Aurelius Atlas Data Governance Solution

Release 0.1

Mar 20, 2024

Menu

Thank you for your interest in Aurelius Atlas Data Governance solution, powered by Apache Atlas.

Here you will find a step-by-step approach how to deploy, operate and use Aurelius Atlas, including a demo environment, videos and possible user stories.

Aurelius Atlas is an open-source solution, which can be used free under the Elastic V2 license agreement.

CHAPTER 1

What is Aurelius Atlas?

Welcome to the **Aurelius Atlas solution** powered by Apache Atlas, Aurelius Atlas is an open-source **Data Governance solution**, based on a selection of open-source tools to facilitate business users to access governance information in an easy consumable way and meet the data governance demands of the distributed data world.

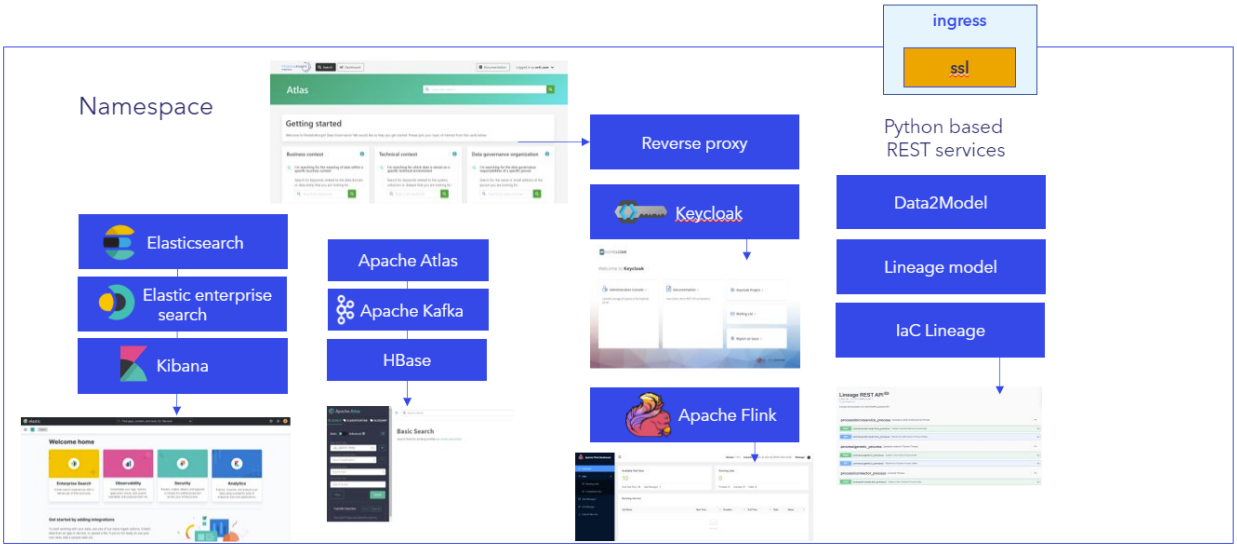
It is a Data Governance solution powered by Apache Atlas for Data in motion **(in transit)**, Data in rest, Data in use. It manages distributed data governance over multicloud environment as well as hybrid **(On-Prem)**

1.1 Components of Aurelius Atlas Helm:

- Apache Atlas
- Kafka UI
- Apache Flink
- Elasticsearch
- Keycloak
- API services

The deployment of our solution is provided as a helm chart so you can roll it out in your Kubernetes cluster.

The solution itself consists of Apache Atlas in the core with Apache Kafka used in HBase, you also publish and make accessible the original Apache user interface.



aureliusdev.westeurope.cloudapp.azure.com/namespace/atlas

In addition to that we deployed a Keycloack which is our identity provider it's open source also, which allows to integrate with all kinds of other identity providers like in our demo environment which you can try by Clicking here.

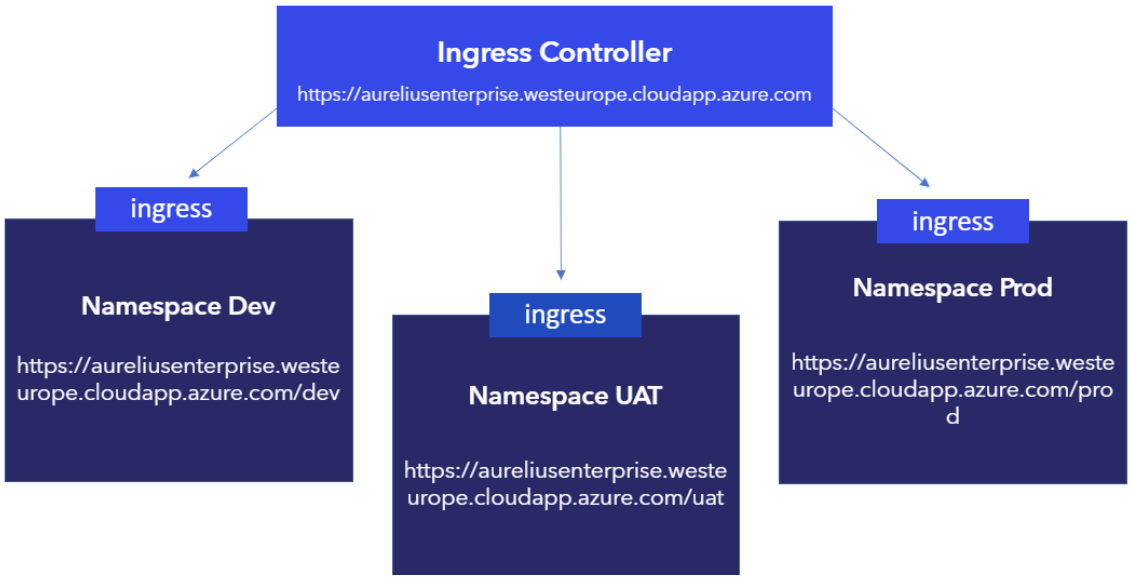
We connect with Gmail, but you can also connect to an active directory somewhere, on top of that we have our actual user interface, which is included in what we call the *Reverse proxy port*.

This port have a lot of uses for searches and full text search but also with different facets, for that we are using the Elastic stack, so an elastic search is and elastic enterprise search and a Kibana just to manage the environment, we also publish the Kibana interface in this helm chart, since the synchronization all changes are directly performed in Apache Atlas but then have to be updated in the elastic environment.

We use Apache Flink and some jobs in there streaming jobs in there to consume the Kafka events from Apache Atlas and translate that into changes in the Elastic Enterprise Search environment using these streams as additional service.

We have rest based services for the data to model and the lineage model both are related required for the lineage graph generation and we have the REST API for integrating our solution with infrastructure as code in an easy way also provided in the image.

Different namespaces on the same cluster for different, independent deployments.



It is possible to deploy the helm chart multiple times in different namespaces, so in our usual environments we have governance set up for the dev environment for the user acceptance environment, and for the production environment, they can all run in the same Kubernetes cluster underneath the same increased controller, and you will always have the same URLs except that the namespace becomes part of the URL and everything will be related there.

So, to understand how these different components work together, click here to go to the technical documentation

If you want to learn more about all the components that made up Aurelius Atlas, Click here

1.2 What do I need to run the application?

To be able to deploy Aurelius Atlas a **Kubernetes cluster will be needed**.

These are some of the components that you need to run the application, be sure that you have them, before running the application.

If you do not have it click on the name to go to the external documentation to set up.

- Apache Atlas
- Kafka UI
- Apache Flink
- Elasticsearch
- Keycloak
- API services

If you already have it, you can go directly to the deploy section by Clicking here.

1.3 Integration Options

Aurelius Atlas has different options to integrate here is an overview of the integration options:

- Identity providers via Keycloak **(AAD, gmail,...)**

- External* Apache Atlas
- External* Elastic
- External* Kafka
- Hadoop
- Azure
- AWS

Click here to know more about the integration options.

CHAPTER 2

Deployment options

2.1 Azure Deploy Aurelius Atlas

2.1.1 Getting started

Welcome to the Aurelius Atlas solution powered by Apache Atlas! Aurelius Atlas is an open-source Data Governance solution, based on a selection of open-source tools to facilitate business users to access governance information in an easy consumable way and meet the data governance demands of the distributed data world.

Here you will find the instillation instructions and the required setup of the kubernetes instructions, followed by how to deploy the chart in different namespaces.

2.1.2 Installation Requirements

This installation assumes that you have:

- a kubernetes cluster running with 2 Node of CPU 4 and 16GB
- Chosen Azure Cli installed
  - az
- kubectl installed and linked to Azure Cli
  - az linked

Further you need the helm chart to deploy all services from https://github.com/aureliusenterprise/Aurelius-Atlas-helm-chart

2.1.3 Required Packages

The deployment requires the following packages:

- **Certificate Manager**

  - To handel and manage the creation of certificates
  - Used in demo: cert-manager
- **Ingress Controller**
  - Used to create an entry point to the cluster through an external IP.
  - Used in demo: Nginx Controller
- **Elastic**
  - Used to deploy elastic on the kubernetes cluster
  - In order to deploy elastic, `Elastic Cluster on Kubernetes (ECK)` must be installed on the cluster. To install ECK on the cluster, please follow the instructions provided on https://www.elastic.co/guide/en/cloud-on-k8s/master/k8s-deploy-eck.html
- **Reflector**
  - Used to reflect secrets across namespaces
  - Used in demo to share the DNS certificate to different namespace

The steps on how to install the required packages

1. Install Certificate manager

Only install if you do not have a certificate manager. Please be aware if you use another manger, some commands later will need adjustments. The certificate manager here is cert-manager.

```
helm repo add jetstack https://charts.jetstack.io
helm repo update
helm install  cert-manager jetstack/cert-manager   --namespace cert-manager   --
↪create-namespace   --version v1.9.1 --set installCRDs=true
```

2. Install Ingress Nginx Controller

Only install if you do not have an Ingress Controller.

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
helm install nginx-ingress ingress-nginx/ingress-nginx --set controller.
↪publishService.enabled=true --set controller.service.annotations."service\.beta\.
↪kubernetes\.io/azure-load-balancer-health-probe-request-path"=/healthz
```

It is also possible to set a DNS label to the ingress controller if you do not have a DNS by adding `--set controller.service.annotations."service\.beta\.kubernetes\.io/azure-dns-label-name"=<label>`

3. Install Elastic

```
kubectl create -f https://download.elastic.co/downloads/eck/2.3.0/crds.yaml
kubectl apply -f https://download.elastic.co/downloads/eck/2.3.0/operator.yaml
```

4. Install Reflector

```
helm repo add emberstack https://emberstack.github.io/helm-charts
helm repo update
helm upgrade --install reflector emberstack/reflector
```

2.1.4 Azure DNS Label

In Azure, it is possible to apply a DNS label to the ingress controller, if you do not have a DNS.

Edit the ingress controller deployment (if not set upon installation)

```
helm upgrade nginx-ingress ingress-nginx/ingress-nginx --reuse-values --set␣
↪controller.service.annotations."service\.beta\.kubernetes\.io/azure-dns-label-name"=
↪<label>
```

Save and exit. Resulting DSN will be `<label>.westeurope.cloudapp.azure.com`

2.1.5 Put ssl certificate in a Secret

Before you start, update zookeeper dependencies:

```
cd charts/zookeeper/
helm dependency update
```

2.1.6 Define a cluster issuer

This is needed if you installed cert-manager from the required packages.

Here we define a CLusterIssuer using cert-manager on the cert-manager namespace

1. Move to the home directory of the chart helm-governance
2. Uncomment templates/prod_issuer.yaml.
3. Update the `{{ .Values.ingress.email_address }}` in values.yaml file and create the ClusterIssuer with the following command

   ```
   helm template -s templates/prod_issuer.yaml . | kubectl apply -f -
   ```

4. comment out prod_issuer.yaml in templates Check that it is running:

   ```
   kubectl get clusterissuer -n cert-manager
   ```

5. It is running when Ready is True.

   ```
   NAME                                    READY   AGE
   letsencrypt-clusterissuer-aureliusdev   True    24h
   ```

2.1.7 Create ssl certificate

This is needed if you installed cert-manager from the required packages.

1. Assumes you have a DNS linked to the external IP of the ingress controller
2. Move to the home directory of the chart helm-governance
3. Uncomment templates/certificate.yaml
4. Update the values.yaml file `{{ .Values.ingress.dns_url}}` to your DNS name
5. Create the certificate with the following command

```
helm template -s templates/certificate.yaml . | kubectl apply -f -
```

6. Comment out certificate.yaml in templates.
7. Check that it is approved.

```
kubectl get certificate -n cert-manager
```

It is running when Ready is True

```
NAME                READY   SECRET                          AGE
cert-aureliusdev    True    letsencrypt-secret-aureliusdev   8h
```

2.1.8 Deploy Aurelius Atlas

1. Create the namespace

```
kubectl create namespace <namespace>
```

1. Update the values.yaml file
   - `{{ .Values.keycloak.keycloakFrontendURL }}` replace it to your DNS name
   - `{{ .Values.kafka-ui. ... .bootstrapServers }}` edit it with your *<namespace>*
   - `{{ .Values.kafka-ui. ... .SERVER_SERVLET_CONTEXT_PATH }}` edit it with your *<namespace>*
2. Deploy the services

```
cd Aurelius-Atlas-helm-chart
helm dependency update
helm install --generate-name -n <namespace>  -f values.yaml .
```

Users with Randomized Passwords

In the helm chart 5 base users are created with randomized passwords stored as secrets on kubernetes.

The 5 base users are:

1. Keycloak Admin User
2. Atlas Admin User
3. Atlas Data Steward User

4. Atlas Data User
5. Elastic User

To get the randomized passwords out of kubernetes there is a bash script get_passwords.

```
./get_passwords.sh <namespace>
```

The above command scans the given <namespace> and prints the usernames and randomized passwords as follows:

```
keycloak admin user pwd:
username: admin
vntoLefBekn3L767
----
keycloak Atlas admin user pwd:
username: atlas
QUVTj1QDKQWZpy27
----
keycloak Atlas data steward user pwd:
username: steward
XFlsi25Nz9h1VwQj
----
keycloak Atlas data user pwd:
username: scientist
PPv57ZvKHwxCUZOG
==========
elasticsearch elastic user pwd:
username: elastic
446PL2F2UF55a19haZtihRm5
----
```

2.1.9 Check that all pods are running

```
kubectl -n <namespace> get all # check that all pods are running
```

Aurelius Atlas is now accessible via reverse proxy at <DNS-url>/<namespace>/atlas/

2.1.10 Initialize the Atlas flink tasks and optionally load sample data

Flink: - For more details about this flink helm chart look at *flink readme <./charts/flink/README.md>*

Init Jobs:

- Create the Atlas Users in Keycloak
- Create the App Search Engines in Elastic

```
kubectl -n <namespace> exec -it <pod/flink-jobmanager-pod-name> -- bash
```

```
cd init
pip3 install m4i-atlas-core@git+https://github.com/aureliusenterprise/m4i_atlas_core.
↪git#egg=m4i-atlas-core --upgrade
cd ../py_libs/m4i-flink-tasks/scripts
/opt/flink/bin/flink run -d -py get_entity_job.py
/opt/flink/bin/flink run -d -py publish_state_job.py
/opt/flink/bin/flink run -d -py determine_change_job.py
/opt/flink/bin/flink run -d -py synchronize_appsearch_job.py
```

```
/opt/flink/bin/flink run -d -py local_operation_job.py
## To Load the Sample Demo Data
cd
cd init
./load_sample_data.sh
```

2.2 Google Deploy Aurelius Atlas

2.2.1 Getting started

Welcome to the Aurelius Atlas solution powered by Apache Atlas! Aurelius Atlas is an open-source Data Governance solution, based on a selection of open-source tools to facilitate business users to access governance information in an easy consumable way and meet the data governance demands of the distributed data world.

Here you will find the instillation instructions and the required setup of the kubernetes instructions, followed by how to deploy the chart in different namespaces.

2.2.2 Installation Requirements

This installation assumes that you have:

- A kubernetes cluster running with 2 Node of CPU 4 and 16GB
- Gcloud Cli installed
  - gcloud
- kubectl installed and linked to Gcloud Cli
  - gcloud linked
- Helm installed locally
- A DomainName

Further you need the helm chart to deploy all services from https://github.com/aureliusenterprise/Aurelius-Atlas-helm-chart

2.2.3 Required Packages

The deployment requires the following packages:

- **Certificate Manager**
  - To handel and manage the creation of certificates
  - Used in demo: cert-manager
- **Ingress Controller**
  - Used to create an entry point to the cluster through an external IP.
  - Used in demo: Nginx Controller
- **Elastic**
  - Used to deploy elastic on the kubernetes cluster

  - In order to deploy elastic, `Elastic Cluster on Kubernetes (ECK)` must be installed on the cluster. To install ECK on the cluster, please follow the instructions provided on https://www.elastic.co/guide/en/cloud-on-k8s/master/k8s-deploy-eck.html

- **Reflector**
  - Used to reflect secrets across namespaces
  - Used in demo to share the DNS certificate to different namespace
- Zookeeper

The steps on how to install the required packages

1. Install Certificate manager

Only install if you do not have a certificate manager. Please be aware if you use another manger, some commands later will need adjustments. The certificate manager here is cert-manager.

```
helm repo add jetstack https://charts.jetstack.io
helm repo update
helm install  cert-manager jetstack/cert-manager   --namespace cert-manager   --
↪create-namespace   --version v1.9.1   --set installCRDs=true   --set   global.
↪leaderElection.namespace=cert-manager
```

- It is successful when the output is like this:

```
NOTES:
cert-manager v1.91 has been deployed successfully
```

2. Install Ingress Nginx Controller

Only install if you do not have an Ingress Controller.

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
helm install nginx-ingress ingress-nginx/ingress-nginx --set controller.
↪publishService.enabled=true
```

3. Install Elastic

```
kubectl create -f https://download.elastic.co/downloads/eck/2.3.0/crds.yaml
kubectl apply -f https://download.elastic.co/downloads/eck/2.3.0/operator.yaml
```

4. Install Reflector

```
helm repo add emberstack https://emberstack.github.io/helm-charts
helm repo update
helm upgrade --install reflector emberstack/reflector
```

5. Update Zookeeper Dependencies

Move to the directory of Aurelius-Atlas-helm-chart

```
cd charts/zookeeper/
helm dependency update
```

2.2.4 Get Ingress Controller External IP to link to DNS

Only do this if your ingress controller does not already have a DNS applied.

2.2.5 Get External IP to link to DNS

```
kubectl get service/nginx-ingress-ingress-nginx-controller
```

Take the external-IP of the ingress controller Link your DNS to this external IP.

2.2.6 Put ssl certificate in a Secret

2.2.7 Define a cluster issuer

This is needed if you installed letsencrypt from the required packages.

Here we define a CLusterIssuer using letsencrypt on the cert-manager namespace:

- Move to the directory of Aurelius-Atlas-helm-chart
- Uncomment prod_issuer.yaml in templates
- Update `{{ .Values.ingress.email_address }}` in values.yaml file
- Create the clusterIssuer with the following command

```
helm template -s templates/prod_issuer.yaml . | kubectl apply -f -
```

- Comment out prod_issuer.yaml in templates

Check that it is running:

```
kubectl get clusterissuer -n cert-manager
```

It is running when Ready is True.

```
NAME                                    READY   AGE
letsencrypt-clusterissuer-aureliusdev   True    24h
```

2.2.8 Create ssl certificate

This is needed if you installed letsencrypt from the required packages.

- Assumes you have a DNS linked to the external IP of the ingress controller

- Move to the directory of Aurelius-Atlas-helm-chart
- Uncomment certificate.yaml in templates
- Update the Values file `{{ .Values.ingress.dns_url}}` to your DNS name
- Create the certificate with the following command

```
helm template -s templates/certificate.yaml . | kubectl apply -f -
```

- Comment out certificate.yaml in templates

Check that it is approved:

```
kubectl get certificate -n cert-manager
```

It is running when Ready is True.

```
NAME                READY   SECRET                          AGE
cert-aureliusdev    True    letsencrypt-secret-aureliusdev   8h
```

2.2.9 Deploy Aurelius Atlas

1. Update the values.yaml file
   - `{{ .Values.keycloak.keycloakFrontendURL }}` replace it to your DNS name
   - `{{ .Values.kafka-ui. ... .bootstrapServers }}` edit it with your <*namespace*>
   - `{{ .Values.kafka-ui. ... .SERVER_SERVLET_CONTEXT_PATH }}` edit it with your <*namespace*>
- Create the namespace

```
kubectl create namespace <namespace>
```

- Deploy the services

```
cd Aurelius-Atlas-helm-chart
helm dependency update
helm install --generate-name -n <namespace>  -f values.yaml .
```

Please note that it can take 5-10 minutes to deploy all services.

Users with Randomized Passwords

In the helm chart 5 base users are created with randomized passwords stored as secrets on kubernetes.

The 5 base users are:

1. Keycloak Admin User
2. Atlas Admin User
3. Atlas Data Steward User
4. Atlas Data User
5. Elastic User

To get the randomized passwords out of kubernetes there is a bash script get_passwords.

```
./get_passwords.sh <namespace>
```

The above command scans the given <namespace> and prints the usernames and randomized passwords as follows:

```
keycloak admin user pwd:
username: admin
vntoLefBekn3L767
----
keycloak Atlas admin user pwd:
username: atlas
QUVTj1QDKQWZpy27
----
keycloak Atlas data steward user pwd:
username: steward
XFlsi25Nz9h1VwQj
----
keycloak Atlas data user pwd:
username: scientist
PPv57ZvKHwxCUZOG
==========
elasticsearch elastic user pwd:
username: elastic
446PL2F2UF55a19haZtihRm5
----
```

```
kubectl -n <namespace> get all # check that all pods are running
```

Atlas is now accessible via reverse proxy at <DNS-url>/<namespace>/atlas/

2.2.10 Initialize the Atlas flink tasks and optionally load sample data

Flink:

- For more details about this flink helm chart look at flinkreadme

Init Jobs:

- Create the Atlas Users in Keycloak
- Create the App Search Engines in Elastic

```
kubectl -n <namespace> exec -it <pod/flink-jobmanager-pod-name> -- bash
```

```
cd init

pip3 install m4i-atlas-core@git+https://github.com/aureliusenterprise/m4i_atlas_core.
↪git#egg=m4i-atlas-core --upgrade

cd ../py_libs/m4i-flink-tasks/scripts

/opt/flink/bin/flink run -d -py get_entity_job.py
/opt/flink/bin/flink run -d -py publish_state_job.py
/opt/flink/bin/flink run -d -py determine_change_job.py
/opt/flink/bin/flink run -d -py synchronize_appsearch_job.py
/opt/flink/bin/flink run -d -py local_operation_job.py
```

```
cd init
./load_sample_data.sh
```

2.3 How to deploy Aurelius Atlas with Docker Compose

2.3.1 Getting started

Welcome to Aurelius Atlas, a powerful data governance solution powered by Apache Atlas! Aurelius Atlas leverages a carefully curated suite of open-source tools to provide business users with seamless access to governance information. Our solution is designed to address the evolving demands of data governance in a distributed data environment, ensuring that you can easily consume and utilize valuable governance insights.

This guide provides comprehensive instructions for setting up the Docker Compose deployment and covers various deployment scenarios. You will find step-by-step instructions to configure the required setup and deploy the system.

2.3.2 Description of system

The solution is based on Apache Atlas for metadata management and governance, and Apache Kafka is utilized for communicating changes in the system between different components. A Kafka Web based user interface is made accessible to have easy access to the Apache Kafka system for maintenance and trouble shooting. Additionally, an Apache server is implemented to handle and distribute frontend traffic to the corresponding components. A custom interface has been developed to enable effective search and browsing functionality using full-text search capabilities, leveraging the power of the Elastic stack. This stack includes Elasticsearch, Enterprise Search, and Kibana. Keycloak serves as the identity provider implementing Single Sign On functionality for all Web based user interfaces. Apache Flink is used to facility the creation of metadata to support the search functionality. Thus, Apache Flink runs streaming jobs that consume Kafka events from Apache Atlas and create metadata in Elastic Enterprise Search.

2.3.3 Hardware requirements

- 4 CPU cores
- 32GB RAM
- 100GB DISK

2.3.4 Installation Requirements

To deploy this solution you will need to install the following components:

- docker
- docker compose

Please ensure that you have these components installed on both the host and client machines for a successful deployment

In addition you need the docker compose file from https://github.com/aureliusenterprise/Aurelius-Atlas-docker-compose.

2.3.5 How to connect to the docker-compose environment?

For the client a local machine is required and for the host a VM or local machine can be used. Below we describe some possible scenarios for this deployment

2.3.6 Deployment on local machine

No additional action is required

2.3.7 Deployment on VM with public domain name

Connect to the VM using as destination its public IP

2.3.8 Deployment on VM without public domain name

In this scenario the following additional components are required:

Host:

- ssh server

Client:

- ssh client

To achieve connectivity with the Host and the Client the following steps have to be taken:

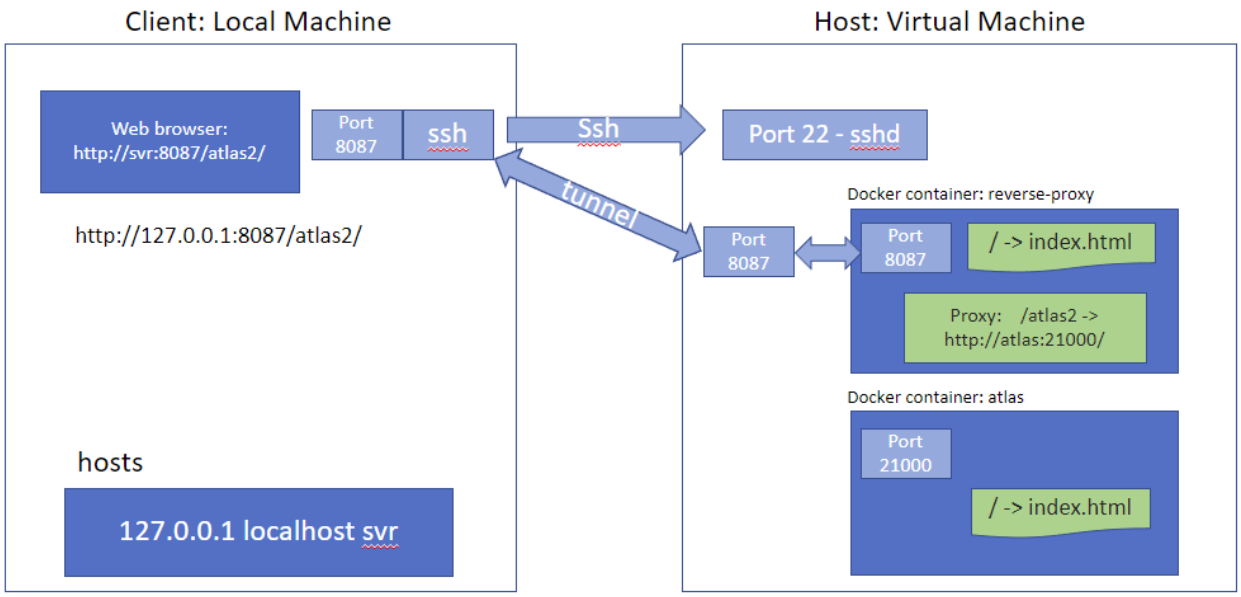
- From the client Connect to the Host using as destination the hosts IP address
- Define a ssh tunnel from the client to the host for port 8087

```
8087 -> 127.0.0.1:8087
```

- Extend hosts file on the client with the following line (admin right required)

```
127.0.0.1       localhost localhost4 $EXTERNAL_HOST
```

This is a representation of the described deployment on VM:



2.3.9 Preparatory Steps

On the host:

1. Start docker (admin rights required):

```
sudo service docker start
```

2. Obtain the IP address or hostname of the host machine's eth0 interface:

- If deployment is on local machine:

```
export EXTERNAL_HOST=$(ifconfig eth0 | grep 'inet' | cut \-d: \-f2 | sed \-e 's/.
↪\*inet \\([^ ]\*\\).\*/\\1/')
```

- If deployment is on a VM:

```
export EXTERNAL_HOST={hostname of VM}
```

3. Run the following script:

```
./retrieve_ip.sh
```

This script updates the values of *$EXTERNAL_HOST* within the templates used to generate the necessary configuration files for the various services.

4. Grant Elasticsearch sufficient virtual memory to facilitate its startup (admin rights required):

```
sudo sysctl -w vm.max_map_count=262144
```

For more details on configuring virtual memory for Elasticsearch, refer to the elastic documentation page

2.3.10 Default Users

By default these roles are created in the different services:

- **Elastic Admin User:** Username: elastic

  Password: elasticpw

- **Keycloak Admin user:** Username: admin

  Password: admin

- **Aurelius/Apache Atlas Admin User:** Username: atlas

  Password: 1234

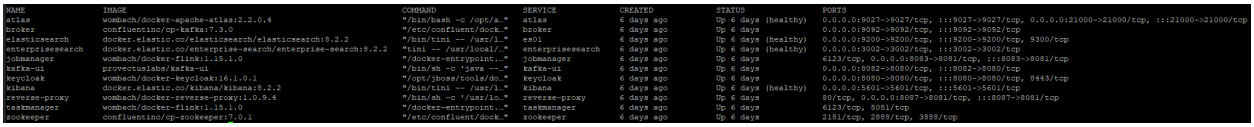
2.3.11 Spin up docker-compose environment

To start up the system, execute the following command on the host.

```
docker compose up -d
```

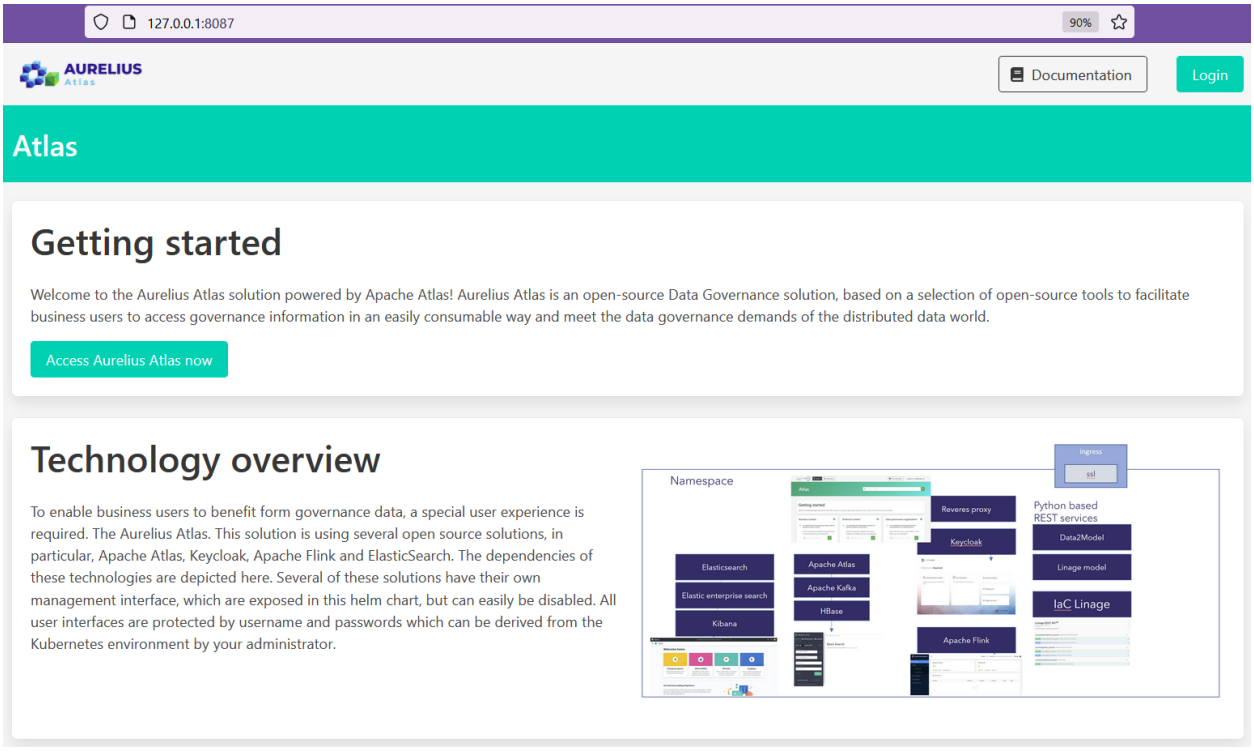
Starting up the system may take several minutes.

This is how the system looks in operational state:



When the Apache Atlas container state changes from starting to healthy, then the system is ready.

You are now able to access Aurelius Atlas at the URL: *http://$EXTERNAL_HOST:8087/*



You can find more information about the product in this page

2.3.12 Notes

- How to restart Apache Atlas?

```
docker exec -it atlas /bin/bash
cd /opt/apache-atlas-2.2.0/bin/
python atlas_stop.py
python atlas_start.py
```

- How to restart reverse proxy?

```
docker exec -it reverse-proxy /bin/bash
apachectl restart
```

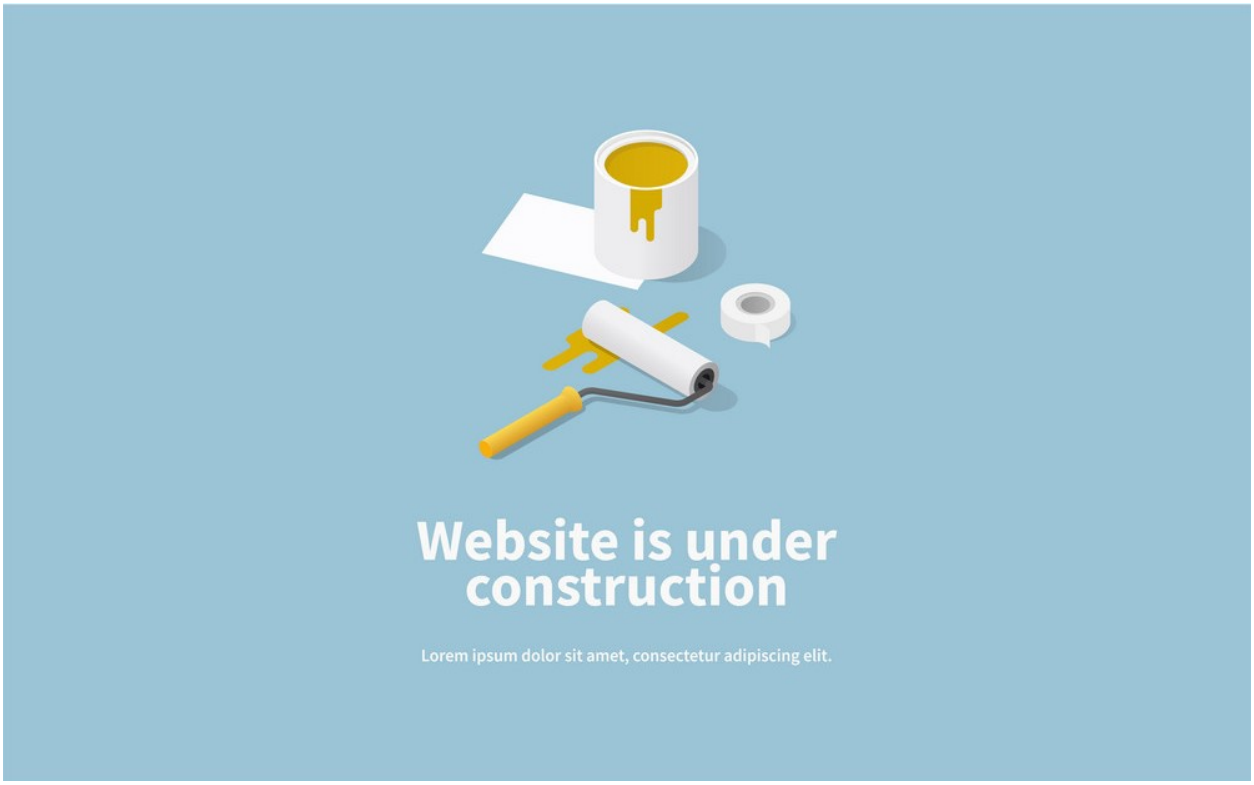
There are multiple deployment options:

- *Helm charts in a kubernetes cluster for Azure*
- *Helm charts in a kubernetes cluster for Google*
- *Standalone deployment using docker compose*

CHAPTER 3

Technical description

COMING SOON…..



CHAPTER 4

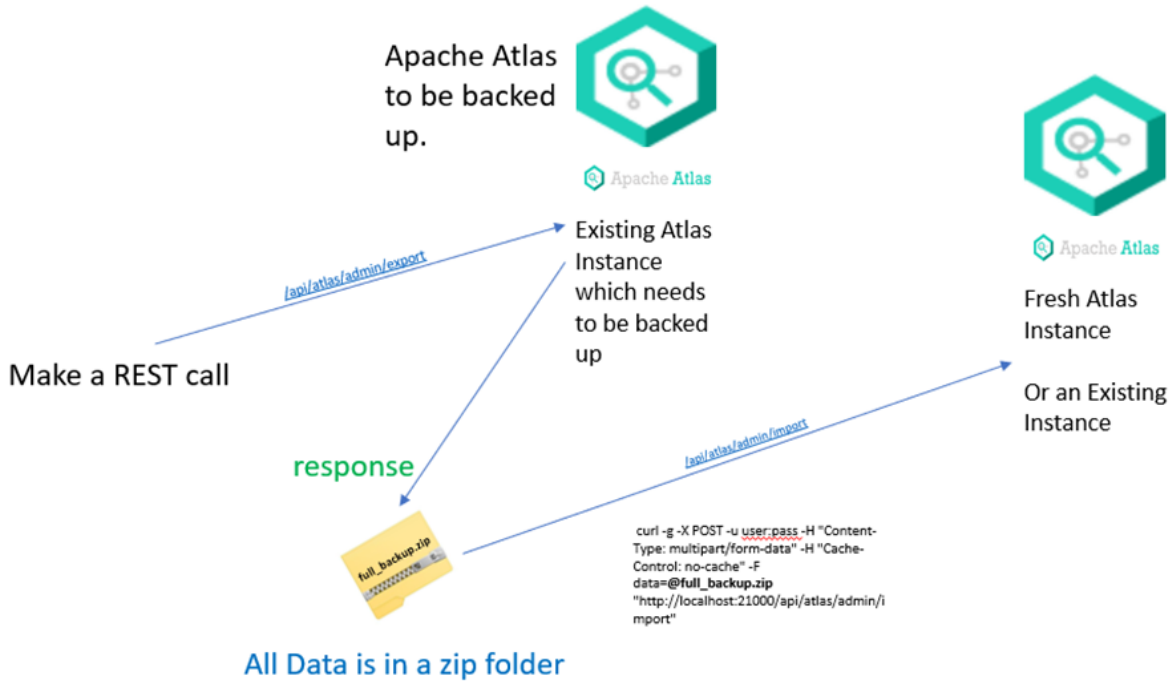
Aurelius Atlas Backup

Here you will find how to back up Aurelius Atlas for moving instances.

This process will result in zip files of the Apache Atlas data and a Snapshot repository of Elasticsearch indices that can be used for backup and in the case of disaster recover process.

4.1 Apache Atlas backup

4.1.1 Apache Atlas Backup Process Overview



4.1.2 Acquire access token for Apache Atlas's admin user

You can use `oauth.sh` script from https://github.com/aureliusenterprise/Aurelius-Atlas-helm-chart. Example usage:

```
export ACCESS_TOKEN=$(./oauth.sh --endpoint https://aureliusdev.westeurope.cloudapp.
↪azure.com/demo/auth/realms/m4i/protocol/openid-connect/token \
--client-id m4i_atlas \
--access atlas $ATLAS_USER_PASSWD)
```

4.1.3 Export data from Apache Atlas

You can use `export-atlas.py` script, that wraps Apache Atlas's Export API to export all data from Atlas. Example Usage:

```
pip install urlpath
python export-atlas.py --token $ACCESS_TOKEN \
--base-url https://aureliusdev.westeurope.cloudapp.azure.com/demo/atlas2/ \
--output out.zip
```

4.1.4 Import Backup to Atlas Instance

Apache Atlas exposes an Import API from where data is imported from a zip file. Admin user need rights are needed to use this api. This command will import a file response.zip in the current directory to a specified atlas instance.

```
curl -g -X POST -H 'Authorization: Bearer <Bearer-Token>' -H "Content-Type: multipart/
↪form-data" -H "Cache-Control: no-cache" -F data=@response.zip <apache-atlas-url>/
↪api/atlas/admin/import
```

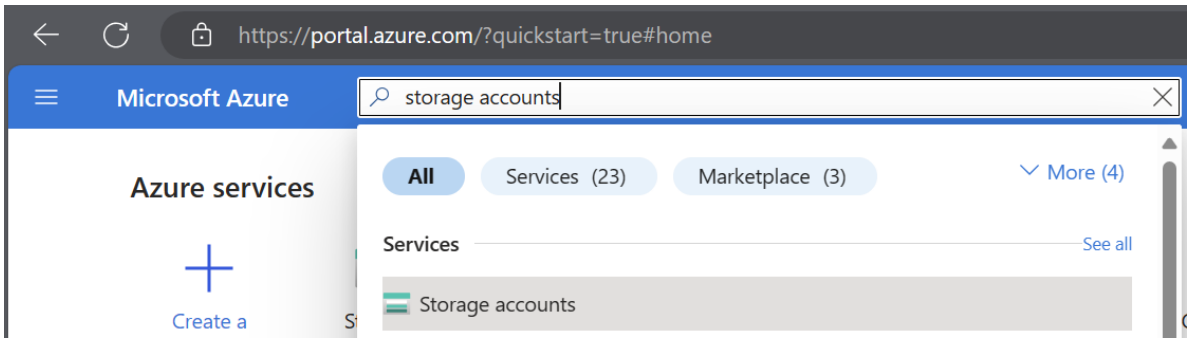
4.2 Elasticsearch backup

For Elasticsearch backup you can use Snapshot and restore API.

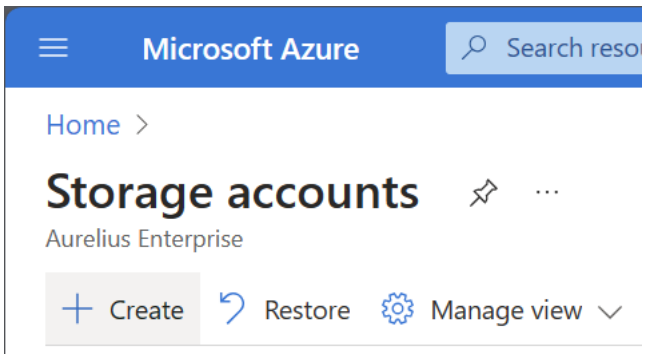
4.2.1 Create a snapshot repository

Create a storage account and a container in Azure

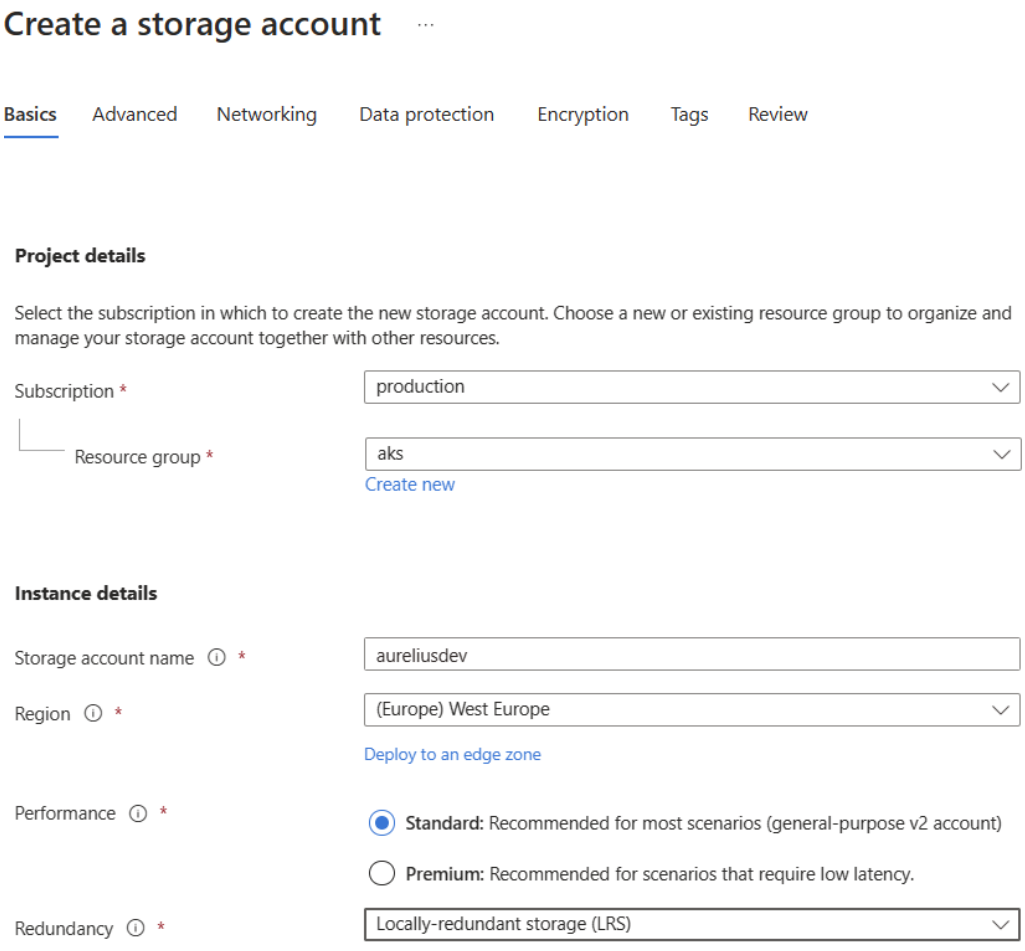
1. Go to https://portal.azure.com/
2. Go to storage accounts service

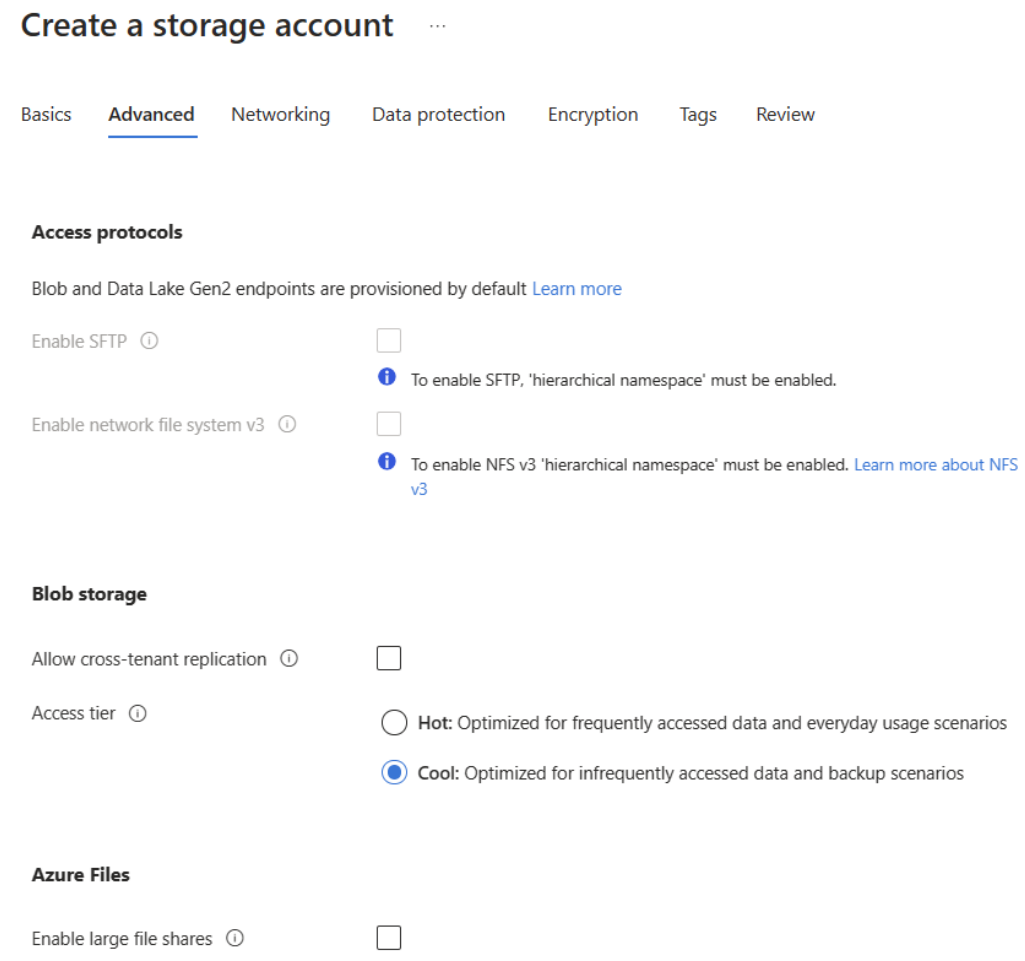


3. Create a new storage account

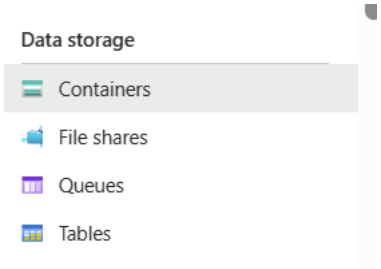


4. Set the account name. Optionally adjust the redundancy and access tier

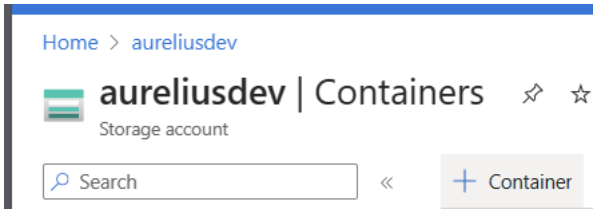

Create a storage account
Basics
Advanced
Networking
Data protection
Encryption
Tags
Review
Project details
Select the subscription in which to create the new storage account. Choose a new or existing resource group to organize and manage your storage account together with other resources.
Subscription *
production
Resource group *
aks
Create new
Instance details
Storage account name *
aureliusdev
Region *
(Europe) West Europe
Deploy to an edge zone
Performance *
Standard: Recommended for most scenarios (general-purpose v2 account)
Premium: Recommended for scenarios that require low latency.
Redundancy *
Locally-redundant storage (LRS)

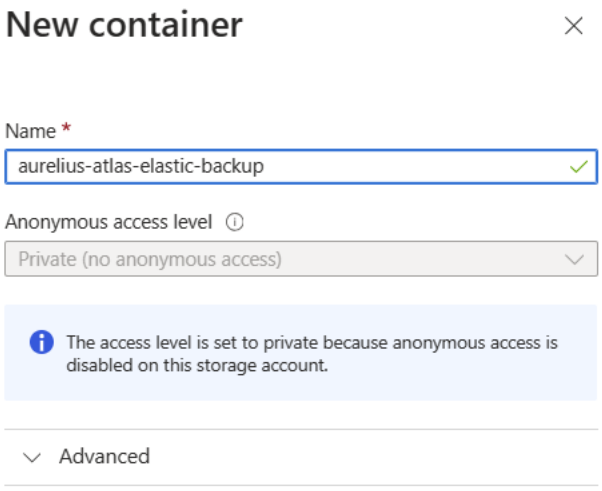


5. Review and create
6. Once the account is created, go to Containers tab

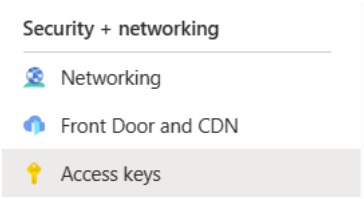


7. Create a new container





8. Go to Access keys tab

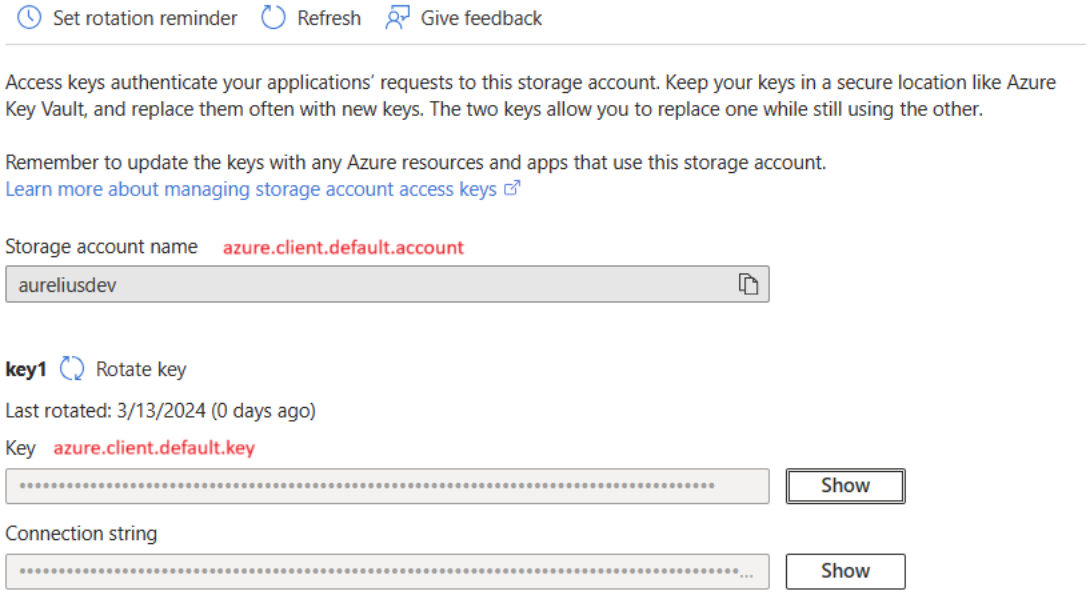


Register a repository

1. Access Elastic's search pod/image, for example:

```
kubectl -n demo exec -it pod/elastic-search-es-default-0 -- bash
```

2. Configure Elasticsearch's keystore with values from the Storage account's Access keys tab.



```
bin/elasticsearch-keystore add azure.client.default.account
bin/elasticsearch-keystore add azure.client.default.key
```

3. Optionally set a password for the keystore

```
bin/elasticsearch-keystore passwd
```

4. Reload secure settings

```
curl -X POST -u "elastic:$ELASTIC_PASSWORD" "https://aureliusdev.westeurope.
↪cloudapp.azure.com/demo/elastic/_nodes/reload_secure_settings?pretty" -H
↪'Content-Type: application/json' -d "
{
    \"secure_settings_password\": \"$ELASTIC_KEYSTORE_PASSWORD\"
}"
```

5. Create the repository

```
curl -X PUT -u "elastic:$ELASTIC_PASSWORD" "https://aureliusdev.westeurope.
↪cloudapp.azure.com/demo/elastic/_snapshot/demo_backup?pretty" -H 'Content-Type:␣
↪application/json' -d "
{
  \"type\": \"azure\",
  \"settings\": {
    \"container\": \"aurelius-atlas-elastic-backup\",
     \"base_path\": \"backups\",
     \"chunk_size\": \"32MB\",
    \"compress\": true
  }
}"
```

4.2.2 Create a snapshot

```
curl -X POST -u "elastic:$ELASTIC_PASSWORD" "https://aureliusdev.westeurope.cloudapp.
↪azure.com/demo/elastic/_snapshot/demo_backup/snapshot_2" -H 'Content-Type:␣
↪application/json' -d '
{
   "indices": ".ent-search-engine-documents-*"
}'
```

CHAPTER 5

Demo environment

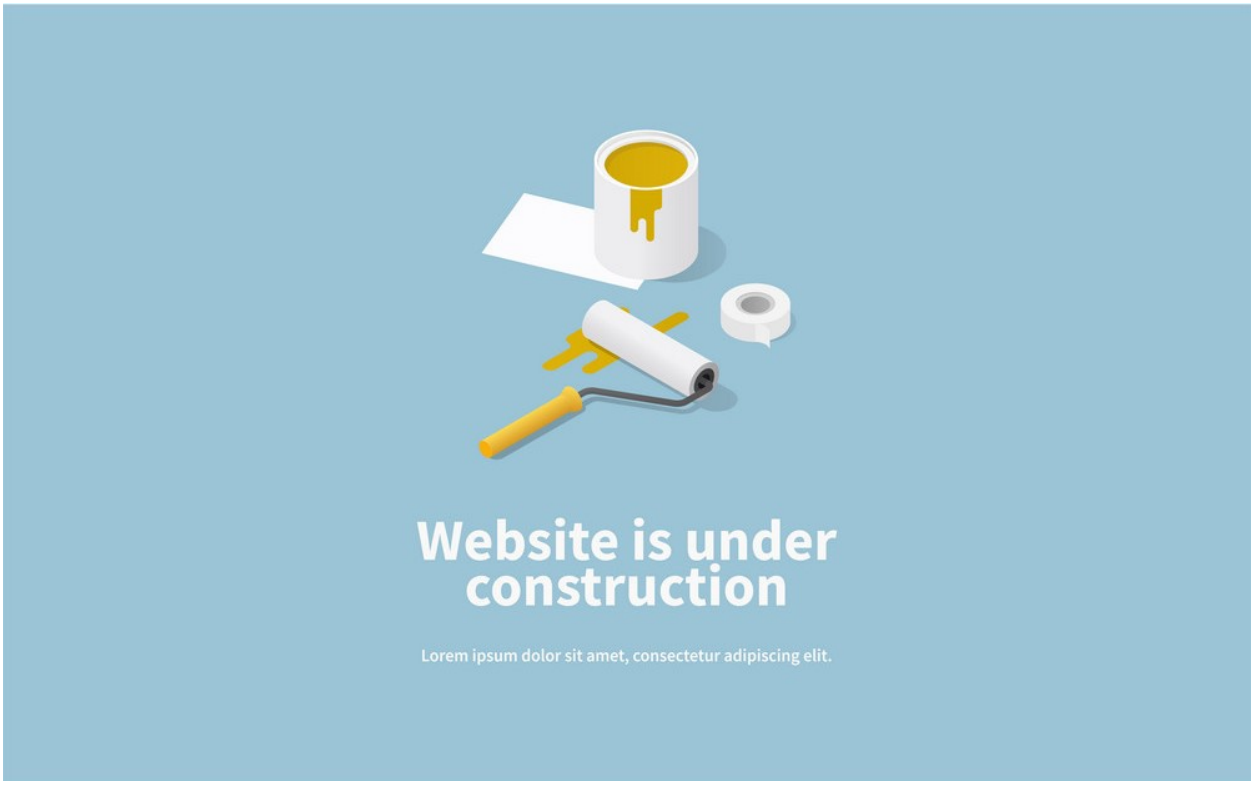
The demo environment is open for anyone who want to try out **Aurelius Atlas**.

- It can be accessed Clicking here
- If you don't know how to use it, you have an user guide, Click here to access
- Some use cases and how to use the front end can be found Click here to go to the business user guid

CHAPTER 6

Integrations

COMING SOON…..



CHAPTER 7

Libraries

7.1 Aurelius Atlas

7.1.1 M4I Atlas Core

API

This README provides documentation for the M4I Atlas Core `api` module, which is designed for interacting with the Apache Atlas API and retrieving authentication tokens from Keycloak.

Features

The API module contains a set of functions that facilitate communication with the Apache Atlas API. These functions provide a convenient and efficient way to interact with the Aurelius Atlas platform. The main features of the API module include:

- Functions for creating, retrieving, updating, and deleting Atlas entities
- Functions for managing entity relationships and classifications
- Support for bulk operations, such as bulk entity creation and deletion
- Error handling and response parsing for API interactions

How to use

To use any of the API functions, import them directly from the library:

```
from m4i_atlas_core import create_entities, create_glossary, ...
```

Submodules

The API module is divided into two submodules:

1. `atlas <#atlas>`__: This submodule contains functions for interacting with the Apache Atlas API, enabling you to create, read, update, and delete entities and their related metadata.
2. `auth <#auth>`__: This submodule is responsible for retrieving authentication tokens from Keycloak, which are required for accessing and utilizing the Apache Atlas API.

Atlas

The `atlas` submodule provides a collection of functions to interact with the Apache Atlas API. These functions enable you to create, retrieve, update, and delete various entities, types, and glossaries in Apache Atlas.

The API functions make extensive use of the data object model included with this library, which corresponds to the data object model for the Apache Atlas API. You can find the official Apache Atlas API documentation at this link.

The following sections include examples demonstrating how to use each API function.

create_entities

The `create_entities` function allows you to create or update multiple entities in Apache Atlas in bulk. It takes in a variable number of `Entity` objects and an optional dictionary of referred entities. It also accepts an optional access token for authentication purposes.

Here's an example of how to use the `create_entities` function:

```
from m4i_atlas_core import Entity, create_entities

entity1 = Entity(...)
entity2 = Entity(...)

mutations = await create_entities(entity1, entity2)

print(mutations)
```

This example creates the two given entities in Apache Atlas. The `create_entities` function returns an `EntityMutationResponse` object containing the details of the entities created or updated.

create_glossary

The `create_glossary` function allows you to create a new glossary in Apache Atlas. It takes in a `Glossary` object and an optional access token for authentication purposes.

Here's an example of how to use the `create_glossary` function:

```
from m4i_atlas_core import Glossary, create_glossary

glossary = Glossary(...)

created_glossary = await create_glossary(glossary)

print(created_glossary)
```

This example creates the given glossary in Apache Atlas. The `create_glossary` function returns a `Glossary` object containing the details of the created glossary.

create_glossary_category

The `create_glossary_category` function allows you to create a new glossary category in Apache Atlas. It takes in a `GlossaryCategory` object and an optional access token for authentication purposes.

Here's an example of how to use the `create_glossary_category` function:

```
from m4i_atlas_core import GlossaryCategory, create_glossary_category

category = GlossaryCategory(...)

created_category = await create_glossary_category(category)

print(created_category)
```

This example creates the given glossary category in Apache Atlas. The `create_glossary_category` function returns a `GlossaryCategory` object containing the details of the created category.

create_glossary_term

The `create_glossary_term` function allows you to create a new glossary term in Apache Atlas. It takes in a `GlossaryTerm` object and an optional access token for authentication purposes.

Here's an example of how to use the `create_glossary_term` function:

```
from m4i_atlas_core import GlossaryTerm, create_glossary_term

term = GlossaryTerm(...)

created_term = await create_glossary_term(term)

print(created_term)
```

This example creates the given glossary term in Apache Atlas. The `create_glossary_term` function returns a `GlossaryTerm` object containing the details of the created term.

create_type_defs

The `create_type_defs` function allows you to create multiple new type definitions in Apache Atlas in bulk. It takes in a `TypesDef` object and an optional access token for authentication purposes.

> **Note**: Only new definitions will be created, and any changes to the existing definitions will be discarded.

Here's an example of how to use the `create_type_defs` function:

```
from m4i_atlas_core import TypesDef, EntityDef, create_type_defs

entity_def = EntityDef(...)

types_def = TypesDef(
    entity_defs=[entity_def]
)

created_type_defs = await create_type_defs(types_def)

print(created_type_defs)
```

This example creates the given entity definition in Apache Atlas. The `create_type_defs` function returns a `TypesDef` object containing lists of type definitions that were successfully created.

delete_entity_hard

The `delete_entity_hard` function allows you to permanently delete one or more entities from Apache Atlas by their `guid`. This operation removes the entities from the database completely.

It takes in a list of `guid` strings and an optional access token for authentication purposes.

> **Note**: This API requires elevated user permissions.

Here's an example of how to use the `delete_entity_hard` function:

```
from m4i_atlas_core import delete_entity_hard

guids = ["1234-5678-90ab-cdef", "abcd-efgh-ijkl-mnop"]

mutations = await delete_entity_hard(guids)

print(mutations)
```

This example permanently deletes the entities with the given `guids` from Apache Atlas. The `delete_entity_hard` function returns an `EntityMutationResponse` object containing the details of the deleted entities.

delete_entity_soft

The `delete_entity_soft` function allows you to mark an entity as deleted in Apache Atlas without completely removing it from the database. The entity's status is set to `DELETED`. It takes in the `guid` of the entity and an optional access token for authentication purposes.

Here's an example of how to use the `delete_entity_soft` function:

```
from m4i_atlas_core import delete_entity_soft

guid = "1234-5678-90ab-cdef"

mutations = await delete_entity_soft(guid)

print(mutations)
```

This example marks the entity with the given `guid` as deleted in Apache Atlas. The `delete_entity_soft` function returns an `EntityMutationResponse` object containing the details of the deleted entity.

get_classification_def

The `get_classification_def` function allows you to retrieve a classification definition from Apache Atlas based on its type name. It takes in the `type_name` of the classification and an optional access token for authentication purposes.

> **Note**: This function is *cached*, meaning that repeated calls with the same parameters will return the cached result rather than making additional requests to the server.

Here's an example of how to use the `get_classification_def` function:

```
from m4i_atlas_core import get_classification_def

type_name = "example_classification"
```

```
classification_def = await get_classification_def(type_name)

print(classification_def)
```

This example retrieves the classification definition with the given `type_name` from Apache Atlas. The `get_classification_def` function returns a `ClassificationDef` object containing the details of the classification definition.

get_entities_by_attribute

The `get_entities_by_attribute` function allows you to retrieve entities from Apache Atlas based on a specified attribute search query. It takes in the `attribute_name`, `attribute_value`, and `type_name` as search parameters, and an optional access token for authentication purposes.

> **Note**: This function is *cached*, meaning that repeated calls with the same parameters will return the cached result rather than making additional requests to the server.

Keep in mind that this search only returns entity *headers*, which include the `guid` and `type_name` of the actual entity. You can use these headers to query the entities API for more information.

Here's an example of how to use the `get_entities_by_attribute` function:

```
from m4i_atlas_core import get_entities_by_attribute

attribute_name = "example_attribute"
attribute_value = "example_value"
type_name = "example_type"

search_result = await get_entities_by_attribute(attribute_name, attribute_value, type_
↪name)

print(search_result)
```

This example retrieves the entities with the given attribute and type from Apache Atlas. The `get_entities_by_attribute` function returns a `SearchResult` object containing the details of the entity headers that match the search query.

get_entities_by_type_name

The `get_entities_by_type_name` function allows you to search for all entities in Apache Atlas whose type matches the given `type_name`. It takes in the `type_name`, an optional `limit` and `offset` for pagination, and an optional access token for authentication purposes.

> **Note**: This function is *cached*, meaning that repeated calls with the same parameters will return the cached result rather than making additional requests to the server.

Keep in mind that this search only returns entity *headers*, which include the `guid` and `type_name` of the actual entity. You can use these headers to query the entities API for more information.

Here's an example of how to use the `get_entities_by_type_name` function:

```
from m4i_atlas_core import get_entities_by_type_name

type_name = "example_type"
```

```
entities = await get_entities_by_type_name(type_name)

print(entities)
```

This example retrieves all entities with the given type from Apache Atlas. The `get_entities_by_type_name` function returns a list of `EntityHeader` objects containing the details of the entity headers that match the search query.

get_entity_audit_events

The `get_entity_audit_events` function allows you to fetch all audit events for an entity in Apache Atlas based on its `guid`. It takes in the `entity_guid` and an optional access token for authentication purposes.

> **Note**: This function is *cached*, meaning that repeated calls with the same parameters will return the cached result rather than making additional requests to the server.

Here's an example of how to use the `get_entity_audit_events` function:

```
from m4i_atlas_core import get_entity_audit_events

entity_guid = "example_guid"

audit_events = await get_entity_audit_events(entity_guid)

print(audit_events)
```

This example fetches all audit events for the entity with the given `guid` from Apache Atlas. The `get_entity_audit_events` function returns a list of `EntityAuditEvent` objects containing the details of the audit events associated with the entity.

get_entity_by_guid

The `get_entity_by_guid` function allows you to fetch the complete definition of an entity in Apache Atlas based on its `guid`. It takes in the guid and an optional `entity_type`, which can be a string or an object of type `T`, where `T` is a subclass of `Entity`.

You can also provide optional parameters like `ignore_relationships` and `min_ext_info` to customize the results, as well as an optional access token for authentication purposes.

> **Note**: This function is *cached*, meaning that repeated calls with the same parameters will return the cached result rather than making additional requests to the server.

Here's an example of how to use the `get_entity_by_guid` function:

```
from m4i_atlas_core import Entity, get_entity_by_guid

guid = "example_guid"

entity = await get_entity_by_guid(guid, Entity)

print(entity)
```

This example fetches the complete definition of the entity with the given `guid` from Apache Atlas. The `get_entity_by_guid` function returns an `Entity` object containing the details of the entity. If the `entity_type` parameter is provided, the function will return an instance of that type.

get_glossary_by_guid

The `get_glossary_by_guid` function allows you to fetch a glossary in Apache Atlas based on its `guid`. It takes in the `guid` of the glossary and an optional access token for authentication purposes.

> **Note**: This function is *cached*, meaning that repeated calls with the same parameters will return the cached result rather than making additional requests to the server.

Here's an example of how to use the `get_glossary_by_guid` function:

```
from m4i_atlas_core import get_glossary_by_guid

guid = "example_glossary_guid"

glossary = await get_glossary_by_guid(guid)

print(glossary)
```

This example fetches the glossary with the given `guid` from Apache Atlas. The `get_glossary_by_guid` function returns a `Glossary` object containing the details of the glossary.

get_glossary_category_by_guid

The `get_glossary_category_by_guid` function allows you to fetch a glossary category in Apache Atlas based on its `guid`. It takes in the `guid` of the glossary category and an optional access token for authentication purposes.

> **Note**: This function is *cached*, meaning that repeated calls with the same parameters will return the cached result rather than making additional requests to the server.

Here's an example of how to use the `get_glossary_category_by_guid` function:

```
from m4i_atlas_core import get_glossary_category_by_guid

guid = "example_glossary_category_guid"

glossary_category = await get_glossary_category_by_guid(guid)

print(glossary_category)
```

This example fetches the glossary category with the given `guid` from Apache Atlas. The `get_glossary_category_by_guid` function returns a `GlossaryCategory` object containing the details of the glossary category.

get_glossary_term_by_guid

The `get_glossary_term_by_guid` function allows you to fetch a glossary term in Apache Atlas based on its `guid`. It takes in the `guid` of the glossary term and an optional access token for authentication purposes.

> **Note**: This function is *cached*, meaning that repeated calls with the same parameters will return the cached result rather than making additional requests to the server.

Here's an example of how to use the `get_glossary_term_by_guid` function:

```
from m4i_atlas_core import get_glossary_term_by_guid

```

```
guid = "example_glossary_term_guid"

glossary_term = await get_glossary_term_by_guid(guid)

print(glossary_term)
```

This example fetches the glossary term with the given `guid` from Apache Atlas. The `get_glossary_term_by_guid` function returns a `GlossaryTerm` object containing the details of the glossary term.

get_glossary

The `get_glossary` function allows you to fetch all glossaries in Apache Atlas with optional pagination and sorting. The function takes in an optional `limit`, `offset`, and sort `order`, as well as an optional access token for authentication purposes.

> **Note**: This function is *cached*, meaning that repeated calls with the same parameters will return the cached result rather than making additional requests to the server.

Here's an example of how to use the get_glossary function:

```
from m4i_atlas_core import get_glossary

limit = 10
offset = 0
sort = 'ASC'

glossaries = await get_glossary(limit=limit, offset=offset, sort=sort)

for glossary in glossaries:
    print(glossary)
```

This example fetches glossaries from Apache Atlas using the specified pagination and sorting options. The `get_glossary` function returns a list of `Glossary` objects containing the details of the glossaries.

get_lineage_by_guid

The `get_lineage_by_guid` function allows you to fetch the lineage of an entity in Apache Atlas given its `guid`.

It takes in the `guid` of the entity, the maximum number of hops to traverse the lineage graph using the `depth` parameter (default is 3), the `direction` parameter to specify whether to retrieve input lineage, output lineage or both (default is both), and an optional access token for authentication purposes.

> **Note**: This function is *cached*, meaning that repeated calls with the same parameters will return the cached result rather than making additional requests to the server.

Here's an example of how to use the `get_lineage_by_guid` function:

```
from m4i_atlas_core import LineageDirection, get_lineage_by_guid

guid = "12345"
depth = 3
direction = LineageDirection.BOTH

lineage_info = await get_lineage_by_guid(guid, depth=depth, direction=direction)
```

```
print(lineage_info)
```

This example fetches the lineage of the entity with the given `guid` from Apache Atlas. The `get_lineage_by_guid` function returns a `LineageInfo` object containing the details of the entity's lineage.

get_lineage_by_qualified_name

The `get_lineage_by_qualified_name` function allows you to fetch the lineage of an entity in Apache Atlas given its `qualified_name` and `type_name`.

It takes in the `qualified_name` and `type_name` of the entity, the maximum number of hops to traverse the lineage graph using the `depth` parameter (default is 3), the `direction` parameter to specify whether to retrieve input lineage, output lineage or both (default is both), and an optional access token for authentication purposes.

> **Note**: This function is *cached*, meaning that repeated calls with the same parameters will return the cached result rather than making additional requests to the server.

Here's an example of how to use the `get_lineage_by_qualified_name` function:

```
from m4i_atlas_core import LineageDirection, get_lineage_by_qualified_name

qualified_name = "example.qualified.name"
type_name = "example_type_name"
depth = 3
direction = LineageDirection.BOTH

lineage_info = await get_lineage_by_qualified_name(qualified_name, type_name,␣
↪depth=depth, direction=direction)

print(lineage_info)
```

This example fetches the lineage of the entity with the given `qualified_name` and `type_name` from Apache Atlas. The `get_lineage_by_qualified_name` function returns a `LineageInfo` object containing the details of the entity's lineage.

get_type_def

The `get_type_def` function allows you to retrieve an entity type definition from Apache Atlas based on its name. It takes in the `input_type` of the entity and an optional access token for authentication purposes.

> **Note**: This function is *cached*, meaning that repeated calls with the same parameters will return the cached result rather than making additional requests to the server.

Here's an example of how to use the `get_type_def` function:

```
from m4i_atlas_core import get_type_def

input_type = "example_entity_type"

entity_def = await get_type_def(input_type)

print(entity_def)
```

This example retrieves the entity type definition with the given `input_type` from Apache Atlas. The `get_type_def` function returns an `EntityDef` object containing the details of the entity type definition.

get_type_defs

The `get_type_defs` function allows you to retrieve all type definitions in Apache Atlas. It takes an optional access token for authentication purposes.

> **Note**: This function is *cached*, meaning that repeated calls with the same parameters will return the cached result rather than making additional requests to the server.

Here's an example of how to use the `get_type_defs` function:

```
from m4i_atlas_core import get_type_defs

type_defs = await get_type_defs()

print(type_defs)
```

This example retrieves all type definitions from Apache Atlas. The `get_type_defs` function returns a `TypesDef` object containing the details of the type definitions.

update_type_defs

The `update_type_defs` function allows you to bulk update all Apache Atlas type definitions. Existing definitions will be overwritten, but the function will not create any new type definitions.

It takes a types parameter, which is a `TypesDef` object containing the type definitions to be updated, and an optional access token for authentication purposes.

Here's an example of how to use the `update_type_defs` function:

```
from m4i_atlas_core import EntityDef, TypesDef, update_type_defs

entity_def = EntityDef(
    category="ENTITY",
    name="example_entity",
    description="An example entity definition"
)

types = TypesDef(entityDefs=[entity_def])

updated_type_defs = await update_type_defs(types)

print(updated_type_defs)
```

This example updates an existing entity definition with the given types parameter in Apache Atlas. The `update_type_defs` function returns a `TypesDef` object containing the details of the type definitions that were successfully updated.

Working with the cache

The library utilizes the `aiocache <https://aiocache.aio-libs.org/en/latest/>`__ library to cache some API function results. Caching can help reduce server load and improve performance by reusing the results from previous API calls with the same parameters.

When you call a cached API function, the cache is automatically checked for the result. If the result is present in the cache, it is returned instead of making a new API call.

```
from m4i_atlas_core import get_entity_by_guid

# Call the function once, making an API call
await get_entity_by_guid("12345")

# Call the function again, returning the result from the cache
await get_entity_by_guid("12345")

# Bypass the cache and make a direct API call
await get_entity_by_guid("12345", cache_read=False)
```

You can interact with the cache for any API function using the `cache` property. The following examples demonstrate how to access and manipulate the cache for the `get_entity_by_guid` function:

```
from m4i_atlas_core import get_entity_by_guid

# Access the cache for the get_entity_by_guid function
cache = get_entity_by_guid.cache

# Delete an item from the cache
await cache.delete("12345")

# Clear the entire cache
await cache.clear()
```

These cache management options enable you to control and optimize the caching behavior of your application, tailoring it to your specific use case.

Auth

The `auth` submodule provides functionality for retrieving authentication tokens from Keycloak, which are required for accessing the Apache Atlas API.

> **Note**: This module is specifically designed for use with Keycloak authentication. When Apache Atlas is configured with basic authentication, obtaining access tokens is not required. Instead, set a username and password in the `ConfigStore` for authentication.

Usage

The `get_keycloak_token` function in the Auth submodule is responsible for retrieving an access token from a Keycloak instance.

To use the `get_keycloak_token` function, first import it:

```
from m4i_atlas_core import get_keycloak_token
```

Next, call the function to retrieve an access token. You can provide your own Keycloak instance and credentials or rely on the pre-configured parameters from the `ConfigStore` as described in the *configuration* section. If you need to use multi-factor authentication, provide the one-time access token (TOTP) as well.

```
# Example: Using pre-configured parameters
access_token = get_keycloak_token()

# Example: Using custom Keycloak instance and credentials
access_token = get_keycloak_token(keycloak=my_keycloak_instance, credentials=("my_
↪username", "my_password"))
```

```
# Example: Using multi-factor authentication (TOTP)
access_token = get_keycloak_token(totp="123456")
```

The `access_token` can then be used to authenticate requests to the Apache Atlas API.

> **Note**: Tokens obtained from Keycloak have a limited lifespan. Once a token expires, you will need to obtain a new access token to continue making authenticated requests.

Configuration

The `get_keycloak_token` function relies on the following values from the `ConfigStore`:

Key	Description	Required
`keycloak.server.url`	The url of the Keycloak server. In case of a local connection, this includes the hostname and the port. E.g. `http://localhost:8180/auth`. In case of an external connection, provide a fully qualified domain name. E.g. `https://www.models4insight.com/auth`.	`True`
`keycloak.client.id`	The name of the Keycloak client.	`True`
`keycloak.realm.name`	The name of the Keycloak realm.	`True`
`keycloak.client.secret.key`	The public RS256 key associated with the Keycloak realm.	`True`
`keycloak.credentials.username`	The username of the Keycloak user.	`False`
`keycloak.credentials.password`	The password of the Keycloak user.	`False`

Please find more detailed documentation about ‘‘ConfigStore‘ here. <./config>‘__

ConfigStore

`ConfigStore` is a powerful, singleton-based configuration store providing an easy-to-use interface to store, retrieve, and manage configuration settings.

* *Error Handling*

Features

- Singleton-based implementation ensures a single source of truth for your configuration settings.
- Ability to load your configuration settings on application start.
- Easy storage and retrieval of configuration settings using simple get and set methods.
- Support for default values and required settings.
- Bulk retrieval and storage of settings using get_many and set_many methods.

How to use

Please find examples of how to use the `ConfigStore` below.

Initializing the ConfigStore

To start using the `ConfigStore`, first import the necessary components and initialize the singleton instance:

```
from config import config
from credentials import credentials

from config_store import ConfigStore

store = ConfigStore.get_instance()
store.load({
   **config,
   **credentials
})
```

In this example, the `config.py` and `credentials.py` files are imported to obtain the necessary configuration parameters and credentials. The `ConfigStore` is then initialized using the `get_instance()` method, and the configuration and credential dictionaries are merged and loaded into the `ConfigStore` using the `load()` method.

Note: It is recommended to initialize the ConfigStore once when the application starts.

Storing Configuration Settings

To store a configuration setting, use the `set` method:

```
store.set("key", "value")
```

To store multiple configuration settings at once, use the set_many method:

```
store.set_many(key1="value1", key2="value2", key3="value3")
```

Retrieving Configuration Settings

To retrieve a configuration setting, use the `get` method. If the key is not present in the `ConfigStore`, it returns `None` by default.

```
value = store.get("key")
```

You can also provide a default value if the key is not found:

```
value = store.get("key", default="default_value")
```

If a key is required and not found in the `ConfigStore`, you can raise a `MissingRequiredConfigException` by setting the required parameter to True:

```
value = store.get("key", required=True)
```

To retrieve multiple configuration settings at once, use the `get_many` method:

```
key1, key2, key3 = store.get_many("key1", "key2", "key3")
```

You can also provide default values and required flags for the keys:

```
defaults = {"key1": "default_value1", "key2": "default_value2"}
required = {"key1": True, "key2": False}

key1, key2, key3 = store.get_many("key1", "key2", "key3", defaults=defaults,␣
↪required=required)
```

If all keys are required, you can use the `all_required` parameter as a shorthand:

```
key1, key2, key3 = store.get_many("key1", "key2", "key3", all_required=True)
```

Resetting the ConfigStore

To reset the ConfigStore and remove all stored configuration settings, use the reset method:

```
store.reset()
```

This will clear the ConfigStore and reset it to an empty state.

Error Handling

The ConfigStore raises a `MissingRequiredConfigException` when a required key is not found and no default value has been provided. This exception can be caught and handled as needed in your application:

```
from m4i_atlas_core import MissingRequiredConfigException

try:
    value = store.get("key", required=True)
except MissingRequiredConfigException as ex:
    # Handle the case of a missing configuration
```

Data Object Model

This section provides an overview of how to use the data object model provided in the library. The data objects are designed to represent various types of entities, attributes, classifications, and other components in Aurelius Atlas. They are used extensively when interacting with the Atlas API.

Features

The entities module provides a collection of data objects designed to represent different types of entities, attributes, classifications, and other components in Aurelius Atlas. The main features of the entities module include:

- Data objects related to the Apache Atlas API
- Data objects related to the Aurelius Atlas metamodel
- Convenience methods for converting data objects to and from JSON format
- Marshmallow schemas for data validation, serialization, and deserialization

How to use

To use the data objects from the library in your code, you can easily import them. For example, if you want to work with the `Entity` data object, you can import it as follows:

```
from m4i_atlas_core import Entity
```

Once you have imported the desired data object, you can create instances, access their properties, and manipulate them as needed.

Submodules

The `entities` module is organized into two main submodules:

- `core`: This submodule includes data objects that correspond to the Apache Atlas API. These objects are used for representing entities, classifications, relationships, and other components as defined in Apache Atlas.

- `data_dictionary`: This submodule contains data objects that are specific to the Aurelius Atlas metamodel. These objects extend or customize the core data objects to better suit the requirements of the Aurelius Atlas platform.

Serialization and deserialization

Each data object is a `` `dataclass <https://docs.python.org/3/library/dataclasses.html>`__ `` and is designed to be easily serialized and deserialized using the `` `dataclasses_json <https://lidatong.github.io/dataclasses-json/>`__ `` library. This allows for convenient conversion between JSON and the corresponding data object instances.

The `dataclasses_json` library provides additional features such as `camelCase` letter conversion and other customizations.

Below are some examples of how to use a data object, such as `BusinessDataDomain`, to convert between its instance and JSON representation.

From JSON to Instance

You can convert JSON data to an `Entity` instance using the `from_json()` method. Suppose you have the following JSON representation of a data domain:

```
{
  "attributes": {
    "key": "value",
    "name": "example",
    "qualifiedName": "data-domain--example"
  },
  "guid": "12345",
  "typeName": "m4i_data_domain"
}
```

The example below demonstrates how to create a `BusinessDataDomain` instance from the given JSON data:

```
from m4i_atlas_core import BusinessDataDomain

json_data = '''JSON string here'''
domain_instance = BusinessDataDomain.from_json(json_data)
```

Unmapped attributes

In the given example, the `key` attribute is not explicitly defined as part of the schema for `BusinessDataDomain`. In such cases, the attributes field of the resulting instance will include an `unmapped_attributes` field. This field offers flexibility when working with entities containing additional or custom attributes not specified in the predefined data model. The `unmapped_attributes` field acts as a catch-all for these attributes, ensuring they are preserved during the conversion process between JSON and the `Entity` instance.

To access an unmapped attribute, you can use the following code:

```
value = domain_instance.attributes.unmapped_attributes["key"]
```

When converting any `Entity` instance back to JSON, the unmapped attributes will be included as part of the `attributes` field once again.

From Instance to JSON

To convert an `Entity` instance back to its JSON representation, use the `to_json()` method. The example below shows how to convert the `BusinessDataDomain` instance we created previously back to its JSON representation:

```
json_data = domain_instance.to_json()
```

This will return a JSON string that represents the data domain instance, including any unmapped attributes.

Marshmallow Schema

Each data object in the library is equipped with a built-in Marshmallow schema. These schemas are valuable tools for validating, serializing, and deserializing complex data structures. By utilizing Marshmallow schemas, you can ensure that the data being passed to or returned from the API adheres to the correct structure and data types.

To access the Marshmallow schema for any data object, use the `schema()` method:

```
from m4i_atlas core import Entity

schema = Entity.schema()
```

Data Validation

Marshmallow schemas associated with the data objects in this library can be employed to perform data validation. The following example demonstrates how to use a Marshmallow schema to validate JSON input data:

```
from m4i_atlas_core import Entity

# Load the schema for the Entity data object
entity_schema = Entity.schema()

# Validate input data
input_data = {
    "guid": "123",
    "created_by": "user",
    "custom_attributes": {"key": "value"},
}

errors = entity_schema.validate(input_data)

if errors:
    print(f"Validation errors: {errors}")
else:
    print("Data is valid")
```

In this example, the `Entity` data object from the library is used to validate the `input_data` JSON using its associated Marshmallow schema. If the data is valid, the `validate` method will not return any errors, and the "Data is valid" message will be displayed. If the data is invalid, a dictionary containing the validation errors will be returned.

This approach can be applied to other data objects in the library for validating JSON input data using their respective Marshmallow schemas. To read more about data validation with Marshmallow, refer to the official documentation.

Bulk Serialization and Deserialization

Marshmallow schemas can be utilized for bulk serialization and deserialization of complex data structures. This is particularly useful when working with lists of data objects.

To serialize a list of data objects into a JSON format, you can use the dump method with the `many=True` option:

```
from m4i_atlas_core import Entity

# Sample list of Entity data objects
entities = [
    Entity(guid="1", created_by="user1", custom_attributes={"key1": "value1"}),
    Entity(guid="2", created_by="user2", custom_attributes={"key2": "value2"}),
]

# Load the schema for the Entity data object
entity_schema = Entity.schema()

# Serialize the list of entities
serialized_data = entity_schema.dump(entities, many=True)

print("Serialized data:", serialized_data)
```

To deserialize a JSON list of data objects, you can use the load method with the `many=True` option:

```
from m4i_atlas_core import Entity

# Sample JSON list of entity data
json_data = [
    {"guid": "1", "created_by": "user1", "custom_attributes": {"key1": "value1"}},
    {"guid": "2", "created_by": "user2", "custom_attributes": {"key2": "value2"}},
]

# Load the schema for the Entity data object
entity_schema = Entity.schema()

# Deserialize the JSON list of entities
deserialized_data = entity_schema.load(json_data, many=True)

print("Deserialized data:", deserialized_data)
```

In both examples, the `many=True` option is specified to indicate that the data being processed is a list. You can apply the same approach with other data objects in the library to perform bulk serialization and deserialization using their corresponding Marshmallow schemas.

Welcome to the M4I Atlas Core library!

This library is designed to streamline your interactions with Aurelius Atlas, providing a comprehensive data object model for all entities related to Aurelius Atlas and a set of functions to facilitate communication with the Aurelius Atlas API.

With this library, you can easily create, retrieve, and manage Atlas entities, enabling a seamless integration with the Aurelius Atlas Data Governance solution.

In this `README`, you will find detailed instructions on how to install, configure, and use the M4I Atlas Core library to simplify your work with the Aurelius Atlas platform.

Features

The M4I Atlas Core library offers a comprehensive set of features designed to simplify your interactions with the Aurelius Atlas platform. The main features of the library include:

- A rich data object model for all entities related to Aurelius Atlas
- A set of API functions to facilitate communication with the Apache Atlas API
- A centralized configuration store for managing settings and credentials

Installation

Below are instructions on various ways to install this project.

Using the Dev Container

This project includes a Visual Studio Code development container to simplify the setup process and provide a consistent development environment. You can use the dev container with either Visual Studio Code locally or with GitHub Codespaces.

Using the Dev Container with Visual Studio Code

Note: The following instructions assume that you have already installed Docker and Visual Studio Code.

1. Install the Remote Development extension pack in Visual Studio Code.
2. Open the project folder in Visual Studio Code.
3. Press `F1` to open the command palette, and then type "Remote-Containers: Open Folder in Container..." and select it from the list. Alternatively, you can click on the green icon in the bottom-left corner of the VS Code window and select "Reopen in Container" from the popup menu.

4. VS Code will automatically build the container and connect to it. This might take some time for the first run as it downloads the required Docker images and installs extensions.
5. Once connected, you'll see "Dev Container: M4I Atlas Core Dev Container" in the bottom-left corner of the VS Code window, indicating that you are now working inside the container.
6. You're all set! You can now develop, build, and test the project using the provided development environment.

Using the Dev Container with GitHub Codespaces

Note: GitHub Codespaces is a paid service. At the time of writing, it offers 60 hours of development time for free every month. Use with care.

1. Ensure that you have access to GitHub Codespaces.
2. Navigate to the GitHub repository for the project.
3. Click the "Code" button and then select "Open with Codespaces" from the dropdown menu.
4. Click on the "+ New codespace" button to create a new Codespace for the project.
5. GitHub Codespaces will automatically build the container and connect to it. This might take some time for the first run as it downloads the required Docker images and installs extensions.
6. Once connected, you'll see "Dev Container: M4I Atlas Core Dev Container" in the bottom-left corner of the VS Code window, indicating that you are now working inside the container.
7. You're all set! You can now develop, build, and test the project using the provided development environment.

Local installation

If you prefer not to use the dev container, you'll need to manually set up your development environment. Please follow the instructions below:

Please ensure your `Python` environment is on version `3.9`. Some dependencies do not work with any previous versions of `Python`.

To install `m4i-atlas-core` and all required dependencies to your active `Python` environment, please run the following command from the project root folder:

```
pip install -e . --user
```

To install the project including development dependencies, please run the following command:

```
pip install -e .[dev] --user
```

How to use

This section provides an overview of how to use the M4I Atlas Core library, including configuration options and example scripts to help you get started.

Submodules

The M4I Atlas Core library consists of several submodules to help you efficiently interact with the Aurelius Atlas platform. Each submodule serves a specific purpose and contains related functionality. Below is a brief description of each submodule:

- *api*: This submodule provides a set of functions that facilitate communication with the Apache Atlas API. It includes functions for creating, retrieving, updating, and deleting entities, as well as handling relationships, classifications, and other aspects of the Aurelius Atlas platform.
- *config*: This submodule includes the `ConfigStore` class, which is responsible for managing configuration settings for the library. It allows you to store, access, and update the configuration settings required to interact with the Atlas API.
- *entities*: This submodule contains the data objects related to the Apache Atlas API and the Aurelius Atlas metamodel.

Authentication

All Aurelius Atlas API endpoints are protected through Keycloak, which requires a valid authentication token for every request. The `api <./api>‘__ module includes functions for retrieving an authentication token from Keycloak. When using API functions, you should pass the authentication token through the `access_token` parameter.

Here's an example of how to authenticate an API request:

```
from m4i_atlas_core import get_entity_by_guid, get_keycloak_token

access_token = get_keycloak_token()

entity = await get_entity_by_guid("1234", access_token=access_token)
```

Refer to the *Configuration* section for details on setting up the required parameters for Keycloak authentication.

Configuration

Before you begin using any functions from the library, you will need to configure certain parameters and credentials for Atlas.

In the scripts directory, make a copy of `config.sample.py` and `credentials.sample.py` and rename the files to `config.py` and `credentials.py`, respectively. Set the configuration parameters and credentials for Atlas as needed.

> **Note**: When using the Dev Container, the sample files are copied for you automatically. However, you will still have to set the configuration parameters yourself.

Name	Required	Description
`atlas.server.url`	True	The base url for the Apache Atlas API. E.g. `https://www.aurelius-atlas.com/api/atlas`.

All configuration parameters should to be loaded into the `ConfigStore` on application startup. *Find more detailed documentation about the ‘‘ConfigStore‘* here. <./config>‘__

Keyloak authentication

When using the default Keycloak authentication, the following additional configuration parameters should be provided:

Name	Required	Description
`keycloak.server.url`	True	The url of the Keycloak server. E.g. `https://www.aurelius-atlas.com/auth`.
`keycloak.client.id`	True	The name of the Keycloak client. The default client id is `m4i_atlas`.
`keycloak.realm.name`	True	The name of the Keycloak realm. The default realm name is `m4i`.
`keycloak.client.secret.key`	True	The public RS256 key associated with the Keycloak realm.
`keycloak.credentials.username`	False	The username of the Keycloak user. The built-in username is `atlas`.
`keycloak.credentials.password`	False	The password of the Keycloak user.

Note: Keycloak credentials for built-in Aurelius Atlas users are automatically generated upon deployment and are available from the deployment log.

Atlas authentication

When Keycloak authentication is disabled, the default Apache Atlas user management system authenticates all requests. In this case, set the following additional configuration parameters:

Name	Required	Description
`atlas.credentials.username`	True	Your username for Apache Atlas. The built-in username is `atlas`.
`atlas.credentials.password`	True	Your password for Apache Atlas.

Example Scripts

The library includes example scripts to demonstrate how to interact with the Atlas API using the provided data object models and functions. These scripts can be found in the scripts directory of the project. Below is a brief overview of some example scripts:

- `load_type_defs.py`: This script loads the type definitions into Atlas. The main function in `load_type_defs.py` can be adjusted to determine which set of type definitions to load. Please note that if a subset of the set already exists, the loading of the type definitions will fail.

Testing

This project uses `pytest` as its unit testing framework. To run the unit tests, please install `pytest` and then execute the `pytest` command from the project root folder.

Unit tests are grouped per module. Unit test modules are located in the same folder as their respective covered modules. They can be recognized by the `test__` module name prefix, followed by the name of the covered module.

7.1.2 Data Quality

Is used to determine the quality of various entities already loaded into DMP's governance tool - Apache Atlas. It verifies data loaded against various m4i types (like m4i_data_domain, m4i_data_entity) on quality measures like completeness, uniqueness etc.

There are two main categories of Data that is generated for each m4i Type entity.

- Attributes related data consists of details about entity attributes where certain quality metrics can be applied like
  - completeness – whether we have a value for an attribute
  - uniqueness – whether values are unique for different entities
- Relationships related data consists of details about entity relationships where certain quality metrics can be applied like
  - completeness – whether we have correct relationships between two entities.

These rules are inherited from *m4i_data_management* repository.

Configuring Rules

An important aspect of Data Quality is the rules that are applied to each entity. There are separate rules for attributes and relationships. However, the structure is same and follows as below.

id: id

expressionVersion: version of expression

expression: expression to evaluate *completeness('name')*

qualifiedName: unique name for the rule example:*m4i_data_domain–name*

qualityDimension: Rule Category - explained below

ruleDescription: Description of the rule ex:*name is not None and is not empty*

active: 0 | 1

type: attribute | relationship

Rule Category	Rule Description
completeness	degree to which data is not null
accuracy	degree to which a column conforms to a standard
validity	degree to which the data comply with a predefined structure
uniqueness	degree to which the data has a unique value
timeliness	the data should be up to date

Example

id: 1

expressionVersion: 1

expression: completeness('name')

qualifiedName: m4i_data_domain–name

qualityDimension: completeness

ruleDescription: name is not None and is not empty

active: 1

type: attribute

Rules are maintained in *rules* directory of the package and can be found for each m4i type.

Running the code

We can execute *run.py* file. This will generates 6 files in output folder of the package. Three each for attributes and relationships. In addition, generated data is pushed to Elasticsearch indexes. We can configure pre-fix of indexes by updating *elastic_index_prefix* for both attributes and relationships related data.

- Summary – gives a summary of the data quality results.
- Complaint Data – gives information about complaints.
- Non-complaint Data – gives information about non-complaints.

Dependency

To Run this package, we need to have below packages installed * *m4i_atlas_core* – communicates with Apache Atlas * *vox-data-management* – communicates for Quality metric already defined * *elasticsearch* – communicates with ElasticSearch

Installation

Please ensure your *Python* environment is set on version *3.7*. Some dependencies do not work with any later versions of *Python*. Basically, this is a requirement for underlying package *m4i_data_management*

To install *m4i-atlas-core* and all required dependencies to your active *Python* environment. Activate it using:

source <venv_name>binactivate or create new *python3.7 -m venv <venv_name>*

Configurations and Credentials

Please make a copy of *config.sample.py* and *credentials.sample.py* and rename the files to *config.py* and *credentials.py* respectively. Please set the configuration parameters and credentials for *atlas* and *elastic* as below.

credentials.py Should contain two dictionaries viz *credential_atlas* and *credential_elastic*

Name	Description
credential_atlas[atlas.credentials.username]	The Username to be used to access the Atlas Instance.
credential_atlas[atlas.credentials.password]	The Password to be used to access the Atlas Instance must correspond to the Username given.
credential_elastic[elastic_cloud_id]	Service URL for Elastic.
credential_elastic[elastic_cloud_username]	The Username to be used to access the Elastic Instance.
credential_elastic[elastic_cloud_password]	The Password to be used to access the Elastic Instance must correspond to the Username given.

config.py Should contain two dictionaries viz *config_elastic* and *config_atlas*

Name	Description
config_elastic[elastic_index_prefix]	Define prefix for the elastic Index where data will be pushed to
config_atlas[atlas.server.url]	The Server URL that Atlas runs on, with */api/atlas* post fix.
config_atlas[atlas.credentials.token]	Add Keycloak access token

Execution

1. Create the Python Environment. How to do this can be found in this file under *Installation*
2. Fill in the Configurations and Credentials as indicated in this file under *Configurations and Credentials*

3. Run *scriptsrun.py* to create 6 files in output folder, 3 each for Attributes and Relationships. Same data is also pushed to Elastic.

1. creates/updates an index for attributes as '<prefix>'_quality_attr_[summary | complaint | non_complaint]
2. creates/updates an index for relationships as '<prefix>'_quality_rels_[summary | complaint | non_complaint]

7.1.3 M4I Data Management

This library contains all core functionality around data management for Models4Insight.

Installation

Please ensure your *Python* environment is on version *3.7*. Some dependencies do not work with any later versions of *Python*.

To install *m4i-data-management* and all required dependencies to your active *Python* environment, please run the following command from the project root folder:

```
1)Set up a virtual environment: Use this command in the root folder,

virtualenv --python "C:\\Python37\\python.exe" venv.


2) Then activate the virtual enviroment with this command:

.\env\Scripts\activate


3) Install the library

pip install -e .


To install `m4i-data-management` including development dependencies, please run the␣
↪following command instead:

pip install -e .[dev]

Install m4i_data_management:
You can clone m4i_data_management from this link https://gitlab.com/m4i/m4i_data_
↪management
```

Please make a copy of *config.sample.py* and *credentials.sample.py* and rename the files to *config.py* and *credentials.py* respectively.

The *config.py* and *credentials.py* files should be located in the root folder of the project, or otherwise on the *PYTHON_PATH*.

Please remember to set the configuration parameters you want to use.

Testing

This project uses *pytest* as its unit testing framework. To run the unit tests, please install *pytest* and then execute the *pytest* command from the project root folder.

Unit tests are grouped per module. Unit test modules are located in the same folder as their respective covered modules. They can be recognized by the *test__* module name prefix, followed by the name of the covered module.

Contacts

Name | Role | Email |
——————— | ——————- | ——————————— |
Thijs Franck | Lead developer | thijs.franck@aureliusenterprise.com |

7.1.4 m4i_data_dictionary_io

This library contains all core functionality for reading Data Dictionary excels and pushing the defined entities in bulk by type to atlas. Data Dictionary is expected to be in the same format as the template Data Dictionary.

Installation

Please ensure your *Python* environment is on version *3.9*. Some dependencies do not work with any previous versions of *Python*.

To install *m4i-data-dictionary-io* and all required dependencies to your active *Python* environment, please run the following command from the project root folder:

pip install -e .

Configurations and Credentials

In the *scripts* directory. Please make a copy of *config.sample.py* and *credentials.sample.py* and rename the files to *config.py* and *credentials.py* respectively. Please set the configuration parameters and credentials for *atlas*.

Server name	Description
atlas.server.url	The Server Url that Atlas runs on, with '/api/atlas' post fix.
atlas.credentials.username	The Username to be used to access the Atlas Instance.
atlas.credentials.password	The Password to be used to access the Atlas Instance must correspond to the Username given.

Execution

1. Create the Python Environment. How to do this can be found in this file under *Installation*
2. Fill in the Configurations and Credentials as indicated in this file under *Configurations and Credentials*
3. Run *main.py* in the terminal to load the definitions.

Testing

This project uses *pytest* as its unit testing framework. To run the unit tests, please install *pytest* and then execute the *pytest* command from the project root folder.

Unit tests are grouped per module. Unit test modules are located in the same folder as their respective covered modules. They can be recognized by the *test__* module name prefix, followed by the name of the covered module.

library	git-lab/github?	purpose	remarks
m4i_governance_data_quality	gitlab	data governance quality checks	merge or branches required
m4i_atlas_core	github		core entities for apache atlas
m4i data management	gitlab	writing and reading from kafka and elastic	many dependencies like confluent kafka and elastic , which are not always required stale branch
linage_restAPI	gitlab and github	backend for publishing data into atlas in a simplified way	in gitlab several unmerged branches
m4i_data_dictionary_io	gitlab	importing Data Dictionary excels	main branch is rc_1.0.0. should be changed to main
Gov UI Backend	gitlab		backend for providing data for the governance dashboard in old UI; main branch is rc_1.0.0. should be changed to main
atlas-m4i-connector	gitlab		integration m4i with atlas; merge required

7.2 Models4Insight

library	git-lab/github?	purpose	remarks
Models4Insight	gitlab and github	Frontend	gitlab version more up to date then github version; 54 branches! Cleanup required?
m4i-keycloak-bulma	gitlab	keycloak	templates for M4I
m4i_analytics_extension	gitlab		extensions to m4i_analytics
analytics library	gitlab and github	functionality to interact with m4i	lot of stuff which is no longer relevant… requires thorough check whether the APIs are still all ok.
RestApi2	gitlab	backend for M4I	
RestUser	gitlab	keycloak integration backend for M4I	
Data2model backend			
model comparison backend			
Consistency check backend			

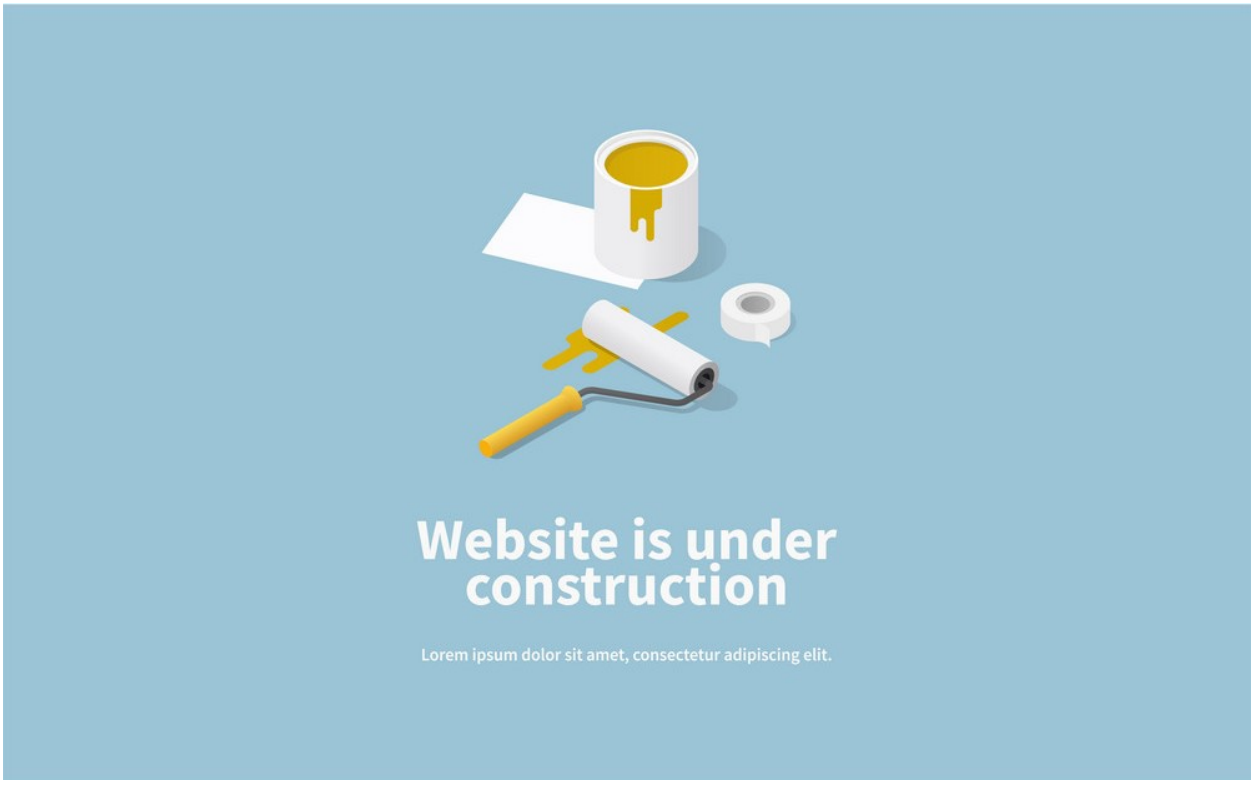
CHAPTER 8

Support / Maintenance

8.1 FAQS

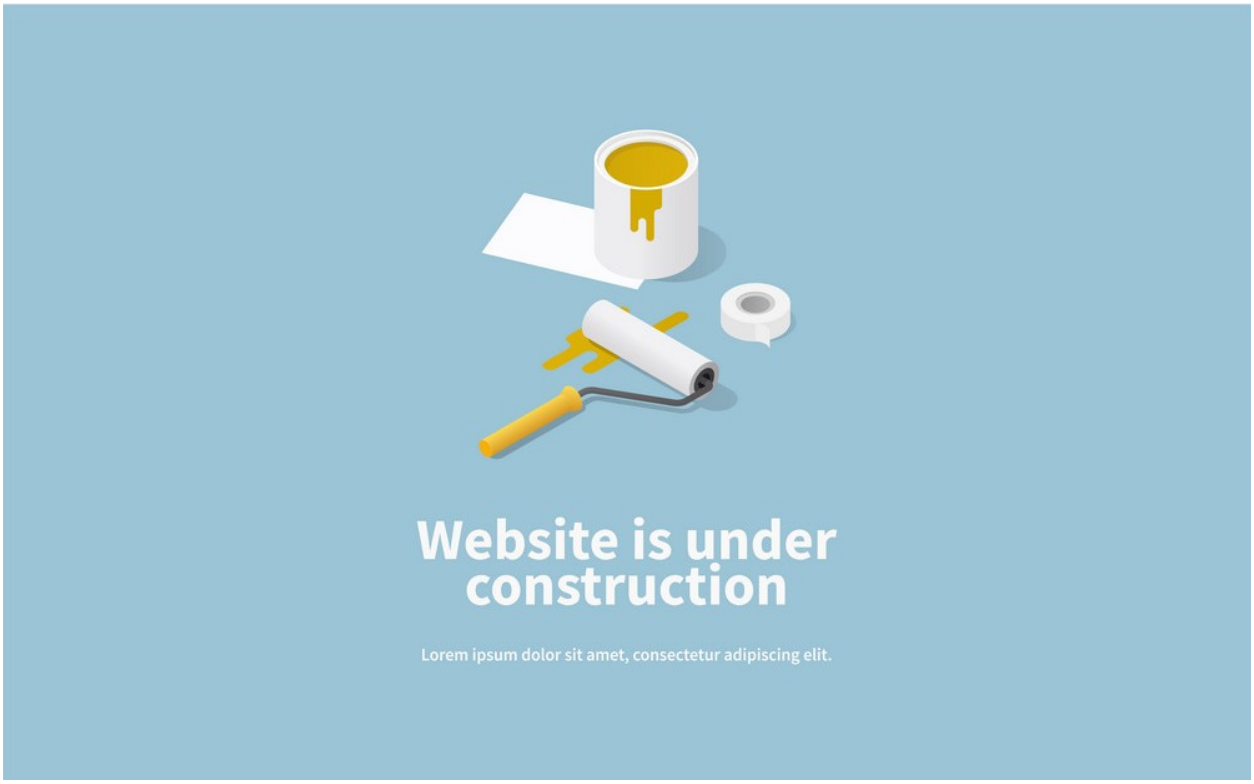
8.1.1 General

COMING SOON…..



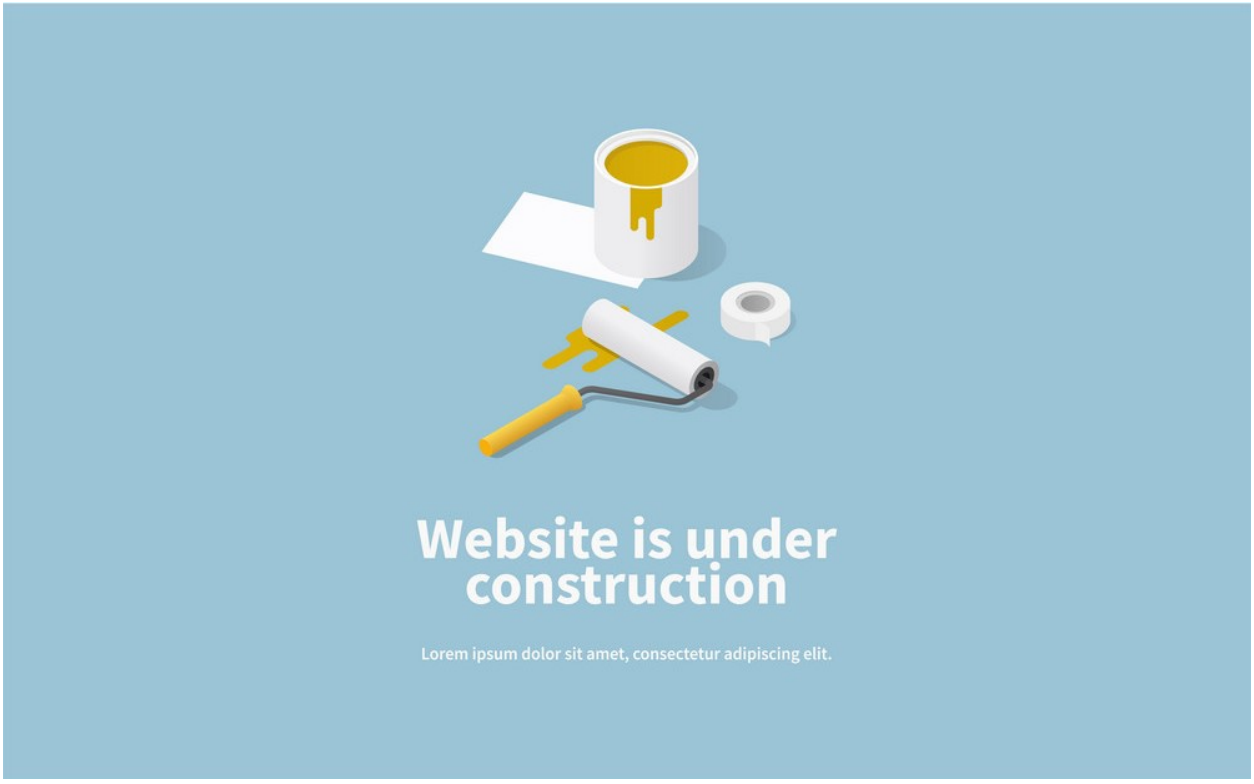
8.1.2 Integrations

COMING SOON…..



8.1.3 Demo environment

COMING SOON…..



8.1.4 Troubleshooting deployment

Connection is not safe

After many deployment attempts, it can happen that the reflector pod is not restarted automatically.

1. Check if there is a secret called *letsencrypt-secret-aureliusdev* in our namespace:

```
kubectl -n <namespace> get secrets
```

2. If it is not there, then find the reflector pod in the default namespace:

```
kubectl get all
```

3. Delete reflector pod (A new one will be created automatically):

```
kubectl -n <namespace> delete pod/<podname>
```

Flink-jobmanager and taskmanager is not running

Flink-jobmanager is not running, and Flink-taskmanager keeps restarting, but other pods are fine.

To check if all pods are running:

```
kubectl -n <namespace> get all
```

Go into the Atlas pod, and see the error message:

```
kubectl -n <namespace> exec -it <pod/chart-id-atlas-0> -- bash
cd opt/apache-atlas-2.2.0/logs
cat application.log
```

If you see an error like: `org.apache.solr.client.solrj.impl.HttpSolrClient$RemoteSolrException: Error from server at http://10.20.129.33:9838/solr: Can not find the specified config set: vertex_index`

Then the *vertex_index* collection could not be created.

To solve it, we can create it manually in Solr client, then restart the Atlas pod.

1. We forward port 9838, so we can access Solr web client:

```
kubectl -n demo port-forward <pod/chart-id-atlas-0> 9838:9838
```

2. Open the web client on *localhost:9838/solr*
3. Go to the *Collections* menu, and add a collection.
   (a) Name: vertex_index
   (b) Config set: _default
   (c) maxShardsPer: -1
4. From another cmd, open the atlas pod again:

```
kubectl -n <namespace> exec -it <pod/chart-id-atlas-0> -- bash
cd opt/apache-atlas-2.2.0/
bin/atlas_stop.py
nohup bin/atlas_start.py &
```

5. You can exit it with CTR+C and to check if it is running:

```
jobs
```

If an entity are not getting created

It could be that a flink job has failed.

1. Check whether all flink jobs are running. if not, then restart them:

```
kubectl -n <namespace> exec -it <pod/flink-jobmanager-pod-name> -- bash

cd py_libs/m4i-flink-tasks/scripts

/opt/flink/bin/flink run -d -py <name_of_job>.py
```

2. Determine if the entity was created within the apache atlas.
3. Determine if the entity was created in the elastic.

PS. Be aware of resource problems

8.2 Contact

- Email

- Website

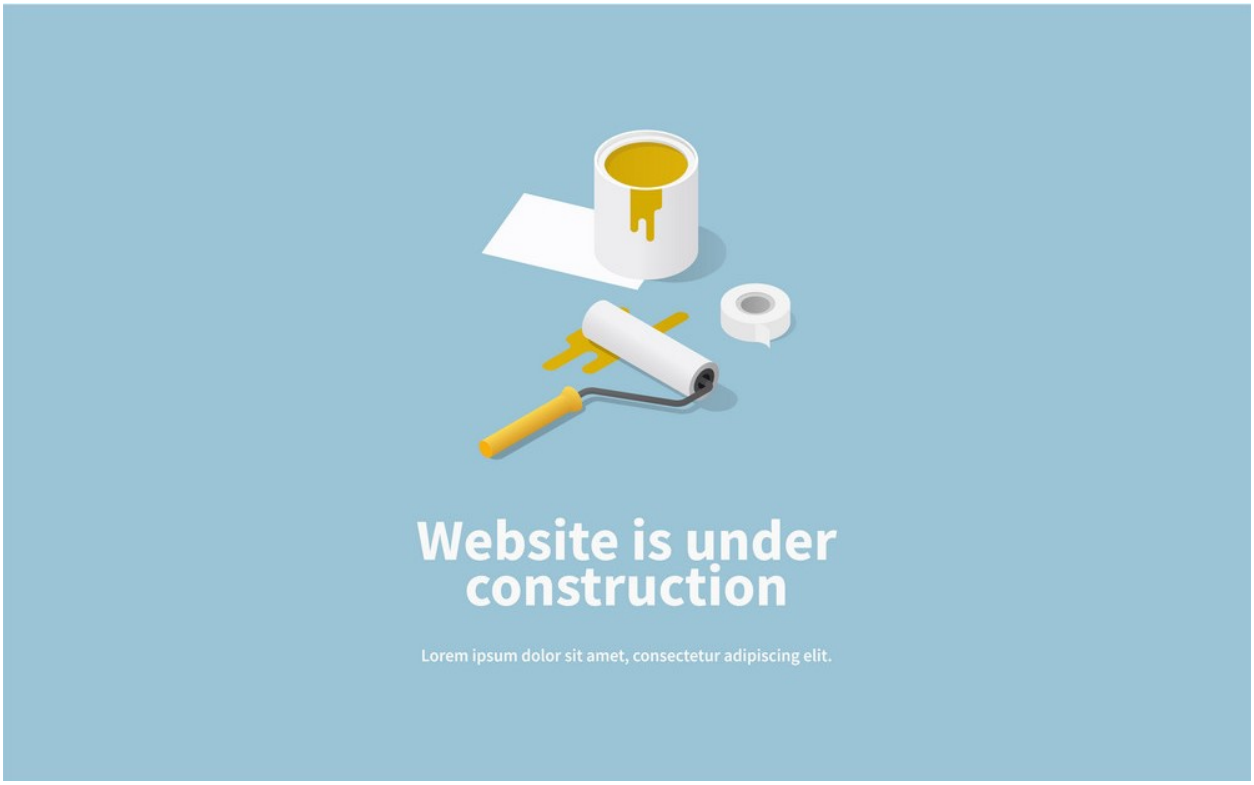
8.3 User comunities

- Github
- Linkedin

CHAPTER 9

About the company

COMING SOON…..



CHAPTER 10

Data Quality

Data quality refers to the overall fitness for use of data. It describes the degree to which data meets the requirements of its intended use, which can vary depending on the context, application, and user. Evaluating data quality ensures that the data is reliable, relevant, and actionable, and can help identify areas for improvement in data collection, storage, and management processes. Ultimately, the goal of data quality management is to ensure that data can be trusted and used effectively to support decision-making, analysis, and other business processes. Since actual data is required for this assessment, this analysis can not be done in Aurelius Atlas itself, but is performed on the related data storage system. The quality results however, can be documented in Aurelius Atlas. This documentation contains the checked rules as well as the actual data quality compliance results.

Data quality results are then propagated along the breadcrumb of the field to datasets, collections and systems on the technical side and to data attributes, data entities and data domains on the business side.

10.1 Conceptual view

Thus, conceptually data quality results can be added in Aurelius Atlas. It consists of 3 parts:

- the actual data quality result
- an associated data quality Atlas entity
- a field which is associated with the quality result

10.1.1 Data quality result

Data quality result consists of multiple fields:

- a unique ID, which can be human readable
- a qualityguid, which is a guid of the actual quality result
- a data quality result (dqscore), which is a boolean value of 0 or 1, where 0 means 0% compliance and 1 means 100% compliance

10.1.2 Data quality rule

A data quality rule is described in Aurelius Atlas as type data quality rule. Currently you can not enter this quality rule via the front end.

A data quality rule consists of :

- name: of the associated rule
- description: explaining the thought behind the rule
- expression: which is constructuced from an expression language on the level of the data quality
- business rule ID: which is usually just a number used for ordering the rules when presented in the front end
- dimension

Rule Category	Rule Description
completeness	degree to which data is not null
accuracy	degree to which a column conforms to a standard
validity	degree to which the data comply with a predefined structure
uniqueness	degree to which the data has a unique value
timeliness	the data should be up to date

10.1.3 Associated field

A field can be used in multiple data quality rules, thus a field may have multiple data quality results of different data quality rule dimensions. A field is referenced by the following information:

- qualified name of the field used for the assessment
- fieldguid, that is the guid of the referenced field
- qualified field name

10.2 Technical view

Technically, data quality is represented in Aurelius Atlas as an Apache Atlas entity and as data in the metadata store (elastic app search). The field as well as a description of the data quality rule are entities in Aurelius Atlas, while the actual data quality result is stored as metadata in elastic app search.

10.2.1 Data quality result

The data quality result in elastic app search is stored in the atlas-dev-quality engine. An example of the required documents is shown below. It contains all the conceptual elements explained in the previous section.

```
{
        "id": "nl3--nl3plant--nl3plant001--workorderid--8",
        "fields": [{
                        "name": "id",
                        "value": "nl3--nl3plant--nl3plant001--workorderid--8",
                        "type": "enum"
                }, {
                        "name": "fieldqualifiedname",
                        "value": "nl3--nl3plant--nl3plant001--workorderid",
```

```
                        "type": "string"
                }, {
                        "name": "fieldguid",
                        "value": "21f89d8f-4e10-4419-b135-6a84d55ed63f",
                        "type": "string"
                }, {
                        "name": "qualityguid",
                        "value": "61484c0e-89db-49ff-a67a-2e3bb2e9219c",
                        "type": "string"
                }, {
                        "name": "dataqualityruledescription",
                        "value": "This field has to be filled at all times",
                        "type": "string"
                }, {
                        "name": "expression",
                        "value": "Completeness('workorderid')",
                        "type": "string"
                }, {
                        "name": "dqscore",
                        "value": "1.0",
                        "type": "float"
                }, {
                        "name": "dataqualityruledimension",
                        "value": "Completeness",
                        "type": "string"
                }, {
                        "name": "businessruleid",
                        "value": "8.0",
                        "type": "float"
                }, {
                        "name": "name",
                        "value": "Rule 8",
                        "type": "string"
                }, {
                        "name": "guid",
                        "value": "61484c0e-89db-49ff-a67a-2e3bb2e9219c",
                        "type": "string"
                }, {
                        "name": "qualityqualifiedname",
                        "value": "nl3--nl3plant--nl3plant001--workorderid--8",
                        "type": "string"
                }, {
                        "name": "datadomainname",
                        "value": "plant data",
                        "type": "string"
                }
        ]
}
```


10.2.2 Data quality rules

Data quality rules are Apache Atlas entities, which can not be entered via the Aurelius Atlas frontend at the moment. We are working on it.

The entity contains the required fields as properties, such that they referential integrity between data quality results and the data quality rule entity are guaranteed. An example of a data quality rule entity in json format as it is stored in Apache Atlas is shown below.

```
{
        "referredEntities": {},
        "entity": {
                "typeName": "m4i_data_quality",
                "attributes": {
                        "expression": "completeness('HIER_ORG')",
                        "qualifiedName": "nl1--nl1hr--nl1hr001--hier_organization--30
↪",
                        "displayName": null,
                        "description": null,
                        "active": true,
                        "businessRuleDescription": "",
                        "ruleDescription": "This field has to be filled at all times",
                        "name": "nl1--nl1hr--nl1hr001--hier_organization--30",
                        "filterRequired": true,
                        "id": 30,
                        "qualityDimension": "Completeness",
                        "expressionVersion": "1",
                        "fields": [{
                                        "guid": "0df94338-1afc-455c-b9d5-c3d0e36d1dac
↪",
                                        "typeName": "m4i_field",
                                        "uniqueAttributes": {
                                                "qualifiedName": "nl1--nl1hr--
↪nl1hr001--hier_organization"
                                        }
                                }
                        ]
                },
                "guid": "3059989c-364d-4404-92ef-c1e719014f00",
                "isIncomplete": false,
                "relationshipAttributes": {
                        "fields": [{
                                        "guid": "0df94338-1afc-455c-b9d5-c3d0e36d1dac
↪",
                                        "typeName": "m4i_field",
                                        "entityStatus": "ACTIVE",
                                        "displayText": "HIER_ORGANIZATION",
                                        "relationshipType": "m4i_data_quality_field_
↪assignment",
                                        "relationshipGuid": "35b3502c-38a7-4524-b266-
↪2fd46888e5f2",
                                        "relationshipStatus": "ACTIVE",
                                        "relationshipAttributes": {
                                                "typeName": "m4i_data_quality_field_
↪assignment"
                                        }
                                }
                        ],
                },
        }
}
```


The relationship attribute fields is referencing the related field. The remaining values are local to the entity and some of them are referenced and/or taken over in the data quality result data structure.

10.2.3 Propagation of data quality results

After creating the data quality rule entity in Apache Atlas and data quality results in the metadata store, the data quality is accessible at the field. To propagate data quality results through the complete governance tree, currently there is a script required which can be called periodically. In a later version of Aurelius Atlas, all changes to data quality or the governance structures in Aurelius Atlas will also propagate data quality results. A description on how to setup the script and how to run it will follow shortly.

10.2.4 Definitions of data quality rules

They are located at the m4i-data-management repository quality rules You can find all the data quality rules, that you wish to apply on a dataset. They are explanations of each rule and examples on how to use them. These are they data quality rules that are applied on a dataset.

Below is a brief description of each rule.

Rule	Description
1. Bijacency	Checks whether or not the values in the given *column_a* and *column_b* only occur as a unique combination.
2. Compare First characters	Checks whether the first 'number_of_characters' values in *first_column_name* and *second_column_name* are similar, and if the values are None or NaN.
3. Check First Characters using Prefix	Checks whether the first 'number_of_characters' values starting without in *first_column_name* and *second_column_name* are similar, and if *column_name* does not start with any of the given *prefixes*, and if the values are None or NaN.
4. Check Completeness	Checks whether the values in the column with the given *column_name* are None or NaN.
5. Check Conditional Completeness	Checks whether or not the values in the given *value_column* are *None* or *NaN*.
6. Check Unallowed Text	Checks if values in the column with the given *value_column* contain a specific unallowed *text*.
7. Check Conditional Value	Checks whether the values in the given *value_column* match (one of) the expected value(s) for a given key in the *key_column*.
8. Check Character Count	Checks how many times the values in the column with the given *column_name* contain a specific character.
9. Check Matching Pattern	Checks whether or not the values in the column with the given *column_name* match the given *pattern*.
10. Check Invalidity	Checks whether or not the values in the column with the given *column_name* does not exist in the given list of *values*.
11. Check Length	Checks if the number of characters of the values in the column with the given *column_name* are equal to the *required_length*.
12. Check Range	Checks whether or not the values in the column with the given *column_name* are: - Greater than or equal to the given *lower_bound*. - Less than or equal to the given *upper_bound*.
13. Check Prefix	Checks whether or not the values in the column with the given *column_name* start with any of the given *prefixes*.
14. Check Unallowed Text	Checks if values in the column with the given *column_name*
15. Check Uniqueness	Checks whether the values in the column with the given *column_name* are unique (duplicate value check).
16. Check Validity	Checks whether or not the values in the column with the given *column_name* exist in the given list of *values*.

10.3 Data Quality Rules and Examples

10.3.1 1. Bijacency

In this example, a dummy dataset is provided and the columns “id” and “name” are compared.

A dummy data set is seen in the code First run a test to see if the columns are bijacent. The columns “id” and “name” are compared.

```
data = DataFrame([
    {
        "id": 1234,
        "name": "John Doe",
        "function": "Developer",
        "from": "01-01-2021"
    },
    {
        "id": 1234,
        "name": "John Doe",
        "function": "Senior developer",
        "from": "01-01-2022"
    }
                ])

result =  bijacency(data, "id", "name")
```

This is the function that we are using: *bijacency(df, “column_a”, “column_b”).* The inputs are the dataset and the column names. The id and name are the same in this example, which means they are bijacent. The output will be 1.

The source code to bijacency is available here

10.3.2 2. Compare First characters

Checks whether the first ‘number_of_characters ‘values in *first_column_name* and *second_column_name* are similar, and if the values are None or NaN.

A dummy dataset is provided and the first two characters of the id and name will be compared.

```
data = DataFrame([
        {
            "id": "NL.xxx",
            "name": "NL.xxx",
        }
                ])

result = compare_first_characters(data, "id", "name", 2)
```

This is the function used in this example: *compare_first_characters(df, “column_a”, “column_b”, num_char).* The inputs are the dataset,the column names and the number of characters.

The source code to *compare_first_characters* is available here

10.3.3 3. Check First Characters using Prefix

This rule does three checks. It checks if the first characters are the same, if the have same prefix and if the values are Nan or none.

A dummy dataset with two columns, id and name is provided

```
data = DataFrame([
        {
            "id": "BE.xxx",
            "name": "BE.xxx",
         }         ])



result = compare_first_characters_starting_without(data, "id", "name", 2, 'BE
↪')
```

A prefix BE is used and the function is *compare_first_characters_starting_without(data, "id", "name", 2, 'BE')* The inputs are the data, the column names, the number of characters and the prefix. The output will be 1, because the charaters are the same and have the prefix too.

The source code to *compare_first_characters_starting_without* is available here

10.3.4 4. Check Completeness

Checks whether the values in the column with the given *column_name* are None or NaN.

We provide a data dummy test in the unit test and we want to check if the column 'name' has a value or not. If it has a value the function will return 1, otherwise it will return 0

```
data = DataFrame([
    {
        "id": 1234,
        "name": NaN,
        "function": "Developer",
        "from": "01-01-2021"
    }
                ])

result = completeness(data, "name")
```

The function is called *completeness(df, "column")*. The inputs are data and the name of the column we want to check. The output will be 0, because the column 'name' has no value in it.

The source code to *completeness* is available here

10.3.5 5. Check Conditional Completeness

The columns "value" and "conditional" are 'None' or 'NaN'. The rows are filtered, where the value of the 'key_column', is not a substring of the given value in the function. In this example the key column in "conditional" and we are seeing if it has a substring of the list values.

```
values = ['.TMP', '.FREE']
['.TMP', '.FREE']
    data = DataFrame([
        {
            "value": "Something",
            "conditional": "xx.FREE.eur"
        }
    ])
```

```
result = conditional_completeness(data, "conditional", "value", values)
```

This is the function of use *conditional_completeness(df, "column_a", "column_b",[list])*. The inputs are data, the name of the columns and the list of given values. The output here will be 1, because they are no empty values in the columns and the column "conditional" has substrings of the given values = ['.TMP', '.FREE']

The source code to *conditional_completeness* is availabe here

10.3.6 6. Check Unallowed Text

The check here is to see if there is unalllowed text in the columns of the dummy dataframe.

```
values = ['.TMP', '.FREE']

unallowed_text_item = "("

data = DataFrame([
    {
        "value": "Something",
        "conditional": "xx.FREE.eur"
    }
])

result = conditional_unallowed_text(data, "conditional", "value", values,␣
↪unallowed_text_item)
```

This is the function of use *conditional_unallowed_text(df, "column_a", "column_b", [list_of_values], "string")*. The inputs are the dataframe, the name of the two columns, the values of the substrings and the unallowed text. The output will be 1 because it containf substrings in the 'conditional' column and doesn't contain the unallowed text in column "Value". If it did the output would be 0.

The source code to *conditional_unallowed_text* is available here

10.3.7 7. Check Conditional Value

The 'value' and 'conditional' column are being checked to see if it contains the expected values of the 'key' values object.

```
values = {"xx.TMP": "XX No Grade"}     (this is dictionary with it's key and␣
↪value)

data = DataFrame([                     (this is our dummy dataset)
    {
        "value": "XX No Grade",
        "conditional": "xx.TMP"
    }
])

result = conditional_value(data, "conditional", "value", values)
```

the function used for this example is called *conditional_value(df, "column_a", "column_b", {dictionary})*. The inputs are data of the dummy dataset, the names of the columns which are "value" and "conditional" and the values, that are the substrings we want to check. The output here will 1, because "value" column, contains an expecetd value. Otherwise it would be 0.

The source code to *conditional_value* is available here

10.3.8 8. Check Character Count

Checks how many times the values in the column with the given *column_name* contain a specific character.

A dummy dataframe is provided with one column called "id".

```
data = DataFrame([
        {
            "id": "12.12"
        }
    ])

result = contains_character(data, "id", ".", 1)
```

This is the function used in this example *contains_character(df, "column", "string", int)*. The inputs are data, name of the column, the character we want to check and 1 is the expected count The check performed here is to if the the id contains "." . The output will be 1 because the "id" column contains "."

The source code to *contains_character* is available here

10.3.9 9. Check Matching Pattern

Checks if the values in the column *name* match the given *pattern*.

A dummy dataset is provided

```
data = DataFrame([
        {
            "name": 'ExampleText'
        }
    ])

result = formatting(data, "name", r'^[a-zA-Z]+$')
```

This is the function used for this example *formatting(df, "column", expression_pattern)*. The inputs are the dataset,the column "name" and the pattern to see if it matches The ouput will be 1 in this example, because 'ExampleText' matches the pattern.

The source code to *formatting* is available here

10.3.10 10. Check Invalidity

The values in the column with the given name *value* are checked if they do not exist in the given list of *exampleValues*.

A list of the example values and a dummy dataframe are provided.

```
exampleValues = ['x', 'X', 'TBD', 'Name']

data = DataFrame([
        {
            "value": "X"
        }
    ])
```

```
result = invalidity(data, "value", exampleValues)
```

The funtion is *invalidity(df, "column", [list])*. The inputs are data, column name and the list of values. The output will be 1 , becaue "X" is in the list of values.

The source code to *invalidity* is available here

10.3.11 11. Check Length

The check performed here is the number of characters of the values in the column *id* are equal to the *required_length*.

A dummy dataframe with column name "id"

```
data = DataFrame([
        {
            "id": "1234"
        }
    ])

result = length(data, "id", 4)
```

The function is *length(df,"column",int)*. The inputs are data, column name and the length of required characters. The output is 1 because the length of id is 4.

The source code to *length* is available here

10.3.12 12. Check Range

The check performed here is the values in the column *column_name* are greater than or equal to the given *lower_bound* or less than or equal to the given *upper_bound*.

A dummy dataframe for this example with column name "value"

```
data = DataFrame([
        {
            "value": 0.1
        }
                ])

result = range(data, "value", 0, 1)
```

The function is *range(df, "column", int1, int2)*. The inputs are the dataframe, the column name and the range (The upper and lower bound) The output will be 1 because 0.1 is between 0 and 1.

The source code to *range* is available here

10.3.13 13. Check Prefix

This example checks if the values in the column *column_name* start with any of the given *prefixes*.

```
data = DataFrame([
        {
            "id": 1234
        }
```

```
                ])

result = starts_with(data, "id", "1")
```

The function is called *starts_with(data, "column", "prefix")*. The inputs are the data the column name and the prefix. The output is 1, because "1" is in the value of the id column.

The source code to *starts_with* is available here

10.3.14 14. Check Unallowed Text

This example checks if the values in the column *Organisation* contain a specific unallowed *text*.

A dummy dataset is provided.

```
data = DataFrame([
    {
        "Organisation": "Something Else"
    }
])

result = unallowed_text(data, "Organisation", "BG Van Oord")
```

The function is called *unallowed_text(df, "column", "sting")*. The inputs are data, the column name and the unallowed text The output is 1 because "BG Van Oord" is not in the "Something Else" of the "Organisation" column.

The source code to *unallowed_text* is available here

10.3.15 15. Check Uniqueness

This example checks if the values in the column *id* are unique. It checks for duplicate values

A dummy dataset is provided

```
data = DataFrame([
        {
            "id": "1234"
        },
        {
            "id": "1234"
        },
        {
            "id": "2345"
        }
                ])

result = uniqueness(data, "id")
```

The function is *uniqueness(data, "id")*. The inputs are the dataset and the name of the column. The output will be 0, because the "id" column conatins duplicate values

The source code to *uniqueness* is available here

10.3.16 16. Check Validity

This example checks if the values in the column *value* exist in the list of exampleValues.

The values in the example list and a dummy dataset are provided

```
exampleValues = ['Definite Contract', 'Indefinite Contract']

data = DataFrame([
        {
            "value": "Definite Contract"
        }
                ])

result = validity(data, "value", exampleValues)
```

The function is *validity(df, "key",[list])*. The inputs are data, the column name and the list of example values. The output is 1, because the value of the column exists in the example list.

The source code to *validity* is available here

10.4 Apply Data Quality Results

The tool checks the quality of your data. To use it, you need to provide a csv file with your data and the rules you want to apply to it. The rules are basically the type of checks you want to do on the attributes of your dataset. The rules you want to define are stored, on Aurelius Atlas and is used to apply the rules to your data. The quality score of your data is calculated based on the applied rules and the results are sent to a Kafka topic. Below is an image that describes the whole process for your better understanding.

1. First upload a file, define the rules that we want to apply to the data. Then push this file to atlas.
2. Then get the data quality rules from atlas and see the data quality results. The quality results have a data quality score. 1 is compiant and 0 is non-compliant
3. Finally push the data quality results to kafka.

10.4.1 How To Perform A Data Quality Check Of Your Data

Here is a link of the repositories you will need:

https://github.com/aureliusenterprise/m4i_atlas_core

https://github.com/AthanasiosAurelius/m4i-data-management

10.4.2 Install M4I Data Management

This library contains all core functionality around data management.

Installation

Please ensure your *Python* environment is on version *3.7*. Some dependencies do not work with any later versions of *Python*.

To install *m4i-data-management* and all required dependencies to your active *Python* environment, please run the following command from the project root folder:

To install *m4i-data-management* including development dependencies, please run the following command instead:

pip install -e .[dev]

Install m4i_data_management: You can clone m4i_data_management from this link https://github.com/AthanasiosAurelius/m4i-data-management Then you install with this command

pip install {path to m4i_data_management}

Do the same for m4i_atlas_core

pip install {path to m4i_atlas_core}

Please make a copy of *config.sample.py* and *credentials.sample.py* and rename the files to *config.py* and *credentials.py* respectively.

The *config.py* and *credentials.py* files should be located in the root folder of the project, or otherwise on the *PYTHON_PATH*.

Please remember to set the configuration parameters you want to use.

10.4.3 How to set up config and credentials file

Here is the exact configuration of the config and credentials, use this to run the example.

```
config = {
    "atlas_dataset_guid": "f686adca-00c4-4509-b73b-1c51ae597ebe",
    "dataset_quality_name": "example_name",
    "atlas": {
        "atlas.server.url": "https://aureliusdev.westeurope.cloudapp.azure.com/anwo/
↪atlas/atlas",
    },
    "keycloak.server.url": "https://aureliusdev.westeurope.cloudapp.azure.com/anwo/
↪auth/",
    "keycloak.client.id": "m4i_public",
    "keycloak.realm.name": "m4i",
    "keycloak.client.secret.key": ""
}

credentials = {
    "keycloak.credentials.username": "atlas",
    "keycloak.credentials.password": "",
    "atlas.server.url":"https://aureliusdev.westeurope.cloudapp.azure.com/anwo/atlas/
↪atlas",
    "atlas.credentials.username":"atlas",
```

```
    "atlas.credentials.password":""
}
```

10.4.4 How to run data quality check

Our tool checks the quality of your data. To use it, you need to provide a csv file with your data and the rules you want to apply to it. The rules are basically the type of checks you want to do on the attributes of your dataset. We store your data and rules on Atlas and use our tool to apply the rules to your data. We then calculate the quality score of your data based on the applied rules and provied a csv output with the results.

These are the steps on how to do it

1. In the run_quality_rules.py we can now run our check. We have to provide a dataset so we can do a quality check. Fill in the path in the get_data_csv(). You will see it on line 63. Make a csv file with example data. Here is a simple example below.

   Just One Column named UID and provide a name. Make an excel file.

   UID example_name

2. Finally we run our check in the run_quality_rules.py In debug mode run the 'asyncio.run(atlas_dataset_quality.run())' it's on line 59

10.4.5 How to create entities and relationships

In the create_push_to_atlas.py a user can create a dataset, field and data quality rule entity and push it to atlas. He can create a relationship between the field and dataset. I will explain how to do it with an example.

1. Define the attributes for each instance

Define the attributes for the dataset instance

```
json_dataset={
    "attributes": {
        "name": "example",
        "qualifiedName": "example100"
    },
    "typeName": "m4i_dataset"
    }
```

Define the attributes for the field instance

```
json_field={
    "attributes": {
        "name": "field",
        "qualifiedName": "example--field"
    },
    "typeName": "m4i_field",
    "relationshipAttributes": {
        "dataset": {
            "guid": "<guid-of-json_dataset>",
            "typeName": "m4i_dataset",
            "relationshipType": "m4i_dataset_fields"
        }
    }
}
```

Define the attributes for the data quality instance

```
json_quality={
    "attributes": {
        "name": "field",
        "qualifiedName": "example--quality",
        "id": 1
    },
    "typeName": "m4i_data_quality"
    }
```

2. Create instances

Create instances of BusinessDataset, BusinessField, and BusinessDataQuality

3. Add relationship between the field and dataset instances

```
field_attributes=field_instance.attributes
field_attributes.datasets= [ObjectId(
            type_name="m4i_dataset",
            unique_attributes= M4IAttributes(
            qualified_name="example100"
        )
        )]
```

4. Push the entities to atlas.

We use the create_entities function that can be found in the m4i_atlas_core. It is important to undertstand what are the inputs. create_entites(dataset_instance,referred_entites,accesss_token). The first input is the instance we created, then the referred entities, which here are non because we are just creating an entity with no relationships and finally the access token.

Push the dataset instance to Atlas

```
async def create_in_atlas(dataset,access_token=access_token):
    mutations_dataset = await create_entities(dataset,referred_entities=None,
↪access_token=access_token)
    print(mutations_dataset)
push_to_atlas= asyncio.run(create_in_atlas(dataset_instance,access_
↪token=access_token))
```

Push the field instance to Atlas

```
async def create_in_atlas_field(field,access_token=access_token):
    mutations_field = await create_entities(field,field,referred_
↪entities=None,access_token=access_token)
    print(mutations_field)
push_field = asyncio.run(create_in_atlas_field(field_instance,access_
↪token=access_token))
```

Push the data quality instance to Atlas

```
async def create_in_atlas_rule(rule,access_token=access_token):
    mutations_rule = await create_entities(rule,referred_entities=None,
↪access_token=access_token)
    print(mutations_rule)
push_rule = asyncio.run(create_in_atlas_rule(rule,access_token=access_token))
```