Implement an Open Metadata Repository Connector

Eager to integrate your own metadata repository into the Egeria ecosystem, but not sure where to start? This article walks through how to do just that: implementing an open metadata repository connector according to the standards of ODPi Egeria.

Integrating a metadata repository into the Open Metadata ecosystem involves coding an Open Metadata Collection Store Connector. These are Open Connector Framework (OCF) connectors that define how to connect to and interact with a metadata repository.

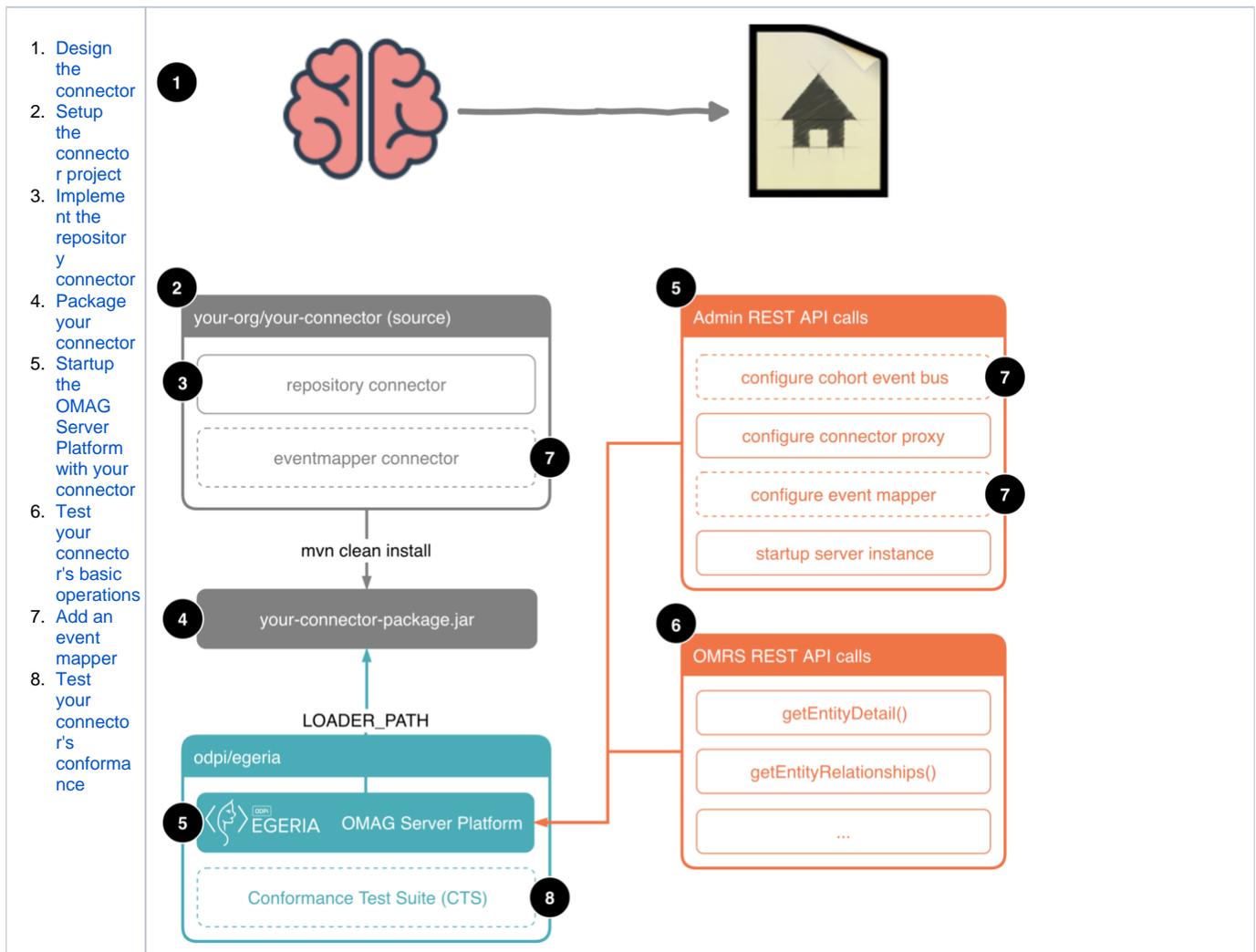Open Metadata Collection Store Connectors are typically comprised of two parts:

1. **The repository connector**: which provides a standard repository interface that communicates using the Open Metadata Repository Services (OMRS) API and payloads.
2. **The event mapper connector**: which captures events when metadata has changed in the metadata repository and passes these along to the Open Metadata Repository Services (OMRS) cohort.

The event mapper connector often calls the repository connector: to translate the repository-native events into Egeria's OMRS events.
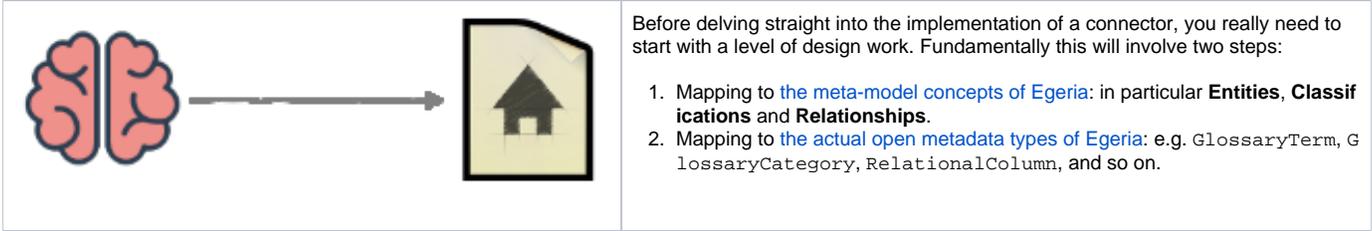
While various patterns can be used to implement these, perhaps the simplest and most loosely-coupled is the adapter. The adapter approach wraps the proprietary interface(s) of the metadata repository to translate these into OMRS calls and payloads. In this way, the metadata repository can communicate as if it were an open metadata repository.

The remainder of this article will walkthrough:

- implementing such an adapter pattern as a connector, and
- using the resulting connector through the proxy capabilities provided by the core of Egeria.

1. Design the connector
2. Setup the connector project
3. Implement the repository connector
4. Package your connector
5. Startup the OMAG Server Platform with your connector
6. Test your connector's basic operations
7. Add an event mapper
8. Test your connector's conformance

Design the connector

Before delving straight into the implementation of a connector, you really need to start with a level of design work. Fundamentally this will involve two steps:

1. Mapping to the meta-model concepts of Egeria: in particular **Entities**, **Classifications** and **Relationships**.
2. Mapping to the actual open metadata types of Egeria: e.g. `GlossaryTerm`, `GlossaryCategory`, `RelationalColumn`, and so on.

Map to the Egeria meta-model concepts

The best place to start with the design work is to understand the meta-model of Egeria itself. Consider how your metadata repository will map to the fundamental Egeria metadata concepts: **Entities**, **Classifications**, and **Relationships**.

When implementing the code described in the remainder of this article, you'll be making use of and mapping to these fundamental Egeria concepts. Therefore, it is well worth your time now understanding them in some detail. This is before even considering specific *instances* of these types like `GlossaryTerm` or `GlossaryCategory`.

Meta-model mapping may be quite a straightforward conceptual mapping for some repositories. For example, Apache Atlas has the same concepts of Entities, Classifications and Relationships all as first-class objects.

On the other hand, not all repositories do. For example, IBM Information Governance Catalog (IGC) has Entities, and a level of Relationships and Classifications — but the latter two are not really first-class objects (i.e. properties and values cannot exist on them).

Therefore you may need to consider

- whether to attempt to support these constructs in your mappings, and
- if so, how to prescriptively represent them (if they are not first-class objects).

For example, in the implementation of the sample IGC connector we suggest using categories with specific names in IGC to represent certain classifications. Additionally, one of the reasons for implementing a read-only connector is that we can still handle relationships without any properties: by simply having the properties of any Egeria relationships we translate *from* IGC all be empty.

Map to the Egeria open metadata types

Once you have some idea for how to handle the mapping to the meta-model concepts, check your thinking by working through a few examples. Pick a few of the open metadata types and work out on paper how they map to your metadata repository's pre-existing model. Common areas to do this would be e.g. `GlossaryTerm`, `GlossaryCategory` for glossary (business vocabulary) content; `RelationalColumn`, etc for relational database structures; and so on.

Most of these should be fairly straightforward after you have an approach for mapping to the fundamental meta-model concepts.

Then you'll also want to decide how to handle any differences in types between the open metadata types and your repository's pre-existing types:

- Can your metadata repository be extended with new types?
- Can your metadata repository's pre-existing types be extended with new properties?
- What impacts might be caused to repositories (and metadata instances) that already exist if you add to or extend the types?
- What impacts will this have on your UI or how users interact with these extensions?

Your answers to these questions will inevitably depend on your specific metadata repository, but should help you decide on what approach you'd like to take:

- Ignore any open metadata types that do not map to your pre-existing types.
- Add any Egeria open metadata types that do not exist in your repository.
- Add Egeria open metadata properties to your pre-existing types when Egeria has additional properties that do not yet exist in your type(s).
- Implement a read-only connection (possibly with some hard-coding of property values) for types that are partially map-able, but not easily extended to support the full set of properties defined on the open metadata type.
- and so on.

Setup the connector project

Implementing an adapter can be greatly accelerated by using the pre-built base classes of Egeria. Therefore building a connector using Java is likely the easiest way to start.

This requires an appropriate build environment comprised of both Java (minimally v1.8) and Maven.

Egeria has been designed to allow connectors to be developed in projects independently from the core itself. Some examples have already been implemented, which could provide a useful reference point as you proceed through this walkthrough:
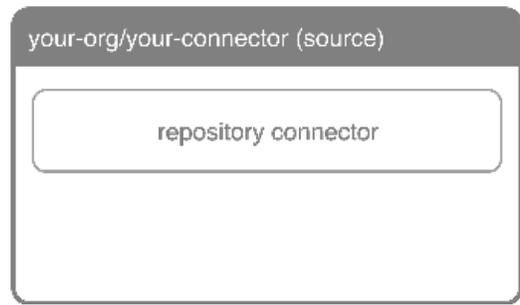
- a sample IBM InfoSphere Information Governance Catalog Repository Connector
- a sample Apache Atlas Repository Connector

Start by defining a new Maven project in your IDE of choice. In the root-level POM be sure to include the following:

```
<properties>
    <open-metadata.version>2.5-SNAPSHOT</open-metadata.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.odpi.egeria</groupId>
        <artifactId>repository-services-apis</artifactId>
        <version>${open-metadata.version}</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.odpi.egeria</groupId>
        <artifactId>open-connector-framework</artifactId>
        <version>${open-metadata.version}</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

Naturally change the version to whichever version of Egeria you'd like to build against. The dependencies listed ensure you'l have the necessary portion of Egeria to build your connector against.

Implement the repository connector

your-org/your-connector (source) repository connector	The repository connector exposes the ability to search, query, create, update and delete metadata in an existing metadata repository. As such, it will be the core of your adapter. You can start to build this within your new project by creating a new Maven module called something like `adapter`.

Within this `adapter` module implement the following:

Implement an OMRSRepositoryConnectorProvider

Start by writing an `OMRSRepositoryConnectorProvider` specific to your connector, which extends `OMRSRepositoryConnectorProviderBase`. The connector provider is a factory for its corresponding connector. Much of the logic needed is coded in the base class, and therefore your implementation really only involves defining the connector class and setting this in the constructor.

For example, the following illustrates this for the Apache Atlas Repository Connector:

```
package org.odpi.egeria.connectors.apache.atlas.repositoryconnector;

import org.odpi.openmetadata.frameworks.connectors.properties.beans.ConnectorType;
import org.odpi.openmetadata.repositoryservices.connectors.stores.metadatacollectionstore.repositoryconnector.
OMRSRepositoryConnectorProviderBase;

public class ApacheAtlasOMRSRepositoryConnectorProvider extends OMRSRepositoryConnectorProviderBase {

    static final String  connectorTypeGUID = "7b200ca2-655b-4106-917b-abddf2ec3aa4";
    static final String  connectorTypeName = "OMRS Apache Atlas Repository Connector";
    static final String  connectorTypeDescription = "OMRS Apache Atlas Repository Connector that processes
events from the Apache Atlas repository store.";

    public ApacheAtlasOMRSRepositoryConnectorProvider() {
        Class connectorClass = ApacheAtlasOMRSRepositoryConnector.class;
        super.setConnectorClassName(connectorClass.getName());
        ConnectorType connectorType = new ConnectorType();
        connectorType.setType(ConnectorType.getConnectorTypeType());
        connectorType.setGUID(connectorTypeGUID);
        connectorType.setQualifiedName(connectorTypeName);
        connectorType.setDisplayName(connectorTypeName);
        connectorType.setDescription(connectorTypeDescription);
        connectorType.setConnectorProviderClassName(this.getClass().getName());
        super.connectorTypeBean = connectorType;
    }
}
```

Note that you'll need to define a unique GUID for the connector type, and a meaningful name and description. Really all you then need to implement is the constructor, which can largely be a copy / paste for most `adapter`s. Just remember to change the `connectorClass` to your own, which you'll implement in the next step (below).

Implement an OMRSRepositoryConnector

Next, write an `OMRSRepositoryConnector` specific to your connector, which extends `OMRSRepositoryConnector`. This defines the logic to connect to and disconnect from your metadata repository. As such the main logic of this class will be implemented by:

- Overriding the `initialize()` method to define any logic for initializing the connection: for example, connecting to an underlying database, starting a REST API session, etc.
- Overriding the `setMetadataCollectionId()` method to create an `OMRSMetadataCollection` for your repository (see next step below).
- Overriding the `disconnect()` method to properly cleanup / close such resources.

Whenever possible, it makes sense to try to re-use any existing client library that might exist for your repository. For example, Apache Atlas provides a client through Maven that we can use directly. Re-using it saves us from needing to implement and maintain various beans for the (de)serialization of REST API calls.

The following illustrates the start of such an implementation for the [Apache Atlas Repository Connector](#):

```java
package org.odpi.egeria.connectors.apache.atlas.repositoryconnector;

import org.apache.atlas.AtlasClientV2;

public class ApacheAtlasOMRSRepositoryConnector extends OMRSRepositoryConnector {

    private String url;
        private AtlasClientV2 atlasClient;
    private boolean successfulInit = false;

    public ApacheAtlasOMRSRepositoryConnector() { }

    @Override
    public void initialize(String                  connectorInstanceId,
                          ConnectionProperties connectionProperties) {
        super.initialize(connectorInstanceId, connectionProperties);

        final String methodName = "initialize";

        // Retrieve connection details
        EndpointProperties endpointProperties = connectionProperties.getEndpoint();
        // ... check for null and handle ...
        this.url = endpointProperties.getProtocol() + "://" + endpointProperties.getAddress();
        String username = connectionProperties.getUserId();
        String password = connectionProperties.getClearPassword();

        this.atlasClient = new AtlasClientV2(new String[]{ this.url }, new String[]{ username, password });

        // Test REST API connection by attempting to retrieve types list
        try {
            AtlasTypesDef atlasTypes = atlasClient.getAllTypeDefs(new SearchFilter());
            successfulInit = (atlasTypes != null && atlasTypes.hasEntityDef("Referenceable"));
        } catch (AtlasServiceException e) {
            log.error("Unable to retrieve types from Apache Atlas.", e);
        }

        if (!successfulInit) {
            ApacheAtlasOMRSErrorCode errorCode = ApacheAtlasOMRSErrorCode.REST_CLIENT_FAILURE;
            String errorMessage = errorCode.getErrorMessageId() + errorCode.getFormattedErrorMessage(this.url);
            throw new OMRSRuntimeException(
                    errorCode.getHTTPErrorCode(),
                    this.getClass().getName(),
                    methodName,
                    errorMessage,
                    errorCode.getSystemAction(),
                    errorCode.getUserAction()
            );
        }

    }

    @Override
    public void setMetadataCollectionId(String metadataCollectionId) {
        this.metadataCollectionId = metadataCollectionId;
        if (successfulInit) {
            metadataCollection = new ApacheAtlasOMRSMetadataCollection(this,
                    serverName,
                    repositoryHelper,
                    repositoryValidator,
                    metadataCollectionId);
        }
    }

}
```

This has been abbreviated from the actual class for simplicity; however, note that as part of the `initialize()` it may make sense to test out the parameters received for configuring the connection, to make sure that a connection to your repository can actually be established before proceeding any further.

(This is also used in this example to setup a flag `successfulInit` to indicate whether connectivity was possible, so that if it was not we do not proceed any further with setting up the metadata collection and we allow the connector to fail immediately, with a meaningful error.)

You may want to wrap the metadata repository client's methods with your own methods in this class as well. Generally think of this class as "speaking the language" of your proprietary metadata repository, while the next class "speaks" Egeria.

Implement an OMRSMetadataCollection

Finally, write an `OMRSMetadataCollection` specific to your repository, which extends `OMRSMetadataCollectionBase`. This can grow to be quite a large class, with many methods, but is essential for the participation of your metadata repository in a broader cohort. In particular, it is heavily leveraged by Egeria's Enterprise Connector to federate actions against your metadata repository. As such, this is how your connector "speaks" Egeria (open metadata).

Ideally your implementation should override each of the methods defined in the base class. To get started:

1. Override the `addTypeDef()` method. For each TypeDef this method should either extend your metadata repository to include this TypeDef, configure the mapping from your repository's types to the open metadata types, or throw a `TypeDefNotSupportedException`. (For those that are implemented it may be helpful to store these in a class member for comparison in the next step.)
2. Override the `verifyTypeDef()` method, which can check the types you have implemented (above) conform to the open metadata TypeDef received (ie. that all properties are available, of the same data type, etc), and that if none have yet been listed as implemented that `false` is returned (this will cause `addTypeDef()` above to automatically be called).
3. Override the `getEntityDetail()` method that retrieves an entity by its GUID.

Note that there are various options for implementing each of these. Which route to take will depend on the particulars of your specific metadata repository:

- In the sample [IBM InfoSphere Information Governance Catalog Repository Connector](#) the mappings are defined in code. This approach was used because IGC does not have first-class Relationship or Classification objects. Therefore, some complex logic is needed in places to achieve an appropriate mapping. Furthermore, if a user wants to extend the logic or mappings used for their particular implementation of IGC, this approach allows complete flexibility to do so. (A developer simply needs to override the appropriate method(s) with custom logic.)
- The sample [Apache Atlas Repository Connector](#) illustrates a different approach. Because the TypeDefs are quite similar to those of Egeria, it is easier to map more directly through configuration files. A generic set of classes can be implemented that use these configuration files to drive the specifics of each mapping. In this case, simple JSON files were used to define the `omrs` name of a particular object or property and the corresponding `atlas` entity / property name to which it should be mapped. While this allows for much more quickly adding new mappings for new object types, it is far less flexible than the code-based approach used for IGC. (It is only capable of handling very simple mappings: anything complex would either require the definition of a complicated configuration file or still resorting to code).
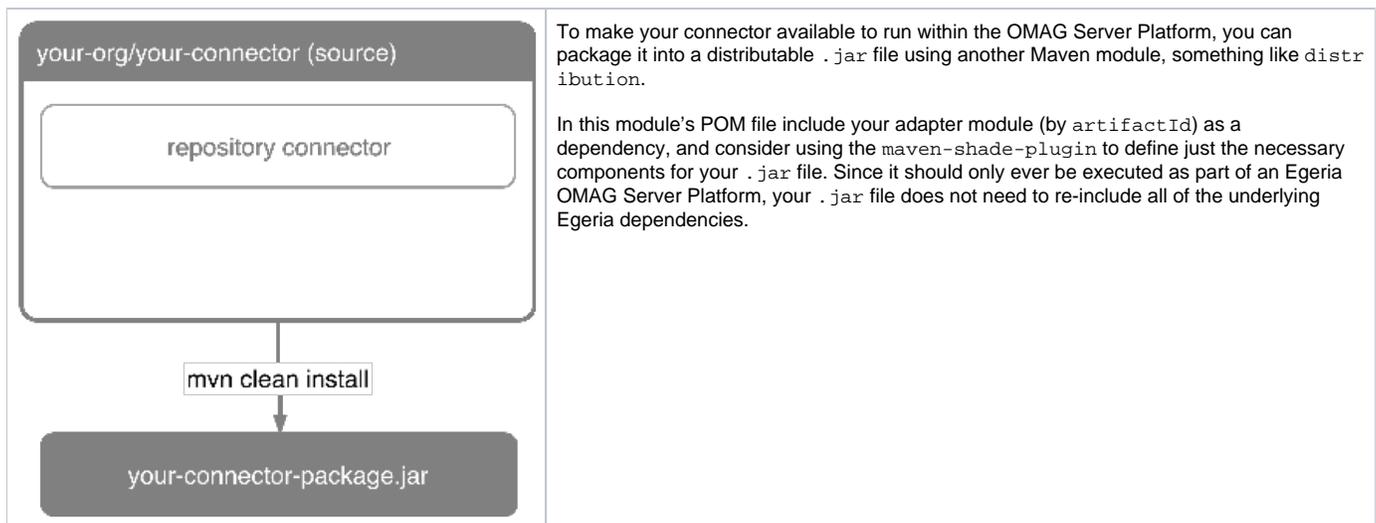
Once these minimal starting points are implemented, you should be able to configure the OMAG Server Platform as a proxy to your repository connector by following the instructions in the next step.

Important: this will *not necessarily* be the end-state pattern you intend to use for your repository connector. Nonetheless, it can provide a quick way to start testing its functionality.

This very basic, initial scaffold of an implementation allows:

- a connection to be instantiated to your repository, and
- translation between your repository's representation of metadata and the open metadata standard types.

Package your connector

To make your connector available to run within the OMAG Server Platform, you can package it into a distributable `.jar` file using another Maven module, something like `distribution`.

In this module's POM file include your adapter module (by `artifactId`) as a dependency, and consider using the `maven-shade-plugin` to define just the necessary components for your `.jar` file. Since it should only ever be executed as part of an Egeria OMAG Server Platform, your `.jar` file does not need to re-include all of the underlying Egeria dependencies.

For example, in our [Apache Atlas Repository Connector](#) we only need to include the `adapter` module itself and the base dependencies for Apache Atlas's Java client (all other dependencies like Egeria core itself, the Spring framework, etc will already be available through the Egeria OMAG Server Platform):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>egeria-connector-apache-atlas</artifactId>
        <groupId>org.odpi.egeria</groupId>
        <version>1.1-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>egeria-connector-apache-atlas-package</artifactId>

    <dependencies>
        <dependency>
            <groupId>org.odpi.egeria</groupId>
            <artifactId>egeria-connector-apache-atlas-adapter</artifactId>
            <version>${open-metadata.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>${maven-shade.version}</version>
                <executions>
                    <execution>
                        <id>assemble</id>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <artifactSet>
                                <includes>
                                    <include>org.odpi.egeria:egeria-connector-apache-atlas-adapter</include>
                                    <include>org.apache.atlas:atlas-client-common</include>
                                    <include>org.apache.atlas:atlas-client-v1</include>
                                    <include>org.apache.atlas:atlas-client-v2</include>
                                    <include>org.apache.atlas:atlas-intg</include>
                                    <include>org.apache.hadoop:hadoop-auth</include>
                                    <include>org.apache.hadoop:hadoop-common</include>
                                    <include>com.fasterxml.jackson.jaxrs:jackson-jaxrs-base</include>
                                    <include>com.fasterxml.jackson.jaxrs:jackson-jaxrs-json-provider</include>
                                    <include>com.fasterxml.jackson.module:jackson-module-jaxb-annotations</include>
                                    <include>com.sun.jersey:jersey-client</include>
                                    <include>com.sun.jersey:jersey-core</include>
                                    <include>com.sun.jersey:jersey-json</include>
                                    <include>com.sun.jersey.contribs:jersey-multipart</include>
                                    <include>javax.ws.rs:jsr311-api</include>
                                </includes>
                            </artifactSet>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

</project>
```

Of course, you do not need to use the `maven-shade-plugin` to accomplish such bundling: feel free to define a Maven assembly or other Maven techniques.

Building and packaging your connector should then be as simple as running the following from the root of your project tree:
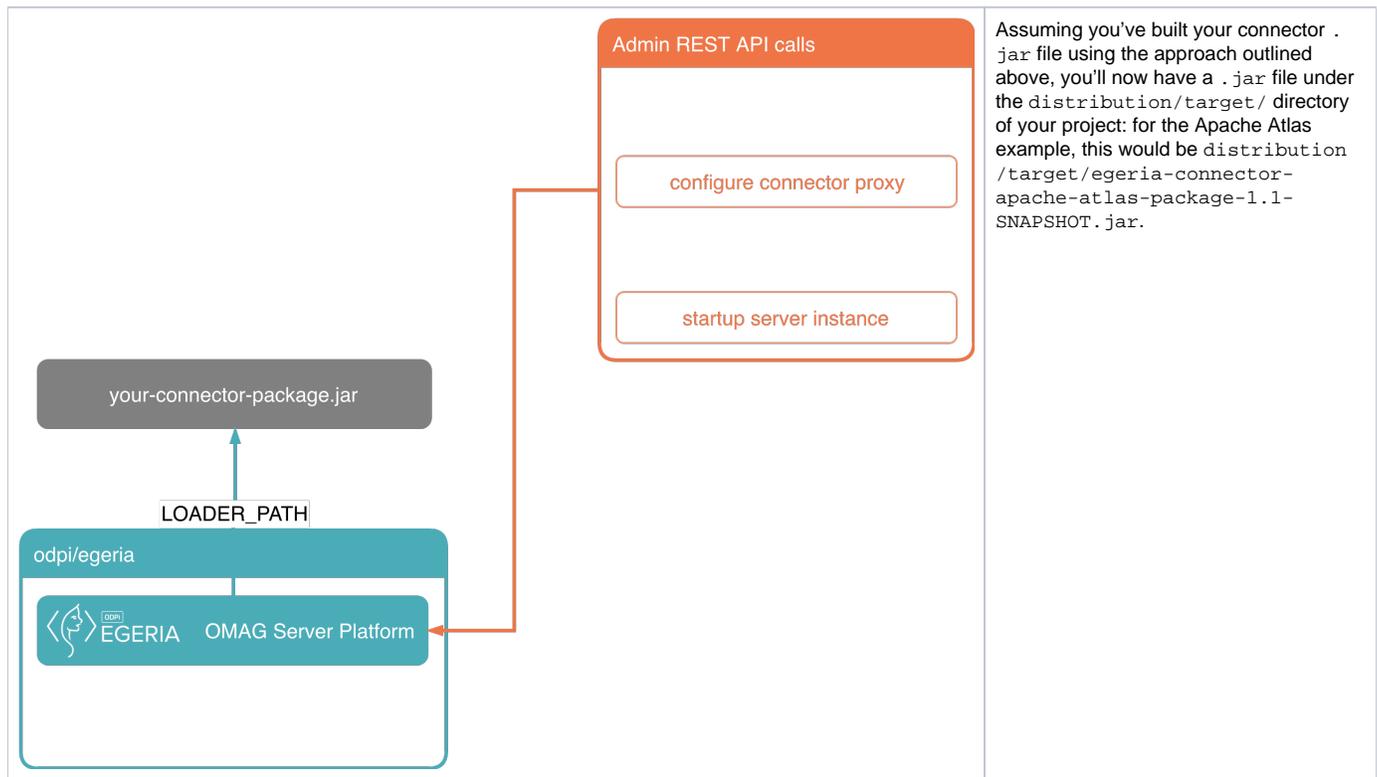
```
mvn clean install
```

Working out exactly which dependencies to include when you are using an external client like Apache Atlas's can be a little bit tricky. Starting small will inevitably result in various errors related to classes not being found: when building you'll see a list of all the classes considered by the shade plugin and which are included and excluded. You can use this to make some educated guesses as to which may still need to be included if you are running into errors about classes not being found. (Ideally you'll have a simple, single jar file / dependency you can directly include instead of needing to work through this, but that won't always be the case.)

Again, since we will just be using this connector alongside the existing OMAG Server Platform, this avoids ending up with a `.jar` file that includes the entirety of the Egeria OMAG Server Platform (and its dependencies) in your connector `.jar` — and instead allows your minimal `.jar` to be loaded at startup of the core OMAG Server Platform and configured through the REST calls covered in section 6.

Of course, if you intend to embed or otherwise implement your own server, the packaging mechanism will likely be different. However, as mentioned in the previous step this should provide a quick and easy initial way of testing the functionality of the connector against the core of Egeria.

Startup the OMAG Server Platform with your connector

Assuming you've built your connector `.jar` file using the approach outlined above, you'll now have a `.jar` file under the `distribution/target/` directory of your project: for the Apache Atlas example, this would be `distribution/target/egeria-connector-apache-atlas-package-1.1-SNAPSHOT.jar`.

When starting up the OMAG Server Platform of Egeria, we need to point to this `.jar` file using either the `LOADER_PATH` environment variable or a `-Dloader.path=` command-line argument to the server start command:

```
export LOADER_PATH=..../distribution/target/egeria-connector-apache-atlas-package-1.1-SNAPSHOT.jar
java -jar server-chassis-spring-1.1-SNAPSHOT.jar
```

or

```
java -Dloader.path=..../distribution/target/egeria-connector-apache-atlas-package-1.1-SNAPSHOT.jar -jar server-
chassis-spring-1.1-SNAPSHOT.jar
```

Either startup should ensure your connector is now available to the OMAG Server Platform to use for connecting to your metadata repository. You may also want to setup the `LOGGING_LEVEL_ROOT` environment variable to define a more granular logging level for your initial testing, e.g. `export LOGGING_LEVEL_ROOT=INFO` before running the startup command above, to receive deeper information on the startup. (You can also setup a similar variable to get even deeper information for just your portion of code by using your unique package name, e.g. `export LOGGING_LEVEL_ORG_ODPI_EGERIA_CONNECTOR_X_Y_Z=DEBUG`.)

Then configure the OMAG Server Platform to use your connector. Note that the configuration and startup sequence is important.

Start with just the following:

Enable the OMAG Server as a repository proxy

Enable the OMAG Server as a repository proxy by specifying your canonical `OMRSRepositoryConnectorProvider` class name for the `connectorProvider={javaClassName}` parameter and `POST`ing to:

```
http://egeriahost:8080/open-metadata/admin-services/users/myself/servers/test/local-repository/mode/repository-
proxy/connection
```

For example, in our Apache Atlas example we would POST with a payload like the following:

```
{
  "class": "Connection",
  "connectorType": {
    "class": "ConnectorType",
    "connectorProviderClassName": "org.odpi.egeria.connectors.apache.atlas.repositoryconnector.
ApacheAtlasOMRSRepositoryConnectorProvider"
  },
  "endpoint": {
    "class": "Endpoint",
    "address": "{{atlas_host}}:{{atlas_port}}",
    "protocol": "http"
  },
  "userId": "{{atlas_user}}",
  "clearPassword": "{{atlas_password}}"
}
```

Start the server instance

Start the OMAG Server instance by `POST`ing to:

```
http://egeriahost:8080/open-metadata/admin-services/users/myself/servers/test/instance
```

During server startup you should then see various messages related to the metadata type registration process as the open metadata types are checked against your repository. (These in turn call the methods you've implemented in your `OMRSMetadataCollection`.) You might naturally need to iron out a few bugs in those methods before proceeding further…

Test your connector's basic operations

Each time you change your connector code, you'll naturally want to re-build it (`mvn clean install`) and restart the OMAG Server Platform. If you are not changing any of the configuration, you can simply restart the OMAG Server Platform and re-run the `POST` to start the server instance (the last step above). If you need to change something in the configuration itself, it will be best to:

1. Stop the OMAG Server Platform.
2. Delete the configuration document (a file named something like `omag.server.test.config`).
3. Start the OMAG Server Platform again.
4. Re-run both steps above (enabling the OMAG Server as a proxy, and starting the instance).

From there you can continue to override other methods of the `OMRSMetadataCollectionBase` class to implement the other metadata functionality for searching, updating and deleting as well as retrieving other instances of metadata like relationships. Most of these methods can be directly invoked (and therefore tested) using the REST API endpoints of the OMAG server.

A logical order of implementation might be:

Read operations

getEntitySummary()

... which you can test through `GET` to:

```
http://egeriahost:8080/servers/test/open-metadata/repository-services/users/myself/instances/entity/
{{guidOfEntity}}/summary
```

getEntityDetail()

... which you can test through `GET` to:

```
http://egeriahost:8080/servers/test/open-metadata/repository-services/users/myself/instances/entity/
{{guidOfEntity}}
```

getRelationshipsForEntity()

... which you can test through `POST` to:

```
http://egeriahost:8080/servers/test/open-metadata/repository-services/users/myself/instances/entity/
{{guidOfEntity}}/relationships
```

... with a payload like the following (to retrieve all relationships):

```
{
  "class": "TypeLimitedFindRequest",
  "pageSize": 100
}
```

These are likely to require the most significant logic for any mappings / translations you're doing between the open metadata types and your own repository. For example, with Apache Atlas these are where we translate between the Apache Atlas native types like `AtlasGlossaryTerm` and its representation in the Apache Atlas java client and the open metadata type `GlossaryTerm` and its representation through the standard OMRS interfaces.

Search operations

The other main area to then implement is searching, for example:

findEntitiesByProperty()

... which you can test through `POST` to:

```
http://egeriahost:8080/servers/test/open-metadata/repository-services/users/myself/instances/entities/by-
property
```

... with a payload like the following (to find only those `GlossaryTerm`s classified as `SpineObject`s and whose name also starts with `Empl`):

```
{
  "class": "EntityPropertyFindRequest",
  "typeGUID": "0db3e6ec-f5ef-4d75-ae38-b7ee6fd6ec0a",
  "pageSize": 10,
  "matchCriteria": "ALL",
  "matchProperties": {
    "class": "InstanceProperties",
    "instanceProperties": {
      "displayName": {
        "class": "PrimitivePropertyValue",
        "instancePropertyCategory": "PRIMITIVE",
        "primitiveDefCategory": "OM_PRIMITIVE_TYPE_STRING",
        "primitiveValue": "\\QEmpl\\E.*"
      }
    }
  },
  "limitResultsByClassification": [ "SpineObject" ]
}
```

findEntitiesByClassification()

... which you can test through `POST` to:

```
http://egeriahost:8080/servers/test/open-metadata/repository-services/users/myself/instances/entities/by-
classification/ContextDefinition
```

... with a payload like the following (to find only those `GlossaryTerm`s classified as `ContextDefinition`s where the `scope` of the context definition contains `local` — note to change the classification type you change the end of the URL path, above):

```
{
  "class": "EntityPropertyFindRequest",
  "typeGUID": "0db3e6ec-f5ef-4d75-ae38-b7ee6fd6ec0a",
  "pageSize": 100,
  "matchClassificationCriteria": "ALL",
  "matchClassificationProperties": {
    "class": "InstanceProperties",
    "instanceProperties": {
      "scope": {
        "class": "PrimitivePropertyValue",
        "instancePropertyCategory": "PRIMITIVE",
        "primitiveDefCategory": "OM_PRIMITIVE_TYPE_STRING",
        "primitiveValue": "*local*"
      }
    }
  }
}
```

findEntitiesByPropertyValue()

... which you can test through `POST` to:

```
http://egeriahost:8080/servers/test/open-metadata/repository-services/users/myself/instances/entities/by-
property-value?searchCriteria=.%2A%5CQaddress%5CE.%2A
```

… with a payload like the following (to find only those `GlossaryTerm`s that contain `address` somewhere in one of their textual properties):

```
{
  "class": "EntityPropertyFindRequest",
  "typeGUID": "0db3e6ec-f5ef-4d75-ae38-b7ee6fd6ec0a",
  "pageSize": 10
}
```

and so on.

You hopefully have access to a search API for your repository so that you can efficiently fulfil these requests. You want to avoid pulling back a large portion of your metadata and having to loop through it in memory to find specific objects. Instead, push-down the search to your repository itself as much as possible…

Once you have those working, it should be relatively easy to go back and fill in areas like the other TypeDef-related methods, to ensure your connector can participate appropriately in a broader open metadata cohort.
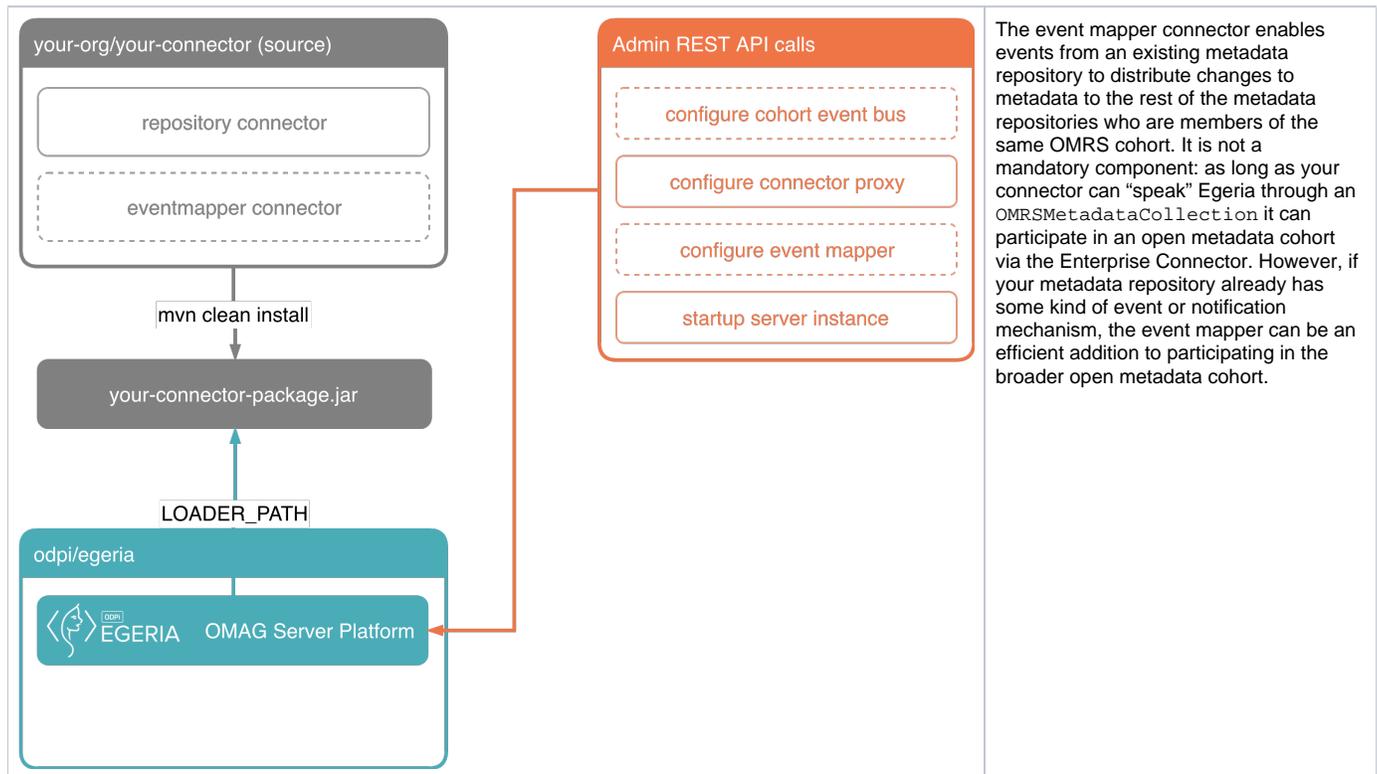
Write operations

While the ordering above is necessary for all connectors, if you've decided to also implement write operations for your repository there are further methods to override. These include:

- creation operations like `addEntity`,
- update operations like `updateEntityProperties`,
- and reference copy-related operations like `saveEntityReferenceCopy`.

If you are only implementing a read-only connector, these methods can be left as-is and the base class will indicate they are not supported by your connector.

Add an event mapper

The event mapper connector enables events from an existing metadata repository to distribute changes to metadata to the rest of the metadata repositories who are members of the same OMRS cohort. It is not a mandatory component: as long as your connector can "speak" Egeria through an `OMRSMetadataCollection` it can participate in an open metadata cohort via the Enterprise Connector. However, if your metadata repository already has some kind of event or notification mechanism, the event mapper can be an efficient addition to participating in the broader open metadata cohort.

Within the same `adapter` Maven module, perhaps under a new sub-package like `...eventmapper`, implement the following:

Implement an OMRSRepositoryEventMapperProvider

Start by writing an `OMRSRepositoryEventMapperProvider` specific to your connector, which extends `OMRSRepositoryConnectorProviderBase`. The connector provider is a factory for its corresponding connector. Much of the logic needed is coded in the base class, and therefore your implementation really only involves defining the connector class and setting this in the constructor.

For example, the following illustrates this for the Apache Atlas Repository Connector:

```
package org.odpi.egeria.connectors.apache.atlas.eventmapper;

import org.odpi.openmetadata.frameworks.connectors.properties.beans.ConnectorType;
import org.odpi.openmetadata.repositoryservices.connectors.stores.metadatacollectionstore.repositoryconnector.
OMRSRepositoryConnectorProviderBase;

public class ApacheAtlasOMRSRepositoryEventMapperProvider extends OMRSRepositoryConnectorProviderBase {

    static final String  connectorTypeGUID = "daeca2f1-9d23-46f4-a380-19a1b6943746";
    static final String  connectorTypeName = "OMRS Apache Atlas Event Mapper Connector";
    static final String  connectorTypeDescription = "OMRS Apache Atlas Event Mapper Connector that processes
events from the Apache Atlas repository store.";

    public ApacheAtlasOMRSRepositoryEventMapperProvider() {
        Class connectorClass = ApacheAtlasOMRSRepositoryEventMapper.class;
        super.setConnectorClassName(connectorClass.getName());
        ConnectorType connectorType = new ConnectorType();
        connectorType.setType(ConnectorType.getConnectorTypeType());
        connectorType.setGUID(connectorTypeGUID);
        connectorType.setQualifiedName(connectorTypeName);
        connectorType.setDisplayName(connectorTypeName);
        connectorType.setDescription(connectorTypeDescription);
        connectorType.setConnectorProviderClassName(this.getClass().getName());
        super.setConnectorTypeProperties(connectorType);
    }

}
```

Note that you'll need to define a unique GUID for the connector type, and a meaningful name and description. Really all you then need to implement is the constructor, which can largely be a copy / paste for most adapters. Just remember to change the `connectorClass` to your own, which you'll implement in the next step (below).

Implement an OMRSRepositoryEventMapper

Next, write an `OMRSRepositoryEventMapper` specific to your connector, which extends `OMRSRepositoryEventMapperBase` and implements `VirtualConnectorExtension` and `OpenMetadataTopicListener`. This defines the logic to pickup and process events or notifications from your repository and produce corresponding OMRS events. As such the main logic of this class will be implemented by:

- Overriding the `initialize()` method to define how you will initialize your event mapper. For example, this could be connecting to an existing event bus for your repository, or some other mechanism through which events should be sourced.
- Overriding the `start()` method to define how to startup the processing of such events.
- Implement the `initializeEmbeddedConnectors()` method to register as a listener to any `OpenMetadataTopicConnectors` that are passed as embedded connectors.
- Implement the `processEvent()` method to define how to process each event received from your repository's event / notification mechanism.

The bulk of the logic in the event mapper should be called from this `processEvent()` method: defining how events that are received from your repository are processed (translated) into OMRS events that deal with Entities, Classifications and Relationships.

Typically you would want to construct such instances by calling into your `OMRSMetadataCollection`, ensuring you produce the same payloads of information for these instances both through API connectivity and the events.

Once you have the appropriate OMRS object, you can make use of the methods provided by the `repositoryEventProcessor`, configured by the base class, to publish these to the cohort. For example:

- `repositoryEventProcessor.processNewEntityEvent(...)` to publish a new entity instance (`EntityDetail`)
- `repositoryEventProcessor.processUpdatedRelationshipEvent(...)` to publish an updated relationship instance (`Relationship`)
- and so on

To add the event mapper configuration to the OMAG Server Platform configuration you started with above, add:

Configure the cohort event bus

This should be done first, before any of the other configuration steps above, by `POST`ing to:

```
http://egeriahost:8080/open-metadata/admin-services/users/myself/servers/test/event-bus?connectorProvider=org.
odpi.openmetadata.adapters.eventbus.topic.kafka.KafkaOpenMetadataTopicProvider&topicURLRoot=OMRSTopic
```

... with a payload like the following:

```
{
  "producer": {
    "bootstrap.servers":"kafkahost:9092"
  },
  "consumer": {
    "bootstrap.servers":"kafkahost:9092"
  }
}
```
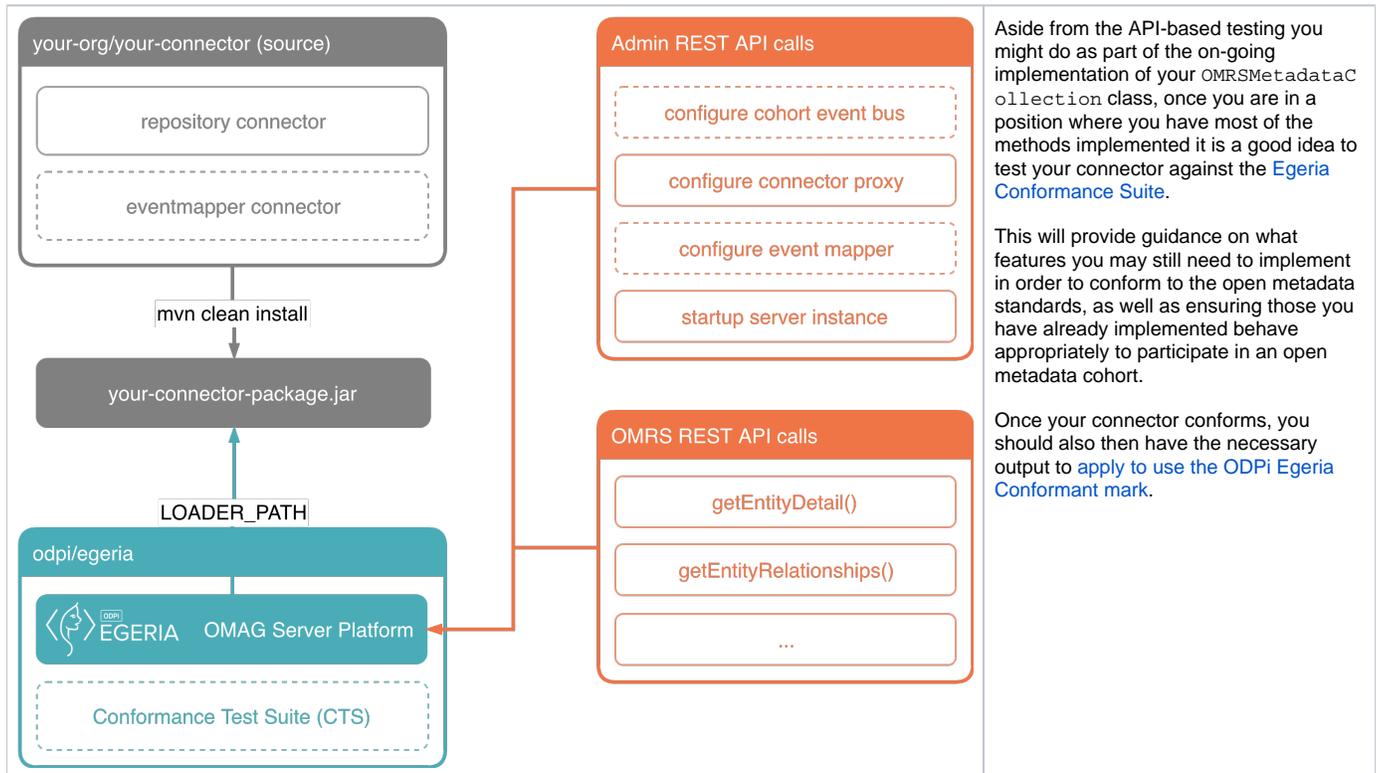
Configure the event mapper

This can be done nearly last, after all of the other configuration steps above *but still before the start of the server instance*. Specify your canonical `OMRSRep ositoryEventMapperProvider` class name for the `connectorProvider={javaClassName}` parameter and connection details to your repository's event source in the `eventSource` parameter by `POST`ing to:

```
http://egeriahost:8080/open-metadata/admin-services/users/myself/servers/test/local-repository/event-mapper-
details
```

For example, in our Apache Atlas example we would `POST` to:

```
http://egeriahost:8080/open-metadata/admin-services/users/myself/servers/test/local-repository/event-mapper-
details?connectorProvider=org.odpi.egeria.connectors.apache.atlas.eventmapper.
ApacheAtlasOMRSRepositoryEventMapperProvider&eventSource=atlashost:9027
```

Test your connector's conformance

Aside from the API-based testing you might do as part of the on-going implementation of your `OMRSMetadataC ollection` class, once you are in a position where you have most of the methods implemented it is a good idea to test your connector against the Egeria Conformance Suite.

This will provide guidance on what features you may still need to implement in order to conform to the open metadata standards, as well as ensuring those you have already implemented behave appropriately to participate in an open metadata cohort.

Once your connector conforms, you should also then have the necessary output to apply to use the ODPi Egeria Conformant mark.

Related articles

- How to find entities and relationships
- Implement an Open Metadata Repository Connector