# **High level Connector module design**

## Main purpose

The primary goal of the connector is to integrate Arno with any third-party tool. The idea is that the connector module will allow share Arno data across a variety of external systems by providing the most standard possible way to “import” and “export” data.

The connector would also allow to access the Arno engine (Dimona, Payroll calculation, ASR) from any external system as a service.

## Data Interface

This interface is responsible for synchronizing data with external systems and can work in both inbound and outbound mode.

In the inbound mode, an external system should be able to call a routine which will allow it to pull data from Arno. In the outbound mode, the connector will push data from Arno to the external system.

### Inbound mode

The connector will provide a web api access to any allowed external system. In this scenario we can consider the use of OData standard. This standard defines some set of best practices and rules for building and consuming web api.

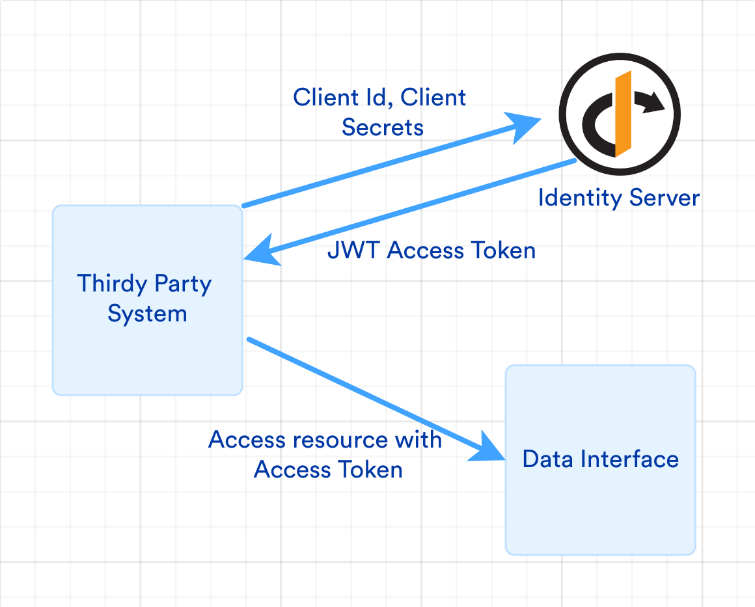
More information can be found here <https://www.odata.org/>

This standard allows the following when querying over data

* Projection (selecting a subset of properties)
* Filtering
* Expanding (selecting linked entities)
* Sorting

Some important points need to be considered when designing this interface.

* **Multi tenancy**: this interface should allow access to different tenants (library). For example, an access to the list of individuals should be achieved through the following url [http://arno.webapi/{library}/odata/individus](http://arno.webapi/%7blibrary%7d/odata/individus) where {library} refers to the library id.
* **Security** : access to the api must be allowed to granted systems only. This implies the use of an Identity Server with a client credentials flow setup. The client will clearly identify which is the calling external system. In addition, we need to make sure the client is allowed to access the selected library. The security settings need to be configurable and follow the steps below
  + - Create a client id, client secrets
    - Select the list of allowed libraries



Another alternative would be the use of GDS (an existing Arno feature used to export/import data). This can be still achieved through a web api. An external system submits a query that can be interpreted and processed. This suggest that the caller has a very good knowledge of the GDS format. This also means that the processing of a GDS file must be accessible to the connector module via an API (Darius?). The same security considerations that apply to OData alternative applies also for GDS.

General considerations

* Darius needs to be redesigned as a middleware without having any dependency to Arno. We can also think of a complete rewrite that targets .Net Core.
* The actual solution used to deploy new modules need also to be simplified in order to be reused within the new api layer. (The removal of pipe communication for example).

### Outbound mode

In this mode, the connector module will call external api to pull data from external system into Arno.

A simple scenario to this would be synchronizing individuals from mpleo using the url <https://demoarnohr.mpleo.net/ws/employee>

Because of the diversity of the different sources and their technical implementations details, we can use an api gateway that will act as an abstraction layer. This layer will provide a better isolation, improve the security, and reduce the complexity.

The api getaway will also be associated to a mapping system that will associate the incoming data to the target data dictionary. This can be configurable.

|  |  |
| --- | --- |
| Incoming data format | Mapping mechanism |
| JSON | Map json tokens |
| XML | Use XPath expressions |

Below a small example from mpleo.

{

"employeeCode": "00001",

"nationalNumber": "70061531370",

"lastName": "Gijs",

"firstName": "Christiane",

"title": "MAD",

"birthPlace": "Roeselare",

"emailProfessional": "gijs.christiane@mpleo.be",

"emailPersonal": "christiane.gijs@gmail.com",

}

This would result in the following map

|  |  |  |
| --- | --- | --- |
| Source | Target | Is Key |
| “emloyeeCode” | V100.F101 | N |
| “nationalNumber” | V100.F105 | Y |
| “lastName” | V100.F102 | N |
| “firstName” | V100.F103 | N |
| “birthdate” | V100.F108 | N |
| “emailProfessional” | V100.L9943.F101 | Y |
| “emailPersonal” | V100.L9941.F101 | Y |

The flag “Is Key” will be used by the system to check whether this is an existing record to update. At least one target should be specified as a key.

This system can be also coupled with a background processing platform (like Hangfire) which can allow running a specific task or job based on a schedule. We can consider for example synchronizing the list of persons each start of a month. Hangfire is compatible with CRON expressions.

## Engine Interface

This purpose of this interface is to provide Arno features as a service. It could be for example a payroll, Dimona declaration or a sickness qualification.

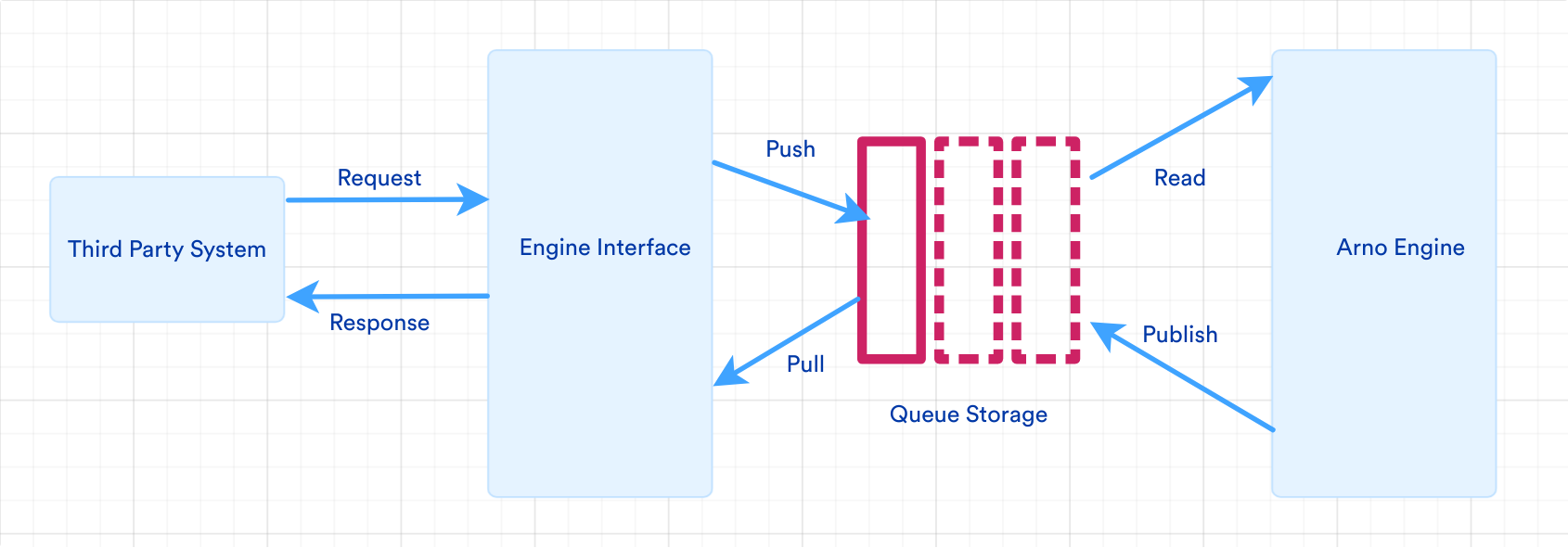
Because these can be considered as long running operation, which is not really adapted for a web api, the idea, is that the system will only push requests to a queue using a message broker platform (like RabbitMQ or Azure Service Bus). The request pushed to the queue can then be later processed and as a result of the process, a new message containing the result can be pushed back to an outgoing queue.

The use of a messaging system provides an asynchronous processing and a strong decoupling between systems.

Depending on the used technology, reading messages from the queue can be event driven. The use of a background processing can also be considered to schedule the treatment of incoming messages.

The external system that initiated the query does not to know about those implementations details, it simply uses the web api to create request and download responses.

The same security considerations that apply to the data interface applies also here.



Request definitions can be reused from the actual “Acerta.Arno.Infrastructure.Arno” project.

Below an example of a request body for a dimona declaration.

{"PersonsVolumeItemPointer":113636848,"BatchMode":false,"TaskType":0,"Module":0,"SubType":0,"Library":"ARNOHR\_TEST"}

## The different scenarios explained by a schema

In the scenario above, the connector initiates a request to access an Mpleo web api. The request is initiated from “Hangfire” and passer to the Api manager which routes the request to Mpleo Api. The http response from the api is then returned back to the initiator which maps and validates the data to the target Arno data dictionary and proceed with database changes through Darius.

## 

A third-party system, requests data from Arno. The request is addressed to Api Manager which will first checks the identity of the caller (subscription, client id, etc.) then routes it to the Arno Web Api. The Arno Web Api processes the request and sends back the data to original caller through the Api Manager. The format of the http response could be Xml/Json or any other format supported.

## 

This scenario reflects the access to the engine interface. It’s the only scenario where a service bus component is involved to handle long running operations. The external system submits a request (could be a dimona declaration) through the Api Manager which will simply push a message to an inbound queue. Hangfire will then read incoming messages and based on its definition runs the corresponding module and task. It acts like the actual implementation of the module host. The result of the process is then saves back to the outbound queue and can be later be accessed on demand by the caller through the Api Manager.