pick the second option to retrieve the list of disabled organizations. You can delete an organization from here, by clicking on the *trashcan* icon.

Server APIs

This section describes the APIs made available by the Org-Manager server. Reading it is not necessary to install or use Org-Manager, as the client takes care of calling APIs appropriately when operations are requested through the UI.

All requests must include the **Authorization** header, containing `Bearer <token value>`. In case an error occurs, each API will return a response with an appropriate status code, usually with some details in the response body.

Assuming the server is being hosted on *localhost* at port 7979, the **Swagger UI** for the Org-Manager APIs is available at <http://localhost:7979/swagger-ui.html>.

Most APIs have security restrictions that require the user to be owner of the organization they are attempting to alter, or to have administrator privileges.

The owner of a specific organization is defined as a user with the following role in AAC: `organizations/<organization_slug>:ROLE_PROVIDER`

A user has administrator privileges when the access token they are using is a client token with the `organization.mgmt` scope, or when they have the following role in AAC: `organizations:ROLE_PROVIDER`

Also keep in mind that, for some of these APIs to work correctly, the access token used must have the following scopes: `profile`, `email`, `user.roles.me`, `profile.basicprofile.me`, `profile.accountprofile.me`.

Create organization

This API is used to create an organization. Its response contains a JSON object that represents the newly created organization. This response will contain an additional field, `id`, necessary to recognize the organization and useful when calling other APIs.

Note that, since the `email` field will be interpreted as the starting owner of the organization, a user with its value as name will be created on the server side. This means that AAC must have a user with this username, otherwise an error will occur.

The `name` and `surname` fields inside the `contacts` object also must match the corresponding fields in AAC.

Requirements: must have administrator privileges

End-point: `/api/organizations`

Method: POST

Body: JSON object describing the organization. The following fields can be defined:

1. `name` – Name of the organization. Required. May contain alphanumeric characters, space, dash (-) or underscore (_). Cannot have the same name as an already existing organization, even if case is different. Any leading or trailing spaces will be ignored, and multiple consecutive spaces will be replaced with a single space.
2. `slug` – Defines the domain of the organization. Optional: if specified, it can only contain alphanumeric lower case characters and underscores. If left out, it will be generated from the name, converting it to lower case and replacing dashes and spaces with underscores.
3. `description` – Description of the organization. Required.
4. `contacts` – Inner JSON object describing the contacts. Required. Its 5 inner properties are:
 - `email` – E-mail. Required. Will be used as name of the owner of the organization.
 - `name` – Name of the contact. Required.
 - `surname` – Surname. Required.
 - `web` – URL. Optional.
 - `phone` – Array of strings for phone numbers. Optional.
 - `logo` – URL. Optional.
 - `tag` – Array of strings for tags. Optional.
 - `active` – Can take `true` or `false` as values. Indicates whether the organization is enabled or disabled. Optional, will default to `true` if omitted.

Sample request body:

```
{
  "name": "My Organization",
  "slug": "my_org",
  "description": "This is my test organization.",
  "contacts": {
    "email": "jsmith@my_org.com ",
    "name": "John",
    "surname": "Smith",
    "web": "http://www.example.com",
    "phone": ["12345", "67890"],
    "logo": "http://www.example.com/images/logo.png"
  },
  "tag": ["test", "testing"],
  "active": "true"
}
```

Search organizations

API for searching organizations. Organizations are searched by name. Responses are returned as pages of size 20.

If the authenticated user has administrator privileges, they will see all organizations, otherwise they will see only organizations they are part of.

End-point: /api/organizations

Method: GET

Parameters:

1. **name** – Name to search. Case insensitive. All organizations with a name that contains this parameter will be returned.
2. **page** – Page to be returned. Can be omitted, since most of the time the organizations returned will be less than 20. Starts from 0, so if you want the second page, use **page=1**.

Sample request URL: `http://localhost:7979/api/organizations?name=Company&page=3`

Update organization

Updates an organization. Only certain fields may be updated. The **id** of the organization must be known, and used in the request URL.

Requirements: must have the administrator privileges, or be owner of the organization.

End-point: /api/organizations/<organization_id>/info

Method: PUT

Body: JSON with the fields to change. Only description, contacts and tags may be changed; any other field present in the request will be ignored. Fields will only be updated if present in the input, so if you do not want to change a field, simply omit it from the request.

Sample request URL: `http://localhost:7979/api/organizations/1/info`

Sample request body:

```
{
  "description": "New description.",
  "contacts": {
    "web": "http://www.test.com",
    "phone": ["12345", "57575"]
  },
  "tag": ["testing"]
}
```

Enable organization

Enables an organization. Simply changes the **active** field to **true**.

Requirements: must have administrator privileges

End-point: /api/organizations/<organization_id>/enable

Method: PUT

Sample request URL: `http://localhost:7979/api/organizations/3/enable`

Disable organization

Disables an organization. Simply changes the `active` field to `false`. Other than the endpoint, it is identical to the **Enable organization** API.

Requirements: must have administrator privileges

End-point: `/api/organizations/<organization_id>/disable`

Method: PUT

Sample request URL: `http://localhost:7979/api/organizations/3/disable`

Delete organization

Deletes an organization. Also unregisters all members belonging to it, deletes all their roles within it, and deletes all tenants within it. An organization must be disabled before it can be deleted.

Requirements: must have administrator privileges

End-point: `/api/organizations/<organization_id>`

Method: DELETE

Sample request URL: `http://localhost:7979/api/organizations/1`

List available components

Lists available components, together with a few properties for each of them.

End-point: `/api/components`

Method: GET

List possible roles for a component

Returns a list of strings, representing what roles may be attributed to a user when added to a tenant within a specific component.

End-point: `/api/components/<component_id>/roles`

Method: GET

Sample request URL: `http://localhost:7979/api/components/nifi/roles`

Configure tenants for an organization

Allows configuring which tenants an organization should have.

Requirements: must have administrator privileges

End-point: `/api/organizations/<organization_id>/configuration`

Method: POST

Body: JSON object containing components and tenants for each component.

1. `componentId` – Identifies the component. Must be chosen among the values that can be found by calling the **Listing available components** API. Note that if a component is not specified in the body, it will not be altered.

2. `tenants` – Array of strings for the tenants. If a component previously contained tenants that are not present in this new array, those tenants will be removed.

Sample request body:

```
[
  {
    "componentId": "nifi",
    "tenants": [
      "trento",
      "ferrara"
    ]
  }, {
    "componentId": "dss",
    "tenants": [
      "reggio"
    ]
  }
]
```

Note that only tenants for the `nifi` and `dss` components will be affected, as no other components are present in the input. For example, tenants for the component `apimanager` will not be altered, since `apimanager` was not specified in the body.

Display tenants of the organization

Displays the tenants that have been configured for the input organization.

Requirements: must have administrator privileges, or be the owner of the organization

End-point: `/api/organizations/<organization_id>/configuration`

Method: GET

Sample request URL: `http://localhost:7979/api/organizations/1/configuration`

List users in an organization

Lists users that belong to the indicated organization. The `id` of the organization must be known. An optional parameter may be specified to act as a filter on the desired users' names.

Requirements: must have administrator privileges, or be the owner of the organization

End-point: `/api/organizations/<organization_id>/members`

Method: GET

Parameters:

1. `username`: If specified, only members whose user name contains this value (case insensitive) will be returned. If omitted, all members of the organization will be returned.

Sample request URL: `http://localhost:7979/api/organizations/1/members?username=john`

Add a user to an organization

Grants a user the roles listed in the request body. All roles they previously had within the organization, but that are not present in this new configuration, will be removed. The user will be added to the organization, in case they were previously not registered. This means that AAC must have a user with this username, otherwise an error will occur. The response will also contain the *id* of the member within the organization.

It is also possible, for administrators only, to grant/revoke the status of owner of the organization through this API.

Requirements: must have administrator privileges, or be owner of the organization (cannot grant/revoke owner status)

End-point: /api/organizations/<organization_id>/members

Method: POST

Body: JSON object containing the user's name and the roles they should have:

1. `username` – Name of the user to add. Must be a valid name recognized by the identity provider.
2. `roles` – Array of JSON objects representing the roles to add. Each role has 2 properties:
 - `contextSpace` – domain of the role. It must be one of the domains registered in the organization. It should have the following structure: `components/<component_id>/<tenant>`
 - `role` – Role of the user in the domain
3. `owner` - Boolean parameter that can only be set by administrators. If this parameter appears in a call performed without administrator rights, it will be ignored.

Sample request URL: `http://localhost:7979/api/organizations/1/members`

Sample request body:

```
{
  "username": "bob@test.com",
  "roles": [{
    "contextSpace": "components/nifi/trento",
    "role": "ROLE_MANAGER"
  }, {
    "contextSpace": "components/nifi/ferrara",
    "role": "ROLE_USER"
  }],
  "owner": "true"
}
```

Remove a user from an organization

Unregisters a user from an organization, stripping them of all roles they had within it. The `id` of the organization, as well as the `id` of the member to remove, must be known.

Requirements: must have administrator privileges, or be the owner of the organization

End-point: /api/organizations/<organization_id>/members/<member_id>

Method: DELETE

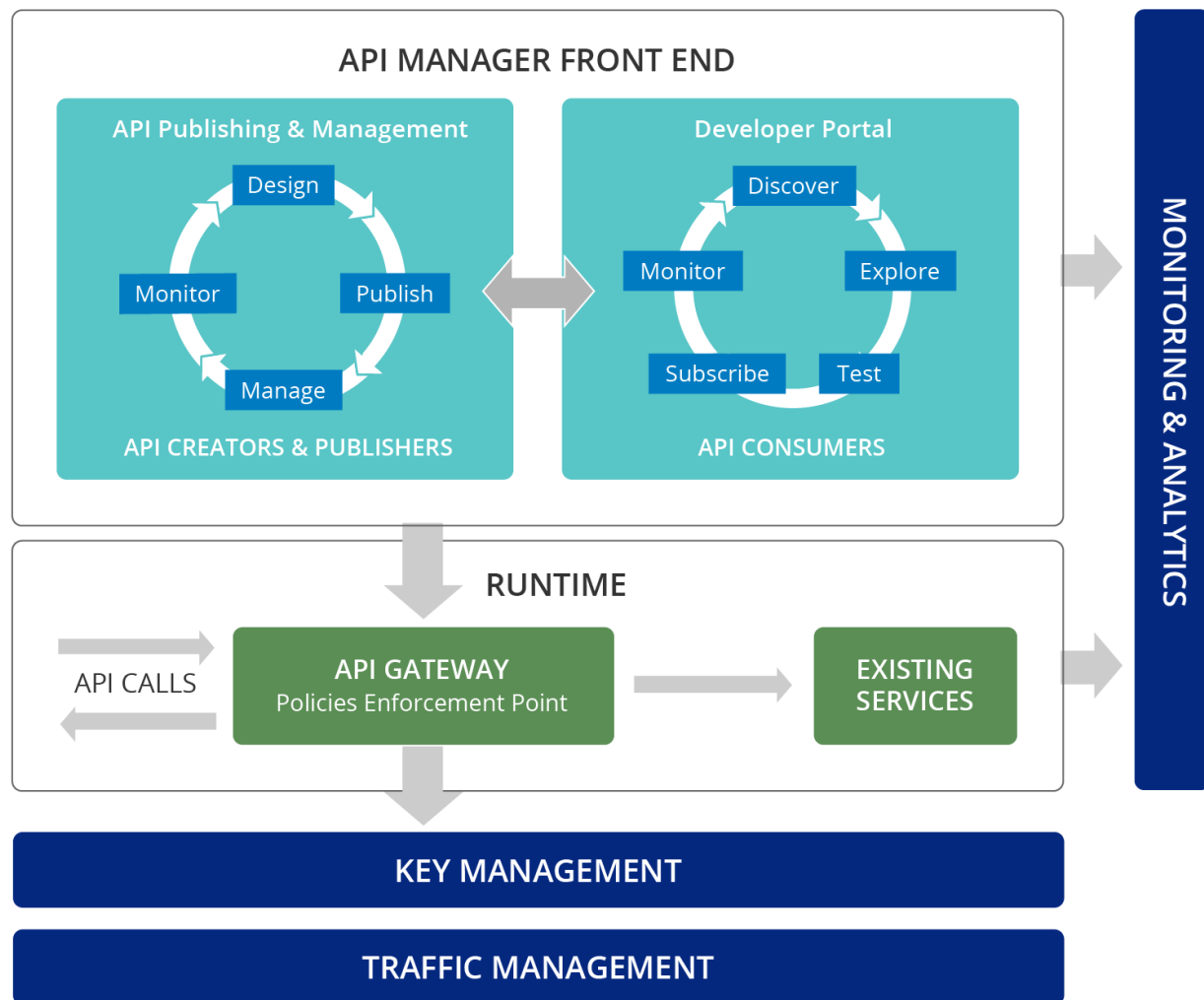
Sample request URL: `http://localhost:7979/api/organizations/1/members/2`

3.1.3 API Manager (WSO2)

As an organization implements SOA, it can benefit by exposing core processes, data and services as APIs to the public.

External parties can mash up these APIs in innovative ways to build new solutions.

A business can increase its growth potential and partnership advancements by facilitating developments that are powered by its APIs in a simple, decentralized manner.



However, leveraging APIs in a collaborative way introduces new challenges in exercising control, establishing trust, security and regulation. As a result, proper API management is crucial.

API Manager is a complete solution for creating, publishing and managing all aspects of an API and its lifecycle. It provides Web interfaces for development teams to deploy and monitor APIs, and for consumers to subscribe to, discover and consume APIs through a user-friendly storefront.

API Manager Installation

WSO2 API Manager is an open source approach that addresses full **API lifecycle management, monetization, and policy enforcement**.

WSO2 API Manager is a unique open approach to full lifecycle API development, integration and management. As part of the larger **Digital Hub Platform**, it is a central component used to deploy and manage **API-driven ecosystems**. It's hybrid integration capabilities further simplify projects that span traditional as well as microservice environments.

It allows extensibility and customization, and ensures freedom from lock-in.

The customized version of APIM inside Digital Hub can be set up following these steps:

Installation Requirements

- *Authentication and Authorization Controller (AAC)*
- MySQL 5.5+

1. Download the API Manager 2.6.0

Get the binary version from [here](#)

2. Database SetUp

Replace the default H2 db with mysql

- using mysql as database

```
- mysql -u root -p
- create database regdb character set latin1 (*);
- GRANT ALL ON regdb.* TO regadmin@localhost IDENTIFIED BY "regadmin";
- FLUSH PRIVILEGES;
- quit;
```

Note: character set is for windows only

- copy mysql connector (i.e. mysql-connector-java-__-bin.jar) into repository/components/lib, downloadable from <https://dev.mysql.com/downloads/connector/j>
- edit /repository/conf/datasources/master-datasources.xml
 - for the datasources **WSO2_CARBON_DB** and **WSO2AM_DB**, make these changes:

```
<url>jdbc:mysql://localhost:3306/regdb</url>
<username>regadmin</username>
<password>regadmin</password>
<driverClassName>com.mysql.jdbc.Driver</driverClassName>
```

- launch the following scripts (for MySQL 5.7 use versions __.5.7.sql):

```
mysql -u regadmin -p -Dregdb < dbscripts/mysql.sql
mysql -u regadmin -p -Dregdb < dbscripts/apimgt/mysql.sql
```

3. AAC integration Authentication and Authorization Controller (AAC)

Clone the project that contains the AAC connector and WSO2 custom theme: <https://github.com/smartcommunitylab/API-Manager>

3.1. APIM-AAC connector

- build `wso2aac.client` project with Maven.
- copy `wso2aac.client-1.0.jar` from the project `API-Manager/wso2aac.client` to the WSO2 directory `repository/components/lib`

3.2. APIM configurations

- In `repository/conf/api-manager.xml`, change **APIKeyManager** and set **ConsumerSecret** with the value found in AAC for the client with `clientId` `API_MGT_CLIENT_ID`

```
<APIKeyManager>
  <KeyManagerClientImpl>it.smartcommunitylab.wso2aac.keymanager.
  ↪AACOAuthClient</KeyManagerClientImpl>
  <Configuration>
    <RegistrationEndpoint>http://localhost:8080/aac</
  ↪RegistrationEndpoint>
    <ConsumerKey>API_MGT_CLIENT_ID</ConsumerKey>
    <ConsumerSecret></ConsumerSecret>
    <Username>admin</Username>
    <Password>admin</Password>
    <VALIDITY_PERIOD>3600</VALIDITY_PERIOD>
    <ServerURL>https://localhost:9443/services/</ServerURL>
    <RevokeURL>https://localhost:8243/revoke</RevokeURL>
    <TokenURL>http://localhost:8080/aac/oauth/token</TokenURL>
  </Configuration>
</APIKeyManager>
```

- In `repository/conf/api-manager.xml`, uncomment **RemoveOAuthHeadersFromOutMessage** and set it to false

```
<RemoveOAuthHeadersFromOutMessage>false</RemoveOAuthHeadersFromOutMessage>
```

- In `repository/conf/identity/identity.xml`

– find

```
<OAuthScopeValidator class="org.wso2.carbon.identity.oauth2.validators.
  ↪JDBCSCOPEValidator"/>
```

– and replace with

```
<OAuthScopeValidator class="it.smartcommunitylab.wso2aac.keymanager.
  ↪CustomJDBCSCOPEValidator"/>
```

– and in **SupportedGrantTypes** section disable **saml2-bearer** and **ntlm** and add:

```
<SupportedGrantType>
  <GrantTypeName>native</GrantTypeName>
  <GrantTypeHandlerImplClass>it.smartcommunitylab.wso2aac.grants.
↪NativeGrantType</GrantTypeHandlerImplClass>
  <GrantTypeValidatorImplClass>it.smartcommunitylab.wso2aac.grants.
↪NativeGrantValidator</GrantTypeValidatorImplClass>
</SupportedGrantType>
```

- In repository/conf/carbon.xml, enable email username

```
<EnableEmailUserName>true</EnableEmailUserName>
```

- In repository/conf/user-mgt.xml, add the following property to <UserStoreManager>

```
<Property name="UsernameWithEmailJavaScriptRegEx">^\S{3,30}$</Property>
```

3.3. APIM theming

- copy the contents of project API-Manager/wso2.custom into the WSO2 directory

4. Keystore configuration

Import and add WSO2 certificate to the default keystore.

Linux

- `sudo rm -f cert.pem && sudo echo -n | openssl s_client -connect localhost:9443 | sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' > ./cert.pem`
- `sudo keytool -import -trustcacerts -file cert.pem -alias root -keystore JAVA_HOME/jre/lib/security/cacerts`

Windows

- `keytool -importkeystore -srckeystore <<wso2_root>>/repository/resources/security/wso2carbon.jks -destkeystore v`

Note: use a temporary password (such '123456') for destination keystore and PEM passphrase, empty password for origin and "wso2carbon" for wso2carbon password.

- `openssl pkcs12 -in wso2.p12 -out wso2.pem`
- Edit wso2.pem and keep only the part between —BEGIN CERTIFICATE— and —END CERTIFICATE—
 - `keytool -import -trustcacerts -file wso2.pem -alias root -keystore "%JAVA_HOME%/jre/lib/security/cacerts"`

Note: java cacerts default password is "changeit"

Warning: keytool is bugged in most java 8 versions, returning a java.util.IllegalFormatConversionException: d != java.lang.String

5. Proxy server configuration (Apache)

5.1. Configure proxy for apps

- Configure proxy publisher and subscriber apps: repository/deployment/server/jaggeryapps/publisher/site/conf/site.json (same for store):
 - context: /publisher
 - host: < mydomain.com >, e.g., am-dev.smartcommunitylab.it
- Configure management console: repository/conf/carbon.xml
 - <HostName>am-dev.smartcommunitylab.it</HostName>
 - <MgtHostName>am-dev.smartcommunitylab.it</MgtHostName>
- Configure WSO2 Tomcat reverse proxy: repository/conf/tomcat/catalina-server.xml
 - Add parameters to 9443 connector:

```
proxyPort="443"
proxyName="am-dev.smartcommunitylab.it"
```

- Configure Apache Virtual Host:
 - * port 80: redirect port 80 to 443
 - * port 443: ProxtPath and ProxyPathReverse / to ip:9443/

5.2. API Gateway

- Configure Gateway endpoint: repository/conf/api-manager.xml:

```
<GatewayEndpoint>http://api-dev.smartcommunitylab.it,https://api-dev.
smartcommunitylab.it</GatewayEndpoint>
```

- Configure axis2 transport Ins (http and https): add the following parameters:

```
<parameter name="proxyPort" locked="false">80</parameter>
<parameter name="hostname" locked="false">api-dev.smartcommunitylab.it</parameter>
```

- Configure Apache Virtual Host:
 - port 80: ProxtPath and ProxyPathReverse / to ip:8280/
 - port 443: ProxtPath and ProxyPathReverse / to ip:8243/

6. API-M Custom User Store Manager

In order to provide the necessary infrastructure for allowing API-M to interact with [Organization Manager](#) it is important to deploy the two new bundles that extend the existing [UserStoreManagerService](#) admin. This extension is done in order to permit the admin account to create,update,delete users and assign/revoke roles within specific tenants.

The configuration steps are the following:

- build [orgmanager-wso2connector](#) project with Maven.

- copy **apim.custom.user.store-0.0.1.jar** from the project `orgmanager-wso2connector/apim.custom.user.store` to the WSO2 directory repository/components/dropins
- copy **apim.custom.user.store.stub-0.0.1.jar** from the project `orgmanager-wso2connector/apim.custom.user.store.stub` to the WSO2 directory repository/components/dropins

As a result the new admin stub can be accessible from the following endpoint: https://protect\T1\textdollarAPIM_URL/services/CustomUserStoreManagerService

Configuration using docker

Docker resources for each of the component of the platform, including APIM, help you build generic Docker images for deploying the corresponding servers in containerized environments.

Configurations, custom JDBC drivers other than the default MySQL JDBC driver provided, extensions and other deployable artifacts are designed to be provided via volume mounts to the containers spawned.

Refer to the [APIM's repository docker files](#).

Set up the parameters in the `apim.env` file according to the explanation provided in the table below:

Property	Default	Description
APIM_USER	admin	The name of the admin user
APIM_PASS	admin	The password of the admin user
APIM_HOSTNAME	api-manager	The name of the running container of apim
APIM_REVERSEPROXY	api.platform.local	The name of reverse proxy in the nginx config
APIM_GATEWAYENDPOINT	api.platform.local	The value of gateway
ANALYTICS_HOSTNAME	am-analytics	The name of the running container of apim analytics
AAC_HOSTNAME	aac	The name of the container running AAC
AAC_CONSUMERKEY	f04ca519XXXX	The value of consumer key from AAC console for APIM Client App
AAC_CONSUMERSECRET	te181bf39XXXX	The value of consumer secret from AAC console for APIM Client App
AAC_REVERSEPROXY	aac.platform.local	The reverse proxy value for AAC
APIM_MYSQL_HOSTNAME	mysql	The name of the running container of mysql instance
APIM_MYSQL_USER	wso2carbon	MySQL db username
APIM_MYSQL_PASS	wso2carbon	MySQL db password

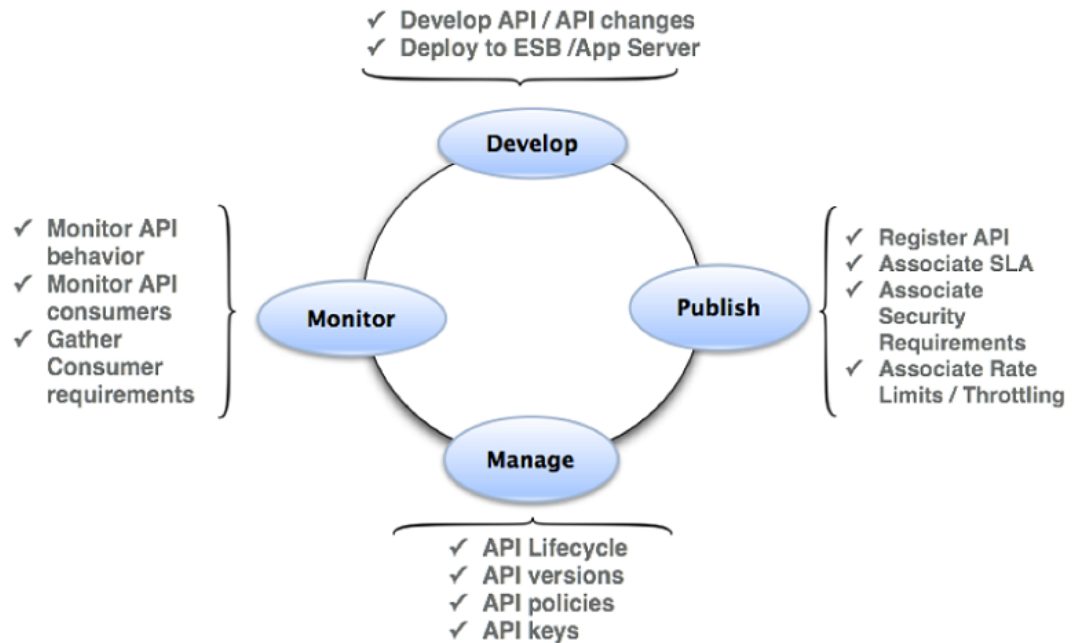
Keep in mind that, if you need to restart the container, you should *remove* it via `docker rm -f container_name_or_id` and then run it again. Stopping and restarting the container without removing it will result in an error.

API Manager Usage

WSO2 API Manager is a platform for creating, managing, consuming and monitoring APIs. It employs proven SOA best practices to solve a wide range of API management challenges such as API provisioning, API governance, API security and API monitoring.

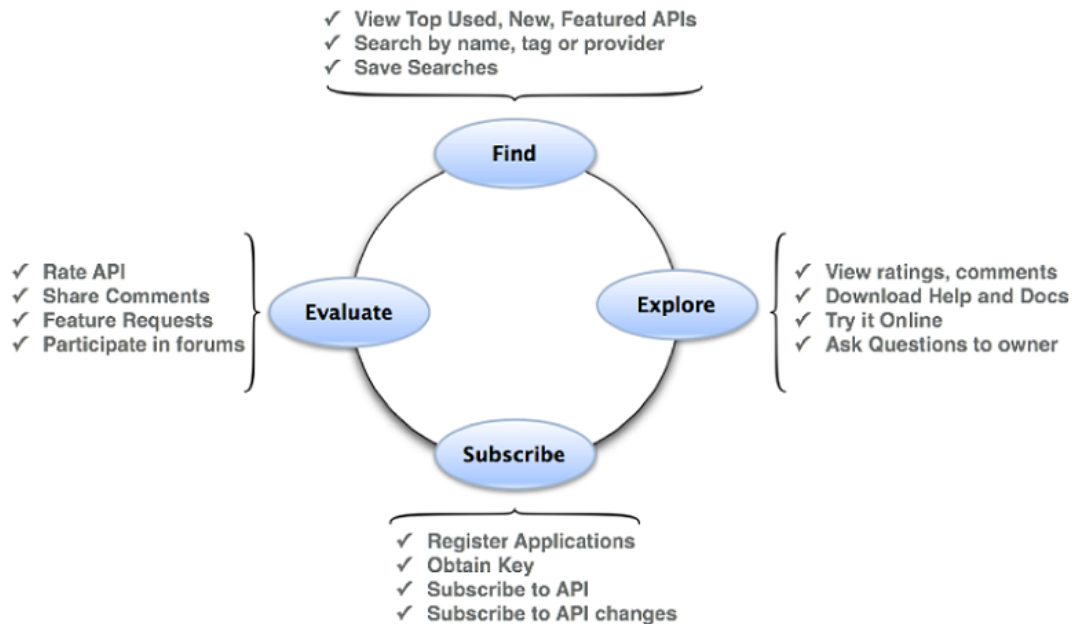
WSO2 API Manager comprises of several modules:

- **API Publisher: Define new APIs and manage them.** A structured GUI designed for API creators to develop, document, scale and version APIs, while also facilitating more API management-related tasks such as publishing APIs, monetization and analyzing statistics.



[LEARN MORE](#)

- **API Store: Browse published APIs and subscribe to them.** Enables developers to discover APIs, test them before consumption, calculate monetization, get feedback and make feature requests.



[LEARN MORE](#)

- **API Gateway: The underlying API runtime** Supports both forms of centralized and decentralized Gateway capabilities with high scalability characteristics to cater traditional and modern enterprise architectures. [LEARN MORE](#)

- **API Key Manager: Performs key generation and key validation functionalities.** The Key Manager is the STS (Security Token Service) that issues tokens that can be used by resource servers for authentication and authorization of requests. It supports a wide range of OAuth grant types and is capable of issuing both opaque and signed self contained tokens. [LEARN MORE](#)
- **API Traffic Manager: Performs rate limiting of API requests** Helps users to regulate API traffic, make APIs and applications available to consumers at different service levels, and secure APIs against security attacks. The Traffic Manager features a dynamic throttling engine to process throttling policies in real-time, including rate limiting of API requests. [LEARN MORE](#)
- **API Analytics: Provide reports, statistics and graphs on the APIs deployed in WSO2 API Manager** API Manager integrates with the WSO2 Analytics platform to provide reports, statistics and graphs on the APIs deployed in WSO2 API Manager. Enable Alerts to monitor APIs and react on anomalies. Gain insights by geography, user categories and more. Configure alerts to monitor these APIs and detect unusual activity, manage locations via geo location statistics and carry out detailed analysis of the logs. [LEARN MORE](#)

API Manager Benefits

- Reduce technology risk with extensible open source platform, rich documentation and community contributions.
- Make data-driven decisions through API usage, analytics and insights.
- The flexibility to deploy on-premise, use the public cloud, or both. Easily migrate in between these environments.
- Integrated platform increases team capabilities and efficiency.
- Designed to fit into Monolithic and Microservice Architectures with gateway and microgateway capabilities.
- Granular access control provides security down to the API level.
- Full Lifecycle API Management reduces need for future investments.

API Analytics

In order to enable and configure the API-M analytics for API Publisher and Subscriber, follow the [official instructions](#).

Some important performance tricks:

- Disable Analytics indexing. This can be done adding the following parameter to the Analytics start command:
`-DdisableIndexing=true`
- Change the default configuration for the analytics scripts: `Analytics Carbon Console > Manage > Batch Analysis > Scripts`. Set the appropriate CRON configurations for the script execution (e.g., each hour `0 0 * ? * * *` instead of each 2 mins). Remember that the configurations are reset after the server restart

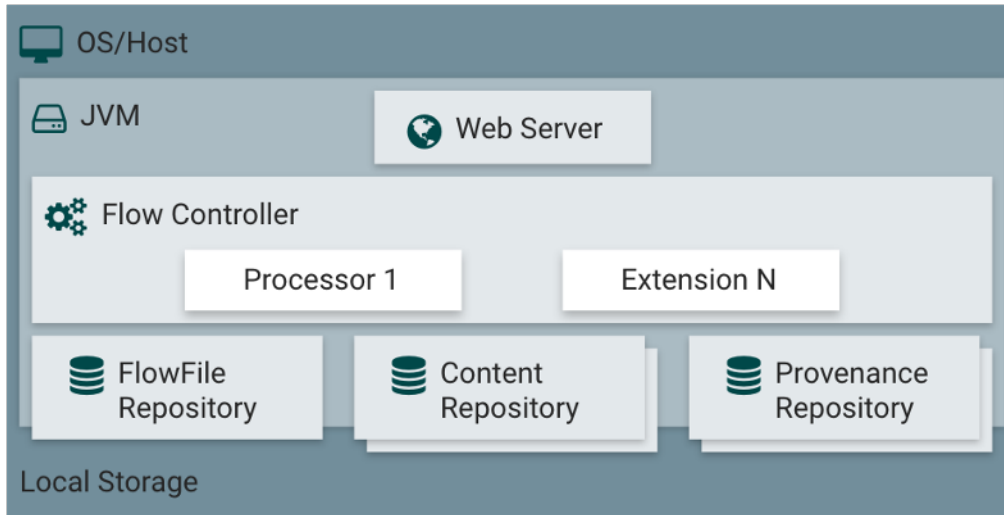
4.1 Overview

4.2 Data Wrangling

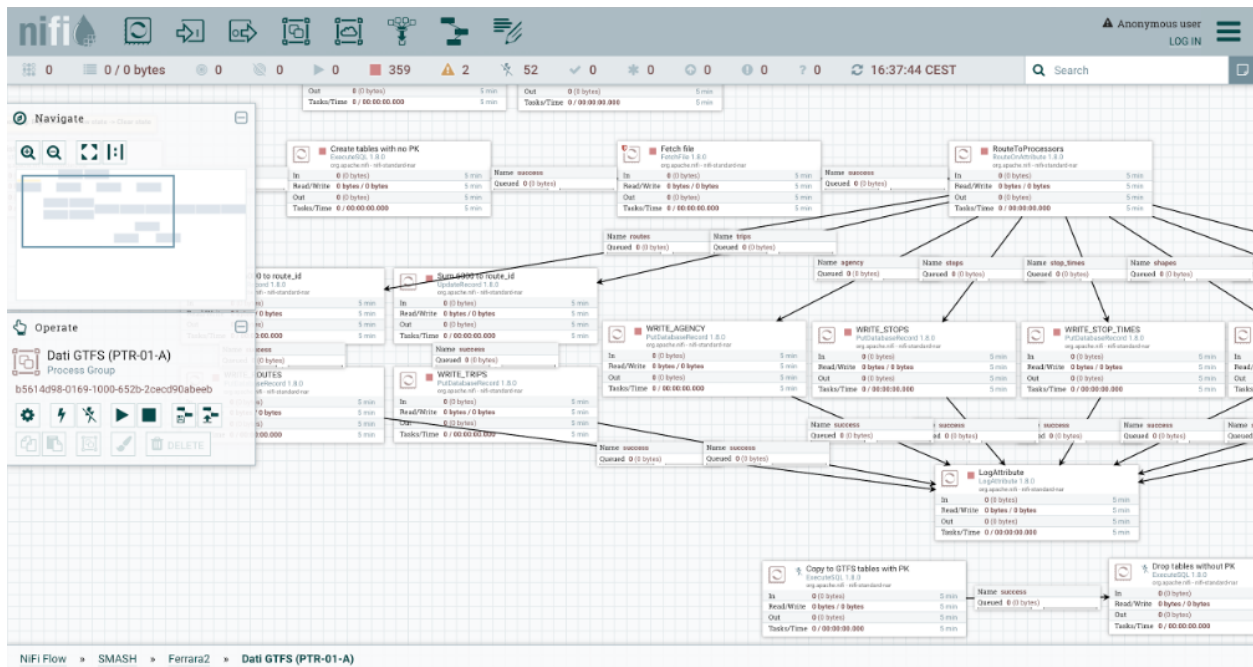
4.2.1 Apache Nifi

Apache NiFi offers the functionality of elaboration, transformation, and distribution of data. NiFi collects these data from the sources of different types and brings it to the corresponding destinations, where the data is stored for further exposure or analysis.

Apache NiFi consists of two elements: data flow modelling environment, where the data transformation process from the source to the destination is defined, and the run-time engine, where the process is being executed. This data elaboration process may be triggered in two different modes. In case of “active” sources (working in “push” mode), such as REST API of the Apache NiFi for process execution, message queues, etc, the process is activated externally. In case of “passive” sources, such as FTP or external APIs, the process may be activated according to the schedule defined by the developer. The intermediate nodes allow for various transformation activities, data aggregation, scripting, etc.



Differently from some other components, e.g., from Apache Kafka, the NiFi component is a “programmable” component and different developers may define and activate different data elaboration processes. These processes are composed of different execution nodes of type “source” (e.g., FTP file) and “processors” (e.g., format conversion), with their specific configuration depending on the nature of the node. NiFi defines a vast range of the pre-defined node types but allows for extensions with new sources and processors. In this way the definition of the processes becomes available for non-experts.



This figure, for example, defines a process for extracting the GTFS data from the ZIP archive available at the specific URL, and for storing the data in corresponding tables of a relational DB.

Apache NiFi Multitenancy Model

The idea of multi-tenancy in NiFi is that process groups represent tenants and have policies defined for them, listing which users or user groups are allowed to view or alter them. User groups are equivalent to teams, so if permission to

view a process group is given to a user group, all users belonging to it can view it.

Users will still be able to see other teams' process groups on the flow, but they will only appear as rectangles that they can neither interact with or view details of. The only information they can see about them is the amount of data they are processing.

The reason is that, if a user couldn't see another team's process group at all (without even knowing it exists), they may create objects on top of it, resulting in a cluttered flow in the eyes of anyone with permissions to view process groups of both teams.

Also, by being able to see the amount of data other process groups are processing, if there is a process group slowing down the whole flow, a user can tell where the problem resides. Without this ability, they would be puzzled, not knowing why their own process group isn't receiving any data.

The implementation of this model is delegated to the NiFi connector that creates the necessary groups and policies using the Apache NiFi REST API. When an organization is created in Organization Manager, a new process group will appear on the NiFi flow, named after the organization.

When new tenants for that organization are defined, each of them will be represented as a new process group nested within the organization's process group, named after the tenant. The NiFi connector will also create multiple user groups for those process groups, with different permissions.

Apache NiFi and AAC Integration

The integration of the Apache NiFi and the user management relies on the pre-defined OpenID Connect integration plugin of NiFi. In this way, the integration allows for SSO using the AAC directly, without the necessity to customize or extend the source code of NiFi. To accomplish this, it is necessary to configure a client app in AAC and use the client ID and secret within Apache NiFi configuration. The validation of the user and their roles within NiFi relies on the data previously stored through NiFi APIs.

The details about the Apache NiFi configuration for the OpenID Connect integration can be found here: <https://nifi.apache.org/docs/nifi-docs/>.

4.2.2 Nuclio

Nuclio (<https://nuclio.io>) is an open source serverless platform built on top of Kubernetes.

It is an highly optimized *Function-As-A-Service* solution, tailored for high performance computing, with minimal maintenance overhead and near-realtime performance.

Being written since its inception with data science and modern computing in mind, Nuclio offers a first-class support for data science (e.g. *machine-learning*) projects, with GPU acceleration and unlimited execution time.

FaaS

The main idea behind the *serverless* moniker is to focus on the development of code, while the execution environment is managed by a third-party, and the developers pay only for the resources consumed by the execution of their code, not for the allocation, provisioning and management of servers. It builds on the idea of *Platform-As-A-Service (PaaS)*, and moves ahead by decoupling application code from *data* and *states*, which are managed in external systems, leaving only the *business logic* in the application layer.

The enabling factor for going *serverless* is the adoption of a **Function-As-A-Service (Faas)** model, where single task applications are designed, developed and deployed as single, composable units of code which are executed in *stateless*, *ephemeral* containers managed by the underlying stack. Furthermore, functions are usually executed in an *event-driven* fashion, an approach which enables rapid and easy scalability, asynchronous execution plans and optimal resource consumption.

Nuclio is a *FaaS* platform, built on Kubernetes, which enables users into developing, executing and monitoring *functions* in a variety of programming languages such as *Python*, *Javascript*, *Java*, *Go* and deploy those blocks as scalable pods within the underlying cluster.

The platform will transparently and automatically handle tasks such as compiling code, building images, defining resources allocations, deploy pods, handle auto-scaling, wire connectivity and handle incoming requests via one of the numerous trigger protocols such as HTTP, MQTT, Kafka etc.

The adoption of the *serverless* pattern encourages developers to focus on well-defined units of business logic, without making premature optimization decisions related to how the logic is deployed or scaled.

As a result, development focus is on a single function or module rather than a service with a large surface area: serverless frees the developer from deployment concerns and allows them to focus on factoring the application following logical encapsulation lines, with tangible gains in speed of iteration, code complexity, quality and performance.

Furthermore, the usage of a pre-defined and highly optimized execution environment, via the encapsulation of the function code performed at compilation by the platform, enables the adoption of state-of-the-art handlers for triggers and connections, with performance and resource usage benefits that are obtained independently of the developers, which won't need to perform long and time-consuming optimization tasks.

Use cases

We can identify three main use cases and event sources:

- Synchronous: service is invoked and provides an immediate response in return (e.g. HTTP request)
- Async: push a message on a queue/broker which will trigger an action later (e.g. email or an S3 change notification)
- Streaming: a continuous data flow needs to be processed by a function and results put in another system

In every one of these cases a function, written in any of the supported programming languages, can fully satisfy the requirements and provide an easy and immediate way to deliver a working solution, without the overhead of defining a project, a stack for an API, a deployment configuration etc.

Some examples are:

- data collection in any form, being it source scraping, external acquisition, direct upload, message-based..
- data transformation, both as real-time streaming and as batch
- change-data-capture for database systems
- data processing, on-demand and scheduled or event-triggered
- data access

Any kind of job, which can be designed as a stateless unit whose execution is externally triggered (e.g. event based) can be easily written as a function, and immediately deployed as an high performance service with minimal effort and cost.

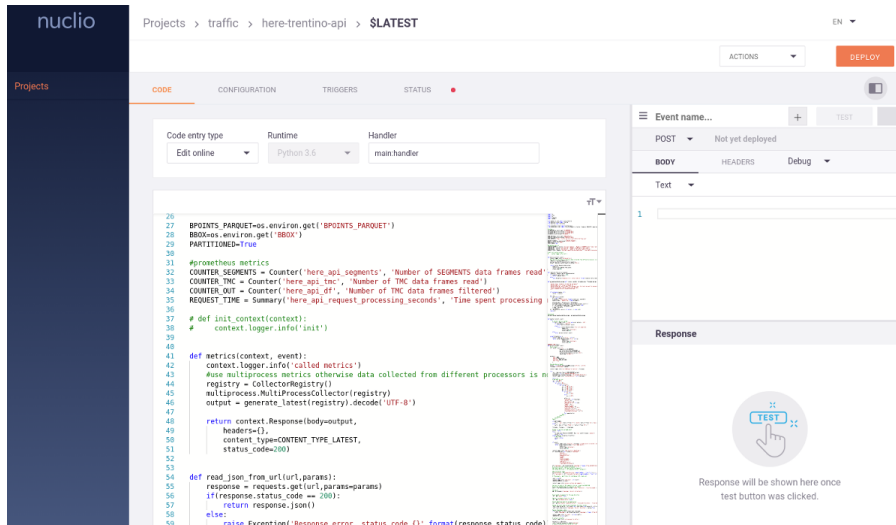
Nuclio dashboard and cli

While the core Nuclio platform is executed as a backend service inside the Kubernetes cluster, with no direct user interaction, the system provides additional components which can handle the:

- definition of functions
- configuration of execution environments
- configuration of deployments

- monitoring of function state
- development console
- log inspection

The **Nuclio dashboard** is the graphical user interface, which enables users to self-manage their platform and all the functions. The *DigitalHub* integration adds an oauth-based login to the web application, thanks to AAC.



A cli tool, named **nuctl**, can be installed locally on developers workstations and be used to perform the same set of operations as the dashboard, in a scriptable and easily operable way.

External access

Each function deployed within the serverless platform can be bounded to an *HTTP* trigger, which exposes a *service* for *synchronous* interaction. Given the fact that any function is executed as an independent *pod* in Kubernetes, we can discover that by enabling the HTTP trigger we can directly invoke the function by calling the dedicated *NodePort* published by Kubernetes.

This approach can satisfy many needs, but is limited to *in-cluster* access, since ports bounded to services are not exposed on the Internet.

To enable the development and testing of functions from the developers workstations, we can adopt 2 solutions:

- call the function via *Nuclio dashboard*, which acts as a *reverse proxy* and exposes a console for testing and debugging
- build a *mTLS tunnel* between the platform and the user pc, via *ghostunnel*

Instead, when we need to expose a function for a broader audience, we can leverage a completely different approach, which integrates with standard Kubernetes guidelines and implements an *Ingress* rule. Each function can thus be *mapped* to a specific path, usually the function name itself, and this information is then written as an annotation to the deployment in Kubernetes, as an *Ingress* definition. An *Ingress controller*, running inside the cluster, acts as a reverse proxy, and listens for requests and provides an autoconfiguring service, able to map path and ports and properly handle outside connectivity.

The result is a completely automated, observable and cloud-native configuration of ingress routes, which provides all the advantages of a proper proxy (SSL support, logging, rate limiting, auditing, authorization etc) to any function, without requiring a manual setup.

Workflows

In many cases functions are deployed and used alone, but sometimes they are invoked as a single processing block part of a bigger process which can be defined as a *workflow*.

While many dedicated tools are designed and developed from the ground up to satisfy the requirements of *workflows* and adopt *flow processing* (e.g. Nifi), serverless platforms are increasingly being used as the base layer for complex *function composition* processes.

Instead of adopting a single server to handle the whole process flow, the approach with serverless is to define many small processing blocks and deploy them as independent functions, which are then connected over some kind of *streaming platform* (such as Kafka) which collects, organizes, distributes and persists data to functions. The main advantages of this solution are *scalability*, *robustness*, *performance*, *observability* and the low level of *complexity* of the single blocks, but the overall design of the workflow and its global status can become complex and obscure.

One viable solution is the integration with **Kubeflow** (<https://www.kubeflow.org/>), an open source project aimed at offering a simple, portable and scalable framework for defining, executing and monitoring data science workflows on *Kubernetes*.

Digital hub integration

Nuclio is deployed inside the Kubernetes cluster on the digital hub cloud.

The integration relies on the definition of *namespaces* in Kubernetes, which isolate tenants in a multi-tenant environment and avoid the reachability of pods from the outside.

Any tenant will have access to a dedicated *Nuclio* deployment, which is a *namespace*, and each authorized user will be able to deploy a function via

- the nuclio *dashboard*
- the cli tool *nucll*

The *authentication* and *authorization* steps are performed via *AAC* for user access, and via *Vault* for credentials and secrets management.

The complete integration will give ensure that functions will be able to *acquire* dynamically the credentials required for the integration with backend services as databases, S3 storage, message brokers etc. thanks to the *role delegation* performed via *AAC* and *Vault*.

This solution will avoid the need to create, store and distribute dedicated credential sets for any given function, or even worse the adoption of a single set of credentials for the whole platform. Instead, by guaranteeing the availability of the *Vault* at runtime, and thanks to the integration with Kubernetes and Nuclio, we could be able to obtain at startup the required accounts, within the *least-required privilege* principle.

Installation

Production-ready deployments require the usage of Kubernetes as cluster-engine. For instructions follow <https://nuclio.io/docs/latest/setup/k8s/getting-started-k8s/>

In order to provide a locally usable development environment, Nuclio also supports the execution within *Docker*, with limitations in terms of performance, scalability and automation support. The use case for Docker is to evaluate the platform and locally test the functions in an initial phase.

To run a local instance of the platform execute the following:

```
docker run -p 8070:8070 -v /var/run/docker.sock:/var/run/docker.sock -v /tmp:/tmp --  
→name nuclio-dashboard quay.io/nuclio/dashboard:stable-amd64
```


By browsing to <http://localhost:8070>, users will be able to create a project and add a function.

When run outside of Kubernetes, the dashboard will simply deploy to the local Docker daemon. Developers will be able to use the usual docker tools to inspect the logs, start/stop the container and access the console.

4.3 Data Management

4.3.1 Resource Manager

The ResourceManager is a core component of the data platform. It has the task to manage resources such as databases or file stores for the various tenants, acting as a *mediator* for managing the lifecycle of resources.

In a complex, multi-tenant and multi-provider environment, it is fundamental to adequately segregate users and roles, mainly for **security**, **privacy** and **accountability** reasons. The ResourceManager is uniquely positioned in the platform architecture to fulfill all these requirements, by acting as:

- a *coordinator* which provides users with the appropriate resources for their scope
- a *manager* which intermediates access to resource providers, avoiding the necessity for a delegation of administrative access to end users for self-management. Admin accounts are managed by the platform, and used exclusively by the resource manager to create **resources** and **limited access credentials** for end users
- a *logger/auditor* for events related to resources, such as creation or removal
- a *registry* for resources and access credentials

Additionally, the ResourceManager can act as **dispatcher** for resources, with the unique ability of connecting external **consumers** to resources, such as data exploration, visualization or analytics softwares.

Note: The software is currently in **beta**, not suitable for deployment outside of test environments.

Design

The core of the ResourceManager is build around 3 components:

- the *resource service*, which alongs the local repository handles the storage and retrieval of resource entities
- the *provider service*, which is responsible for the communication with backend resource providers and handles the *creation*, *update*, *check* and *delete* of resources
- the *consumer service*, which providers resource consumers with details of resource-related events, along with all the data required for resource access such as *user credentials*, *connection URIs* and properties.

The various components are interconnected both in a **synchronous way**, via direct service call, and in an **asynchronous way**, via an internal event bus built on *message delivery*.

The idea is to provide an agile and extensible building block for the data platform, with a small deployment footprint, completely manageable via API calls to fulfill the *integration*, *automation* and *observability* requirements.

Architecture

The software is divided into 4 logical layers:

- the *access layer* which exposes the API (REST controllers, services), the user interface, the Swagger UI and documentation

- the *service layer*, which handles the business logic, the authorization and permissions system, the event bus and the system configuration
- the *persistence layer* which handles the safe storage of resource definitions and consumer registrations
- the *adapter/connectors layer*, which provides the required glue between external actors (providers and consumers) and the service layer

Additionally, specific components are vertically integrated to offer *audit*, *log* and other common functionalities.

The modular system allows the development of specific *providers* and *consumers* by implementing well-defined interfaces, and the configuration system permits the activation of only specific classes.

Scopes

One of the project goals is providing a way to properly share resource providers between different, isolated and dedicated **scopes**. Do note that this does not mean that *resources* are shared between projects, tenants or users, but that the *providers*, as in database clusters, storages etc, can be used at the same by different entities while keeping the single resources isolated.

Scopes are the internal definition for areas which logically *contain* resources and consumers. A group of users belonging to the same scope will see the same slice of the system: all the resources and consumers defined for the current scope, and nothing else.

The advantage of sharing backend systems, in a proper multi-tenant way, with clear boundaries relies on proper *scoping* of resources inside backends, and at the same time on **single-scope user credentials**, which ensure that users won't have the ability to access resources owned by another scope.

This approach must also ensure that no name collision between isolated scopes happens: since users have no visibility of other scopes, they can not verify if a given identifier (for example a database *name*) is already used on the backend server. At the same time, backend systems do not possess information about *scopes* and thus can not autonomously decide. This means that, without proper supervision, **id collisions** could happen. Depending of the kind of systems involved, this could mean a simple error or the unwanted sharing of a single resource between different scopes.

As an example, let's review the following cases:

- an SQL provider such as PostgreSQL will throw an error on creation of a database if the name is already taken, leading to an **error**
- a NoSQL provider such as MongoDB, via automatic collection creation, will implicitly create a database on a write access, or reuse one if it already exists, leading to **data sharing**

The ResourceManager, thanks to the **scope** concept, properly isolates tenants by ensuring both **collision-free scoped identifiers**, guaranteeing uniqueness on any given provider, and by providing dedicated **scoped user credentials**, which ensure limited access to backend systems.

Security

Due to its unique positioning in the platform architecture, the ResourceManager is able to ensure a proper separation of duties between *end users* managing their resources and accessing them via the various internal or external consumers, and the *administrators* which install, configure and manage resource providers such as RDBMS, cloud storage backends, compute units etc.

By leveraging administrators from the tedious task of creating resources for end users, the ResourceManager can at the same time ensure that the **privileged credentials** required to manage resource providers are never leaked to end-users, by contextually providing **single-scope user credentials** for each and every resource.

This design choice ensures that at any given time, the leakage or exposition of a single account won't give access to other resources offered by the same provider. While potentially cumbersome for end-users, the security and privacy objectives are valued as primary targets for the data platform.

Since the ResourceManager is given the task of creating user credentials for direct resource access, **secure storage** of sensitive data is a mandatory requirement. An encryption schema, based on AES symmetric cipher, ensures that data at rest in the local database is properly secured, and decrypted only for authorized access.

Properties for encryption configuration are :

```
#encrypt
encrypt.enabled=true
encrypt.key=*****
```

These can be defined via an `application-local.properties` file or via ENV variables.

Future development should introduce:

- additional per-scope keys for storage, or externally-provided keys (for example KMS or SSE-C approaches)
- temporary credentials generation for limited scope/time access to resources

In particular, an hypothetical system composed of an *identity provider*, a *key provider* and the *ResourceManager* as separated entities could ensure that all the user credentials, along with any user-provided data such as custom properties, annotations and eventually *data exports* are adequately protected via encryption, all within a Zero-Trust deployment where not even the manager has direct access to protected data.

Authentication and Authorization

The ResourceManager is a backend component of the data platform. As such, its target is to provide a single service without replicating the capabilities of other pre-existing components. User *authentication* is one of these shared functionalities, which can be delegated to a dedicated **identity provider**. The system supports external authentication via **OAuth2**, where a user accessing the API must provide a valid **access token**, an opaque string generated by the identity provider after a successful authentication. Via backend call to a validation endpoint, the ResourceManager can retrieve the user **identification** for the *bearer token*, along with all the information required to properly **authorize** the request.

The ResourceManager does employ a scoped *Role-Based Access control*, where roles are bounded to the specific scope (ie tenant/group/project...). The permissions related to the *entities* and the relative *actions* are statically connected to a set of roles, where each of these defines a specific model in the system.

- `ROLE_ADMIN` identifies the administrator for the scope, which can perform **any operation on any entity**, and has complete access to all the data
- `ROLE_RESOURCE_ADMIN` identifies a limited administrator with all the permissions related to *resource* management and access, and no permissions for the *consumers*
- `ROLE_CONSUMER_ADMIN` identifies a limited administrator, with all the permissions on *consumers* but no management rights for *resources*
- `ROLE_USER` identifies an **authenticated user** which has complete *read-access* to both resources and consumers, included protected data, but no admin rights.

As previously stated, roles are bounded to a **scope**, which is the system definition of a shared space between users. This means that a single user can possess different roles, impersonating the relative actor, within different domains, for example a mere `USER` in a given shared scope, and `ADMIN` in another personal space.

The configuration for authentication and authorization relies on an external OAuth2 provider, which should ensure user authentication and then provide the information about roles assignment for any given scope. If a user does not possess a role in a given scope, it is deemed as *unauthorized* by the backend.

Additionally, a **scope filter** can be configured via the `scope.list` property, which requires that any requested or claimed scope is included in the given list. This can be leveraged to limit access to a given resource manager to only users belonging to a given group or tenant.

The properties for configuration are :

```
#oauth
rest.security.issuer-uri=
security.oauth2.client.client-id=resourcemanager
security.oauth2.client.client-secret=*****
security.oauth2.client.access-token-uri=${rest.security.issuer-uri}/protocol/openid-
↪connect/token
security.oauth2.client.user-authorization-uri=${rest.security.issuer-uri}/protocol/
↪openid-connect/auth
security.oauth2.resource.id=resourcemanager
security.oauth2.resource.token-info-uri=${rest.security.issuer-uri}/protocol/openid-
↪connect/token/introspect
security.oauth2.resource.user-info-uri=${rest.security.issuer-uri}/protocol/openid-
↪connect/userinfo
security.oauth2.resource.jwk.key-set-uri=${rest.security.issuer-uri}/protocol/openid-
↪connect/certs

#scopes
scopes.enabled=true
scopes.default=default
scopes.list=
scopes.roles.mapping.admin=resourcemanager/<scope>:ROLE_ADMIN
scopes.roles.mapping.resourceAdmin=
scopes.roles.mapping.consumerAdmin=
scopes.roles.mapping.user=resourcemanager/<scope>:ROLE_USER
```

These can be defined via an `application-local.properties` file or via ENV variables.

Providers

Resource providers are backend services which offer a specific kind of resource, such as *SQL databases*, *NoSQL* data stores, *Object or file storages*. While management of such backend services is demanded to operators and platform managers, **resources** lifecycle is directly handled by end users via `ResourceManager`.

A *provider adapter* connects the management stack with a specific instance of a particular software, such as a single PostgreSQL database server. When an authorized user requests the creation of a new resource, the adapter will translate the operation into a series of backend calls, via native protocols, aimed at creating a single resource, uniquely assigned to a single-scope user credential, which will then be made available to end-users.

The same translation process happens for all the significative operations supported by the platform:

- **CREATE** requires the creation of a resource to the provider, with an optional name and provider-specific properties
- **UPDATE** communicates the modification of user-editable properties such as attributes, tags or extensions
- **CHECK** instructs the adapter to verify the existance and accessibility of the resource and the related user-account, and to eventually fix access problems
- **DELETE** requires the removal of the resource, the user account and all the related data, such as tables, temporary files, caches, backups etc

Future development could introduce additional operations such as:

- **EXPORT** to extract data from a backend into a downloadable archive

- `IMPORT` to import a source dataset into a newly instantiated resource, or deploy a model
- `BACKUP` to require a server-side backup of a resource, along with all the data, for versioning or data recovery
- `CLONE` to quickly reproduce a project or an environment via cloning resources and data repositories along with all the data
- `CLEANUP` to restore a resource to a clean state, which could be empty or conform to an seed template

Currently, the software assumes that any given provider has a single instantiation, since the configuration is provided statically at startup and can not be changed during execution. This means that, in a case where multiple resource backends happen to offer the same provider, multiple resource managers would be deployed, in a *one-to-one* mapping where each would exclusively manage a single instance of the provider.

A future evolution could introduce support for multiple providers of the same class, if the current model results too limited.

The provider system supports **unmanaged resources**, which are externally defined and only registered into the ResourceManager. This mode enables the propagation of resource events and configurations to *consumers*, without at the same delegating the administrative rights to the resource manager. The resources `ADDED` in this way will be effectively *read-only* for the system and the users. The removal of an unmanaged resource will simply delete the registration, leaving the provider and the real resource intact.

Consumers

Resource consumers are softwares which connect to resource *providers* in order to leverage their capabilities and data. As an example, data exploration softwares like *SQLPad*, *Dremio* or data visualization and analysis such as *Superset* or *Metabase*.

Setting up these softwares requires the insertion of connection information, along with credentials, for each resource in each and every provider. This tedious task, entirely performed by end users in a self-managed system, can and should be automated in order to provide:

- auto-configuration
- consistency
- error prevention
- audit
- security along privilege separation

By demanding configuration to the ResourceManager, users will access such tools with limited credentials, without delegating the ability to administrate resource connections and spaces. Instead, the ResourceManager will handle the privileged access, and provide configuration detail for each supported resource, depending on both providers and consumers, according to:

- scope
- resource type
- resource provider
- tags

Future improvements could eventually provide additional security, by defining **single-consumer credentials** for each resource, with appropriate privileges connected with the specific capabilities of the software. For example, a data visualization component could leverage *read-only* access with limited visibility of sensitive data, while a CRUD application could receive full *read-write-delete* privileges over the same resource, with a dedicated user account.

Tags

Tags are user-defined strings which can be used to describe *resources* and *consumers*. The main objective of their usage is the grouping of entities under logical groups, which are meaningful for end users. Additionally, *tags* are used internally to filter resources and events during the dispatch of actions, effectively limiting the visibility to consumers.

For example, a consumer such as SQLPad tagged with `developement` could obtain visibility on *development* resources as staging and test databases, and enable users to explore, validate and analyze data, while on the same **scope** a consumer tagged with `production` such as a reporting tool would never gain access to development resources.

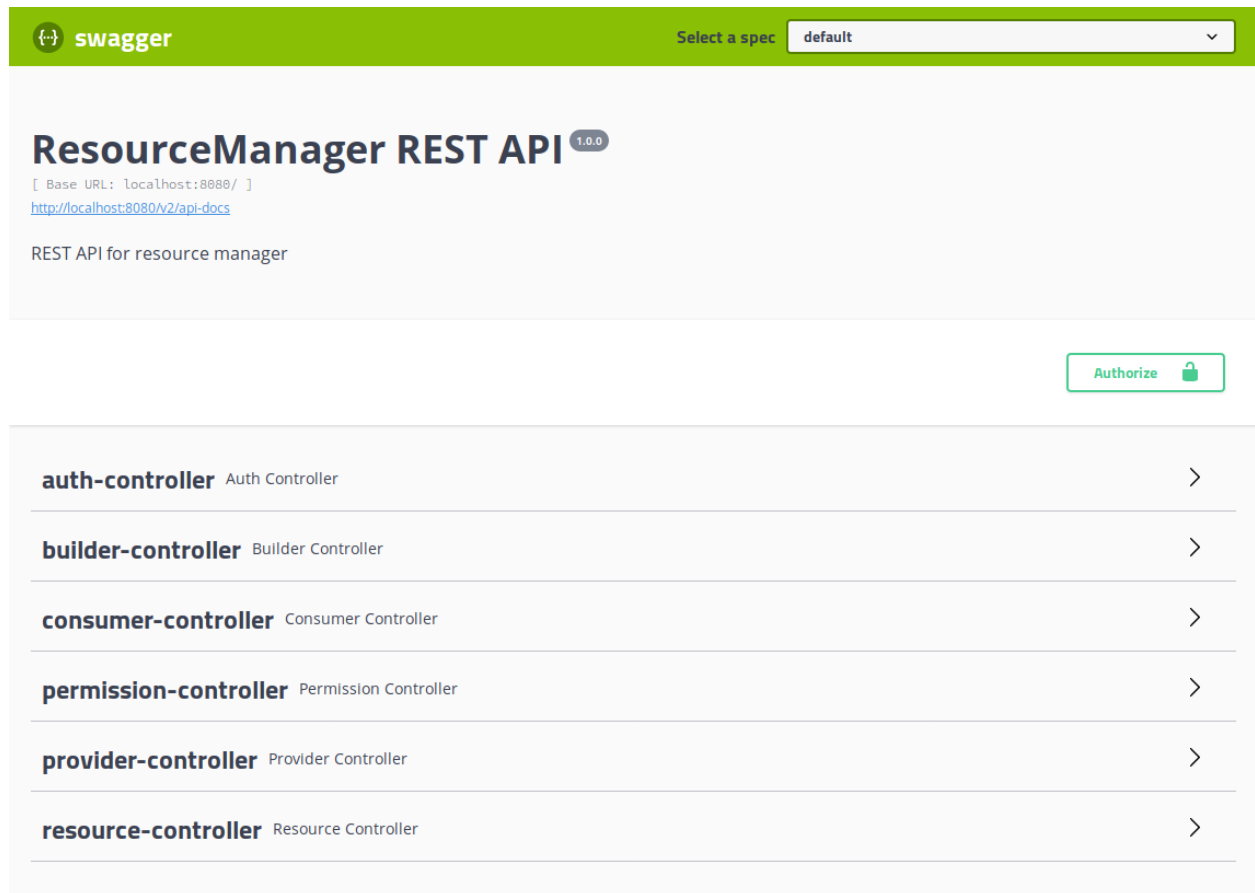
API

The software exposes a REST API, accessible via HTTP at `http://resourcemanager:8080/api`.

Every management and access operation, both for *resource* and *consumer* entities is available via API. Exposed endpoints are:

- `resources` for resource management
- `consumers` for consumer management
- `builders` for inquiring the available consumer builders
- `providers` for inquiring the avialble resource providers
- `auth` for performing UI login via OAuth2
- `permission` for introspection of user permission and roles, used by UI after auth

For detailed documentation and a playground, access the included **SwaggerUI** at `http://resourcemanager:8080/swagger-ui.html`.



The image shows the Swagger UI for the ResourceManager REST API. At the top, there's a green header with the Swagger logo and a dropdown menu labeled 'Select a spec' with 'default' selected. Below the header, the title 'ResourceManager REST API' is displayed with a version badge '1.0.0'. Underneath, it shows the base URL 'localhost:8080/' and a link to 'http://localhost:8080/v2/api-docs'. A description 'REST API for resource manager' is also present. On the right side, there is an 'Authorize' button with a lock icon. Below this, a list of controllers is shown, each with a name, a description, and a right-pointing chevron icon:

- auth-controller** Auth Controller
- builder-controller** Builder Controller
- consumer-controller** Consumer Controller
- permission-controller** Permission Controller
- provider-controller** Provider Controller
- resource-controller** Resource Controller

API access requires a valid `Authorization` header, which should carry a valid `Bearer` token obtained by clients from the OAuth2 provider. The backend will validate the token and use it to recover user information such as *username*, *scopes* and *roles*.

The ResourceManager is natively *multi-tenant* thanks to *scopes*.

As such, each API call should include the specific *scope* either via:

- HTTP Header `X-Scope`, example `X-Scope: tenant123`, or
- PATH `/api/c/<scope>/`, example `/api/c/tenant123/resources/1`

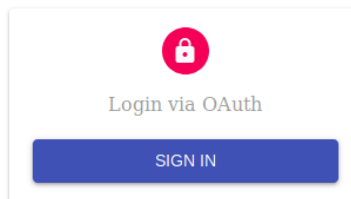
When requests do not carry a *scope* identifier, the system will refer them to the **default scope**, as configured in the relative property. This can be used to deploy a *single-tenant* instance, with only a single scope enabled and set as default, and avoid the hassle of including it into each and every call.

User Interface

Included with each ResourceManager release, a React-admin based user interface provides a simplified graphical way for the management of resources and consumers. The UI respects authentication and roles, same as the backend.

Login

Access the interface at `http://resourcemanager:8080` and login via OAuth2.



Resource management

List and filter resources for a given scope.

Resources

Resources

Consumers

Id

Type

Provider

Managed

tags

> 12

sql

postgresSql

✗

two ONE

SHOW

> 18

sql

cockroachDB

✓

SHOW

> 19

object

minio

✓

SHOW

Rows per page: 10 1-3 of 3

Resource detail

Access resource details along with connection information and user credentials.

The screenshot shows the 'Resource detail' page for 'Resource #12'. The page has a blue header bar with a menu icon, the title 'Resource #12', a refresh icon, and a user profile icon. Below the header, there is a sidebar with a grid icon and a list icon. The main content area has a tabbed interface with four tabs: 'SUMMARY' (selected), 'STATUS', 'PROPERTIES', and 'TAGS'. An 'EDIT' button with a pencil icon is located in the top right corner of the main content area. The 'SUMMARY' tab displays the following information:

Id	12
Type	sql
Provider	postgresSql
Uri	psql://postgres:dsaa@172.18.0.2:5432/default_test_yypvd

Resource creation

Create a new resource from available providers, either via **creation** for managed or **registration** for unmanaged, which will require the full URI.

Create a Resource

type

sql

provider

postgresSql

properties

+

 ADD

Managed

SAVE

Consumer management

List and filter consumers

Consumers

+ CREATE

EXPORT

Id	Type	Consumer	tags	
> 13	sql	sqlpad	test one	SHOW
> 14	sql	dremio	ONE the123	SHOW

Rows per page: 10 1-2 of 2

Consumer creation

Create a new consumer, based on available builder, by providing connectin details and consumer-specific properties as for example *endpoint*, **username** etc.

Note: The UI is currently a beta software, with limited scope (only *default**) and provided AS-IS mainly for testing and development.

Building from source

The codebase can be cloned via `git` from the software repository on the local computer.

```
git clone
```

The software is a SpringBoot project, built with Maven. As such, given the availability of JDK 1.8 and all the dependencies, the software can be built via:

```
mvn clean install
```

The software will lack the UI, which is separately managed.

The *user interface* is in a dedicated repository and is a React (JS) application. NPM can serve as both the package manager and the build handler.

In order to package the interface and all the assets (images, css, libs) into a deployable artifact, execute:

```
npm run package
```

This will produce a `dist` folder with all the resulting files. To compile a `release` build of the whole stack, copy all these assets into the `public resources` folder of the maven project.

```
cp dist/* $PRJ_FOLDER/src/resources/public/
```

Then rebuild the maven project

```
mvn clean install
```

The resulting artifact will contain both the backend and the UI, which will be served at the same port by the internal Jetty http server.

Installation

Obtain the release artifact, then configure the `application.properties` either via config file, with a `*-local.properties` file in the classpath, or via environmental variables.

At minimum, the software expects a valid configuration for:

- local JPA repositories
- OAuth2
- scopes default and role mapping
- at least one resource provider

Example configuration

```
# scopes
scopes.enabled=true
scopes.default=default
scopes.list=
scopes.roles.mapping.admin=resourcemanager/<scope>:ROLE_ADMIN
scopes.roles.mapping.resourceAdmin=
scopes.roles.mapping.consumerAdmin=
scopes.roles.mapping.user=resourcemanager/<scope>:ROLE_USER

# encrypt
encrypt.enabled=true
encrypt.key=aNdRgUkXp2s5v8y0

# oauth
rest.security.issuer-uri=http://localhost:8180/auth/realms/test
security.oauth2.client.client-id=resourcemanager
security.oauth2.client.client-secret=*****
security.oauth2.client.access-token-uri=${rest.security.issuer-uri}/protocol/openid-
↪connect/token
security.oauth2.client.user-authorization-uri=${rest.security.issuer-uri}/protocol/
↪openid-connect/auth
security.oauth2.resource.id=resourcemanager
security.oauth2.resource.token-info-uri=${rest.security.issuer-uri}/protocol/openid-
↪connect/token/introspect
security.oauth2.resource.user-info-uri=${rest.security.issuer-uri}/protocol/openid-
↪connect/userinfo
security.oauth2.resource.jwk.key-set-uri=${rest.security.issuer-uri}/protocol/openid-
↪connect/certs

# providers
providers.mysql.enable=true
providers.mysql.host=172.17.0.2
providers.mysql.port=3306
providers.mysql.username=root
providers.mysql.password=secret-pw
```

(continues on next page)

(continued from previous page)

```
# consumers
consumers.log.enable=true
```

4.3.2 Dremio

Dremio is a *Data-as-a-Service* platform, which enables data analysts and scientists to autonomously explore, validate and curate data from a variety of sources, all in a single, unified and coherent interface. Built for teams, Dremio leverages spaces and virtual datasets to offer a data platform with the following features.

The official website is <https://www.dremio.com>.

The official documentation with details on how to use Dremio is <https://docs.dremio.com/>.

Data Sources

Dremio supports modern data lakes built on a variety of different systems and provides:

- native integrations with major RDBMS such as PostgreSQL, MySQL, MS SQL, IBM DB2, Oracle
- NoSQL integration for modern datastores as MongoDB, Elasticsearch
- support for file based datasources, cloud storage systems, NAS

Data Exploration

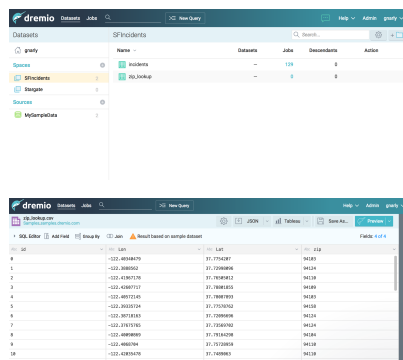
Dremio offers a unified view across all datasets connected to the platform, with:

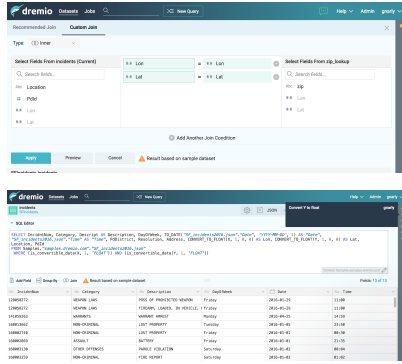
- live data visualization during query preparation and execution, with dynamic previews
- optimized query pushdown for all sources in native query language
- virtual datasets based on complex queries available as sources for analytics and external tools

Cloud Ready

Dremio is architected for cloud environments, with elastic computing abilities and dynamic horizontal scaling. Data reflections can be stored into distributed storage platforms such as *S3*, *HDFS*, *ADLS*.

Screenshots from the Website





Installation

Dremio is a Java software and requires a compatible JDK installed. The current version supports only OpenJDK 1.8 and Oracle JDK 1.8.

System requirements are:

- a supported Linux distribution: RHEL/CentOS 6.7+/7.3+, SLES 12+, Ubuntu 14.04+, Debian 7+
- at least 4 CPU cores and 8GB RAM for starting the software.

Given the nature of data analysis, and the distributed design of the software, production deployments should follow the following indications:

Node role	Hardware required
Coordinators	8 CPU/16GB RAM recommended
Executors	4 CPU/16GB RAM minimum 16 CPU/64GB RAM recommended

You can read more at <https://docs.dremio.com/deployment/rpm-tarball-install.html>.

Platform Fork

The integration of Dremio into the Digital Hub platform required extending the open source version, which lacks some enterprise features, to support:

- **external user authentication** via OAuth2.0 and OpenID Connect
- **multitenancy** (see [Multitenancy and Organizational Model](#))

Without these extensions, Dremio supports internal authentication only and grants administrator privileges to all users, hence every user can access any resource indiscriminately.

As far as **authentication** is concerned, the following features have been implemented:

- OAuth2.0 support, with access via the secure *authorization_code* flow and a native Dremio token integration for UI
- automatic user creation and personal space (user home) definition upon valid OAuth2.0 access
- distinction between ADMIN role and USER role, which reflects on the UI in that admin actions and menus are hidden to unprivileged users
- OAuth2.0 login in the UI

Additionally, the upstream support service, which exposes metrics, interactive chat and debug information to dremio.com for licensed enterprise environments, is disabled by default. This should be reviewed in privacy-sensitive environments, as the complete deactivation of user and session data leakage to dremio.com and its partners requires the explicit configuration of various properties in `dremio.conf`.

The **multitenancy model** implemented in the fork is structured as follows:

- admin privileges are not assignable, ADMIN (Dremio admin or system admin) role is reserved to `dremio` user, every other user is assigned either TENANT ADMIN role or USER role
- each user is associated to a single tenant
- the tenant is attached to the username with the syntax `<username>@<tenant>`
- all APIs accessible to regular users are protected so that non-ADMIN users can only access resources within their own tenant
- when a resource belongs to a tenant (i.e. is shared among all its users), such tenant is specified as a prefix in the resource path with the syntax `<tenant>__<rootname>/path/to/resource`

In Dremio, resources are either containers (spaces, sources, homes) or inside a container (folders, datasets), therefore spaces and sources are prefixed with their tenant, while folders and datasets inherit it from their container, which is the root of their path, and do not need to be prefixed. For example, in the following resource tree, `myspace`, `myfolder` and `mydataset` all belong to `mytenant`:

```
mytenant__myspace
├── myfolder
│   └── mydataset
```

The ADMIN user can access any resource. Regular users (i.e. tenant admins and users) can only access resources inside their own home or belonging to their tenant. This implies that users can only query data and access job results according to these constraints.

Note: Currently, when non-ADMIN users create a new source or space (sample sources included), that is **automatically prefixed** with their own tenant. Non-ADMIN users cannot create sources or spaces with a different tenant than their own.

Configuration for OAuth2.0

Note: The configuration described below uses [AAC](#) as the authentication provider, however any standard OAuth2.0 provider can be used.

1. Configuring a client application on AAC

On your AAC instance, create a new client app named `dremio` with the following properties:

- Identity providers : `internal`
- Redirect URIs: `<dremio_url>/apiv2/oauth/callback`
- Grant types: `authorization_code`
- Authentication methods: `client_secret_basic`, `client_secret_post`, `none`
- Token type: `JWT`

- Selected scopes: user.roles.me, user.spaces.me, openid, profile, email

Under “Hooks & Claims”, set:

- Unique spaces prefix: components/dremio
- Custom claim mapping: enable
- Custom claim mapping function:

```
function claimMapping(claims) {
  var valid = ['ROLE_USER'];
  var owner = ['ROLE_OWNER'];
  var prefix = "components/dremio/";

  //fetch username where we find it
  var username = claims["username"];
  if(!username) {
    username = claims ["preferred_username"];
  }
  if(!username) {
    username = claims ["email"];
  }

  if ("spaceRoles" in claims && "space" in claims) {
    var space = claims["space"];
    //can't support no space selection performed
    if (Array.isArray(space)) {
      space = null;
    }
    //lookup for policy for selected space
    var tenant = null;
    if(space) {
      for (var role of claims["spaceRoles"]) {
        if (role.startsWith(prefix + space + ":")) {
          var p = role.split(":")[1]

          //replace owner with USER
          if (owner.indexOf(p) !== -1) {
            p = "ROLE_USER"
          }

          if (valid.indexOf(p) !== -1) {
            tenant = space
            break;
          }
        }
      }
    }

    if (tenant) {
      tenant = tenant.replace(/\.\/g, '_');
      claims["dremio/tenant"] = tenant;
      claims["dremio/username"] = username+'@'+tenant;
      claims["dremio/role"] = "admin";
    }
  }

  return claims;
}
```


This function adds a custom claim holding a single user tenant, as AAC supports users being associated to multiple tenants while Dremio does not. During the authorization step on AAC, the user will be asked to select which tenant to use.

2. Configuring Dremio

Open your `dremio.conf` file and add the following configuration:

```
services.coordinator.web.auth: {
  type: "oauth",
  oauth: {
    authorizationUrl: "<aac_url>/oauth/authorize"
    tokenUrl: "<aac_url>/oauth/token"
    userInfoUrl: "<aac_url>/userinfo"
    callbackUrl: "<dremio_url>"
    jwksUrl: "<aac_url>/jwk"
    clientId: "<your_client_id>"
    clientSecret: "<your_client_secret>"
    tenantField: "dremio/tenant"
    scope: "openid profile email user.roles.me user.spaces.me"
    roleField: "dremio/role"
    jwtIssuer: "<expected_token_issuer>"
    jwtAudience: "<expected_token_audience>"
  }
}
```

The `tenantField` property matches the claim defined in the function above, which holds the user tenant selected during the login. Dremio will associate it to the username with the syntax `<username>@<tenant>`. That will be used as username in Dremio.

The `roleField` property matches another claim defined in the function, which holds the role of the user (either “user” or “admin”) within the selected tenant. Such roles correspond to READ and WRITE privileges over tenant data.

Additionally, to fully disable `dremio.com` intercom, add also:

```
services.coordinator.web.ui {
  intercom: {
    enabled: false
    appid: ""
  }
}
```

Building from Source

Dremio is a *maven* project, and as such can be properly compiled, along with all the dependencies, via the usual `mvn` commands:

```
mvn clean install
```

Since some modules require license acceptance and checks, in automated builds it is advisable to skip those checks to avoid a failure:

```
mvn clean install -DskipTests -Dlicense.skip=true
```

The `skipTests` flag is useful to speed up automated builds, for example for Docker container rebuilds, once the CI has properly executed all the tests.

During development of new modules or modifications, it is advisable to disable the *style-checker* via the `-Dcheckstyle.skip` flag. In order to build a single module, for example *dremio-common*, use the following syntax:

```
mvn clean install -DskipTests -Dlicense.skip=true -Dcheckstyle.skip -pl :dremio-common
```

To test the build, you can execute only the *distribution* module, which will produce a complete distribution tree under the `distribution/server/target` folder, and a **tar.gz** with the deployable package named *dremio-community-{version}-{date}-{build}*, for example `./distribution/server/target/dremio-community-3.2.1-201905191350330803-1a33f83.tar.gz`.

```
mvn clean install -DskipTests -Dlicense.skip=true -pl :dremio-distribution
```

The resulting archive can be installed as per upstream instructions.

Note: The first time you open Dremio, you will be asked to create an administrator account. The admin user **must** have the username `dremio`, as that is currently the only user that can have admin privileges.

Additional Changes in the Fork

Source Management

Differently from the original implementation, in which source management was restricted to ADMIN only, users with TENANT ADMIN role are allowed to manage (create, update and delete) sources in addition to spaces *within their tenant*, while the other users can only manage spaces.

Arrow Flight and ODBC/JDBC Services

While internal users can use their credentials to connect to Dremio Arrow Flight server endpoint and ODBC and JDBC services, users that log in via OAuth need to set an internal password in order to connect to Dremio with some client. Such password can be set in the Dremio UI on the Account Settings page.

Connecting WSO2 DSS to Dremio via JDBC

The fork includes an [OSGi bundle](#) for Dremio JDBC Driver that can be used with WSO2 Data Services Server. In order to use it, copy the JAR file to `<DSS_PRODUCT_HOME>/repository/components/dropins` and restart DSS.

DSS Datasource Configuration

A DSS data source can be connected to Dremio by configuring the following properties:

- Datasource Type: RDBMS
- Database Engine: Generic
- Driver Class: `com.dremio.jdbc.Driver`
- URL: `jdbc:dremio:direct=localhost:31010`

- User Name: <dremio_username>
- Password: <dremio_password>

When you create a datasource that connects to Dremio, you will likely get a warning on the DSS console that a default logger will be used for the driver logs.

Dremio APIs

Many features of Dremio are available via the Dremio REST API. Two versions of the API currently coexist:

- v2 is still used internally, although it should be dismissed in the future
- v3 is documented on the Dremio docs as the official REST API and is progressively replacing v2 also internally

Here is a collection of all the **v3 endpoints** with links to the corresponding Dremio docs pages, if any. Note that access to some stats APIs has been restricted to ADMIN (i.e. Dremio system admin) in the fork, while regular users have been granted access to source management APIs (if they are tenant admins). The required permission is marked in **bold** in the tables whenever it differs from the official documentation.

The API path is <dremio_url>/api/v3.

Catalog API:

Path	Method	Docs	Permis- sion
/catalog	GET	https://docs.dremio.com/rest-api/catalog/get-catalog.html	user
	POST	https://docs.dremio.com/rest-api/catalog/post-catalog.html	user
/catalog/{id}	GET	https://docs.dremio.com/rest-api/catalog/get-catalog-id.html	user
	POST	https://docs.dremio.com/rest-api/catalog/post-catalog-id.html	user
	PUT	https://docs.dremio.com/rest-api/catalog/put-catalog-id.html	user
	DELETE	https://docs.dremio.com/rest-api/catalog/delete-catalog-id.html	user
/catalog/{id}/refresh	POST	https://docs.dremio.com/rest-api/catalog/post-catalog-id-refresh.html	user
/catalog/{id}/metadata/refresh	POST	Refresh of physical dataset metadata	user
/catalog/by-path/{path}	GET	https://docs.dremio.com/rest-api/catalog/get-catalog-path.html	user
/catalog/search	GET	Item research given a query string	user
/catalog/{id}/collaboration/tag	GET	https://docs.dremio.com/rest-api/catalog/get-catalog-collaboration.html	user
	POST	https://docs.dremio.com/rest-api/catalog/post-catalog-collaboration.html	user
/catalog/{id}/collaboration/view	GET	https://docs.dremio.com/rest-api/catalog/get-catalog-collaboration.html	user
	POST	https://docs.dremio.com/rest-api/catalog/post-catalog-collaboration.html	user

Reflection API:

Path	Method	Docs	Permis- sion
/reflection	POST	https://docs.dremio.com/rest-api/reflections/post-reflection.html	user
/reflection/{id}	GET	https://docs.dremio.com/rest-api/reflections/get-reflection-id.html	user
	PUT	https://docs.dremio.com/rest-api/reflections/put-reflection.html	user
	DELETE	https://docs.dremio.com/rest-api/reflections/delete-reflection.html	user
/dataset/{id}/reflection	GET	Reflections used on a dataset	user
/dataset/{id}/reflection/recommendations	POST	Reflections recommended for a dataset	user

Job API:

Path	Method	Docs	Permis- sion
/job/{id}	GET	https://docs.dremio.com/rest-api/jobs/get-job.html	user
/job/{id}/results	GET	https://docs.dremio.com/rest-api/jobs/get-job.html	user
/job/{id}/cancel	POST	https://docs.dremio.com/rest-api/jobs/post-job.html	user
/job/{id}/reflection/{reflectionId}	GET	Retrieval of a reflection job status	user
/job/{id}/reflection/{reflectionId}/cancel	POST	Cancellation of a running reflection job	user

SQL API:

Path	Method	Docs	Permission
/sql	POST	https://docs.dremio.com/rest-api/sql/post-sql.html	user

User API:

Path	Method	Docs	Permission
/user	POST	User creation	admin
/user/{id}	GET	https://docs.dremio.com/rest-api/user/get-user.html	user
	PUT	User update	user
/user/by-name/{name}	GET	https://docs.dremio.com/rest-api/user/get-user.html	user

Cluster Statistics API:

Path	Method	Docs	Permission
/cluster/stats	GET	Stats about sources, jobs and reflections	admin

Job Statistics API:

Path	Method	Docs	Permission
/cluster/jobstats	GET	Stats about the number of jobs per type over ten days	admin

User Statistics API:

Path	Method	Docs	Permission
/stats/user	GET	Stats about user activity	admin

Info API:

Path	Method	Docs	Permission
/info	GET	Basic information about Dremio	user

Source API (deprecated in favour of Catalog API, will be removed):

Path	Method	Docs	Permission
/source	GET	https://docs.dremio.com/rest-api/sources/get-source.html	user
	POST	https://docs.dremio.com/rest-api/sources/post-source.html	user
/source/{id}	GET	https://docs.dremio.com/rest-api/sources/get-source.html	user
	PUT	https://docs.dremio.com/rest-api/sources/put-source.html	user
	DELETE	https://docs.dremio.com/rest-api/sources/delete-source.html	user
/source/type	GET	https://docs.dremio.com/rest-api/sources/source-types.html	user
/source/type/{name}	GET	https://docs.dremio.com/rest-api/sources/source-types.html	user

4.3.3 SQLPad

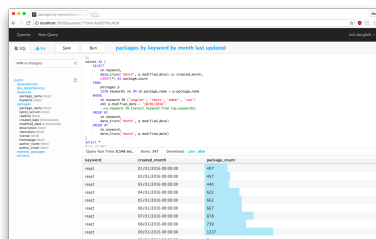
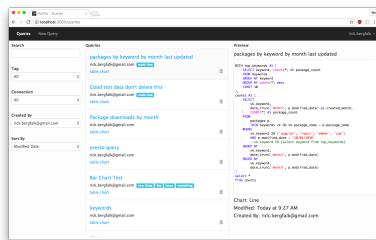
SQLPad is a web app which enables end users to connect via browser to various SQL servers, explore data by writing and running complex SQL queries, and eventually visualize the results. Additionally, SQLPad includes a simple chart interface able to produce interactive graphs based on SQL results.

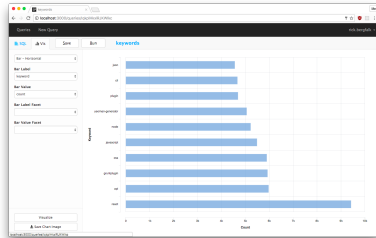
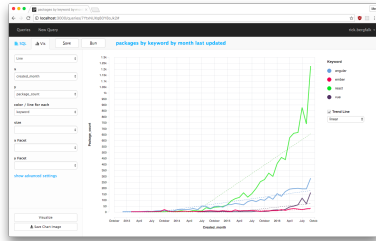
SQLPad is the ideal tool for data analysts and scientists who want to perform data exploration and visualization via SQL, all via an intuitive and simple web app.

Reference website <https://rickbergfalk.github.io/sqlpad/>

Screenshots

from website





Installation

SQLPad is a Node.js application. As such, it can be installed via NPM and then directly run:

```
npm install sqlpad -g
sqlpad
```

Alternatively, SQLPad can be executed via Docker, thanks to the official image available at DockerHUB: <https://hub.docker.com/r/sqlpad/sqlpad/>

Configuration for both the Docker version and the standalone one can be expressed via environmental variables, eg More at <https://rickbergfalk.github.io/sqlpad/installation-and-administration/>

Platform fork

In order to integrate SQLPad with OAuth2 login, we have forked the original project at <https://github.com/smartcommunitylab/sqlpad>

This version, based on the stable **v2 branch** enables the integration of SQLPad with an OAuth2 compliant identity provider, such as **AAC**.

The configuration for establishing the connection requires the following settings (via ENV):

- **PUBLIC_URL**: Public URL used for OAuth setup and links in email communications. Protocol is expected to be provided. Example: <https://mysqlpad.com>
- **OAUTH_AUTHORIZATION_URL**: Authorization endpoint.
- **OAUTH_TOKEN_URL**: Token endpoint.
- **OAUTH_USERINFO_URL**: UserInfo endpoint.
- **OAUTH_CLIENT_ID**: Client ID.
- **OAUTH_CLIENT_SECRET**: Client Secret.

Additionally, the updated branch implements a basic role authorization schema, which can distinguish between *admins* (with the ability to manage users and connections) and *users* (which can write and execute SQL queries). In order to leverage this functionality, the following settings are required:

- `OAUTH_ROLE_FIELD`: OAuth token mapping field for Role discovery, example *authorization*.
- `OAUTH_ROLE_PREFIX`: OAuth token claim prefix for roles, example *sqlpad*:

Available roles are *admin/users*. Following the example configuration, a token exposing the following claim enables the user to access as admin:

```
{ authorization: ["sqlpad:admin"]}
```

while the following would map to the *user* role:

```
{ authorization: ["sqlpad:user"]}
```

When the claim is missing or does not contain a valid mapping, access is rejected. Note: Tokens are fetched from UserInfo endpoint.

Building from source

Since SQLPad is a Node.js application, developers need to install `node v8` or later. After cloning the repository, install all the dependencies via npm:

```
npm install
```

The code for *frontend* and *backend* are separated into different subfolders. In order to compile an executable bundle, run the `build.sh` script to compile and package the UI:

```
scripts/build.sh
```

Afterwards, the `server.js` script can be run via node:

```
cd server
node server.js --dir ../db --port 3010 --base-url '/sqlpad'
```

This will execute the backend application, which will provide access to both the API backend and the static bundled frontend UI over the configured port.

ODBC

Starting from version 2.7.0, SQLPad supports ODBC connections.

As per doc <https://github.com/rickbergfalk/sqlpad/wiki/ODBC>:

To enable this support, your setup must meet the `odbc` module requirements, as well as the requirements for building native modules in node.js via `node-gyp`.

When `odbc` can successfully build, an `odbc` option should be available on the connections page.

An initial default query (using SQL standard `INFORMATION_SCHEMA`) has been provided to pull the schema from your `odbc` connection, but it may not work with the database you are using, and you may need to provide an alternate query.

4.4 Data Storages

4.4.1 Minio

Minio (<https://min.io/>) is a high-performance distributed object storage system, open source, which provides us with a scalable, lightweight and secure data lake. Minio is an open source *S3 Compatible* storage system, which can be directly used by every application able to understand the *Simple Storage Service (S3)* protocol.

Being purpose-built to serve objects instead of files or data tables, Minio can be used to store unstructured/structured/semi-structured data inside a flat addressing space, while at the same time offering advanced querying and lifecycle capabilities.

Minio can be used to store any kind of data in the same space, independently of size, type and structure, and access it via S3 over the network, all while respecting and enforcing user permissions, policies and data-retention locks as defined by administrators.

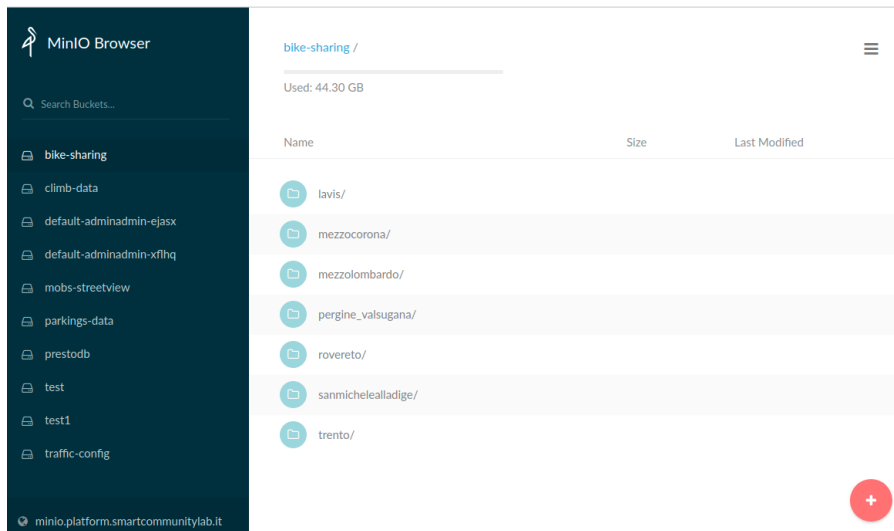
One of the design principles of object storage is to abstract the lower layers away from users and applications. Thus, data is exposed and managed as objects instead of files or blocks. Object storage allows the addressing and identification of individual objects by more than just file name and file path, it adds a unique identifier for each object within a bucket, or across the entire system, to support much larger namespaces and eliminate name collisions.

Furthermore, Minio explicitly separates file metadata from data to support additional capabilities: instead of a fixed set of metadata available in file systems (filename, creation date, type, etc.), object storage provides for full function, custom, object-level metadata, where users can store any kind of metadata meaningful for the application, such as secure hashes, signatures, tags etc.

Minio Browser

An additional package available within the Minio distribution, the *Minio Browser* is a graphical user interface dedicated to management of objects within the storage system. Structured as an independent web application, it can be accessed over the network via browser and enables users to upload, download, delete, move, rename objects of any size and kind.

Screenshots



Installation

Being a *cloud-native* application, Minio can be installed by pulling an image within a cloud environment, either local or remote, and deploy a container attached to a permanent storage such as physical or virtual disks, network shares or cloud storage systems.

In order to test locally it is enough to leverage *Docker* to run the container via

```
$ docker pull minio/minio
$ docker run -p 9000:9000 minio/minio server /data
```

and then the service will be accessible over the port 9000.

In a proper cloud environment, *Kubernetes* will provide the base stack and the administrators will be able to either leverage the preconfigured *helm charts* or the custom *operator*. See <https://docs.min.io/docs/minio-deployment-quickstart-guide.html> for further information.

Digital Hub integration

A cloud native system such as Minio is inherently designed to be utilized concurrently by many different users and tenants. Minio natively supports:

- multiple users
- multiple spaces (*buckets*) with dedicated access and lifecycle policies
- complex policies with the ability to discriminate actions and capabilities for users based on various attributes
- externally provided *Key Management Systems (KMS)* to securely handle and distribute encryption keys

Nevertheless, it is mandatory for a properly usable product to integrate a management system, able to create, manage and destroy resources and coordinate user and policies.

The *Resource Manager* integration provides a central place for handling different kind of resources, such as minio buckets, by offering users a self-action portal, all while ensuring the correct management of policies and permissions.

The RM will enrich user-created buckets with a dedicated access policy, which will enable all tenant members to access the system, as dictated by *AAC* and *OrgManager* roles.

Furthermore, the *Minio Browser* will be integrated directly with *AAC* in order to enable direct access via user credentials, by converting at login tenants and groups to the proper bucket policy defined by the *RM* at bucket creation.

Eventually, end users will be able to autonomously provision a private bucket in Minio via *Resource Manager*, and the directly access the Browser with *AAC* Single-Sign-On (SSO), while their applications will dynamically provision access credentials via *RM* and *Vault*.

This approach ensures that credentials and policies will be securely handled, without limiting end user capabilities or introducing additional steps.

4.4.2 PostgreSQL

PostgreSQL (<https://www.postgresql.org/>) is a powerful, flexible and advanced open-source relational database system which provides a secure, stable and reliable storage for structured datasets, by leveraging the * Relational Database Management System (RDBMS) * model to build a consistent, safe and performing data layer.

At the same time, PostgreSQL is a general purpose and object-relational database management system, with *user-defined* data types, functions and functional languages, posing as a solid and reliable platform for many useful extensions which can add interesting functionalities to the base.

PostGIS

PostGIS (<https://postgis.net/>) is an open source *OCG compliant* spatial database extender for PostgreSQL. It add advanced spatial and geographical capabilities, and specific geometry data types to the database.

Among many capabilities, notable features are:

- processing and analytic functions for both vector and raster data
- 3D measurement functions, 3D spatial index support and 3D surface area types
- seamless raster/vector analysis support
- network topology support
- SQL/MM topology support
- spatial reprojection SQL callable functions
- support for importing and rendering vector data from textual formats such as KML, GeoJSON, WKT
- rendering raster data in various image formats such as GeoTIFF, PNG, JPG

In order to enable PostGIS in a specific database, after installing the plugin, it is mandatory to leverage the SQL console to activate the various extensions needed:

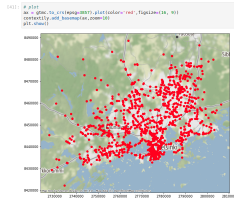
```
-- Enable PostGIS (as of 3.0 contains just geometry/geography)
CREATE EXTENSION postgis;
-- enable raster support (for 3+)
CREATE EXTENSION postgis_raster;
-- Enable Topology
CREATE EXTENSION postgis_topology;
-- Enable PostGIS Advanced 3D
-- and other geoprocessing algorithms
-- sfcgal not available with all distributions
CREATE EXTENSION postgis_sfcgal;
-- fuzzy matching needed for Tiger
CREATE EXTENSION fuzzystmatch;
-- rule based standardizer
CREATE EXTENSION address_standardizer;
-- example rule data set
CREATE EXTENSION address_standardizer_data_us;
-- Enable US Tiger Geocoder
CREATE EXTENSION postgis_tiger_geocoder;
```

DigitalHub integration

In order to properly utilize and share the database system with the various tenants, the hub will leverage the *ResourceManager* as the only operator able to create, alter and manage databases within the server, and will also integrate users via dedicated read/write roles.

This approach ensures that no user will be able to interfere and access datasets of other tenants, while giving at the same time authorized operators the ability to autonomously create and manage databases within their own slice of the share PostgreSQL cluster.

Credentials will be securely handled by the platform, thanks to the integration between AAC and *ResourceManager*, ensuring that users and applications will use a properly scoped and limited access to the system.



Installation

The Jupyter notebook can be installed locally on a pc by using a *Python* package manager such as `pip` or `conda`. While Jupyter can run code in many languages, to install the application Python is a requirement. Either one of the following instructions will install Jupyter:

```
$ pip install jupyterlab
```

for *pip*, otherwise when using *conda* type:

```
$ conda install -c conda-forge jupyterlab
```

(see <https://jupyter.readthedocs.io/en/latest/install.html>)

If the local environment does not provide the necessary packages, it is possible to leverage *anaconda* to install everything required. See <https://jupyter.readthedocs.io/en/latest/install.html> for detailed instructions.

Do note that all the libraries, packages and tools needed to execute the code will need to be manually installed via the same package manager used to install *jupyterlab*, to avoid inconsistencies and conflicts.

Alternatively, an easier, more modern and immediate way to experience Jupyter is to leverage *docker* to run a local container with everything installed and pre-configured, ready to use.

Project Jupyter maintains a set of images with up-to-date dependencies and stacks, pre-configured for various scenarios such as

- minimal-notebook
- scipy-notebook
- pyspark-notebook
- datascience-notebook
- tensorflow-notebook

and many others at DockerHub <https://hub.docker.com/u/jupyter>.

See the complete guide at <https://jupyter-docker-stacks.readthedocs.io/en/latest/using/selecting.html#jupyter-datascience-notebook>.

Running a notebook is as easy as launching the chosen image and access via browser to the local URL as detailed by the execution log

```
$ docker run -p 8888:8888 jupyter/scipy-notebook:2c80cf3537ca
```

```
Executing the command: jupyter notebook
```

```
[I 15:33:00.567 NotebookApp] Writing notebook server cookie secret to /home/jovyan/.  
→ local/share/jupyter/runtime/notebook_cookie_secret
```

```
[W 15:33:01.084 NotebookApp] WARNING: The notebook server is listening on all IP_  
→ addresses and not using encryption. This is not recommended.
```

```
[I 15:33:01.155 NotebookApp] Serving notebooks from local directory: /home/jovyan
```

(continues on next page)

(continued from previous page)

```
[I 15:33:01.156 NotebookApp] 0 active kernels
[I 15:33:01.156 NotebookApp] The Jupyter Notebook is running at:
[I 15:33:01.157 NotebookApp] http://[all ip addresses on your system]:8888/?
↪token=112bb073331f1460b73768c76dff2f87ac1d4ca7870d46a
[I 15:33:01.157 NotebookApp] Use Control-C to stop this server and shut down all
↪kernels (twice to skip confirmation).
[C 15:33:01.160 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
    http://localhost:8888/?token=112bb073331f1460b73768c76dff2f87a
```

For further instructions see <https://jupyter-docker-stacks.readthedocs.io/en/latest/index.html>.

JupyterHub

JupyterHub is a project aimed at bringing the Jupyter Notebook to groups of users, without the need to install it locally. Thanks to the hub, users are able to access, over the network, a dedicated instance hosted on a shared computational environment.

Every kind of user, from students to researchers and data scientists, can get their work done in their own dedicated workspace on shared resources which can be managed efficiently by system administrators.

JupyterHub is aimed at cloud environments, and is natively deployable to *kubernetes clusters* in an automated, scalable and configurable way.

The approach taken by the hub is to dynamically instantiate a pod when authorized users need it, running the chosen Jupyter Notebook image, and then after a given amount of inactivity destroy the pod and release the resources to the cluster.

The administrators can personalize and pre-configure the set of images available to users, which will then be able to autonomously install packages into their own ephemeral environment as needed.

Obviously, by employing *non-persistent* containers to serve users, all the data and code uploaded and produced during the lifetime of the pod will be lost. While this may be of little inconvenience for testing and small tasks, many use cases require the persistence of data and code during different sessions.

The idea that we will explore in building the *DigitalHub* is to consolidate all the persistent data into an external storage, being it a database, a data-warehouse or even a cloud object storage, and then automatically connect those resources to the Jupyter Notebook environment, in order to let users transparently access them. This way, all the temporary data can be left on locally attached disks, to maximize performances and minimize the latency, and the datasets, outputs and relevant objects will be persisted and eventually versioned and tracked inside the data lake.

The same approach will be taken for the scripts and code blocks, this time leveraging *git* to automatically version and sync the code to a dedicated repository.

Ideally, users will be able to dynamically provision a complete, ready to use computing environment within seconds, either on the shared cloud or inside a personal workstation, and access all the datasets and code all the time.

4.5.2 Apache Spark

Spark (<https://spark.apache.org/>) is a general-purpose distributed data processing engine, suitable for use in a wide range of circumstances.

The official tagline is *unified analytics engine for large-scale data processing*.

On top of the its core data processing engine, Spark offers libraries for SQL, machine learning, graph computation, and stream processing, which can be used together in an application.

Programming languages supported by Spark include: Java, Python, Scala, and R.

Application developers and data scientists integrate Spark into their applications to rapidly query, analyze, and transform data at scale, also thanks to the availability of a number of high-level operators that make it easy to build and scale parallel applications.

Some of the tasks most frequently associated with Spark are:

- ETL and SQL batch jobs across huge data sets
- processing of streaming data from sensors (e.g. IoT)
- machine learning tasks
- complex analytics
- data integration

Spark is able to handle tasks which span several petabytes of data at a time, distributed across a cluster of thousands of physical or virtual servers.

The main advantage of Spark is the ability to build complex and very performing data pipelines, which combine very different techniques and processes into a single, coherent whole. In modern cloud computing, the discrete tasks of acquiring, selecting, transforming and analyzing data usually requires the integration of many different systems and processing frameworks. Thanks to Spark, developers have the ability to combine these together, crossing the boundaries between batch, streaming, and interactive workflows.

Furthermore, Spark executes tasks as in-memory operations, with the option to *spill to disk* intermediate results when memory constrained. This approach guarantees a fast data pipeline, easy to configure, run and maintain.

4.5.3 Presto

Presto (<https://prestosql.io/>) is an open source distributed SQL query engine, originally developed at Facebook, which is able to optimally respond to all the complex analytical queries needed by data scientists by performing fast in-memory operations and directly leveraging the data lake abilities.

The main factor which drives the adoption of such a tool is its unique ability to natively query data directly from where it is stored, being it a RDBMS, an object storage such as Minio, or even an Hadoop cluster, without the need for complex, costly and lengthy ingestion steps, which usually involve copying and transforming data via ETL pipelines. Instead, by directly connecting to backend systems which hold the data, and by leveraging their native capabilities and querying languages, Presto can offer an unified ANSI SQL console with a complete view over the datasets.

By performing in-memory computations, and leveraging a distributed query engine optimized for low latency, Presto is able to offer remarkable performance for queries which span very large datasets (up to *petabytes*), stored inside different backend systems, all in a transparent way.

Furthermore, Presto offers many advanced capabilities, such as spatial and geographical support, which can be used independently of the presence of the same tools in the backend system.

When paired with an advanced storage system such as Minio, Presto can exploit the advanced querying features of the *S3 Select* protocol and minimize the amount of data transferred from the backend systems, while also leveraging the push-down of predicates to further reduce the amount of work needed to compile the query results.

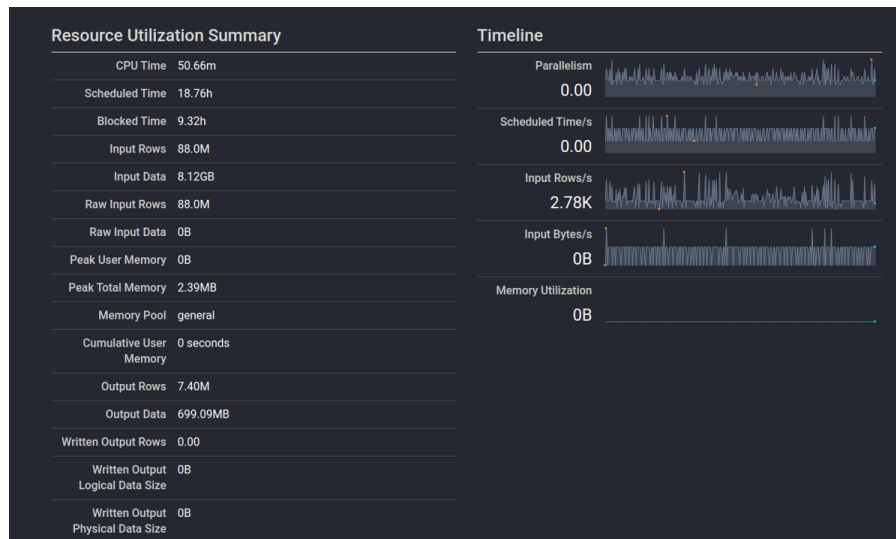
Digital Hub integration

Given the specialized and complex tasks performed by Presto, and the vast amount of resources needed to properly handle large datasets, the integration is still in a preliminary phase.

The design of the system requires a direct integration between *Presto* and the *Resource Manager*, in order to make Presto aware of the various storage systems and datasets such as RDBMS databases, document collections or object storage buckets.

Furthermore, *Presto* will have to discover or obtain proper access credentials, either via user delegation through AAC or via direct access to the credentials *Vault*.

Screenshots



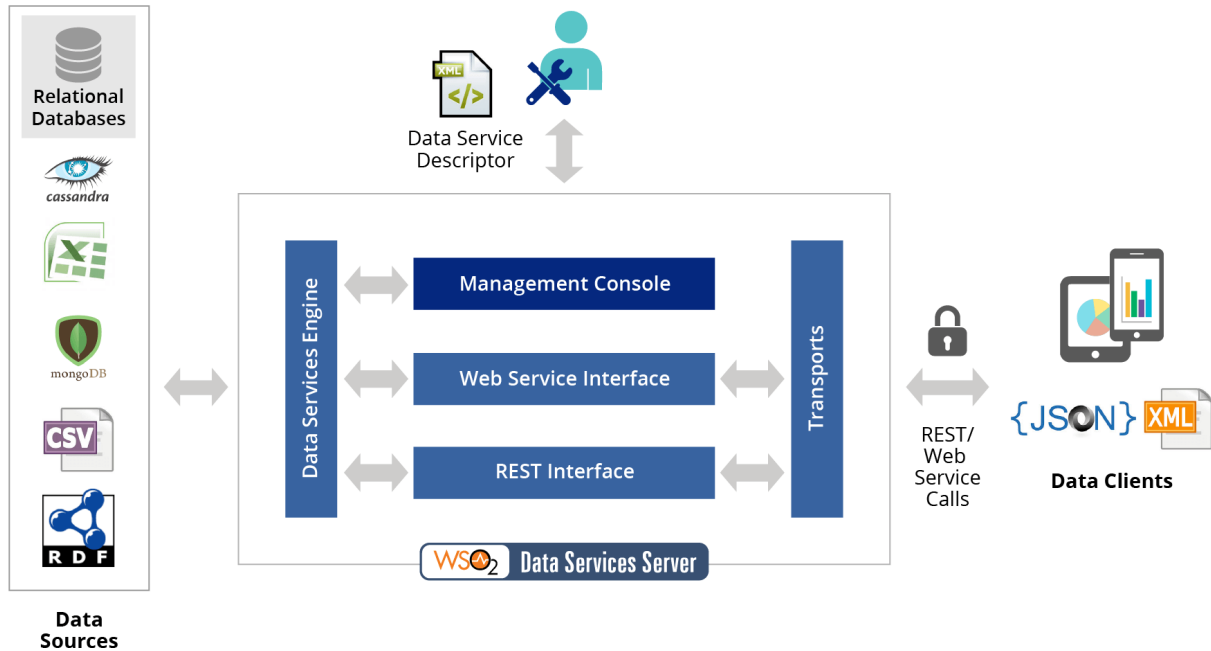
4.6 Data Interfaces

4.6.1 Data Services Server (WSO2)

Behind most applications are heterogeneous data stores. Most businesses require secure and managed data access across these federated data stores, data service transactions, data transformation, and validation. An organization's data exposed as a service, decoupled from the infrastructure where it is stored is called data services in service-oriented architecture (SOA).

Data services provide a convenient mechanism to configure a Web service interface for data in various datasources such as relational databases, CSV files, Microsoft Excel sheets, Google spreadsheets etc. These data services provide unprecedented data access and straightforward integration with business processes, mashups, gadgets, business intelligence, and mobile applications.

WSO2 Data Services Server augments service-oriented architecture development efforts by providing an easy-to-use platform for integrating data stores, creating composite data views, and hosting data services. It supports secure and managed data access across federated data stores, data service transactions, and data transformation and validation using a lightweight, developer friendly, agile development approach. It provides federation support, combining data from multiple sources in single response or resource and also supports nested queries across data sources. WSO2 Data Services Server is based on Java OSGi technology, which allows components to be dynamically installed, started, stopped, updated, and uninstalled while eliminating component version conflicts.



Data Services Server Installation

WSO2 Data Services Server is based on Java OSGi technology, which allows components to be dynamically installed, started, stopped, updated, and uninstalled while eliminating component version conflicts. Given below are the steps you need to follow to build the product from source code:

1. Git clone the product-level repository <https://github.com/coinnovationlab/product-dss.git>.
2. Use one of the following commands to build your repository and create complete release artifacts of WSO2 DSS.

Command	Description
<code>mvn clean install</code>	The binary and source distributions.
<code>mvn clean install -Dmaven.test.skip=true</code>	The binary and source distributions, without running any of the unit tests
<code>mvn clean install -Dmaven.test.skip=true -o</code>	The binary and source distributions, without running any of the unit tests, in offline mode. This can be done only if you have already built the source at least once.

Warning: Certificate validation errors

Product DSS makes use of an old version of the Eclipse Equinox framework that is signed with a certificate that is not trusted by recent versions of Java. If the Maven build fails with the error **One or more certificates rejected. Cannot proceed with installation**, that means you have to import the certificate to the Java certificate store.:

```
jar -xvf $MAVEN_HOME/repository/org/eclipse/tycho/tycho-p2-runtime/0.13.0/eclipse/
  ↳ plugins/org.eclipse.equinox.launcher_1.2.0.v20110725-1610.jar
cd META-INF
openssl pkcs7 -in ECLIPSEF.RSA -print_certs -inform DER -out EEF.cer`` This_
  ↳ certificate can be found also in the root of Product DSS repo.
keytool -import -file ECLIPSEF.cer -alias eclipse -keystore "$JAVA_HOME/jre/lib/
  ↳ security/cacerts"-storepass changeit -noprompt
```


3. You can find the new binary pack (ZIP file) in the <PRODUCT_HOME>/modules/distribution/target directory
4. Unzip this file in order to start the configuration of the environment
5. Download the JDBC driver for MySQL from <https://dev.mysql.com/downloads/connector/j> and copy it to <PRODUCT_HOME>/repository/components/lib directory.
6. Download the PostgreSQL driver from <https://jdbc.postgresql.org/download.html> and copy it to <PRODUCT_HOME>/repository/components/lib directory.
7. Open ./repository/conf/carbon.xml and set the value of Offset <Offset>0</Offset> (optional if you have several WSO2 products installed)
8. Changing the default database (using mysql instead of h2 default db for WSO2 products):

```
mysql -u root -p
create database dss
GRANT ALL ON dss.* TO dssadmin@localhost IDENTIFIED BY "dssadmin";
mysql -u dssadmin -p dss < '<PRODUCT_HOME>/dbscripts/mysql.sql'; OR mysql5.7.sql
```

9. Open the <PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml file and locate the <datasource> configuration element. You simply have to update the URL pointing to your MySQL database, the username, and password required to access the database and the MySQL driver details. See the example below::

```
<datasource>
  <name>WSO2_CARBON_DB</name>
  <description>The datasource used for registry and user manager</description>
  <jndiConfig>
    <name>jdbc/WSO2CarbonDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:mysql://localhost:3306/$dbname?autoReconnect=true</url>
      <username>$user</username>
      <password>$password</password>
      <driverClassName>com.mysql.jdbc.Driver</driverClassName>
      <maxActive>50</maxActive>
      <maxWait>60000</maxWait>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
      <defaultAutoCommit>true</defaultAutoCommit>
    </configuration>
  </definition>
</datasource>
```

10. Run the product in the following modes:

- **Default**

To start the server, you run the script wso2server.bat (on Windows) or wso2server.sh (on Linux/Solaris) from the bin folder:

```
On Windows:      <PRODUCT_HOME>\bin\wso2server.bat --run
On Linux/Solaris: sh <PRODUCT_HOME>/bin/wso2server.sh
```

- Custom (Including Security Integrations with AAC Component)

- In the file repository/conf/security/authenticators.xml put the following xml configuration

```
<!-- Example AAC OAUTH2 provider →
<Authenticator name="OAUTH2SSOAuthenticator" disabled="false">
  <Priority>3</Priority>
  <Config>
    <Parameter name="OauthProviderName">AAC</Parameter>
    <Parameter name="LoginPage">/carbon/admin/login.jsp</Parameter>
    <Parameter name="ServiceProviderID">carbonServer</Parameter>
    <Parameter name="IdentityProviderSSOServiceURL">http://
↪localhost:8080/aac</Parameter>
    <Parameter name="LandingPage">https://mydomain.com/dss_proxy_context_
↪path/carbon/oauth2-sso-accs/custom_login.jsp</Parameter>
    <Parameter name="RedirectURL">https://mydomain.com/dss_proxy_context_
↪path/oauth2_accs</Parameter>
    <Parameter name="UserProvisioningEnabled">true</Parameter>
    <Parameter name="TenantProvisioningEnabled">true</Parameter>
    <Parameter name="TenantDefault">testdomain.com</Parameter>
    <Parameter name="ClientID">YOUR_AAC_CLIENT_ID</Parameter>
    <Parameter name="ClientSecret">YOUR_AAC_CLIENT_SECRET</Parameter>
    <Parameter name="AuthorizationURL">http://localhost:8080/aac/oauth/
↪authorize</Parameter>
    <Parameter name="TokenURL">http://localhost:8080/aac/oauth/token</
↪Parameter>
    <Parameter name="CheckTokenEndpointUrl">http://localhost:8080/aac/
↪oauth/introspect</Parameter>
    <Parameter name="APIUserInfoURL">http://localhost:8080/aac/
↪basicprofile/me</Parameter>
    <Parameter name="APIRoleInfoURL">http://localhost:8080/aac/userroles/
↪me</Parameter>
    <Parameter name="GetRolesOfTokenURL">http://localhost:8080/aac/
↪userroles/token</Parameter>
    <Parameter name="ApiKeyCheckURL">http://localhost:8080/aac/
↪apikeycheck</Parameter>
    <Parameter name="MaxExpireSecToken">86400</Parameter>
    <Parameter name="ScopesListUserInfo">profile.basicprofile.me profile.
↪accountprofile.me user.roles.me user.roles.read</Parameter>
    <Parameter name="UserNameField">username</Parameter>
    <Parameter name="RoleContext">YOUR_ROLE_CONTEXT</Parameter>
    <Parameter name="SelectTenantURL">https://mydomain.com/dss_proxy_
↪context_path/carbon/oauth2-sso-accs/select_tenant.jsp</Parameter>
    <Parameter name="TenantSelectedURL">https://mydomain.com/dss_
↪proxy_context_path/forwardtenant</Parameter>
    <Parameter name="OauthProviderName">AAC</Parameter>
    <Parameter name="SecurityFilterClass">org.wso2.carbon.dataservices.
↪core.security.filter.ServicesSecurityFilter</Parameter>
  </Config>
</Authenticator>
```

You can find a description of each of these parameters [here](#):

- Edit the file repository/conf/tomcat/web.xml by adding the cookie-config tag::

```
<session-config>
  <session-timeout>30</session-timeout>
  <cookie-config>
    <name>JSESSIONID_DSS</name>
```

(continues on next page)

(continued from previous page)

```
</cookie-config>
</session-config>
```

10. You can populate the environment with the default samples inside the product by running:

```
cd <PRODUCT_HOME> samples; ant clean; ant
```

11. Regarding the OAUTH2 provider configuration you need to put the redirect link to: `https://mydomain.com/dss_proxy_context_path/oauth2_acs`

Description of Custom Authentication Config Parameters

In order to define the necessary parameters of the custom authentication of DSS using OAUTH2 protocol we need to add the proper values inside the file `<PRODUCT_HOME>/repository/conf/security/authenticators.xml`

```
<Parameter name="PROPERTY-NAME">PROPERTY-VALUE</Parameter>
```

PROPERTY-NAME	DESCRIPTION
Oauth-Provider-Name	The name of the provider that will appear on the login button
LoginPage	The default login page
IdentityProviderSSOServiceURL	The URL of the OAUTH2 provider
Landing-Page	The custom login page that contains the 'Login With' button.If empty it will directly redirect to the IdentityProviderSSOServiceURL
RedirectURL	The callback page where the service redirects the user-agent after an authorization code is granted
UserProvisioningEnabled	Boolean value to create or not the user on the fly if it doesn't exist
TenantDefault	The default tenant to whom attach the users if the tenant doesn't exist
TenantProvisioningEnabled	Boolean value to create or not the tenant on the fly if it doesn't exist. If the value is false then the user will belong to the TenantDefault
ProvisioningDefaultRole	The default roles to which will belong the users created on the fly.
ClientID	The Clientid value of the Client Application
ClientSecret	The ClientSecret value of the Client Application
AuthorizationURL	The API authorization endpoint
TokenURL	The API token endpoint: the application requests an access token from the API, by passing the authorization code along with the authentication details, including the client secret, to the API token endpoint.
CheckTokenEndpointUrl	The endpoint to get the information regarding the generated token.
GetRolesOfTokenURL	The endpoint to check the validity of the token provided in the odata/rest requests.
ApiKeyCheckURL	The endpoint to check the validity of the apiKey provided in the odata/rest requests.
MaxExpireSecToken	The period of expiration of the client_credentials token if this is an unlimited generated token. The token is being saved in Cookies and can be regenerated according to the expire_in parameter if this is >0 or if it is unlimited it will be regenerated after \$MaxExpireSecToken seconds.
APIUserInfoURL	The URL to call the API that gives information about the user profile
APIRoleInfoURL	The URL to call the API that gives information about the roles of the user
ScopesListUserInfo	Specifies the level of access that the application is requesting
UserNameField	The name of the field that contains the username from the APIUserInfoURL request
RoleContext	The context that the roles must have in order to be considered inside the DSS installation (ex: components/dss.super)
SelectTenantURL	The redirection page to select the desired tenant to whom belongs the user
TenantSelectedURL	The servlet name to select the desired tenant to whom belongs the user
Security-	The name of the class that implements the security filter of the odata/rest requests

Data Services Server Usage

OData requests

1.If the data service is inside the super tenant:

```
https://mydomain.com/dss_proxy_context_path/odata/ODataSampleService/default/$metadata
```

2.If the data service is inside a specific tenant:

```
https://mydomain.com/dss_proxy_context_path/odata/t/domain.com/ODataSampleService/  
↪default/$metadata
```

REST Requests

1.If the data service is inside the super tenant:

```
https://mydomain.com/dss_proxy_context_path/services/AccountDetailsService/Account?  
↪AccountId=1
```

2.If the data service is inside a specific tenant:

```
https://mydomain.com/dss_proxy_context_path/services/t/domain.com/  
↪AccountDetailsService/Account?AccountId=1
```

Securing Private Data Services

In this version of Data Services Server it is taken in consideration the security policy of accessing data services exposed over REST or OData format. For this purpose inside the configuration of a service it is added new parameter to specify if it is going to be a public or private service.

After being specified as private, the data service is going to be accessed in two ways:

1.Providing Authorization Token Header

```
https://mydomain.com/dss_proxy_context_path/odata/ODataSampleService/default/$metadata  
Authorization Header: Bearer $token$
```

2.Providing ApiKey Parameter:

```
https://mydomain.com/dss_proxy_context_path/odata/ODataSampleService/default/  
↪$metadata?apikey=$apikey$
```

REST API for dataservices CRUD

All WSO2 products expose their management functionality as admin services, which are SOAP services. In this fork of DSS we have exposed OSGi-level functionalities as REST APIs. Having said that, it is possible to create,update,delete and read the list of dataservices through this API.

1. REST API Usage

All the list of webservices is exposed in the following endpoint:

`https://mydomain.com/dss_proxy_context_path/rest/`

They are protected by OAuth2 token-based authentication method, therefore the token must be provided in the Header or it can be used the apikey as a query parameter.

The API can be accessed also by using Basic Authorization Header : `Basic base64Encode(username:password)`

1.1. Save dataservice

To save or update the configuration of one dataservice inside specific tenant the following call should be performed:

```
POST /{tenantDomain}/saveDataService      HTTPS/1.1
Accept: application/json
Host: https://mydomain.com/dss_proxy_context_path/rest/
a. Authorization: Bearer <token-value>
b. ApiKey query param https://wso2server.com/dss/rest/{tenantDomain}/saveDataService?
   ↪apikey=APIKEY_VALUE
```

1.2. Get dataservice configuration

To get the configuration of one dataservice inside specific tenant the following call should be performed:

```
GET /{tenantDomain}/getDataService?serviceid=DATASERVICENAME      HTTPS/1.1
Host: https://mydomain.com/dss_proxy_context_path/rest
a. Authorization: Bearer <token-value>
b. ApiKey query param https://wso2server.com/dss/rest/{tenantDomain}/getDataService?
   ↪serviceid=DATASERVICENAME&apikey=APIKEY_VALUE
```

1.3. Get list of dataservices

To filter the list of dataservices inside specific tenant is important to provide the term you want to search and also provide the page number starting from 0. The following call should be performed:

```
GET /{tenantDomain}/listDataService?search=TERM_TO_SEARCH&page=0      HTTPS/1.1
Host: https://mydomain.com/dss_proxy_context_path/rest
Accept: application/json
a. Authorization: Bearer <token-value>
b. ApiKey query param https://wso2server.com/dss/rest/{tenantDomain}/listDataService?
   ↪search=TERM_TO_SEARCH&page=0&apikey=APIKEY_VALUE
```

1.4. Delete dataservice

To delete one dataservice inside specific tenant the following call should be performed:

```
DELETE /{tenantDomain}/dataService/DATASERVICENAME      HTTPS/1.1
Host: https://mydomain.com/dss_proxy_context_path/rest
```

(continues on next page)

(continued from previous page)

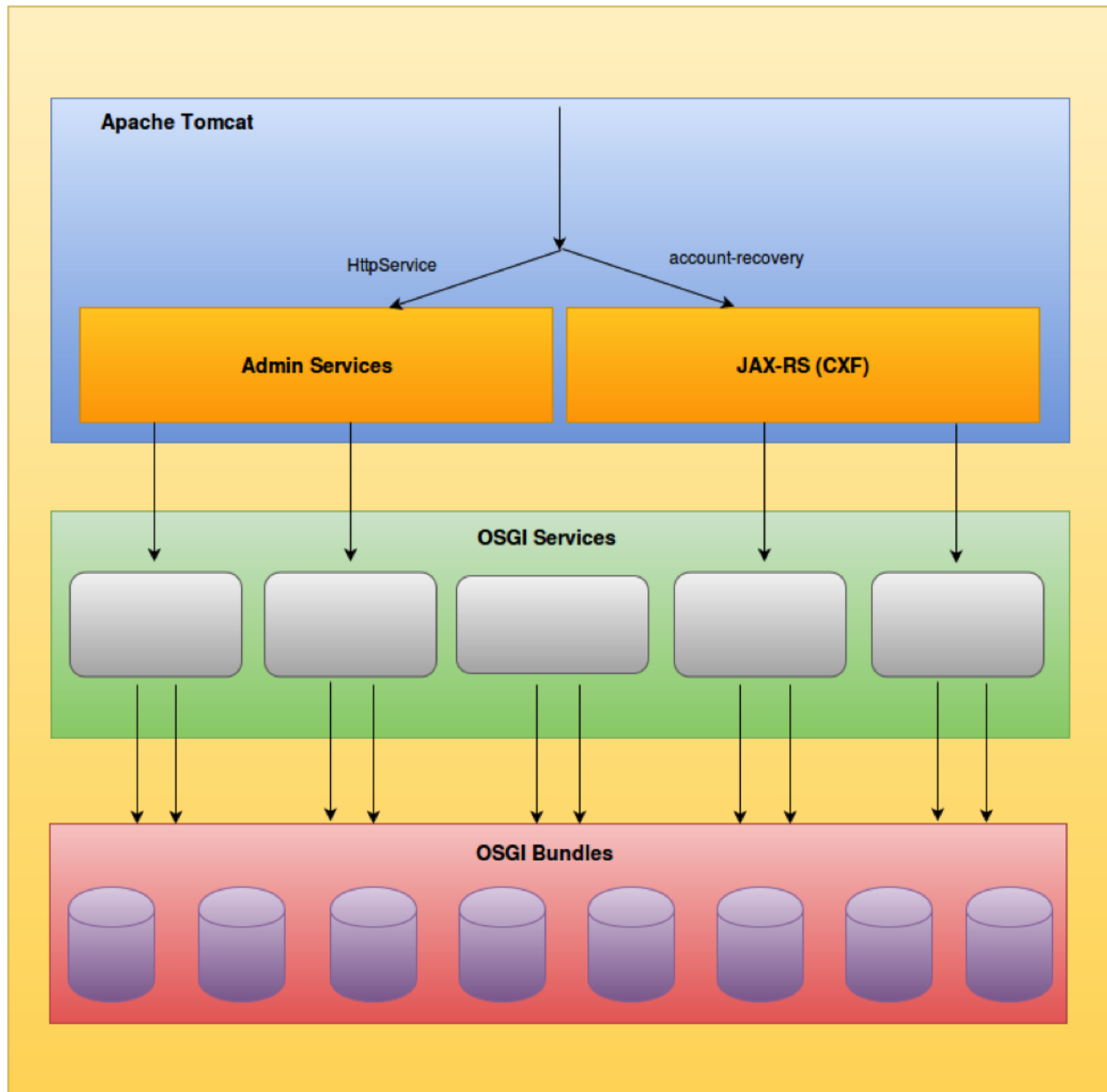
```

a. Authorization: Bearer <token-value>
b. ApiKey query param https://wso2server.com/dss/rest/{tenantDomain}/dataService/
   ↪ DATASERVICENAME&apikey=APIKEY_VALUE

```

Consider the fact that you can delete a dataservice that could be placed inside specific folder, in that case you can provide the exact value: /dataService/FOLDERNAME/DATASERVICENAME

Technical implementation



In order to expose a SOAP-based admin service as a REST API there are several tasks that need to be done:

- Create DataServiceManager packages to be integrated inside carbon.data OSGi bundles.

- Create Jax-RS WebApp in order to provide the REST API that is going to consume the OSGi service.
- Provide multitenancy data services creation through the API, extend admin services in order to switch the AxisConfiguration according to current PrivilegedCarbonContext.
- Integrate the OAUTH2 security filter in order to allow only authorized access to the REST API for dataservices CRUD.
- Integrate the Basic Auth security filter in order to allow only authorized access to the REST API for dataservices CRUD.

Deployment Guidelines in Production

1. Re-generate the keystore:

These commands allow you to generate a new Java Keytool keystore file, create a CSR, and import certificates. Execute the commands inside the folder <PRODUCT_HOME>/repository/resources/security.

- **Generate a Java keystore and key pair** `keytool -genkey-alias mydomain -keyalg RSA -keystore key-store.jks -keysize 2048 -validity 3650 -dname "CN=mydomain,OU=test,O=test,L=test,S=test,C=TS" -storepass mypassword -keypass mypassword`
- **Generate a certificate signing request (CSR) for an existing Java keystore** `keytool -certreq-alias mydomain -keystore keystore.jks -file mydomain.csr`
- **Import a root or intermediate CA certificate to an existing Java keystore** `keytool -import-trustcacerts -alias root -file Thawte.crt -keystore keystore.jks`
- **Import a signed primary certificate to an existing Java keystore** `keytool -import-trustcacerts -alias mydomain -file mydomain.crt -keystore keystore.jks`
- **Generate a keystore and self-signed certificate** `keytool -genkey-keyalg RSA -alias selfsigned -keystore keystore.jks -storepass password -validity 360 -keysize 2048`
- **Change a Java keystore password** `keytool -storepasswd-new new_storepass -keystore keystore.jks`
- **Check a particular keystore entry using an alias** `keytool -list-v -keystore keystore.jks -alias mydomain`

1.1 Example for generating a new keystore:

- `keytool -genkey -alias mydomain -keyalg RSA -keystore wso2carbonDev.jks -keysize 2048 -validity 3650 -dname "CN=mydomain.com,OU=test,O=test,L=test,S=test,C=TS" -storepass mypassword -keypass mypassword`
- `keytool -export -alias mydomain.com -file pubkey.cer -keystore wso2carbonDev.jks -storepass mypassword -noprompt`
- `keytool -import -trustcacerts -alias mydomain.com -file pubkey.cer -keystore client-truststoreDev.jks -storepass mypassword -noprompt`

1.2 Example for importing a new certificate:

- `keytool -importcert -file $somepath/someserver.com -keystore client-truststore.jks -alias "Some-Server"`

3. Update the following files with the new keyStore name and domain alias:

- <PRODUCT_HOME>/repository/conf/carbon.xml
- <PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml
- <PRODUCT_HOME>/repository/conf/security/secret-conf.properties
- <PRODUCT_HOME>/repository/conf/sec.policy

3. Setting Up Secure Vault Configuration (OPTIONAL)

Encrypt sensitive data in configuration files stored in file system

1. Locate <PRODUCT_HOME>/repository/conf/security/cipher-text.properties file. This file contains the alias names and the corresponding plain text password in square brackets.
2. Locate <PRODUCT_HOME>/bin/ciphertool.sh and run ciphertool.sh -Dconfigure
3. **Edit <PRODUCT_HOME>/repository/conf/security/secret-conf.properties:**
keystore.identity.alias=mydomain.com

This security configuration will require to restart DSS providing also the password of the keystore. It is important to mention the fact that when changing the admin password you have to set the new value in the file <PRODUCT_HOME>/repository/conf/user-mgt.xml and using ciphertool it will be part of the sensitive data that are going to be encrypted.

4. Adding a Custom Proxy Path

4.1. Install and configure a reverse proxy

Example - Apache virtual host configuration

```
<VirtualHost *:443>
ServerAdmin webmaster@localhost
ServerName mydomain.com
ServerAlias mydomain.com
ProxyRequests Off
<Directory "/">
    RewriteEngine On
    RewriteRule csrfPrevention.js$ https://innovation.deda.com/dss/carbon/admin/js/
    ↪csrfPrevention.js [P]
</Directory>

<Proxy>
    Order deny,allow
    Allow from all
</Proxy>
SSLEngine On
SSLProxyEngine On
SSLCertificateFile /etc/apache2/ssl/server.crt
SSLCertificateKeyFile /etc/apache2/ssl/server.key
SSLCACertificateFile /etc/apache2/ssl/ca.crt
ProxyPass /dss https://mydomain.com:9444/
ProxyPassReverse /dss https://mydomain.com:9444/
</VirtualHost>
```

4.2. Configure DSS with proxy context path

4.2.1 Edit <PRODUCT_HOME>/repository/conf/carbon.xml

- Change <HostName>mydomain.com</HostName>
- Change <MgtHostName>mydomain.com</MgtHostName>
- Change <KeyAlias>mydomain.com</KeyAlias>
- Comment the value of ServerURL and uncomment the following: <ServerURL>https://mydomain.com:\${carbon.management.port}\${carbon.context}/services/</ServerURL>
- Uncomment <ProxyContextPath>dss</ProxyContextPath> <MgtProxyContextPath>dss</MgtProxyContextPath>

4.2.2 Edit <PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml

Locate HTTPS connector and set

```
proxyPort="443" proxyName="mydomain.com/dss"
```

4.2.3 Edit <PRODUCT_HOME>/repository/conf/security/Owasp.CsrfGuard.Carbon.properties

```
org.owasp.csrfguard.UnprotectedMethods=GET,POST # this is to support oauth2 POST_
↪servlet (/forwardmultitenant)
```

4.2.4 Remember to update the file authenticators.xml according to the new proxy config:

Example:

```
<Parameter name="LandingPage">https://mydomain.com/dss/carbon/oauth2-sso-acs/custom_
↪login.jsp</Parameter>
```

4.6.2 GeoServer

GeoServer is an open source software server written in Java that allows users to share and edit geospatial data.

Designed for interoperability, it publishes data from any major spatial data source using open standards.

GeoServer is the reference implementation of the Open Geospatial Consortium (OGC) Web Feature Service (WFS) and Web Coverage Service (WCS) standards, as well as a high performance certified compliant Web Map Service (WMS). GeoServer forms a core component of the Geospatial Web.

In order to manage the configuration, Geoserver provides:

- an interactive interface (via the [web admin interface](#))
- a programmatic interface (through the [REST API](#))

Installation

1. GeoServer requires a Java 8 environment (JRE) to be installed on your system. You can download [JRE 8](#) from [Oracle](#).

Note: Note Java 9 is not currently supported.

2. Select the version of GeoServer that you wish to [download](#). If you're not sure, select [Stable](#).
3. Select Platform Independent Binary on the download page.
4. Download the archive and unpack to the directory where you would like the program to be located.

Note: Note A suggested location would be `/usr/share/geoserver`.

5. Add an environment variable to save the location of GeoServer by typing the following command:

```
echo "export GEOSERVER_HOME=/usr/share/geoserver" >> ~/.profile
. ~/.profile
```

6. Make yourself the owner of the geoserver folder. Type the following command in the terminal window, replacing `USER_NAME` with your own username :

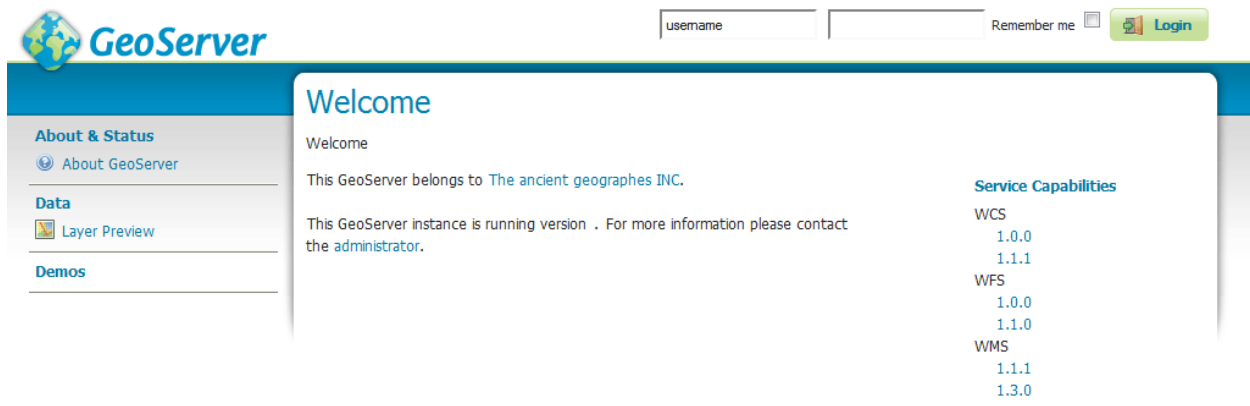
```
sudo chown -R USER_NAME /usr/share/geoserver/
```

7. Start GeoServer by changing into the directory `geoserver/bin` and executing the `startup.sh` script:

```
cd geoserver/bin
sh startup.sh
```

8. In a web browser, navigate to <http://localhost:8080/geoserver>.

If you see the GeoServer logo, then GeoServer is successfully installed.



Services

OGC services are the primary method of publishing data in GeoServer.

GeoServer serves data using standard protocols established by the [Open Geospatial Consortium](#):

These services are the primary way that GeoServer supplies geospatial information.

- **Web Map Service (WMS)** - GeoServer provides support for Open Geospatial Consortium (OGC) Web Map Service (WMS) versions 1.1.1 and 1.3.0. This is the most widely used standard for generating maps on the web, and it is the primary interface to request map products from GeoServer. Using WMS makes it possible for clients to overlay maps from several different sources in a seamless way. It supports requests for map images (and other formats) generated from geographical data.
- **Web Feature Service (WFS)** - GeoServer provides support for the Open Geospatial Consortium (OGC) Web Feature Service (WFS) specification, versions 1.0.0, 1.1.0, and 2.0.0. WFS defines a standard for exchanging vector data over the Internet. With a compliant WFS, clients can query both the data structure and the source data. It supports requests for geographical feature data (with vector geometry and attributes).
- **Web Coverage Service (WCS)** - is a standard created by the OGC that refers to the receiving of geospatial information as 'coverages' (rasters): digital geospatial information representing space-varying phenomena. One can think of it as Web Feature Service (WFS) for raster data. It gets the 'source code' of the map, but in this case it's not raw vectors but raw imagery. An important distinction must be made between WCS and Web Map Service (WMS). They are similar, and can return similar formats, but a WCS is able to return more information, including valuable metadata and more formats. It additionally allows more precise queries, potentially against multi-dimensional backend formats.
- **Web Processing Service (WPS)** - is an OGC service for the publishing of geospatial processes, algorithms, and calculations. The WPS service is available as an extension for GeoServer providing an execute operation for data processing and geospatial analysis. WPS is not a part of GeoServer by default, but is available as an extension. The main advantage of GeoServer WPS over a standalone WPS is direct integration with other GeoServer services and the data catalog. This means that it is possible to create processes based on data served in GeoServer, as opposed to sending the entire data source in the request. It is also possible for the results of a process to be stored as a new layer in the GeoServer catalog. In this way, WPS acts as a full remote geospatial analysis tool, capable of reading and writing data from and to GeoServer.
- **Catalog Services for the Web (CSW)** - GeoServer supports retrieving and displaying items from the GeoServer catalog using the CSW service.

Data Management

Data Settings

GeoServer permits to connect, load and publish data from a wide variety of sources.

Refer to the [official documentation](#) to learn how to use the GeoServer web interface to accomplish most common tasks, along with the different data formats served by GeoServer.

Note: Note There are many more data sources available through [Extensions](#) and [Community](#) modules. If you are looking for a specific data format and not finding it here, please check those sections.

Workspace

Analogous to a namespace, a workspace is a container which organizes other items.

In GeoServer, a workspace is often used to group similar layers together.











Layers may be referred to by their workspace name, colon, layer name (for example `topp:states`). Two different layers can have the same name as long as they belong to different workspaces (for example `sf:states` and `topp:states`).

Workspaces

Manage GeoServer workspaces

 Add new workspace

 Remove selected workspace(s)

     Results 1 to 7 (out of 7 items)			<input type="text" value="Search"/>
<input type="checkbox"/>	Workspace Name	Default	Isolated
<input type="checkbox"/>	cite	✓	
<input type="checkbox"/>	it.geosolutions		
<input type="checkbox"/>	nurc		
<input type="checkbox"/>	sde		✓
<input type="checkbox"/>	sf		
<input type="checkbox"/>	tiger		✓
<input type="checkbox"/>	topp		
     Results 1 to 7 (out of 7 items)			

Layers

In GeoServer, the term “layer” refers to a raster or vector dataset that represents a collection of geographic features. Vector layers are analogous to “featureTypes” and raster layers are analogous to “coverages”.

All layers have a source of data, known as a Store. The layer is associated with the Workspace in which the Store is defined.

In the Layers section of the web interface, you can view and edit existing layers, add (register) a new layer, or remove (unregister) a layer.

The Layers View page displays the list of layers, and the Store and Workspace in which each layer is contained.

Layers







































Manage the layers being published by GeoServer

 Add a new layer

 Remove selected layers

<<
<
1
>
>>
Results 1 to 19 (out of 19 items)

Search

<input type="checkbox"/>	Type	Title	Name	Store	Enabled	Native SRS
<input type="checkbox"/>		A sample ArcGrid file	nurc:Arc_Sample	arcGridSample		EPSG:4326
<input type="checkbox"/>		Pk50095	nurc:Pk50095	img_sample2		EPSG:32633
<input type="checkbox"/>		mosaic	nurc:mosaic	mosaic		EPSG:4326
<input type="checkbox"/>		North America sample imagery	nurc:Img_Sample	worldImageSample		EPSG:4326
<input type="checkbox"/>		Spearfish archeological sites	sf:archsites	sf		EPSG:26713
<input type="checkbox"/>		Spearfish bug locations	sf:bugsites	sf		EPSG:26713
<input type="checkbox"/>		Spearfish restricted areas	sf:restricted	sf		EPSG:26713
<input type="checkbox"/>		Spearfish roads	sf:roads	sf		EPSG:26713
<input type="checkbox"/>		Spearfish streams	sf:streams	sf		EPSG:26713
<input type="checkbox"/>		Spearfish elevation	sf:sfdem	sfdem		EPSG:26713
<input type="checkbox"/>		World rectangle	tiger:giant_polygon	nyc		EPSG:4326
<input type="checkbox"/>		Manhattan (NY) points of interest	tiger:poi	nyc		EPSG:4326
<input type="checkbox"/>		Manhattan (NY) landmarks	tiger:poly_landmarks	nyc		EPSG:4326
<input type="checkbox"/>		Manhattan (NY) roads	tiger:tiger_roads	nyc		EPSG:4326
<input type="checkbox"/>		USA Population	topp:states	states_shapefile		EPSG:4326
<input type="checkbox"/>		Tasmania cities	topp:tasmania_cities	taz_shapes		EPSG:4326
<input type="checkbox"/>		Tasmania roads	topp:tasmania_roads	taz_shapes		EPSG:4326
<input type="checkbox"/>		Tasmania state boundaries	topp:tasmania_state_boundaries	taz_shapes		EPSG:4326
<input type="checkbox"/>		Tasmania water bodies	topp:tasmania_water_bodies	taz_shapes		EPSG:4326

<<
<
1
>
>>
Results 1 to 19 (out of 19 items)

Stores

A store connects to a data source that contains raster or vector data.


A data source can be a file or group of files, a table in a database, a single raster file, or a directory (for example, a Vector Product Format library).

The store construct allows connection parameters to be defined once, rather than for each dataset in a source. As such, it is necessary to register a store before configuring datasets within it.




















Stores

Manage the stores providing data to GeoServer

 Add new Store

 Remove selected Stores

<< < 1 > >> Results 1 to 9 (out of 9 items)

 Type	Workspace	Store Name	Enabled?
	nurc	arcGridSample	
	nurc	img_sample2	
	nurc	mosaic	
	nurc	worldImageSample	
	sf	sfdem	
	sf	sf	
	tiger	nyc	
	topp	states_shapefile	
	topp	taz_shapes	

<< < 1 > >> Results 1 to 9 (out of 9 items)

While there are many potential formats for data sources, there are only four kinds of stores. For raster data, a store can be a file. For vector data, a store can be a file, database, or server.



Layer Groups

A layer group is a container in which layers and other layer groups can be organized in a hierarchical structure. A layer group can be referred to by a single name in WMS requests. This allows simpler requests, as one layer can be specified instead of multiple individual layers. A layer group also provides a consistent, fixed ordering of the layers it contains, and can specify alternate (non-default) styles for layers.

Layers

 Add Layer...

 Add Layer Group...

 Add Style Group... 

Drawing order	Type	Layer	Default Style	Style	Remove
1 	Layer	sf:sfdem		dem	
2  	Layer	sf:streams		simple_streams	
3  	Layer	sf:roads	<input type="checkbox"/>	line	
4  	Layer	sf:restricted		restricted	
5  	Layer	sf:archsites		point	
6 	Layer	sf:bugsites		capitals	

<< < 1 > >> Results 1 to 6 (out of 6 items)

DigitalHub integration

The extended [authentication and authorization module](#) allows GeoServer to authenticate against the OAuth2 Protocol .

In order to let the module work, it's mandatory to setup and configure both `oauth2` and `oauth2-aac` extension.

The module is a sample extension for [AAC](#) OAuth2 Provider.

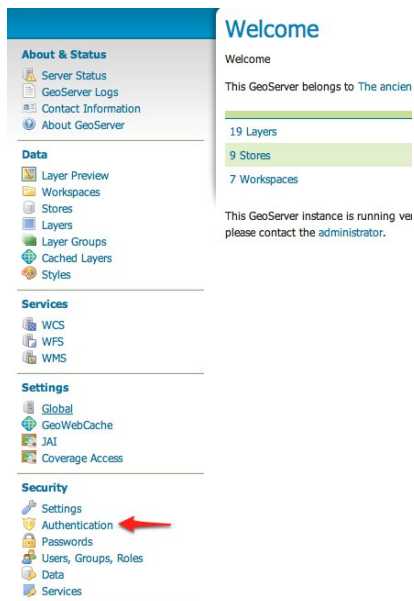
This module contains the implementation of the GeoServer security filter, the base classes for the OAuth2 Token services and the GeoServer GUI panel.

1. Configure the AAC authentication provider

The first thing to do is to configure the AAC OAuth2 Provider and obtain Client ID and Client Secret keys. Refer to the [AAC](#) documentation for related detailed configurations.

2. Configure the GeoServer OAuth2 filter

1. Start GeoServer and login to the web admin interface as the admin user (default: admin/geoserver).
2. Click the Authentication link located under the Security section of the navigation sidebar.



3. Scroll down to the Authentication Filters panel and click the Add new link.

Authentication

Authentication providers and settings



Logout settings


Redirect URL after logout (empty, absolute or relative to context root)

SSL settings

SSL Port (default is 443)

Authentication Filters

 Add new 


 Remove selected

- Click the OAuth2 link.

New Authentication Filter

Create and configure a new Authentication Filter

J2EE - Delegates to servlet container for authentication

OAuth2 - Authenticates by looking up for a valid OAuth2 access_token key sent as URL param 

Anonymous - Authenticates anonymously performing no actual authentication

Remember Me - Authenticates by recognizing authentication from a previous request

Form - Authenticates by processing username/password from a form submission

X.509 - Authenticates by extracting the common name (cn) of a X.509 certificate

HTTP Header - Authenticates by checking existence of an HTTP request header

Basic - Authenticates using HTTP basic authentication

Digest - Authenticates using HTTP digest authentication

Credentials From Headers - Authenticates by looking up for credentials sent in headers

Name

Role source

Scieglierne uno ▼

- Fill in the fields of the settings form as follows:

```
"Enable Redirect Authentication EntryPoint" = False
"Access Token URI" = https://AAC_URL/oauth2/token
"User Authorization URI" = http://AAC_URL/eauth/authorize
"Redirect URI" = http://localhost:8080/geoserver
"Check Token Endpoint URL" = http://AAC_URL/resources/token
"Logout URI" = http://AAC_URL/logout?target=http://localhost:8080/geoserver
"Scopes" = profile.basicprofile.me profile.accountprofile.me user.roles.me user.
<->roles.read
```

- Update the filter chains by adding the new OAuth2 filter.

Once everything has been configured you should be able to see the new oauth2 filter available among the Authentication Filters list.

Authentication

Authentication providers and settings

Logout settings

Redirect URL after logout (empty, absolute or relative to context root)

SSL settings

SSL Port (default is 443)

Authentication Filters

Add new

Remove selected

Search

<input type="checkbox"/>	Name	Type
<input type="checkbox"/>	anonymous	Anonymous authentication
<input type="checkbox"/>	basic	Basic HTTP authentication
<input type="checkbox"/>	form	Form authentication
<input type="checkbox"/>	oauth2	Authentication using a OAuth2
<input type="checkbox"/>	rememberme	Remember me authentication

<<

<

1

>

>>

Results 1 to 5 (out of 5 items)

Through this it will be always possible to modify / update the filter options, or create more of them.
The next step is to add the filter to the Filter Chains you want to protect with OAuth2 also

Filter Chains

Add service chain

Add HTML chain

Position	Name	Patterns	Check HTTP Method	HTTP methods	No Security	HTTP Session	SSL	Role Filter
<div><div>↓</div></div>	web	/web/**,/gwc/rest/web/**,/				✓		
<div><div>↑</div><div>↓</div></div>	webLogin	/j_spring_security_check,/j_spring_security_check/				✓		
<div><div>↑</div><div>↓</div></div>	webLogout	/j_spring_security_logout,/j_spring_security_logout/						
<div><div>↑</div><div>↓</div></div>	rest	/rest/**						
<div><div>↑</div><div>↓</div></div>	gwc	/gwc/rest/**						
<div><div>↑</div></div>	default	/**						

<<

<

1

>

>>

Results 1 to 6 (out of 6 items)

7.Select the OAuth2 Filter for each filter chain you want to protect with OAuth2.

If you need to protect all the GeoServer services and the GeoServer Admin GUI too with OAuth2, you need to add the oauth2 filter to all the following chains

- web
- rest
- gwc
- default

The order of the authentication filters depends basically on which method you would like GeoServer to try first.

Note: During the authentication process, the authentication filters of a Filter Chain are executed serially until one succeed (for more details please see the section [Authentication chain](#))

Available		Selected
	⇒	rememberme
	⇐	form
	↑	basic
	↓	oauth2
		anonymous

Note: Remember that the anonymous filter must be always the last one.

8. Save. It's now possible to test the authentication

4.6.3 Comprehensive Knowledge Archive Network (CKAN)

CKAN is an Open Source platform for making catalogs of open data, such as spreadsheets, geospatial GeoJSON data, OData, etc.

CKAN is inspired by the package management systems as in Linux OS. CKAN codebase is maintained by Open Knowledge Foundation.

It is widely used and adopted by national and local governments, research institutions, and other organizations who collect a lot of data. Once your data is published, users can use its faceted search features to browse and find the data they need, and preview it using maps, graphs and tables – whether they are developers, journalists, researchers, NGOs, citizens, or even your own staff.

CKAN supports multi-tenant data structure model, allowing different organizations to publish and manage their proprietary datasets.

The CKAN platform provides both Web-based GUI and the REST APIs to automate the procedures of data provisioning and publications.

The role of CKAN in the platform is to provide a catalogue of data for the community of the platform users. CKAN has become a reference model for the management and exposure of the Open Data due to

- possibility to customize both the functionality and the catalogue through the extensions and UI themes
- support for various different formats and plugins for the management and representation
- rich and extensible model of the metadata, which allows for the integration of the specific models and reference ontologies.
- large community support and adoption.

The integration of CKAN within the platform is based on

- the APIs provided by the CKAN distribution out of the box (e.g., APIs for the user management and access control, publication and management of data sets).

- the extensions for the 3rd party authentication plugins.

It is important to note, however, that the integration of CKAN within the platform is optional.

It is possible to rely on the Open Data portals already available in the specific contexts (e.g., existing portals provided by the PAs).

The official CKAN documentation is available here : <https://ckan.org/documentation-and-api/>

CKAN Integration

Different modes for the installation of CKAN platform is described in the official documentation (<https://docs.ckan.org/en/latest/maintaining/installing/index.html>).

The integration within the platform relies upon the organization and user model of the CKAN platform and the multi-tenancy model supported by the DigitalHub platform. To guarantee the alignment between the two models it is necessary to

- align the tenants (organizations) and the corresponding users
- align the authentication mechanism to support the appropriate authorization models and Single Sign-On.

More specifically, multitenancy in CKAN is based on the concept of Organization. The datasets published are associated to a specific organization and managed by that organization. The model of roles made available by AAC and managed by the Organization Manager allows for modelling these concepts.

To allow for authentication and authorization using AAC, CKAN provides extension plugins, in particular, OAuth2.0 authentication plugin. The existing plugins, however, do not allow for a sophisticated user and organization management, and have been extended within the DigitalHub. In particular, the new CKAN extension allows for

- authentication of the users based on AAC OAuth2.0 integration
- understanding the roles and organizations of the authenticated user
- registering dynamically the organization and the users within CKAN

CKAN DigitalHub OAuth2.0 Plugin Installation

The plugin is a Python module that is installed within the CKAN platform following the standard CKAN process.

To install the plugin, enter your virtualenv and install the package using pip as follows:

```
pip install git+https://github.com/scc-digitalhub/ckanext-oauth2.git
```

Add the following to your CKAN .ini (generally /etc/ckan/default/production.ini) file::

```
ckan.plugins = oauth2 <other-plugins>

## OAuth2 configuration

ckan.oauth2.register_url = https://YOUR_OAUTH_SERVICE/users/sign_up
ckan.oauth2.reset_url = https://YOUR_OAUTH_SERVICE/users/password/new
ckan.oauth2.edit_url = https://YOUR_OAUTH_SERVICE/settings
ckan.oauth2.authorization_endpoint = https://YOUR_OAUTH_SERVICE/authorize
ckan.oauth2.token_endpoint = https://YOUR_OAUTH_SERVICE/token
ckan.oauth2.profile_api_url = https://YOUR_OAUTH_SERVICE/user
```

(continues on next page)

(continued from previous page)

```

ckan.oauth2.client_id = YOUR_CLIENT_ID
ckan.oauth2.client_secret = YOUR_CLIENT_SECRET
ckan.oauth2.scope = profile other.scope
ckan.oauth2.rememberer_name = auth_tkt
ckan.oauth2.profile_api_user_field = JSON_FIELD_TO_FIND_THE_USER_IDENTIFIER
ckan.oauth2.profile_api_fullname_field = JSON_FIELD_TO_FIND_THE_USER_FULLNAME
ckan.oauth2.profile_api_mail_field = JSON_FIELD_TO_FIND_THE_USER_MAIL
ckan.oauth2.authorization_header = OAUTH2_HEADER
ckan.oauth2.profile_api_groupmembership_field = JSON_FIELD_TO_FIND_USER_ROLES_IN_
↳ORGANIZATIONS
ckan.oauth2.sysadmin_group_name = NAME_OF_THE_SYSADMIN_GROUP

```

The `profile_api_groupmembership_field` should be a list of strings or JSON objects with the `org` field representing the organization, and `role` representing the role. The first case is for compatibility with the original plugin. May contain the `sysadmin_group_name` value to map the user into the CKAN sysadmin role.

The `role` value should be one of `admin`, `editor`, or `member`. If the value is different, it will be cast to `member`.

You can also use environment variables to configure this plugin, the name of the environment variables are::

```

CKAN_OAUTH2_REGISTER_URL
CKAN_OAUTH2_RESET_URL
CKAN_OAUTH2_EDIT_URL
CKAN_OAUTH2_AUTHORIZATION_ENDPOINT
CKAN_OAUTH2_TOKEN_ENDPOINT
CKAN_OAUTH2_PROFILE_API_URL
CKAN_OAUTH2_CLIENT_ID
CKAN_OAUTH2_CLIENT_SECRET
CKAN_OAUTH2_SCOPE
CKAN_OAUTH2_REMEMBERER_NAME
CKAN_OAUTH2_PROFILE_API_USER_FIELD
CKAN_OAUTH2_PROFILE_API_FULLNAME_FIELD
CKAN_OAUTH2_PROFILE_API_MAIL_FIELD
CKAN_OAUTH2_AUTHORIZATION_HEADER
CKAN_OAUTH2_PROFILE_API_GROUPMEMBERSHIP_FIELD
CKAN_OAUTH2_SYSADMIN_GROUP_NAME

```

The callback URL that you should set on your OAuth 2.0 is: https://YOUR_CKAN_INSTANCE/oauth2/callback, replacing `YOUR_CKAN_INSTANCE` by the machine and port where your CKAN instance is running.

4.6.4 Goodtables

Goodtables is a continuous validation service for tabular data. Internally developed by the SCC Lab, it is modelled around the service provided by the *Open Knowledge Foundation* at <https://goodtables.io>.

The service provides the users with the ability to continuously validate the format and correctness of every CSV or TSV file uploaded on the S3 data lake (backed by Minio). By simply registering their *bucket* with the service, Goodtables will autonomously process each file uploaded to S3 and perform a series of validation checks, producing an accurate analysis report that users will be able to check to know the results.

The idea is to reduce the effort needed to validate the files, and thus improve the quality of datasets produced and shared within the digital hub by end users. At first, the service will be served as an *opt-in* options, connected only to the data lake, to help scientists and users in improving the quality of data.

The second step will be a direct connection with the data catalogue *CKAN*, where each tabular dataset inserted will have to undergo a validation check by *goodtables* before being published. This way, data will be continuously checked, reducing the likelihood of malformed or incomplete files published.

Goodtables-py

The actual validation framework used by the service is *goodtables-py* (<https://github.com/frictionlessdata/goodtables-py>), a python library and toolkit specifically designed to perform a series of validation checks on tabular data, both on the *structural* and the *content* level.

The main feature of the library (*from website*) are:

- Structural checks: Ensure that there are no empty rows, no blank headers, etc.
- Content checks: Ensure that the values have the correct types (“string”, “number”, “date”, etc.), that their format is valid (“string must be an e-mail”), and that they respect the constraints (“age must be a number greater than 18”).
- Support for multiple tabular formats: CSV, Excel files, LibreOffice, Data Package, etc.
- Parallelized validations for multi-table datasets
- Command line interface

The actual implementation (as of 2019) supports only structural checks, while the support for *content checks* depends on the adoption of a *content schema* which describes the required fields and their types. While *goodtables-py* offers an open schema format, available at <https://frictionlessdata.io/specs/>, it is yet to be decided how to introduce the support in the platform.

Continuous validation service

The service is designed as a backend system, developed in Java with the *spring boot* framework, which provides an API and an execution service able to perform various kind of checks when triggered.

While the inspiration is taken by *goodtables.io* (<https://goodtables.io/about>), we found out the implementation and the approach taken by that project too restricting and narrow focused for the adoption within the Digital Hub.

The idea of offering a self-managed portal for users, where they will be able to register their resources and have them checked, is much more useful if applied to a variety of formats and repositories than restricting to *github repositories* and *tabular data* as the original project. We envision a solution where users will be able to check both the structure and the content of resources like tabular data, geographic and spatial datasets (geojson, kml), plain json (with json schema), xml (plus xsd) etc.

To support this vision we built a custom system, where data processors and validators are decoupled from the management system and from the *triggers*.

The high level architecture is made of :

- a suite of *validators*, mostly built on open source projects with reputable history and recognition
- a suite of *triggers* able to receive an event and start the validation process on a resource
- a suite of *connectors* designed to collect resources (files) from repositories and also write results
- a repository of *registrations*, which represent the user request for checking a kind of file inside a specific repository
- an API for programmatic access
- a user interface for self-management

The backend layer connects all the various parts and also leverages AAC to obtain roles and identities, and the *Vault* to collect single-use credentials for resource access.

Any kind of trigger could start the validation process, which could involve one of the many validators available. The process will then proceed with the parsing of the event generated by the trigger, in an asynchronous way to free the trigger thread. The result will instruct the system in the lookup of the corresponding resource definition. If one (or

more) of the *registrations* matches the event received, the system will then validate the accessibility of the remote file, by requesting an appropriate set of credentials, and then fork a dedicated process to execute the validation script defined inside the registration. The executor will then process the validation, in an isolated thread, and eventually output the results, which will be stored inside the primary database, along with the registrations, and also written in an adequate format inside the remote storage, along with the source file, if requested.

The whole system is designed to be easily extended, by adding *connectors*, *validators* for new file types, and scaled by increasing the thread pool size for executors.

The triggers available as of end of 2019 are:

- *MQTT*: the service subscribes a set of topics and listens for messages describing the upload of new files.
- *HTTP*: the client uploads a file via POST request

The S3 (Minio) integration leverages the MQTT trigger.

The validators available are:

- Goodtables-py for CSV,TSV
- JSON (without schema)

Validation API

Goodtables exposes a complete API where users and developers will be able to:

- register new repositories and require checks
- review the validation results
- perform validation checks on files via upload

The details of the API are not finalized.

User interface

The service offers a *self-management* portal for end-users, which enables them into autonomously registering and reviewing validation tasks within any of the repositories available on Goodtables.

The *authentication* and *authorization* steps are performed via AAC.

4.7 Data Access

4.7.1 Ghostunnel

Ghostunnel is a SSL/TLS proxy with mutual authentication.

See <https://github.com/square/ghostunnel>

The HUB leverages the proxy to provide remote access to backend services like database servers (eg Postgres, MySQL). In order to ensure the security of data and processes, it is mandatory to limit access only to authorized users. Furthermore, it is advisable to expose only the *resource* (ie the database) to clients, and avoid the usage of network tunnels like VPN, SSH port-forward etc which expose the whole network to unsecured client devices.

The adopted solution leverages a couple of *ghostunnel* proxies, configured with mutual authentication:

- the **server-side** proxy is exposed on the internet, possesses a long-duration certificate and accepts connections only from clients which present a valid certificate. This proxy exposes an external TCP port which is binded to the internal database server.
- the **client-side** proxy verifies the server certificate, presents a *short-lived* certificate with a valid identity (gathered from OIDC - see *smallstep ca*) and after the successful handshake will expose a local port (on the client) which exposes the server-side database as local

This schema ensures that

- clients can access cloud databases with any tool, for any kind of operation, even for days-long backup/restores
- database servers are never exposed on the public network
- communication is encrypted via TLS, without dependencies on external softwares, libraries etc
- connection requests are validated against a real identity, provided by the identity provider (AAC)
- only real users with a valid *role* will be able to obtain a temporary certificate, independently from database credentials
- certificates are provisioned and invalidated programmately, without operator access
- certificates can have a short duration, from minutes to less than a day, an approach which could avoid the need for CRLs

1. Server proxy

The **server-side** proxy is run via docker or kubernetes with the following parameters:

- the address (host+port) of the internal database server
- the address (host+port) of the external public mapping
- the root certificate for the CA
- the server certificate and private key for mutual authentication
- a *policy* which checks for additional properties on valid certificate (eg groups, cn, san etc).

Example execution for exposing a local MySQL on 3306 on the local port 8306 with TLS.

```
$ docker run --rm squareup/ghostunnel server --listen localhost:8306 --target_
↪localhost:3306 --cert ghost.crt --key ghost.key --cacert root_ca.crt --allow-all
```

2. Client proxy

The **client-side** proxy is executed from the binary and requires the same parameters as the server side, but obviously in an inverted fashion.

- the address (host+port) of the cloud database server
- the address (host+port) of the local port mapping
- the root certificate for the CA
- the client certificate and private key for mutual authentication
- a *policy* which checks for additional properties on valid certificate (eg groups, cn, san etc).

Example execution for a cloud Postgres, with a combined keystore (cert+key) and the override of the server identity.


```
$ ~/ghostunnel-v1.4.1-linux-amd64-with-pkcs11 client --listen localhost:5432 --target_
↪13.80.78.44:8543 --cacert root_ca.crt --keystore test.pem --override-server-name_
↪ghostunnel
```

Do note the `--override-server-name` flag which is required if the server identity (CN) does not match the DNS. In this case the provided IP can not match the identity CN=ghostunnel, and the certificate has not SAN fields.

4.7.2 Smallstep CA & cli

Smallstep (<https://smallstep.com/>) provides tools for **identity-driver security**.

In the DigitalHUB context, the *Step Certificates (CA)* project is used as a PKI certification authority in order to implement a Zero Trust schema for cloud databases.

Via secure TLS proxies (see ghostunnel) cloud databases are safely accessible from end clients, thanks to **mutual authentication**: both servers and clients have to possess a valid and mutually accepted certificate in order to establish an encrypted communication channel.

Step Certificates and the associated *Step cli* are the building blocks for a system which can automatically craft and deliver *identity based* certificates to end users, via a connection to an *OIDC identity provider* such as AAC.

Users can perform the authentication process via browser, securely accessing the IdP in a private and safe way which doesn't expose their credentials to the command line tools.

After completing the *authorization request*, the IdP will provide *step* a JWT encoded OpenID connect **identity token**.

Step will then ask the **certification authority** to produce a short-lived X509 certificate to represent such identity. After collecting both the *public certificate* and the associated *private key*, users can utilize them to establish a connection with a proxy like *ghostunnel*.

1. Server configuration

Smallstep Certificates can be executed via Docker or kubernetes.

See the official doc at <https://github.com/smallstep/certificates/>

For running the CA in production, operators will have to provide a valid configuration with

- root and intermediate CA
- secrets for private keys
- JSON configuration for identity providers
- certificate configuration (duration, renewal..)

Everything can be set up by executing the docker image interactively with a local volume, see the upstream docs. To add an OIDC provisioner execute the *step cli* tool and provide:

- valid client-id and client-secret for Oauth
- the *.well-known* configuration URL

Example

```
$ step ca provisioner add aac --type oidc --configuration-endpoint "http://127.0.0.
↪1:9090/aac/.well-known/openid-configuration" --client-id XX --client-secret XX
```

After gathering the required files, the CA can be executed as a daemon and directly exposed on the web via port mapping. Avoid the usage of a proxy like Nginx, because this would result in the addition of another certificate and a double TLS layer. Do note that the public facing port should match the locally configured port in order to avoid errors with clients.

Important: Save the CA fingerprint because clients will need it to validate the CA certificate on bootstrap.

2. Client execution

Client side, the only software required is the *step cli* which can be downloaded as a binary release from github. The process for obtaining a valid certificate is:

- import the root CA certificate from the public facing URL via `step bootstrap`. This will require the public URL and the CA fingerprint.
- generate a certificate for a given identity via `step ca certificate`.

Example

```
$ step ca certificate test test.crt test.key
```

Do note that the given identity (“test” in the example) has to match the identity provided by the IdP via OIDC token. Furthermore, the server component will validate again the JWT token against the JWKS keys associated with the IdP, to avoid spoofing.

Example certificate `$ step certificate inspect test.pem`

```
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 168759030285531132221109068689398630028_
    ↪ (0x7ef5ce96530097408229d3d02e8d068c)
    Signature Algorithm: ECDSA-SHA256
    Issuer: CN=step-ca Intermediate CA
    Validity
      Not Before: Jun 28 13:56:35 2019 UTC
      Not After : Jun 29 13:56:35 2019 UTC
    Subject: CN=17
    Subject Public Key Info:
      Public Key Algorithm: ECDSA
      Public-Key: (256 bit)
      X:
        f6:a2:6d:d2:b3:98:76:ec:4e:b1:90:99:8c:96:29:
        ad:25:79:ef:70:d6:94:80:a7:94:e4:87:80:f8:fc:
        53:a3
      Y:
        47:54:26:cd:b9:f5:5f:d2:e5:39:d9:05:77:df:3f:
        d5:13:2d:a3:1f:e1:e5:b1:1e:08:f8:23:d8:e9:30:
        31:98
      Curve: P-256
    X509v3 extensions:
      X509v3 Key Usage: critical
        Digital Signature, Key Encipherment
      X509v3 Extended Key Usage:
        TLS Web Server Authentication, TLS Web Client Authentication
      X509v3 Subject Key Identifier:
```

(continues on next page)

(continued from previous page)

```
53:BD:55:0C:39:1C:68:25:1A:48:04:D4:CD:3C:91:92:5A:75:45:EC
X509v3 Authority Key Identifier:
  keyid:94:7C:89:2F:BC:14:0F:B4:FE:CC:23:2A:EF:44:8A:C0:4C:90:60:54
X509v3 Subject Alternative Name:
  email:admin
X509v3 Step Provisioner:
  Type: OIDC
  Name: aac
  CredentialID: da894353-1c0b-4fad-9d0f-cf83e89166ae
```

```
Signature Algorithm: ECDSA-SHA256
30:46:02:21:00:be:24:a8:d7:e0:8c:f3:fb:62:27:3c:2a:3e:
3b:08:9e:4e:86:89:d8:93:a2:37:c9:74:da:81:70:27:aa:3f:
fc:02:21:00:8f:a2:18:da:15:d9:92:a4:48:c1:0d:99:cc:ef:
f0:ef:7a:b5:6f:42:e0:7d:69:75:78:b0:55:9e:3d:c2:fa:91
```


IoT layer allows for the realization of the Internet of Things scenarios. The primary object of the layer is to provide the necessary infrastructure for managing the IoT applications and collect the data from different devices within these applications. The layer should be flexible in order to support variety of communication protocols, including REST, MQTT, LoRaWAN, NB-IoT, etc.

5.1 ThingsBoard

ThingsBoard represents a software platform that exposes the APIs to communicate with the devices and collect the data, to configure and manage the IoT applications (devices, events, commands, data routes) and perform basic IoT data monitoring activities. The collected data may be stored within the platform or routed to external storage (e.g., to the Data Lake via corresponding data stream connectors).

More specifically, the features of the ThingsBoard platform include

- Wide range of connectivity options, from REST API, to MQTT, CoAP, etc.
- Extensibility options for connectivity (e.g., LoRaWAN, NB-IoT, etc), rule engine, server integrations.
- Provision devices, assets and customers and define relations between them.
- Collect and visualize data from devices and assets.
- Analyze incoming telemetry and trigger alarms with complex event processing.
- Control your devices using remote procedure calls (RPC).
- Build work-flows based on device life-cycle event, REST API event, RPC request, etc
- Design dynamic and responsive dashboards and present device or asset telemetry and insights to your customers
- Enable use-case specific features using customizable rule chains.
- Push device data to other systems.

What is important for the DigitalHub platform and its architecture, ThingsBoard supports multitenancy out of the box. This allows for defining centrally the ThingsBoard tenants using the corresponding administration API, create and associate users, and bootstrap applications automatically.

By default, DigitalHub considers ThingsBoard Community Edition distributed under Apache License. However, the possibility to integrate the Enterprise Edition is possible as well.

Full documentation of the platform is available here: <https://thingsboard.io/docs/>.

The installation and integration instructions are available under the platform *installation section*.

Visualization, Dashboards, and Data Applications

Data visualization is used to provide an intuitive insight on the data phenomena that has been elaborated on top of platform data and services. The visualization is provided in a form of various dashboards and dashboard-based applications and may be used in different scenarios. Common to the visualization problem are the following requirements:

- Possibility to have a wide range of visualization types (charts, maps) and interactions (filters, details, drill down)
- Possibility to control the access to the dashboards, ranging from public to private based on the user roles and permissions. Related to this is the possibility to share the dashboard.
- Integration with the other platform components, both for data sources and for access control.

Visualization tools that can access Digital Hub may be divided in two categories:

- Tools that work directly with data sources, such as “online” storages (e.g., PostgreSQL) or data lake (e.g., file in MinIO). This includes tools such as SQLPad and Grafana.
- Tools that connect to the services and data exposed as “Data Interfaces”: DSS, GeoService, API REST, etc. For this purpose the platform adopts Cyclotron.

6.1 Grafana

Grafana is the open-source platform for monitoring and observability.

Grafana allows you to query, visualize, alert on and understand your metrics no matter where they are stored.

Create, explore, and share dashboards with your team and foster a data driven culture.

It is a tool for beautiful monitoring and metric analytics & dashboards for Graphite, InfluxDB, Prometheus and more

The official Grafana documentation is available at the following link: <https://grafana.com/docs/grafana/latest/>

6.1.1 Installation

It is possible to install Grafana in different modes on different systems.

Referring to the documentation <https://grafana.com/docs/grafana/latest/installation/> , Grafana is very easy to install and run using the official Docker container:

```
$docker run -d -p 3000:3000 grafana/grafana
```

as a result, the service will be accessible over the port 3000, <http://localhost:3000>

6.1.2 Features

- **Visualize:** Fast and flexible client side graphs with a multitude of options. Panel plugins for many different way to visualize metrics and logs.
- **Dynamic Dashboards:** Create dynamic & reusable dashboards with template variables that appear as drop-downs at the top of the dashboard.
- **Explore Metrics:** Explore your data through ad-hoc queries and dynamic drilldown. Split view and compare different time ranges, queries and data sources side by side.
- **Explore Logs:** Experience the magic of switching from metrics to logs with preserved label filters. Quickly search through all your logs or streaming them live.
- **Alerting:** Visually define alert rules for your most important metrics. Grafana will continuously evaluate and send notifications to systems like Slack, PagerDuty, VictorOps, OpsGenie.
- **Mixed Data Sources:** Mix different data sources in the same graph! You can specify a data source on a per-query basis. This works for even custom datasources.

6.1.3 Concepts

Dashboard

The dashboard is where it all comes together. A dashboard is a set of one or more panels organized and arranged into one or more rows.

The time period for the dashboard can be controlled by the Time range controls in the upper right of the dashboard.

Dashboards can use templating to make them more dynamic and interactive.

Dashboards can use annotations to display event data across panels. This can help correlate the time series data in the panel with other events.

Dashboards can be shared easily in a variety of ways.

Dashboards can be tagged, and the dashboard picker provides quick, searchable access to all dashboards in a particular organization.

Data source

Grafana can visualize, explore, and alert on data from many different databases and cloud services. Each database or service type is accessed from a data source.

Each data source has a specific query editor that is customized for the features and capabilities that the particular data source exposes. The query language and capabilities of each data source are || obviously very different. You can combine data from multiple data sources into a single dashboard, but each panel is connected to a specific data source that belongs to a particular organization.

Organization

Grafana supports multiple organizations in order to support a wide variety of deployment models, including using a single Grafana instance to provide service to multiple potentially untrusted organizations.

Each organization can have one or more data sources.

All dashboards are owned by a particular organization.

Note: Most metric databases do not provide per-user series authentication. This means that organization data sources and dashboards are available to all users in a particular organization.

Panel

The panel is the basic visualization building block in Grafana. Each panel has a Query Editor specific to the data source selected in the panel.

The query editor allows you to extract the perfect visualization to display on the panel.

There are a wide variety of styling and formatting options for each panel. Panels can be dragged and dropped and rearranged on the Dashboard. They can also be resized.

Panels can be made more dynamic with Dashboard Templating variable strings within the panel configuration. The template can include queries to your data source configured in the Query Editor.

Panels can be shared easily in a variety of ways.

Query editor

The query editor exposes capabilities of your data source and allows you to query the metrics that it contains.

Use the query editor to build one or more queries in your time series database. The panel instantly updates, allowing you to effectively explore your data in real time and build a perfect query for that particular panel.

You can use template variables in the query editor within the queries themselves.

This provides a powerful way to explore data dynamically based on the templating variables selected on the dashboard.

Grafana allows you to reference queries in the query editor by the row that they're on. If you add a second query to graph, you can reference the first query by typing in #A. This provides an easy and convenient way to build compound queries.

Row

A row is a logical divider within a dashboard. It is used to group panels together.

Rows are always 12 “units” wide. These units are automatically scaled dependent on the horizontal resolution of your browser.

User

A user is a named account in Grafana.

A user can belong to one or more organizations and can be assigned different levels of privileges through roles.

Grafana supports a wide variety of internal and external ways for users to authenticate themselves.

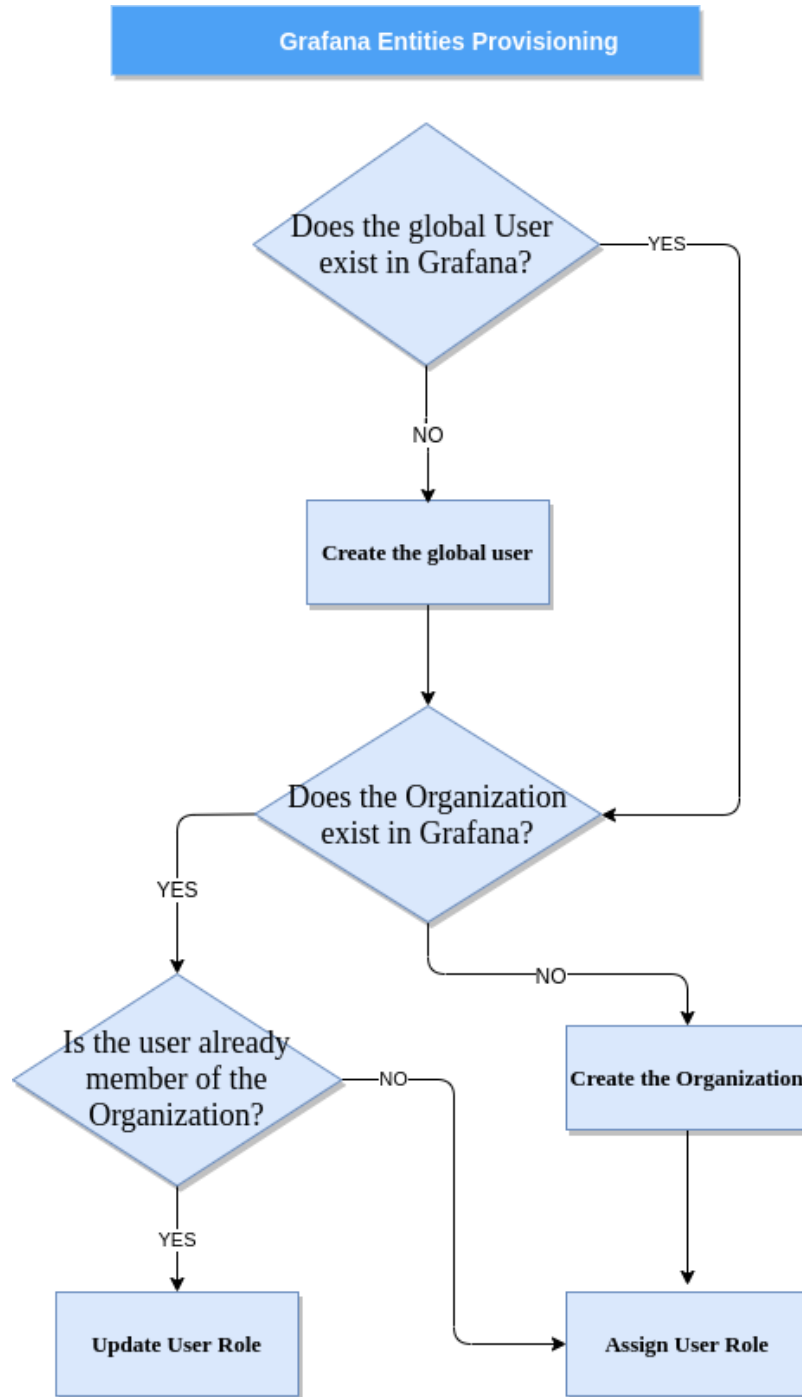
These include from its own integrated database, from an external SQL server, or from an external LDAP server.

6.1.4 Integration

Integration within the DigitalHub platform

Grafana has been integrated within the DigitalHub platform and aligned with its **multitenancy model**. Since Grafana supports multiple organizations in order to support a wide variety of deployment models, each organization can have one or more data sources. All dashboards are owned by a particular organization.

In order to allow the authentication and authorization using [AAC](#), within the platform has been integrated a **serverless Grafana Connector** for Organizations and Users provisioning. This serverless function creates the organizations and the users on the fly and assigns the proper roles and organizations to the authenticated user.



1. Create AAC OAuth keys

You can configure AAC OAUTH2 authentication services with Grafana using the generic oauth2 feature.

To enable the AAC OAuth2 you must register **Grafana Client Application** with AAC. AAC will generate a client ID and secret key for you to use.

The callback URL must match the full HTTP address that you use in your browser to access Grafana, **“https://<grafana domain>/login/generic_oauth”**.

In the section of **Roles & Claims** put the right endpoint of the deployed serverless function. You should use a custom

claim function for elaborating the list of roles for Grafana component, similar to the content inside the file `customClaims.js` in the repository of [Grafana Connector](#)

2. Enable AAC OAuth in Grafana

Finally set up the **generic oauth module** in the `conf/custom.ini` file like this:

```
##### Generic OAuth #####
[auth.generic_oauth]
name = AAC
enabled = true
allow_sign_up = true
client_id = CLIENT_ID_VALUE
client_secret = CLIENT_SECRET_VALUE
scopes = profile.basicprofile.me profile.accountprofile.me user.roles.me user.roles.
↳read email openid
email_attribute_name = email
email_attribute_path = email
#profile.basicprofile.me
role_attribute_path = user.roles.me
auth_url = http://YOUR_OAUTH_SERVICE/eaauth/authorize
token_url = http://YOUR_OAUTH_SERVICE/oauth/token
api_url = http://YOUR_OAUTH_SERVICE/userinfo
team_ids =
allowed_organizations =
tls_skip_verify_insecure = true
tls_client_cert =
tls_client_key =
tls_client_ca =
send_client_credentials_via_post = true
```

Restart the Grafana back-end. You should now see a AAC login button on the login page.
You can now login with your AAC accounts.

6.2 Cyclotron

Cyclotron is included in the platform as a fork of [Expedia's software](#). Head over to <https://www.cyclotron.io/> for an in-depth description of its functionalities. The fork introduces the following features:

- security based on Authentication and Authorization Control Module (AAC)
- new widgets: time slider, OpenLayers map, Google charts, Deck.gl, widget container
- new data sources: OData
- parameter-based interaction between dashboard components

6.2.1 AAC Integration

Authentication Methods

The forked version of Cyclotron supports OAuth2/OpenID as login method, via *implicit flow*. The module is usable with any *OAuth2/OIDC-compliant* identity provider, but some advanced functionalities such as role mapping and permission evaluators are available only when using AAC as IdP.

On Cyclotron **web app**, login can be performed:

- via LDAP by providing username and password
- via OAuth2/OpenID, i.e., being redirected to an identity provider (e.g. AAC) for authentication

On Cyclotron **API**, requests can be authenticated:

- via session key, which is the internal mechanism used by Cyclotron web app to authenticate its requests to the API
- via `Authentication HTTP header`
- via API key

The Authentication header must provide a valid token issued by the IdP. It is used by the web app if login is performed via OAuth2/OpenID and can be used to request API services directly:

```
GET /dashboards/mydashboard HTTP/1.1
Host: localhost:8077
Accept: application/json
Authorization: Bearer <my_token>
```

The API key is issued by IdP and can be passed as query parameter in the URL to request API services:

```
http://localhost:8077/dashboards/mydashboard?apikey=<my_apikey>
```

Authentication Configuration with OAuth2

Remember to set the same configuration (when needed) to both backend and frontend, without exposing private variables.

Frontend Configuration

Open `cyclotron-site/_public/js/conf/configService.js` and set the following properties under authentication:

```
authentication: {
  enable: true,
  authProvider: 'aac',
  authorizationURL: 'http://localhost:8080/aac/eauth/authorize',
  clientId: '<clientId>',
  callbackDomain: 'http://localhost:8088',
  scopes: 'openid profile user.roles.me'
}
```

Backend Configuration

Open `cyclotron-svc/config/config.js` and update the properties according to your needs:

```
enableAuth: true,
authProvider: 'aac',
oauth: {
  useJWT: <true|false>,
  clientId: '<clientId>',
  clientSecret: '<clientSecret>',
  jwksEndpoint: 'http://localhost:8080/aac/jwk',
  tokenIntrospectionEndpoint: 'http://localhost:8080/aac/oauth/introspect',
  userProfileEndpoint: 'http://localhost:8080/aac/userinfo',
  parentSpace: 'components/cyclotron',
  editorRoles: ['ROLE_PROVIDER', 'ROLE_EDITOR']
}
```

In order to use **JWTs** as bearer tokens, and locally verify them, please set `useJWT:true` and provide only one of these two configurations:

- populate `jwksEndpoint` with the JWKS uri to use public/private key verification via RSA
- set `clientSecret` and leave `jwksEndpoint` empty to use symmetric HMAC with `clientSecret` as key

Examples:

```
//JWT + public RSA key
oauth: {
  useJWT: true,
  clientId: '<clientId>',
  clientSecret: '',
  jwksEndpoint: 'http://localhost:8080/aac/jwk',
  tokenIntrospectionEndpoint: '',
  userProfileEndpoint: '',
},

//JWT + private HMAC key
oauth: {
  useJWT: true,
  clientId: '<clientId>',
  clientSecret: '<clientSecret>',
  jwksEndpoint: '',
  tokenIntrospectionEndpoint: '',
  userProfileEndpoint: '',
},
```

Do note that the default validation will check for a valid signature and for the correspondence between `clientId` and audience. If you want to also validate the *issuer* of JWT tokens set the corresponding property in config:

```
oauth: {
  issuer: <issuer>
}
```

Alternatively, you can use **opaque tokens** as bearer, and thus leverage *OAuth2 introspection* plus *OpenID userProfile*. This configuration requires `useJWT:false` and all the endpoints properly populated (except `jwksEndpoint`).

Example:

```
//opaque oauth
oauth: {
  useJWT: false,
  clientId: '<clientId>',
```

(continues on next page)

(continued from previous page)

```

clientSecret: '<clientSecret>',
jwksEndpoint: '',
tokenIntrospectionEndpoint: 'http://localhost:8080/aac/oauth/introspect',
userProfileEndpoint: 'http://localhost:8080/aac/userinfo',
},

```

Role mapping

By default, valid users are given the permission to create and manage their own dashboards, but can not access private dashboards without a proper role. A dashboard is private if the ability to **view** and/or **edit** it is restricted to specific users or *groups*.

Cyclotron supports two different roles:

- viewers
- editors

When using an external IdP (such as AAC) it is possible to map **roles** and **groups** by defining a context for the component space and a mapping for the **editor** role (i.e. a list of external roles that must be mapped as **editor** role in Cyclotron):

```

oauth: {
  parentSpace: 'components/cyclotron',
  editorRoles: ['ROLE_PROVIDER', 'ROLE_EDITOR']
},

```

As such, Cyclotron dynamically assigns roles to users at login, by deriving their **group memberships** and their role inside such groups from the IdP user profile. Any role that is not included in `editorRoles` will be mapped as **viewer**.

By setting `parentSpace` we define a prefix for roles obtained from the IdP, which is then used to derive the group from the following pattern:

```
<parentSpace>/<groupName>:<roleName>
```

For example, the upstream role `components/cyclotron/testgroup:ROLE_PROVIDER` with the given configuration can be translated to:

- (parentSpace=components/cyclotron)
- group=testgroup
- role=editor

because the upstream `ROLE_PROVIDER` role is recognized as an editor role.

The upstream role `components/cyclotron/testgroup:ROLE_USER` will be translated to:

- (parentSpace=components/cyclotron)
- group=testgroup
- role=viewer

Without a direct mapping to a given group, the system won't assign any role to the current user in such group. The user won't thus be able to access any dashboard restricted to that group.

Client Application Configuration on AAC

Log in to AAC as a provider user and click “New App” to create a client application. In the Settings tab:

- add Cyclotron website as redirect URL: `http://localhost:8088/`, `http://localhost:8088` (change the domain if it runs on a different host and port)
- check all the Grant Types and at least `internal` as identity provider (this must be approved on the Admin account under tab Admin -> IdP Approvals)

In the API Access tab:

- under OpenID Connect, check `openid`
- under User Profile Service, check `profile.basicprofile.me` to give access to user profiles to the client app
- under Role Service, check `user.roles.me` to give access to user roles

You can find `clientId` and `clientSecret` properties in the Overview tab. Add `clientId` to `cyclotron-site/_public/js/conf/configService.js` and `cyclotron-svc/config/config.js`. Add `clientSecret` as well if needed.

Now you can (re)start Cyclotron API and website with authentication enabled. Most services will now be protected and will require login and specific privileges.

NOTE: if you need to change the API port you can do it in the configuration file, but changing Cyclotron website port can only be done in `cyclotron-site/gulpfile.coffee`, inside the Gulp task named `webserver` (line 281): update `port` and `open` properties as needed.

Using Cyclotron API

AAC Roles and Cyclotron Permissions

NOTE: refer to AAC documentation on its Data Model if you are not familiar with the concepts of “role” and “space”.

If you use AAC as authentication provider, then Cyclotron *groups* correspond to AAC *spaces*. By default, owners of a space in AAC have the role `ROLE_PROVIDER`. In the AAC console, in the tab User Roles, owners (providers) of a space can add other users to it and assign them roles.

Suppose you configured `oauth.editorRoles=['ROLE_PROVIDER', 'ROLE_EDITOR']` and the following AAC roles exist:

- user A is provider of space T1 and user of space T2:

```
components/cyclotron/T1:ROLE_PROVIDER
components/cyclotron/T2:ROLE_USER
```

- user B is user of space T1:

```
components/cyclotron/T1:ROLE_USER
```

- user C is editor of space T1:

```
components/cyclotron/T1:ROLE_EDITOR
```

When these users log in to Cyclotron via AAC they are assigned the following property:

- user A: `memberOf: ['T1_viewers', 'T1_editors', 'T2_viewers']`
- user B: `memberOf: ['T1_viewers']`

- user C: memberOf: ['T1_viewers', 'T1_editors']

NOTE: editors can also view, i.e., users A and C being members of T1_editors are also members of T1_viewers; but viewers cannot edit, i.e., groups <group_name>_viewers cannot be assigned as editors of a dashboard.

Creating Private and Public Dashboards

When authentication is enabled, if a dashboard has no editors or viewers specified when it is created, by default both edit and view permissions are restricted to the dashboard creator only, who is allowed to change this behaviour later on. In order to allow anyone to view a dashboard, even anonymously, its view permissions can be given to the system role **Public**. Edit permissions can be given to **Public** as well, in which case the dashboard is editable by every user.

If you want to restrict access to a dashboard, you can give view/edit permissions either to specific users or to groups you are a member of. Some examples are provided in the next section.

Restricting Access to Dashboards in JSON

If you create a dashboard as a JSON document (either by POSTing it on the API or via JSON document editor on the website), this is its skeleton:

```
{
  "tags": [],
  "name": "foo",
  "dashboard": {
    "name": "foo",
    "pages": [],
    "sidebar": {
      "showDashboardSidebar": true
    }
  },
  "editors": [],
  "viewers": []
}
```

Resuming the example above, suppose user A wants to restrict access to its dashboard:

- dashboard editors list: can contain only group **T1_editors** or its members (e.g. user C)
- dashboard viewers list: can contain groups **T1_viewers** (*not* T1_editors as it is already a subset of T1_viewers, since editors are automatically also viewers) and **T2_viewers** or their members (e.g. users B and C)

Each editor or viewer specified in the lists must have three mandatory properties: `category` (either “User” or “Group”), `displayName` (used for readability purpose) and `dn` (unique name that identifies the user or group; corresponds to `distinguishedName` property in Cyclotron API User model).

Example 1. User A restricts edit permissions to themselves and gives view permissions to group T2:

```
"editors": [{
  "category": "User",
  "displayName": "John Doe",
  "dn": "A"
}],
"viewers": [{
  "category": "Group",
  "displayName": "T2",
  "dn": "T2_viewers"
}]
```

Example 2. User A restricts both edit and view permissions to group T2, i.e., every T2 member can view but only editor members can edit:

```
"editors": [{
  "category": "Group",
  "displayName": "T2",
  "dn": "T2_editors"
}],
"viewers": [{
  "category": "Group",
  "displayName": "T2",
  "dn": "T2_viewers"
}]
```

In short: use `<group_name>_editors` syntax for editors and `<group_name>_viewers` syntax for viewers.

NOTE: the system role **Public** is represented by the following properties:

```
{
  "category": "System",
  "displayName": "Public",
  "dn": "_public"
}
```

Testing

If you want to test the authorization mechanism (e.g. on Postman), you can use the following URLs, setting the `Authorization` header with an appropriate token:

- to create a dashboard, POST on `http://localhost:8077/dashboards`
- to update a dashboard, PUT on `http://localhost:8077/dashboards/{dashboard_name}`
- to retrieve a dashboard, GET on `http://localhost:8077/dashboards/{dashboard_name}`

6.2.2 Time Slider Widget

The Time Slider widget wraps `noUiSlider` JavaScript range slider.

Configuration

The following table illustrates the configuration properties available (required properties are marked in **bold**).

Property	JSON Key	Default	Description
Minimum date-time	min-Value		Starting value of the slider in the same format as <i>momentFormat</i>
Maximum date-time	max-Value		Ending value of the slider in the same format as <i>momentFormat</i>
Date-time Format	moment-Format	YYYY-MM-DD HH:mm	Any valid moment.js format
Step	step	1	Number of time units (e.g. days) between slider values
Direction	direction	ltr	Left-to-right or right-to-left
Orientation	orientation	horizontal	Horizontal or vertical slider
Player	player		Play/pause button for automatic sliding (more details below)
Time Unit	time-Unit	days	Each slider value corresponds to a minute, hour, day or month
Pips	pips		Scale/points shown near the slider (more details below)
Tooltips	tooltips	false	Show tooltips
Show Two Handles	twoHandles	false	Show two handles instead of one. In this case automatic sliding cannot be used. The handles will be placed at the first and last steps unless you specify a different position with Initial Handle Position.
Initial Handle Position	handle-Position	minValue [, maxValue]	Initial position of the handle, in the same format as <i>momentFormat</i> . If the slider has two handles, use two comma-separated values. Inline JavaScript can also be used, e.g. “ <code>\${Cyclotron.parameters.handlePos}</code> ”.
Specific Events	specificEvents		Array of parameters (defined in the Parameters section) and widget events that can trigger their change (more details below).

Player

Player property specifies an optional play/pause button that will cause the slider to slide automatically after each interval of the specified length. The button is displayed before (if horizontal) or on top of (if vertical) the slider.

Property	JSON Key	Default	Description
Show Player	showPlayer	false	Show player
Interval	interval	1	Seconds between each sliding
Start On Page Ready	startOnPageReady	false	Start the player automatically when the page is loaded

Pips

Pips are the points or scale displayed along the slider. Mode property configures the way pips are displayed:

- Range mode places a pip for every point specified in the range
- Steps mode places a pip for every step
- Positions mode places pips at percentage-based positions (percentages can be specified in Values option)
- Count mode generates a fixed number of pips (specified in Values option)
- Values mode is the same as Positions mode but with values instead of percentages

More details and examples are provided in the [noUiSlider documentation](#).

Property	JSON Key	Default	Description
Mode	mode	steps	Determines where to place pips
Values	values		Additional options for Positions, Count and Values mode. For Count mode provide an integer. For Positions and Values mode provide comma-separated integers.
Stepped	stepped	false	Match pips with slider values
Density	density	1	Number of pips in one percent of the slider range
Format	format	true	Show pip values in the same format as slider format

Specific Events

Events generated exclusively by the Slider Widget (see [Parameter-based Interaction](#)). They produce a value that will be stored in the given Parameter. Note that Parameters must be defined beforehand in the Parameters section of the dashboard. If the slider has two handles, you can use the *Section* property to specify which handle will generate the event.

Event	Value
dateTimeChange	Date or time selected with a handle

Examples

Basic slider with four pips matching slider values

```
{
  "maxValue": "2018-03-30",
  "minValue": "2017-09-10",
  "momentFormat": "YYYY-MM-DD",
  "pips": {
    "density": 2,
    "mode": "count",
    "stepped": true,
    "values": "4"
  },
  "widget": "slider"
}
```

Vertical slider with Player and Tooltips

```
{
  "direction": "rtl",
  "maxValue": "2018-03-30",
  "minValue": "2017-09-10",
  "momentFormat": "YYYY-MM-DD",
  "orientation": "vertical",
  "player": {
    "interval": 2,
    "showPlayer": true
  },
  "tooltips": true,
  "widget": "slider"
}
```

Slider with two handles and Parameter generation

```
{
  "maxValue": "2017-09-10 23:00",
  "minValue": "2017-09-10 00:00",
  "momentFormat": "YYYY-MM-DD HH:mm",
  "specificEvents": [{
    "event": "dateTimeChange",
    "paramName": "lowerValue",
    "section": "first"
  }, {
    "event": "dateTimeChange",
    "paramName": "upperValue",
    "section": "second"
  }],
  "timeUnit": "hours",
  "tooltips": true,
  "twoHandles": true,
  "widget": "slider"
}
```

6.2.3 OpenLayers Widget

The OpenLayers Map Widget wraps [OpenLayers version 4.6.5](#) to create dynamic maps and represent geographical data.

Configuration

The following table illustrates the configuration properties available (required properties are marked in **bold**).

Property	JSON Key	Default	Description
Center	center.x, center.y		X and Y coordinates to center the map on
Zoom	zoom		Zoom level (0 is zoomed out)
Data Source	dataSource		Name of the Data Source providing data for the overlays
Layers	layers	One OSM layer	Array of layers that will be added to the map (more details below).
Overlay Groups	overlay-Groups		Groups of overlays that will be displayed over the map, attached to a given position (more details below)
Controls	controls		Array of controls that will be added to the map (more details below)
DataSource Mapping	data-SourceMap-ping		Mapping between overlay properties and Data Source fields (more details below)
Specific Events	speci-ficEvents		Array of parameters (defined in the Parameters section) and widget events that can trigger their change (more details below)
Projection	projection	EPSG : 385	Projection

Layers

Layers property defines the layers that will be added to the OpenLayers map. Every layer must have a source, except VectorTile layers. Note that although OpenLayers provides many optional properties to configure layers, the current implementation of the OpenLayers Map Widget does not support all of them. If no layer type nor source are specified, a single Tile layer with an OSM (Open Street Map) source will be added to the map.

The following table lists which sources are valid for each type of layer:

Type	Source	Configuration Required	Notes
Image	ImageArcGISRest	No	
	ImageCanvas	Yes	
	ImageMapGuide	No	
	ImageStatic	Yes	
	ImageWMS	Yes	
	Raster	Yes	
Tile	BingMaps	Yes	
	CartoDB	Yes	
	OSM	No	
	Stamen	Yes	
	TileArcGISRest	No	
	TileDebug	Yes	
	TileJSON	Yes	
	TileUTFGrid	Yes	
	TileWMS	Yes	
	WMTS	Yes	
	XYZ	No	
	Zoomify	Yes	
Heatmap	Cluster	Yes	Requires <i>Weight</i> property
	Vector	No	Requires <i>Weight</i> property
Vector	Cluster	Yes	Can use Style function property
	Vector	No	Can use Style function property
VectorTile	VectorTile	No	

The configuration of sources is described in the documentation of [OpenLayers API](#). Note that the *Configuration* property can contain JavaScript code (see examples).

Overlay Groups

Overlays are elements that will be displayed over the map and attached to a given position. Each of them can have its own style, positioning and content. They can be interactive, that is, you can click on them and use Parameters to store the Name of the selected overlays (see Specific Events).

Every overlay must be part of a *group*, even if there is only one group on the map, and each group of overlays can have one overlay selected at a time. Let's say you want to place some red circles on cities and change them to blue when you click on them: if you want to have only one blue city at a time, you can have all the overlays in one group; if you want to have any number of blue cities at a time, you must add every city in a different group.

Note that overlays within the same group have their own template and style. The concept of group is only related to selection.

Each group has the following properties (required properties are marked in **bold**):

Property	JSON Key	Description
Name	name	Unique identifier for the group
Overlay Initially Selected	initiallySelected	Name of the overlay that will be assigned the <i>CSS Class On Selection</i> when the page is loaded. If not provided, all overlays will have the regular <i>CSS Class</i> until one is clicked. To use this field, you must provide names for the overlays.
Overlays	overlays	Array of overlays

Each overlay has the following properties (required properties are marked in **bold**):

Property	JSON Key	Default	Description
Name	name		Unique identifier for the overlay
CSS Class	cssClass		CSS class name (defined beforehand in the Styles section of the dashboard)
CSS Class On Selection	cssClass-Selected		Optional CSS class name (defined beforehand in the Styles section of the dashboard) to apply when the overlay is selected
Position	position		X and Y coordinates where the overlay will be attached
Positioning	positioning	top-left	Where the overlay is placed with respect to <i>Position</i>
Template	template		HTML template for overlay content

Note that *Name* property is not required, but it will be generated automatically if not provided, therefore if you want it to be stored in a Parameter you will not recognize which overlay the name belongs to if you do not provide it.

Instead of configuring the overlays in the Overlay Groups property, it is possible to provide them via a Data Source (see Data Source Mapping).

Controls

Controls are graphical elements on a fixed position over the map. They can provide user input or information.

Control	Description
Attribution	Informational button showing the attributions for the map layer sources
MousePosition	Shows the coordinates of the mouse cursor
OverviewMap	Shows an overview of the main map
ScaleLine	Shows a scale line with rough Y-axis distances
Zoom	Buttons for zoom-in and zoom-out
ZoomSlider	Slider for zooming in or out
ZoomToExtent	Button for zooming to an extent (default is zooming to zoom 0)

More information on controls can be found in the [OpenLayers API documentation](#), e.g. CSS classes to modify control styles.

Data Source Mapping

As an alternative to *Overlay Groups* property, a Data Source can also be used to provide overlays for the map. In this case, a mapping must be provided for the Widget to correctly read the dataset. The dataset structure differs a bit from the Data Sources for the other Widgets. It recalls the same structure of the *Overlay Groups* property, i.e., an array of groups, each one having a name, the optional name of an overlay pre-selected at loading and an array of overlays with their own properties (coordinates, CSS class, HTML template, etc.).

The Data Source must return a result like this:

```
[{
  "groupID": "g1",                                //unique group name (will be
↪assigned randomly if missing)
  "selectedOverlay": "ov1",                        //optional
  "overlays": [{                                   //list of overlays
    "css": "my-css-class",
    "cssSelected": "my-css-class-sel",
    "id": "ov1",
    "content": "<div>OV 1</div>",
    "coordinates": [11, 46],                       //alternatively you can have
↪separate X and Y fields
    "positioning": "top-left"
  },
  {
    "css": "my-css-class",
    "cssSelected": "my-css-class-sel",
    "id": "ov2",
    "content": "<div>OV 2</div>",
    "coordinates": [10, 45],
    "positioning": "bottom-right"
  }
]}]
```

You must use the *Data Source Mapping* property set to specify which dataset fields contain each piece of data necessary for configuring the overlays, that is, how to interpret the keys of the objects returned by the Data Source. The following table illustrates how to configure *Data Source Mapping* properties (required properties are marked in **bold**):

Property	JSON Key	Description
Identifier Field	identifier-Field	Name of the field containing a unique identifier for the group. If it is not specified and the datasource provides more than one group, each group will be assigned a random ID.
Overlay Initially Selected Field	initiallySelectedField	Name of the field containing the ID of the pre-selected overlay. If not provided, all overlays will have the regular CSS class until one is clicked.
Overlay List Field	overlayList-Field	Name of the field containing the list of overlays
CSS Class Field	cssClass-Field	Name of the field containing the CSS class for the overlay (must be defined beforehand in the Styles section)
CSS Class On Selection Field	cssClassOnSelectionField	Name of the field containing the CSS class for the overlay after its selection (must be defined beforehand in the Styles section)
Overlay Identifier Field	overlayId-Field	Name of the field containing a unique identifier for the overlay. If it is not specified, a random ID will be assigned.
Position Field	position-Field	Name of the field containing an array with the coordinates ([x_coord, y_coord]) for the overlay. Use <i>xField</i> and <i>yField</i> if coordinates are in two separate fields.
X Coordinate Field	xField	Name of the field containing X coordinate for the overlay
Y Coordinate Field	yField	Name of the field containing Y coordinate for the overlay
Positioning Field	positioning-Field	Name of the field containing the overlay positioning
Template Field	template-Field	Name of the field containing the HTML template for the overlay

A widget that uses the Data Source in the example above would need the following *Data Source Mapping*:

```

"dataSourceMapping": {
  "identifierField": "groupID",
  "initiallySelectedField": "selectedOverlay",
  "overlayListField": "overlays",
  "cssClassField": "css",
  "cssClassOnSelectionField": "cssSelected",
  "overlayIdField": "id",
  "positionField": "coordinates",
  "positioningField": "positioning",
  "templateField": "content"
}

```

Note that if the Data Source provides more than one group, each object in the array must have the same keys (e.g. in every object, the group ID can be found in a field named “groupID”).

Specific Events

Events generated exclusively by the OpenLayers Widget (see [Parameter-based Interaction](#)). They produce a value that will be stored in the given Parameter. Note that Parameters must be defined beforehand in the Parameters section of the dashboard.

The subproperty *Section* can be used to specify the name of the map portion that triggers the event. If the event is triggered by the map itself, you can leave this option empty.

Event	Value	Section
clickOnOverlay	Name of the overlay clicked	Name of an overlay group (e.g. “g1”)
clickOnWMSLayer	Feature Info	Name of a WMS layer (e.g. “topp:states”), obtained with <code>getGetFeatureInfoUrl</code>

Examples

Many examples of maps can be found on the [OpenLayers website](#), although not all of them are reproducible with Cyclotron.

Although *Configuration* property has the structure of a JSON object in the editor, it is a JavaScript string that is evaluated at dashboard loading, therefore it can contain JavaScript code (e.g. object instantiation).

OSM layer and WMS layer with zoom control

```
{
  "center": {
    "x": "11.123251",
    "y": "46.044685"
  },
  "controls": [{
    "control": "Zoom"
  }],
  "layers": [{
    "source": {
      "name": "OSM"
    },
    "type": "Tile"
  }, {
    "source": {
      "configuration": "{\r\n  \"url\": \"http://my.geoserver.com/wms\", \r\n  \"\r\n  \"params\": {\r\n    \"FORMAT\": \"image/png\", \r\n    \"VERSION\": \"1.1.1\", \r\n    \"STYLES\": \"\", \r\n    \"LAYERS\": \"topp:states\"}\r\n  }",
      "name": "ImageWMS"
    },
    "type": "Image"
  }],
  "widget": "openLayersMap",
  "zoom": 8
}
```

Default OSM layer and some overlays

This example uses air quality data stations located in Trentino to illustrate how to configure selectable overlays. Each overlay is positioned over a station and is represented as a circle whose color is assigned randomly among five options. As an overlay is clicked on, it becomes the currently selected one and the circle enlarges. The ID of the currently selected overlay is held by the Parameter “currentStation”. When you select an overlay, “currentStation” is updated and such update triggers the refresh of the Data Source, which in turn changes the color and template of the overlays.

The following dashboard components are required:

- Parameter:

```
{
  "name": "currentStation",
  "defaultValue": "7"
}
```

- CSS Style:

```
.station {
  opacity: .8;
  border-radius: 50%;
  width: 50px;
  height: 50px;
  line-height: 50px;
  text-align: center;
  font-size: 12px;
}

.station.sel {
  width: 70px !important;
  height: 70px !important;
  border: 2px solid #ddd;
  padding: 20px 20px;
  margin: auto;
}

.station.level1 {
  background-color: #7BBB6D;
}
.station.level2 {
  background-color: #BBCE55;
}
.station.level3 {
  background-color: #EDC12F;
}
.station.level4 {
  background-color: #F09227;
}
.station.level5 {
  background-color: #E64770;
}
```

- JSON Data Source

- Name: air-quality-stations
- URL: <https://am-test.smartcommunitylab.it/dss/services/ariadb/Stations>
- Post-Processor:

```
e = function(dataSet){
  res = [];
  var stations = dataSet.Entries.Entry; //array of objects with keys: id, name, ↵
  ↵X, Y

  _.each(stations, (station) => {
    var level = Math.floor(Math.random() * (6 - 1)) + 1;
    station.css = 'station level' + level;
    station.cssSelected = 'sel';
```

(continues on next page)

(continued from previous page)

```

        station.template = '<div>' + level + '</div>';
    });
    stationGroup = {};
    stationGroup.currentlySelected = '7';
    stationGroup.stationOverlays = stations;
    res.push(stationGroup);
    return res;
}

```

– Subscription to Parameters: `currentStation`

- OpenLayers Map Widget:

```

{
  "center": {
    "x": "11.123251",
    "y": "46.044685"
  },
  "dataSource": "air-quality-stations",
  "dataSourceMapping": {
    "cssClassField": "css",
    "cssClassOnSelectionField": "cssSelected",
    "initiallySelectedField": "currentlySelected",
    "overlayIdField": "id",
    "overlayListField": "stationOverlays",
    "templateField": "template",
    "xField": "X",
    "yField": "Y"
  },
  "specificEvents": [{
    "event": "clickOnOverlay",
    "paramName": "currentStation"
  }],
  "widget": "openLayersMap",
  "zoom": 11
}

```

Parametric ImageWMS layer

In this example, the OpenLayers Widget uses the value generated by a [Time Slider Widget](#) as `TIME` parameter for the WMS layer.

The following dashboard components are required:

- Parameter:

```

{
  "name": "sliderValue",
  "defaultValue": "2017-09-10"
}

```

- Slider Widget:

```

{
  "maxValue": "2018-03-30",
  "minValue": "2017-09-10",
}

```

(continues on next page)

(continued from previous page)

```

    "momentFormat": "YYYY-MM-DD",
    "pips": {
      "density": 2,
      "mode": "count",
      "stepped": true,
      "values": "4"
    },
    "specificEvents": [{
      "event": "dateTimeChange",
      "paramName": "sliderValue"
    }],
    "tooltips": true,
    "widget": "slider"
  }
}

```

- OpenLayers Map Widget:

```

{
  "center": {
    "x": "11.123251",
    "y": "46.044685"
  },
  "layers": [{
    "source": {
      "name": "OSM"
    },
    "type": "Tile"
  }, {
    "source": {
      "configuration": "{\n  \"url\": \"http://maps.dedagroup.it/geoserver/\n→mobility/wms\", \n  \"params\": {\n    \"FORMAT\": \"image/png\", \n    \n→\"VERSION\": \"1.1.1\", \n    \"LAYERS\": \"mobility:trento_archilday_total_week\"\n→\", \n    \"TIME\": \"#{sliderValue}T00:00:00.000Z\" \n  } \n\", \n    \"name\": \"ImageWMS\"
      },
      "type": "Image"
    },
    "parameterSubscription": ["sliderValue"],
    "widget": "openLayersMap",
    "zoom": 11
  }
}

```

The WMS layer configuration is repeated below for easier reading:

```

{
  "url": "http://maps.dedagroup.it/geoserver/mobility/wms",
  "params": {
    "FORMAT": "image/png",
    "VERSION": "1.1.1",
    "LAYERS": "mobility:trento_archilday_total_week",
    "TIME": "#{sliderValue}T00:00:00.000Z"
  }
}

```

Vector layer with GeoJSON features

This example reproduces [GeoJSON example](#).

In order to add GeoJSON features to the OpenLayers widget, you can either pass it an array of [Feature](#) or the URL of a GeoJSON file. In the first case, features can be added as a script in the Scripts section of the dashboard:

```
var geoJsonFeatures = {
  "type": "FeatureCollection",
  "crs": {
    "type": "name",
    "properties": {
      "name": "EPSG:3857"
    }
  },
  "features": [{
    "type": "Feature",
    "geometry": {
      "type": "Point",
      "coordinates": [0, 0]
    }
  }, {
    "type": "Feature",
    "geometry": {
      "type": "LineString",
      "coordinates": [[4e6, -2e6], [8e6, 2e6]]
    }
  }, {
    "type": "Feature",
    "geometry": {
      "type": "LineString",
      "coordinates": [[4e6, 2e6], [8e6, -2e6]]
    }
  }, {
    "type": "Feature",
    "geometry": {
      "type": "Polygon",
      "coordinates": [[[-5e6, -1e6], [-4e6, 1e6], [-3e6, -1e6]]]
    }
  }, {
    "type": "Feature",
    "geometry": {
      "type": "MultiLineString",
      "coordinates": [
        [[-1e6, -7.5e5], [-1e6, 7.5e5]],
        [[1e6, -7.5e5], [1e6, 7.5e5]],
        [[-7.5e5, -1e6], [7.5e5, -1e6]],
        [[-7.5e5, 1e6], [7.5e5, 1e6]]
      ]
    }
  }, {
    "type": "Feature",
    "geometry": {
      "type": "MultiPolygon",
      "coordinates": [
        [[[-5e6, 6e6], [-5e6, 8e6], [-3e6, 8e6], [-3e6, 6e6]]],
        [[[-2e6, 6e6], [-2e6, 8e6], [0, 8e6], [0, 6e6]]],
        [[[1e6, 6e6], [1e6, 8e6], [3e6, 8e6], [3e6, 6e6]]]
      ]
    }
  }
]
```

(continues on next page)

(continued from previous page)

```

    ]
  }, {
    "type": "Feature",
    "geometry": {
      "type": "GeometryCollection",
      "geometries": [{
        "type": "LineString",
        "coordinates": [[-5e6, -5e6], [0, -5e6]]
      }, {
        "type": "Point",
        "coordinates": [4e6, -5e6]
      }, {
        "type": "Polygon",
        "coordinates": [[[1e6, -6e6], [2e6, -4e6], [3e6, -6e6]]]
      }
    ]
  }
]}
}

```

Then you can add to the map widget a *vector* layer with the following source:

- Name: Vector
- Configuration:

```

{
  "features": (new ol.format.GeoJSON()).readFeatures(geoJsonFeatures).
  ↪concat(new ol.Feature(new ol.geom.Circle([5e6, 7e6], 1e6)))
}

```

- Notice the use of *geoJsonFeatures* variable that you created in the script
- Here *concat()* function is used to add a circle, which is a feature type not supported natively by GeoJSON

As an alternative to adding features as a Script, you can give a URL in the source configuration of the Vector layer. The Configuration property would be:

```

{
  "format": new ol.format.GeoJSON(),
  "url": "https://path/to/your/geojsonfile.json"
}

```

If you want to give the features a proper style, you can use the Style Function property to provide a *StyleFunction*:

```

e = function(feature) {
  var image = new ol.style.Circle({
    radius: 5,
    fill: null,
    stroke: new ol.style.Stroke({color: 'red', width: 1})
  });

  var styles = {
    'Point': new ol.style.Style({
      image: image
    }),
    'LineString': new ol.style.Style({
      stroke: new ol.style.Stroke({

```

(continues on next page)

(continued from previous page)

```

        color: 'green',
        width: 1
    })
  }},
  'MultiLineString': new ol.style.Style({
    stroke: new ol.style.Stroke({
      color: 'green',
      width: 1
    })
  }},
  'MultiPoint': new ol.style.Style({
    image: image
  }},
  'MultiPolygon': new ol.style.Style({
    stroke: new ol.style.Stroke({
      color: 'yellow',
      width: 1
    }),
    fill: new ol.style.Fill({
      color: 'rgba(255, 255, 0, 0.1)'
    })
  }},
  'Polygon': new ol.style.Style({
    stroke: new ol.style.Stroke({
      color: 'blue',
      lineDash: [4],
      width: 3
    }),
    fill: new ol.style.Fill({
      color: 'rgba(0, 0, 255, 0.1)'
    })
  }},
  'GeometryCollection': new ol.style.Style({
    stroke: new ol.style.Stroke({
      color: 'magenta',
      width: 2
    }),
    fill: new ol.style.Fill({
      color: 'magenta'
    }),
    image: new ol.style.Circle({
      radius: 10,
      fill: null,
      stroke: new ol.style.Stroke({
        color: 'magenta'
      })
    })
  }},
  'Circle': new ol.style.Style({
    stroke: new ol.style.Stroke({
      color: 'red',
      width: 2
    }),
    fill: new ol.style.Fill({
      color: 'rgba(255,0,0,0.2)'
    })
  })
})

```

(continues on next page)

(continued from previous page)

```
};

return styles[feature.getGeometry().getType()];
}
```

Notes

You might notice that some layers displaying images fetched via HTTP requests (e.g. ImageWMS sources) might refresh only at page reload or when the map is zoomed in or out. This happens because the browser is caching previous results and does not automatically perform a new request if the URL is unchanged. One way to avoid the use of cached content is to make every URL unique by adding a parameter that changes every time you need a refresh.

In this example:

```
"layers": [{
  "source": {
    "configuration": "{
      \"url\": \"http://my.geoserver.com/wms\",
      \"params\": {
        \"FORMAT\": \"image/png\",
        \"LAYERS\": \"my:layer\",
        \"requestVersion\": \"#{reqVersion}\"
      }
    }",
    "name": "ImageWMS"
  },
  "type": "Image"
}],
"parameterSubscription": ["reqVersion"]
```

the map widget subscribes to the Parameter *reqVersion* and its WMS layer adds it to the parameters that will be appended to the WMS server URL. The server will ignore it, but whenever *reqVersion* is updated, the browser will find that the URL has changed and request new content.

6.2.4 Google Charts Widget

The Google Charts Widget uses the wrapper directive for Google Chart Tools [angular-google-charts](#) to create several types of charts.

Configuration

This widget requires a Data Source to provide data for the chart. The following table illustrates the configuration properties available (required properties are marked in **bold**):

Property	JSON Key	Default	Description
Data Source	dataSource		Name of the Data Source providing table-like data for this Widget
Chart Type	chartType		Type of the chart
Options	options		JSON object with configuration options for the chart. Refer to the Google Charts API documentation of the selected chart type to see the list of available options.
Columns	columns		Array of columns of the table-like dataset. If not provided, the column labels will be inferred from the keys of the first dataset object. Note that this property must be used if some columns contain dates, times or datetimes or if the column role should differ from “data” (e.g. for annotations; see roles).
Formatters	formatters		Optional JavaScript functions that will be applied to all the values of the specified datasource columns

Options

Options property provides configuration options for the chart. Please refer to the [Google Charts API documentation](#) of the selected chart type to see the list of available options.

Columns

Columns property specifies how to read dataset columns, i.e., their names and types. While columns with strings, numbers and booleans can be easily identified, this property is useful for datasets containing dates or times, which are otherwise not recognized by the directive.

Property	JSON Key	Default	Description
Name	name		Indicates the name of the data source field to use for this column
Type	type	string	Type of the column values
Role	role	data	Optional role of the column

Note that columns with role “annotation” must immediately follow the series column that should be annotated, and columns with role “annotationText” must follow the annotated column.

Formatters

Formatters property specifies an array of JavaScript functions to format data in a given dataset column.

Property	JSON Key	Description
Column Name	columnName	Indicates the name of the datasource column
Formatter	formatter	The function must take a single value as argument and return the new value

In the following example, dates in the column named Day are given a different datetime format:

```
"formatters": [{
  "columnName": "Day",
```

(continues on next page)

(continued from previous page)

```

    "formatter": "p = function(value){ return moment(value, 'YYYY-MM-DD').format('dd_
↪DD'); }"
  }]
}

```

Examples

Basic column chart

JavaScript data source:

- Name: prices
- Processor:

```

e = function(promise) {
  var data = [{
    Product: 'Book',
    Price: 15.00
  }, {
    Product: 'Pencil',
    Price: 0.50
  }, {
    Product: 'Water bottle',
    Price: 1.50
  }, {
    Product: 'Bread',
    Price: 2.0
  }, {
    Product: 'Cake',
    Price: 20.00
  }]

  return promise.resolve(data);
}

```

Google Charts widget:

```

{
  "chartType": "ColumnChart",
  "dataSource": "prices",
  "options": "{\n  \"legend\": {\"position\": \"bottom\"},\n}",
  "widget": "gchart"
}

```

Column chart with formatter

```

{
  "chartType": "ColumnChart",
  "dataSource": "prices",
  "formatters": [{
    "columnName": "Price",
    "formatter": "p = function(value){\n  return value - (value*10)/100;\n}"
  }],
}

```

(continues on next page)

(continued from previous page)

```
"options": "{\n  \"legend\": {\"position\": \"bottom\"},\n  \"title\": \n→\"Product Prices with 10% Discount\", \n  \"colors\": [\"#ff00ee\"], \n}\",\n\"widget\": \"gchart\"\n}
```

Line chart with date column

JavaScript data source:

- Name: values
- Processor:

```
e = function(promise) {\n  var data = [{\n    Day: new Date(2018, 01, 01),\n    Value: 15.00\n  }, {\n    Day: new Date(2018, 01, 02),\n    Value: 0.50\n  }, {\n    Day: new Date(2018, 01, 03),\n    Value: 1.50\n  }, {\n    Day: new Date(2018, 01, 04),\n    Value: 2.0\n  }, {\n    Day: new Date(2018, 01, 05),\n    Value: 20.00\n  }];\n\n  return promise.resolve(data);\n}
```

Google Charts widget:

```
{\n  \"chartType\": \"LineChart\", \n  \"columns\": [{\n    \"name\": \"Day\", \n    \"type\": \"date\" \n  }], \n  \"dataSource\": \"values\", \n  \"options\": \"{\\n  \\\"legend\\\": {\\\"position\\\": \\\"bottom\\\"},\\n  \\\"title\\\": \n→\"Value by Day\", \\n  \\\"colors\\\": [\\\"#ff00ee\\\"], \\n}\", \n  \"widget\": \"gchart\" \n}
```

6.2.5 Deck.gl Widget

The Deck.gl Widget wraps [Deck.gl](#) to create a stack of visual layers, usually representing geographical data.

Configuration

The following table illustrates the configuration properties available.

Property	JSON Key	Description
Data Source	dataSource	Name of the Data Source providing data for one layer
View State	viewState	viewState property (see viewState). Zoom defaults to 0.
Additional Deck Properties	additionalDeck-Props	Additional supported Deck properties (e.g. <code>getTooltip</code>). See Deck
Layers	layers	Deck layers

Layers

Layers property defines a set of transparent overlays, each one representing a data collection that will be added to the deck. Layers can be interactive if the appropriate event (e.g. on click, on hover, on drag) handlers are configured (see [Adding interactivity](#)).

Deck.gl provides several layer types. The widget currently supports all the [Core Layers](#).

Property	JSON Key	Default	Description
Type	type		Layer type (required)
Configuration Properties	config-Properties		Object with layer properties. Despite having the structure of a JSON object in the editor, it is a JavaScript string that is evaluated at dashboard loading, therefore it can contain JavaScript (e.g. functions). See supported properties (note that they may vary depending on the layer type).
Use Data Source	use-Data-Source	false	Use the Data Source to provide data for this layer (alternative to data property). The data will be used for the first layer found with this property set to true.

Examples

Basic ScatterplotLayer

This example adapts the snippet used on Deck.gl [Getting Started](#) page. The Data Source provides data to create a circle whose color is changed at every Data Source refresh (5 seconds).

Create a new Data Source:

- Type: JavaScript
- Processor:

```
e = function(promise) {
  return [{
    "position": [-122.45, 37.8],
    "color": [Math.floor(Math.random() * Math.floor(256)), 0, 0], //random red
    "radius": 100000
  }];
}
```

- Auto-Refresh: 5

Create a new Deck.gl widget:

- Data Source: the one you just created
- Longitude: -122.45
- Latitude: 37.8
- Zoom: 4
- ScatterplotLayer:
 - Configuration Properties (functions can also be defined in the Scripts section of the dashboard and referenced here):

```
{
  "getFillColor": d => d.color,
  "getRadius": d => d.radius
}
```

- Use Data Source: true

Adding a tooltip

To add a tooltip (or any other Deck event handler) you can use *Additional Deck Properties* property (see documentation about the [picking info object](#)):

```
{
  "getTooltip": tooltipFunc
}
```

In the Scripts section, define a function `tooltipFunc` that returns a div with the mouse position:

```
tooltipFunc = function(info){
  return {
    html: ('<div>' + info.x + ', ' + info.y + '</div>'),
    style: {'background-color': 'yellow'}
  }
}
```

Multiple interactive layers

This example implements Deck.gl [basic example](#), which displays airports around the world. Whenever an airport is clicked, an alert will pop up with the airport name.

Create a new Deck.gl widget with the following *View State*:

```
{
  "bearing": 0,
  "latitude": "51.47",
  "longitude": "0.45",
  "pitch": 30,
  "zoom": 4
}
```

Add a ScatterplotLayer with the following *Configuration Properties*:

```
{
  "id": "base-map",
  "data": "https://d2ad6b4ur7yvpq.cloudfront.net/naturalearth-3.3.0/ne_50m_admin_0_
↪scale_rank.geojson",
  "stroked": true,
  "filled": true,
  "lineWidthMinPixels": 2,
  "opacity": 0.4,
  "getLineDashArray": [3, 3],
  "getLineColor": [60, 60, 60],
  "getFillColor": [200, 200, 200]
}
```

Add another ScatterplotLayer with the following *Configuration Properties*:

```
{
  "id": "airports",
  "data": "https://d2ad6b4ur7yvpq.cloudfront.net/naturalearth-3.3.0/ne_10m_airports.
↪geojson",
  "filled": true,
  "pointRadiusMinPixels": 2,
  "opacity": 1,
  "pointRadiusScale": 2000,
  "getRadius": f => (11 - f.properties.scalerank),
  "getFillColor": [200, 0, 80, 180],
  "pickable": true,
  "autoHighlight": true,
  "onClick": info => info.object && alert(`${info.object.properties.name} (${info.
↪object.properties.abbrev})`)
}
```

Add an ArcLayer with the following *Configuration Properties*:

```
{
  "id": "arcs",
  "data": "https://d2ad6b4ur7yvpq.cloudfront.net/naturalearth-3.3.0/ne_10m_airports.
↪geojson",
  "dataTransform": d => d.features.filter(f => f.properties.scalerank < 4),
  "getSourcePosition": f => [-0.4531566, 51.4709959],
  "getTargetPosition": f => f.geometry.coordinates,
  "getSourceColor": [0, 128, 200],
  "getTargetColor": [200, 0, 80],
  "getWidth": 1
}
```

6.2.6 Widget Container

The Widget Container widget allows to position other widgets within itself as if it were a page inside the page, i.e., with its own grid layout and theme. Its configuration is similar to that of a regular Page.

In order to add widgets to the container, its **Name property** must be specified. Each widget has an optional **Container property** which holds the name of the Container widget that should contain it. If this property is specified, the widget will be added to the container instead of the Page and its dimensions will be determined by the container grid layout.

Layout Configuration

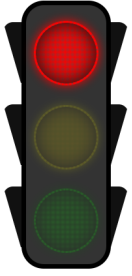
The following table illustrates the layout configuration properties available.

Property	JSON Key	Default	Description
Grid Columns	grid-Columns		Total number of horizontal grid squares available in the grid. The grid squares will be scaled to fit the container. If omitted, the number of columns is calculated dynamically.
Grid Rows	gridRows		Total number of vertical grid squares available in the grid. The grid squares will be scaled vertically to fit the container. If omitted, the grid squares will be literally square, i.e., the height and width will be the same. When omitted, the widgets may not fill the entire container, or they may scroll vertically. Use this property to make widgets scale vertically to fit the container.
Grid Container Height Adjustment	grid-HeightAdjustment	0	Adjustment (in pixels) to the height of the container when calculating the grid layout. If the value is positive, the container height will have the adjustment added to it (making each row taller). If it is negative, the container height will be reduced (making each row shorter).
Grid Container Width Adjustment	grid-WidthAdjustment	0	Adjustment (in pixels) to the width of the container when calculating the grid layout. If the value is positive, the container width will have the adjustment added to it (making each column wider). If it is negative, the container width will be reduced (making each column skinnier).
Gutter	gutter	10	Space (in pixels) between widgets positioned in the grid
Border Width	border-Width		Pixel width of the border around each widget. Can be set to 0 to remove the border. If omitted, the theme default will be used.
Margin	margin	10	Empty margin width (in pixels) around the outer edge of the container. Can be set to 0 to remove the margin.
Scrolling Enabled	scrolling	true	Enables vertical scrolling of the container to display content longer than the current size

Example


The following image shows a sample dashboard with a few widgets, both displayed on the page and inside a container. The dashboard JSON definition can be found below.

42



Container
↗

42



Name	Description	Link	Type
foo	foo	/ex1	Widget
bar	bar	/ex2	Widget

Name	Description	Link	Type
foo	foo	/ex1	Widget
bar	bar	/ex2	Widget

```
{
  "dataSources": [{
    "name": "some-links",
    "preload": true,
    "processor": "p = function () {\n    return [{\n      description: 'foo',\n      link: '/ex1',\n      name: 'foo',\n      type: 'Widget'\n    }, {\n      description: 'bar',\n      link: '/ex2',\n      name: 'bar',\n      type: 'Widget'\n    }];\n}",
    "type": "javascript"
  }],
  "name": "example-widget-containers",
  "pages": [{
    "frequency": 1,
    "layout": {
      "gridColumns": 4,
      "gridRows": 4
    },
    "widgets": [{
      "gridHeight": 1,
      "gridWidth": 4,
      "numbers": [{
        "number": "42"
      }],
      "orientation": "horizontal",
      "widget": "number"
    }, {
      "gridHeight": 2,
      "gridWidth": 1,
      "rules": {
        "red": "${true}"
      },
      "tooltip": "Time To Stop",
      "widget": "stoplight"
    }, {
      "gridHeight": 3,
      "gridWidth": 3,
```

(continues on next page)

(continued from previous page)

```

        "layout": {
            "gridColumns": 2,
            "gridRows": 4
        },
        "name": "Cont",
        "title": "Container",
        "widget": "widgetContainer"
    }, {
        "container": "Cont",
        "gridHeight": 1,
        "gridWidth": 1,
        "name": "contained-number",
        "numbers": [{
            "number": "42"
        }],
        "orientation": "horizontal",
        "widget": "number"
    }, {
        "container": "Cont",
        "gridHeight": 1,
        "gridWidth": 1,
        "name": "contained-stoplight",
        "rules": {
            "red": "${true}"
        },
        "tooltip": "Time To Stop",
        "widget": "stoplight"
    }, {
        "columns": [{
            "label": "Name",
            "link": "#{link}",
            "name": "name"
        }, {
            "label": "Description",
            "name": "description"
        }, {
            "name": "link"
        }, {
            "name": "type"
        }],
        "container": "Cont",
        "dataSource": "some-links",
        "gridHeight": 2,
        "gridWidth": 2,
        "name": "contained-table",
        "widget": "table"
    }, {
        "columns": [{
            "label": "Name",
            "link": "#{link}",
            "name": "name"
        }, {
            "label": "Description",
            "name": "description"
        }, {
            "name": "link"
        }, {

```

(continues on next page)

(continued from previous page)

```

        "name": "type"
      }},
      "dataSource": "some-links",
      "gridHeight": 1,
      "gridWidth": 1,
      "widget": "table"
    }]
  },
  "parameters": [],
  "sidebar": {
    "showDashboardSidebar": false
  },
  "theme": "light"
}

```

6.2.7 OData Data Source

The Odata Data Source loads data via HTTP GET requests to Odata services. It is a more specialized alternative to the JSON Data Source.

Please note that it may be necessary to set the optional *Proxy* property in the Data Source in order to access web sites or services in other domains. This property lists other Cyclotron servers which may have the required connectivity. By default, this Data Source uses the current Cyclotron environment to proxy the request.

Configuration

The following table illustrates the configuration properties available:

Property	JSON Key	Default	Description
URL	url		OData Service URL (required)
Response Adapter	responseAdapter	raw	How the response will be converted to Cyclotron's data format
Query Parameters	queryParams		Optional query parameters. If there are already query parameters in the URL, these will be appended. The keys and values are both URL-encoded.
Auto-Refresh	refresh		Number of seconds before Data Source reload
Pre-Processor	preProcessor		JavaScript function that can inspect and modify the Data Source properties before its execution. It takes the Data Source object as an argument returns it modified or a new one.
Post-Processor	postProcessor		JavaScript function that can inspect and modify the result before returning it
Proxy Server	proxy		Alternative proxy server to route the requests through
Options	options		Optional request parameters that are passed to the library making the request
AWS Credentials	awsCredentials		AWS IAM signing credentials for making authenticated requests. If set, the request will be signed before it is sent.
OAuth2.0 Client Credentials	oauth2ClientCredentials		OAuth2.0 client credentials for making authenticated requests. If set, the request will be added an Authorization header before it is sent.

URL

The *URL* property specifies the OData service URI for the request. Query parameters can either be part of the URL or added via the Query Parameters property, however the alternative is not valid for system query options (i.e. those prefixed with the dollar sign \$), which must be written in the URL.

Response Adapter

The *responseAdapter* property is used to correctly read the result returned by OData. By default its value is *raw*, which means that no adaptation occurs and the result is returned as it is, for further elaboration in the *post-processor* or in case the response contains a single raw value (e.g. produced by \$count or \$value query options).

If an Entity Set is requested to OData, then the *entity set* response adapter may be used to retrieve the array of objects returned by OData (i.e. the “value” of the JSON response in OData V4).

If a Single Entity is requested, then the *single entity* adapter may be used to remove OData metadata from the result and retrieve the single object.

If a Primitive Property is requested, then the *primitive property* adapter may be used to retrieve from the result the raw value of the property.

The data that is sent to the *Post-Processor* will depend on the setting of the response adapter, in that the function receives the output of whichever *responseAdapter* has been selected. In order to manually inspect the entire response, set it to *raw*.

6.2.8 Parameter-based Interaction

Cyclotron provides support for dashboard parameters, i.e., values that can be used in the configuration of dashboard components such as widgets and data sources to filter/drive the visualization of the information (e.g. a dynamic URL), as well as to hold constant information shared by different components (e.g. connection information).

Parameters can be defined in the dedicated editor section, where an input form (e.g. textbox, checkbox, dropdown menu, etc.) and a change event handler (i.e. `function(parameterName, newValue, oldValue){}`, called when the parameter is updated to perform some operation, like re-executing a data source) can optionally be configured.

In the original implementation, parameters can only be set *explicitly*, either:

- via dashboard URL as a query parameter: `http://cyclotron/my-dashboard?numberOfHours=8`
- via their input form in the Header widget, which can additionally display an “Update” button with an associated event handler
- within JavaScript code, via the global `Cyclotron` object: `Cyclotron.parameters.numberOfHours = 24;`

In order to overcome this limit and allow for more interaction between dashboard components, the platform fork includes a mechanism of automatic update and consequent reload of the affected components.

Parameter Generation

Some widgets (e.g. time slider, OpenLayers map) generate events that can trigger the update of a parameter, that is, some value is produced and automatically assigned to the parameter. For example, the event generated by the time slider is the selection of a date/time by sliding and the value produced is the selected date/time.

When the event is fired, the change (in the form of the message `parameter:<param_name>:update`) is broadcasted to the whole dashboard by a built-in function, so that any components (data sources or widgets) using that parameter react to the update.

Subscription to Parameter Updates

On the other side, the configuration of widgets and data sources can be made *parametric* via the use of wildcards, which will be replaced with the current parameter value at loading. The wildcard syntax is `{parameterName}` and the widget/data source must *subscribe* to that parameter, so that whenever it is updated they are updated consequently. For example, a parameter defined in the editor as

```
{
  "name": "currentStation",
  "defaultValue": "7"
}
```

can be used in the configuration of a data source as follows:

```
{
  "name": "stationData",
  "type": "json",
  "parameterSubscription": ["currentStation"],
  "url": "https://localhost:8080/dss/services/db/{currentStation}"
}
```

If a widget or data source subscribes to a parameter, a listener associated to that parameter change broadcast is set and when an update is broadcasted (as the message `parameter:<param_name>:update`), the wildcards are replaced with the new value and the component is reloaded.

6.2.9 Installation

Note that the detailed installation procedure is only summarized here and is better described on [Expedia's page](#).

Requirements:

- [AAC](#)
- Node.js
- MongoDB 2.6+ ([installation instructions](#))

Installation steps:

0. Ensure that MongoDB and AAC are running. Cyclotron will automatically create a database named “cyclotron”.
1. Clone this repository.
2. Install the REST API and create the configuration file `cyclotron-svc/config/config.js`. Paste in it the content of `sample.config.js`, which contains the configurable properties of the API, such as MongoDB instance and AAC endpoints.
3. Install Cyclotron website.
4. Start both:
 - API: from `cyclotron-svc` run the command `node app.js`
 - website: from `cyclotron-site` run the command `gulp server`

Now Cyclotron is running with its default settings and authentication is disabled.

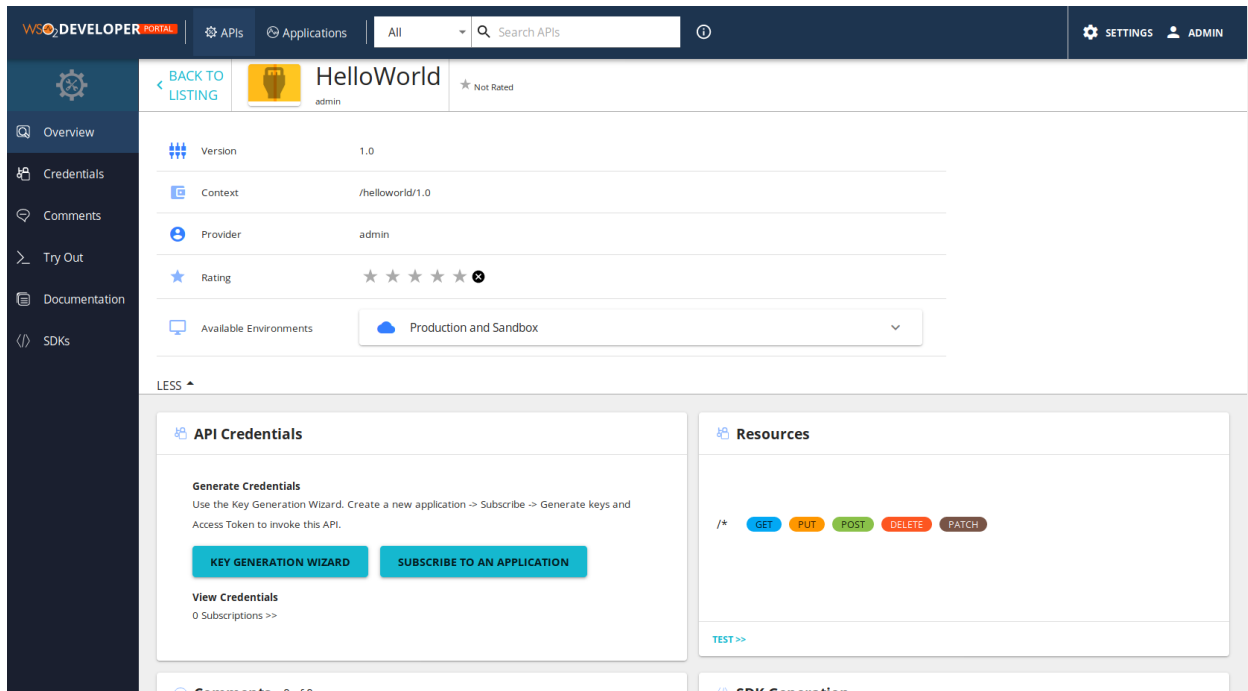
Community Layer

The Community Layer of the DigitalHub platform aims at opening the assets managed by the platform towards the community of developers, data consumers, and even end users. In this way, the APIs, Datasets, visualizations, are made opening in a standard, interoperable way, enriched with the documentation and access management capabilities. Besides, the Community Layer provides means for engaging the end users in the data collection and application usage through gamification functionalities.

To accomplish this, the Community Layer relies on the following base elements: Service Portal, Data Portal, and Gamification Engine.

7.1 Service Portal

Service Portal in DigitalHub relies on the API Store component of the *API Manager*.



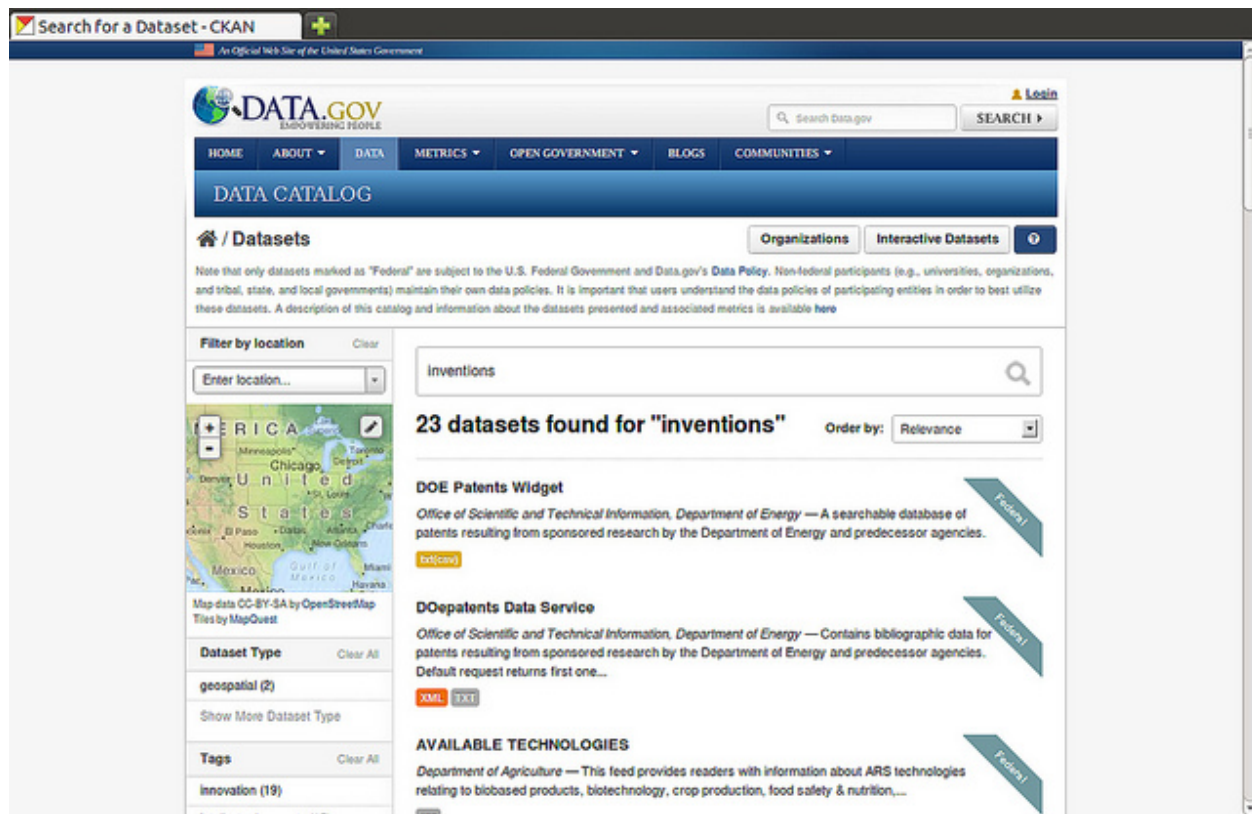
The Open Service exposed in ServiceHub through API Manager component are made available to the community via the API Store, where the APIs are divided across tenants and categorized. Through the store the API consumers may reference the API documentation, both formal (e.g., service interface definitions in Open API specification), and informal (e.g., tutorials, descriptions, wikis, etc), comment and rate the API, and even subscribe the APIs and test them directly. Furthermore, the API store Portal allows for

- downloading the autogenerated API SDK libraries,
- manage credentials for accessing the APIs
- collect statistics of the usage of the subscribed client app.

Further details on configuring and using the API Store portal are available [here](#).

7.2 Data Portal

Data Portal in DigitalHub relies on the [CKAN platform](#), an Open Source platform for making catalogs of open data, such as spreadsheets, geospatial GeoJSON data, OData, etc.



CKAN functionality allows the data providers to manage and publish collections of data. Each data collection, or dataset, may be provided with the necessary metadata regarding the data structure, attributes, formats, updates, etc.

Once the data is published, users can use its faceted search features to browse and find the data they need, and preview it using maps, graphs and tables - whether they are developers, journalists, researchers, NGOs, or citizens.

An important feature of CKAN is the extensibility, which allows one to create new module to deal with the specific data types and formats (e.g., OData, STA), visualization tools or widgets, data validation tools, authentication, etc.

Using CKAN, the community can access the data made available by the DigitalHub platform, e.g., GeoServer, DSS, API Manager. The data may be exposed in a public or protected manner, using the security features provided by AAC on top of the

7.3 Gamification Engine

Gamification Engine is the tool for implementing the user community engagement mechanisms within mobile and web applications. Gamification Engine allows for capturing the game model concepts, such as points, levels, badges, challenges, leaderboards, and define the rules for their assignments in correspondence to the actions performed by the users.

7.3.1 Game Model

Each game model defined with the gamification engine consists of a set of elements, such as game actions, game concepts, rules, and tasks. During the game execution, the game state is being populated and maintained. The game state is represented with the state of its individual players, namely user state, such as current user points, badges earned, etc.

Game

The game is a container of other element and is defined with the following set of attributes:

- **id:** automatically assigned from system. It uniquely identifies the game within the engine instance. The game ID is important for relating the actions to the game state during the execution.
- **name:** the game name.
- **owner:** the user owner of the game. This is the ID of the game manager, and is relevant for the multitenant execution environment.
- **game model:** the set of concept instances used by this specific game (different types of points, badges, rules, classifications, etc).

Game Action

The game actions represent the game-specific events that triggers changes in the game state. The game actions may be external and internal. The external actions correspond to the user actions (when triggered, e.g., through a mobile app), while the internal ones correspond to the system triggers (e.g., timers to recalculate periodic user classifications). Each action may carry some data with it.

Game Concepts

The game concepts represent elements like points and badges. Each game may define its own set of points (green points, community points, etc.), as well as badge collections. Participating to the game (through the actions), the user can earn or lose the points (i.e., her state changes).

Game Rules

A rule defines how the state of the users changes in reaction to some actions. Rules follow the “event-condition-action” paradigm, where event corresponds to the game action triggering the rule, condition represent the situation around the current user state and the action payload, and the action defines how the state should evolve (e.g., points that are added, badge that is earned, and the notification is sent).

Currently, the Drools Rule (<http://www.drools.org/>) language is used for specifying the game rules.

Game Tasks

The task defines a periodically scheduled internal actions and their characteristics. A typical example of the game task is the classification task, where for some specific type of points the activity is triggered on a fixed schedule (e.g., daily) to incentivate best users.

7.3.2 Gamification Engine Tools

The functionality of the Gamification Engine is made available via two key modules:

- **Gamification Engine API:** allows for interacting with the engine programmatically, e.g., to get information about the leaderboard, the user game status, triggering actions, reading notifications, etc.
- **Gamification Engine Management API:** the set of API interfaces for creating and managing game rules, concepts, and definitions.
- **Gaimification Engine Console:** the administration UI for accessing the management functionality of the engine

- Gamification Status Console: the UI for accessing and visualizing the current state of the game, the users, and for performing some of the operations directly on the game data.

The detailed information about the Gamification Engine setup, the model, and its usage may be found here: <https://github.com/smartcommunitylab/smartcampus.gamification/wiki/>.

8.1 Prerequisites

- Kubernetes 1.14+ with RBAC enabled
- Helm 2.14+

8.2 Install core components

Use Git to clone DigitalHub Platform repository

```
$ git clone https://github.com/scc-digitalhub/platformdocs.git
```

8.2.1 nginx-ingress

nginx-ingress is an Ingress controller that uses ConfigMap to store the nginx configuration.

Create an ingress controller

```
$ kubectl create namespace ingress
$ helm install --namespace ingress --name my-release stable/nginx-ingress
```

During the installation, a public IP address is created for the ingress controller.

To get the public IP address use the following command:

```
$ kubectl get service --namespace ingress
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
↪ PORT(S)	AGE		

(continues on next page)

(continued from previous page)

my-release-ingress-controller	LoadBalancer	10.0.147.113	YOUR_EXTERNAL_	
↪ IP 80:30058/TCP, 443:32725/TCP	26d			
my-release-ingress-controller-metrics	ClusterIP	10.0.48.136	<none>	⌋
↪ 9913/TCP	25d			
my-release-ingress-default-backend	ClusterIP	10.0.190.184	<none>	⌋
↪ 80/TCP	26d			

8.2.2 cert-manager

cert-manager is a Kubernetes addon to automate the management and issuance of TLS certificates from various issuing sources.

Installing with Helm

Install the *CustomResourceDefinition* from jetstack repo.

```
$ kubectl apply --validate=false -f https://raw.githubusercontent.com/jetstack/cert-
↪ manager/v0.13.0/deploy/manifests/00-crds.yaml
```

Create cert-manager namespace.

```
$ kubectl create namespace cert-manager
```

Add the jetstack helm repository

```
$ helm repo add jetstack https://charts.jetstack.io
```

Update your local Helm chart repository cache.

```
$ helm repo update
```

Install cert-manager with helm.

```
$ helm install \
--name cert-manager \
--namespace cert-manager \
--version v0.13.0 \
jetstack/cert-manager
```

Check in cert-manager namespace if all pods are up & running

```
$ kubectl get pods -n cert-manager
```

NAME	READY	STATUS	RESTARTS	AGE
cert-manager-587dc68fc4-kp8hk	1/1	Running	0	3h4m
cert-manager-cainjector-67ff67fd45-vws89	1/1	Running	0	3h4m
cert-manager-webhook-5c8cf6d9d4-8lv6p	1/1	Running	0	3h4m

Create *ClusterIssuer* definition.

```
$ cat <<EOF > clusterissuer-test.yaml
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
```

(continues on next page)

(continued from previous page)

```

name: letsencrypt-staging
spec:
  acme:
    # You must replace this email address with your own.
    # Let's Encrypt will use this to contact you about expiring
    # certificates, and issues related to your account.
    email: user@example.com
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      # Secret resource used to store the account's private key.
      name: example-issuer-account-key
    # Add a single challenge solver, HTTP01 using nginx
    solvers:
    - http01:
        ingress:
          class: nginx
EOF

```

Install *ClusterIssuer*.

```
$ kubectl apply -f clusterissuer-test.yaml
```

8.2.3 Install MySQL

Create one secret with init script e another with root credentials.

```
$ kubectl create secret generic mysql-dbscripts --from-file=mysql/init-scripts/
$ kubectl create secret generic mysql-db-ps --from-literal=rootps=rootpassword
```

Deploy MySQL container.

```
$ kubectl apply -f mysql/
```

8.2.4 Install PostgreSQL

Create one secret with init script e another with root credentials.

```
$ kubectl create secret generic postgres-dbscripts --from-file=postgresql/init-
↪scripts/
$ kubectl create secret generic postgrescrt --from-literal=user=rootuser --from-
↪literal=ps=rootpassword
```

Deploy PostgreSQL container.

```
$ kubectl apply -f postgresql/
```

8.3 Install platform components

8.3.1 AAC

Configuration

Configure AAC using environment variables in `aac/aac-configmap.yaml` file.

See documentation for details: <https://digitalhub.readthedocs.io/en/latest/docs/service/aac.html>

Installation

```
$ kubectl apply -f aac/
```

8.3.2 Org-Manager

Configuration

Configure Org-Manager using environment variables in `org-manager/org-manager-configmap.yaml` file.

See documentation for details: <https://digitalhub.readthedocs.io/en/latest/docs/service/orgman.html>

Installation

```
$ kubectl apply -f org-manager/
```

8.3.3 WSO2 API Manager with APIM-Analytics

1. APIM-Analytics

Configuration

Configure APIM-Analytics using environment variables in `apim-analytics/apim-analytics-configmap.yml` file.

See documentation for details: <https://digitalhub.readthedocs.io/en/latest/docs/service/apim.html>

Installation

```
$ kubectl apply -f apim-analytics/
```

2. API-Manager

Configuration

Configure API-Manager using environment variables in `api-manager/apim-configmap.yml` file.

See documentation for details: <https://digitalhub.readthedocs.io/en/latest/docs/service/apim.html>

Installation

```
$ kubectl apply -f api-manager/
```

8.3.4 Dremio

Configuration

Configure Dremio using environment variables in dremio/dremio-configmap.yaml file.

See documentation for details: <https://digitalhub.readthedocs.io/en/latest/docs/data/dremio.html>

Installation

```
$ kubectl apply -f dremio/
```

8.3.5 Dss

Configuration

Configure Dss using environment variables in dss/dss-configmap.yaml file.

See documentation for details: <https://digitalhub.readthedocs.io/en/latest/docs/data/dss.html>

Installation

```
$ kubectl apply -f dss/
```

8.3.6 JupyterHub

Installation & Configuration

Install JupyterHub using Helm Chart.

```
$ helm repo add jupyterhub https://jupyterhub.github.io/helm-chart/
$ helm repo update
$ helm install --name jupyterhub --namespace jupyterhub jupyterhub/jupyterhub --
↪values config.yaml
```

The JupyterHub Helm chart is configurable by values in your config.yaml.

See documentation for details: <https://zero-to-jupyterhub.readthedocs.io/en/latest/>

8.3.7 Minio

Installation & Configuration

Install Minio using Helm Chart.

```
$ helm install --name minio --namespace minio stable/minio --values values.yaml
```

The Minio Helm chart is configurable by values in your values.yaml.

See documentation for details: <https://docs.min.io/docs/deploy-minio-on-kubernetes.html>

8.3.8 Nifi

Configuration

Use the tls-toolkit command line utility to automatically generate the required keystores, truststore.

See documentation for details: https://nifi.apache.org/docs/nifi-docs/html/toolkit-guide.html#tls_toolkit

Create a Secret that holds both keystore and truststore.

```
$ kubectl create secret generic nifi-keystore --from-file=keystore.jks --from-  
↪file=truststore.jks
```

Installation

```
$ kubectl apply -f nifi/
```

8.3.9 Nuclio with OAuth2_Proxy

1. Nuclio

Install Nuclio using Helm Chart.

```
$ helm repo add nuclio https://nuclio.github.io/nuclio/charts  
$ helm repo update  
$ helm install --namespace nuclio --name nuclio nuclio/nuclio --values values.yaml
```

The Nuclio Helm chart is configurable by values in your values.yaml.

See documentation for details: <https://github.com/nuclio/nuclio/tree/master/hack/k8s/helm/nuclio>

2. OAuth2_Proxy

Configuration

Configure OAuth2_Proxy using environment variables in nuclio/oauth2_proxy.yml file.

See documentation for details: https://pusher.github.io/oauth2_proxy

Installation

```
$ kubectl apply -f nuclio/
```

8.3.10 Resource Manager

Configuration

Configure Resource Manager using environment variables in resource-manager/resource-manager-configmap.yaml file.

See documentation for details: <https://digitalhub.readthedocs.io/en/latest/docs/data/rm.html>

Installation

```
$ kubectl apply -f resource-manager/
```

8.3.11 SQLPad

Configuration

Configure SQLPad using environment variables in sqlpad/sqlpad-configmap.yaml file.

See documentation for details: <https://digitalhub.readthedocs.io/en/latest/docs/data/sqlpad.html>

Installation

```
$ kubectl apply -f sqlpad/
```

8.3.12 Thingsboard

Installation & Configuration

Refer to the documentation to run ThingsBoard in Microservices Mode.

Kubernetes resources configuration for ThingsBoard Microservices <https://github.com/thingsboard/thingsboard/tree/master/k8s#kubernetes-resources-configuration-for-thingsboard-microservices>

ThingsBoard documentation: <https://thingsboard.io/docs/>