# Intro

Data Workbench uses external data sources in order to provide data from those sources to customers/users in a unified form. These data sources are currently provided and maintained by various departments inside DNV. They expose certain REST APIs which are then utilized by Data Workbench.

In order to integrate with such an API, the notion of Connector was introduced in the Data Workbench. Connector embodies an integration layer between a data provider and the Data Workbench. It is authorized at a provider’s system following an existing authorization policy on the provider’s side and it uses an API exposed by the provider in order to query data. Received data is then transformed to a format used by Data Workbench. Also, when the provider offers functionality for downloading binary data, this integration layer is responsible for storing this data into cloud file stores for further use by customers/users.

# Standardized External Connector

Data Workbench offers a few Connectors implemented by the Data Workbench team. These connectors wrap APIs of external data providers. Since those APIs are completely different, the Connectors by implication must be rather different too. Hence, we are talking about custom connectors. This means that they have to query data according to the specific API of the providers, while their output is uniform in order to give customers seamless experience both in the Data Workbench portal and in the 'Share as API' feature.

The next step is to give providers a tool for seamless integration with the Data Workbench. This means effectively that providers can follow certain requirements which will help to set up a new Connector without new custom implementation in the Data Workbench. New connectors can be added to the Data Workbench as plug-ins.

The Data Workbench team has designed a standardized API for external providers. This API consists of a few endpoints. Some of these endpoints are mandatory, while others are optional. These endpoints are assumed to follow a standardized predefined protocol which covers:

* REST API verbs
* paths
* contracts for payloads: both request payloads and response payloads

External provider implements API according to this protocol and hosts it in its own environment. Data Workbench should be able to access this external API. The latter means that if there any firewalls at the provider’s host, they should allow queries coming from the Data Workbench.

# Main notions

Here we describe a few main notions from the Data Workbench Connector subsystem. Some of them are reflected in the standardized external API and thus important to be aware of. Some of them are not reflected in the API but are helpful for understanding how the whole integration works.

## Schema

Schemas are not exposed in the External API. But it's nice to have some basic knowledge about their existence.

As was mentioned earlier, at the current stage Data Workbench works with tabular data. Schemas in the Data Workbench describe such tables. They define each column of a correspondent table: name of the column, its type, if it can be used for filtering and if yes then how (types of possible conditions).

It's easy to imagine a provider which offers data of different nature. So, then it can be represented as a few tables with own schemas. In such a case we will configure a Schema for each 'table'. Essentially, we can have multiple schemas associated with the same data provider.

Schemas must be created prior to creation of a Connector. Currently this can be done only via internal API which is not exposed for public use (i.e., only DW team can create schemas). However, in feature such API will be exposed publicly. We have plans for adding helper classes for public API in the Connector SDK.

## Connector

Connector identifies a single data provider. When we create an instance of a new connector, we configure it with the following metadata:

* all technicalities needed for server-to-server communication, e.g., base URL of a provider's API, server-to-server API keys, etc.
* references to schemas supported by this type of Connector together with custom settings per each schema (more on this below)
* definition of settings which must be provided when a new connection for the given connector is being created (more on this below)

As mentioned above, before we can create a new Connector, we should have created all relevant schemas.

Connectors can be created only via internal API. This functionality is not going to be exposed publicly.

## Connection

Connection binds together a Connector and a Tenant, or to be more precise, a Connector and a tenant's Workspace. You can think Connection as a data channel of a given Connector's type allocated to a given Tenant.

When we create an instance of a new connection, we should specify a concrete Connector, a workspace (~ a Tenant), and values of connection settings required by the given Connector (more on this below).

Currently connection can be created only via internal API which is not exposed for public use (i.e., only DW team can create connections). However, in feature such API will be exposed publicly. We have plans for adding helper classes for public API in the Connector SDK.

# Restricting access to the provider's data

Every data provider most likely restricts access to its data. There may be some restrictions scoped to tenant and/or to users at tenant's organization.

As mentioned earlier, Connection is related to a specific tenant. So, this is a level where certain restrictions on data access can be imposed. In terms of API, usually data access restrictions are represented by profile IDs, access tokens, API keys, etc. Every data provider has its own model of access policies. The total number, names and types of used authorization keys/IDs may vary from provider to provider. External Connector API is assumed to satisfy the needs of various providers. Thus, authorization mechanism is configured across Connector and Connection in a generic way.

1. When an instance of a Connector is being created, it can be configured with definition of settings which are mandatory for each connection of this Connector. It's an array of items with the following schema:  
     
    {  
    "key": <string>,  
    "name": <string>,  
    "description": <string>  
    }  
     
   As you can see, this structure only describes a setting, it does not provide a value. As soon as connection settings are defined in a Connector, they become mandatory in all connections created from this connector. Thus, these settings can describe such mandatory data as authorization tokens. For example:  
     
   "connectionSettings": [  
    {  
    "key": "ProfileId",  
    "name": "Profiled ID",  
    "description": "Profile ID of a user in Contoso Ltd"  
    },  
    {  
    "key": "AccessToken",  
    "name": "Access Token",  
    "description": "Access token of a user generated by Contoso Ltd auth server"  
    }  
   ]
2. When an instance of a Connection is being created, it should be configured with settings together with respective values in accordance with their definition in the Connector:  
   "connectionSettingValues": [  
    {  
    "key": <string>,  
    "value": <string>  
    },  
    …  
   ]  
     
   If we take the example from above, connection settings can look like this:  
     
   "connectionSettingValues": [  
    {  
    "key": "ProfileId",  
    "value": "test\\MyID"  
    },  
    {  
    "key": "AccessToken",  
    "value": "gfdhjkgfjlew8r327u09opjdkvnsm"  
    }  
    ]

Connection settings can be used not only for authorization. Basically, any metadata specific for a pair Tenant-Connector can be stored there and added to data query requests for consumption by provider. For example, this can include certain references to a provider's environment, etc.

It's important to emphasize here that essentially connection settings are represented by a dictionary (string ↔ string). This means that settings can contain any data, they are a convention between the provider and the Data Workbench. When a new Connector is configured with certain mandatory connection settings, it's assumed that the provider will understand what these settings mean when it will be receiving them in data queries. Basically, prior to creating a connector, the provider must decide what kind of additional data **specific to the given tenant** it will need in data queries: auth data, environment data, etc. Then this metadata can be stored in the connection.

Below it will be described how connections settings are used with the External API.

# Matching external data sources with schemas

Each data provider may provide access to data of a different nature. In the Data Workbench we describe it with the help of schemas which can be thought of as tables' schemas. But how can provider know which internal data source (table, API, 3d party provider, etc.) to query in order to return data needed by given schema in the Data Workbench? For this we are going to use Schema settings in the Connector.

As mentioned earlier, when we create a new Connector, among other things we configure it with references to schemas supported by this type of Connector together with custom settings per each schema. Custom settings are essentially a key-value dictionary with items defined according to this schema:

{

"key": <string>,

"value": <string>,

"type": <string>,

"description": <string>

}

For example, let's assume that we are setting up a Connector for hypothetical IoT data provider. This provider internally distinguishes its data by type of API and by set of sensors. In the Data Workbench we want to create two schemas: one for IoT data from oil tankers where sensors are installed on a surface of vessels (whatever that means...), and another for IoT data from container ships where sensors are installed in a captain's cabin.

1. We create two schemas in the Data Workbench. Each has a unique ID, name and a set of columns with their metadata. Let's assume that the schema for tankers has got ID="ABC123" and the schema for container ships has got ID="DEF987".
2. We create a Connector for IoT provider. Here we reference the schemas from above. But we also add custom settings for each of these schemas:  
   "supportedSchemaVersions": {  
    "ABC123": [  
    {  
    "key": "API",  
    "value": "TankersApi",  
    "type": "string",  
    "description": "IoT provider will use its internal Tankers API for this schema"  
    },  
    {  
    "key": "SensorsSet",  
    "value": "Surface",  
    "type": "string",  
    "description": "IoT provider will query data from Surface sensors set"  
    }  
    ],  
    "DEF987": [  
    {  
    "key": "API",  
    "value": "ContainerShipApi",  
    "type": "string",  
    "description": "IoT provider will use its internal Container Ships' API for this schema"  
    },  
    {  
    "key": "SensorsSet",  
    "value": "CaptainCabin",  
    "type": "string",  
    "description": "IoT provider will query data from sensors set in a captain's cabin"  
    }  
    ]  
   }
3. When the Data Workbench queries data from the IoT provider, it queries it for a certain schema. So, a request's payload among other things contains also settings specific for this schema. These settings are fetched from the connector configuration.
4. The provider when receives a query, uses these schema settings in order to figure out how it should satisfy the request: which API to use, or which database, or which sensors set, etc.

Same as with connection settings, it's important to mention here that these custom settings are represented by a simple dictionary (string ↔ string). This means that settings can contain any data, they are a convention between the provider and the Data Workbench. When a new Connector is configured with certain custom schema settings, it's assumed that the provider will understand what these settings mean when it will be receiving them in data queries. Basically, prior to creating a connector, the provider must decide what kind of metadata **specific for the data query** it will need in requests. Then this metadata can be stored in the connector.

# External API

External API is a set of API endpoints which should be exposed by the provider. This API will be used by the Data Workbench for establishing a connection and for querying data. Currently we assume REST API, but later some other protocols can be introduced (e.g., gRPC).

External API can be divided into two groups: mandatory and optional.

## Mandatory API

### *Health check*

The endpoint has the following path:

**GET /healthcheck**

The endpoint will be called on attempt to create or update a connector. At this point the Data Workbench needs to verify that with all provided settings it can reach the provider’s API. This way all essential settings will be verified, e.g., a base URL, potentially some server-to-server authentication mechanisms like API-key, subscription key, etc.

Also, this endpoint may be called regularly for a general status check of all dependencies.

This endpoint doesn’t have any contracts either for input or for output. It is only supposed to return HTTP status 200.

#### Connection Validation

The endpoint has the following path:

**POST /validate**

This endpoint is supposed to be called on a new connection creation or on update of an existing connection. Its purpose is to validate connection settings at the provider's system before they will be persisted in the Data Workbench internal storage.

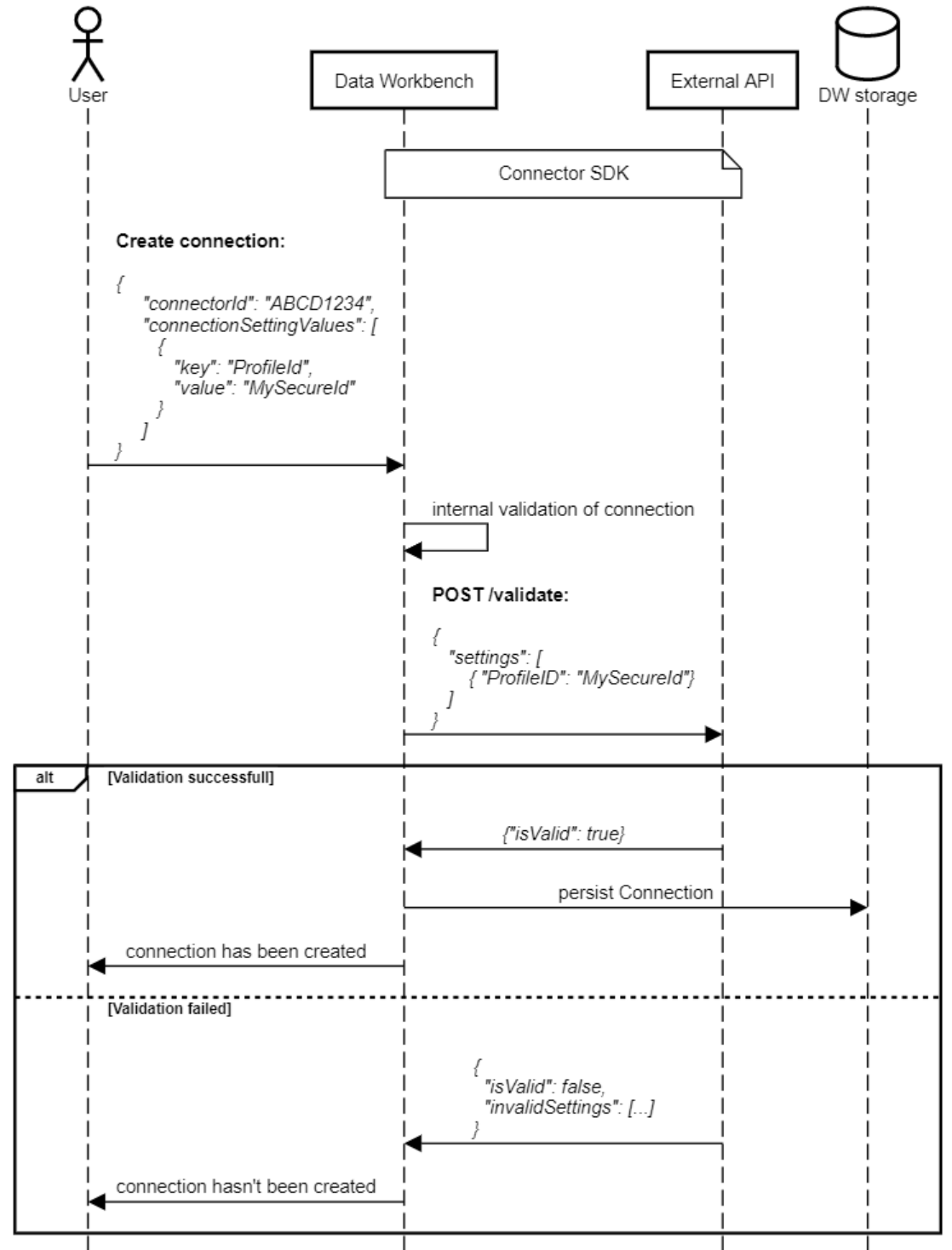
Input:

* connection settings (an array of key-value pairs)

Output:

* validation result (successful or not)
* detailed information about validation failures (if there were any).

As it was explained above, a connection object in the Data Workbench keeps data specific for a pair Connector-Tenant. It can be auth data (IDs, tokens, etc.), certain references specific to environment, and so on. At that moment when a new connection is being created in the Data Workbench, all provided settings have to be validated by the data provider. This means, for example, that provided auth tokens will be checked if they are authentic and if they authorize access to certain data in the provider's system. If this validation fails, the connection can't be created, and thus querying data will be impossible. If the validation is successful, the connection settings will be persisted in the Data Workbench. On later stages, when Data Workbench will query data for the provider, these settings will be added to requests' payloads. Then the provider can use them for data access control.



### *Data Query*

The endpoint has the following path:

**POST /query**

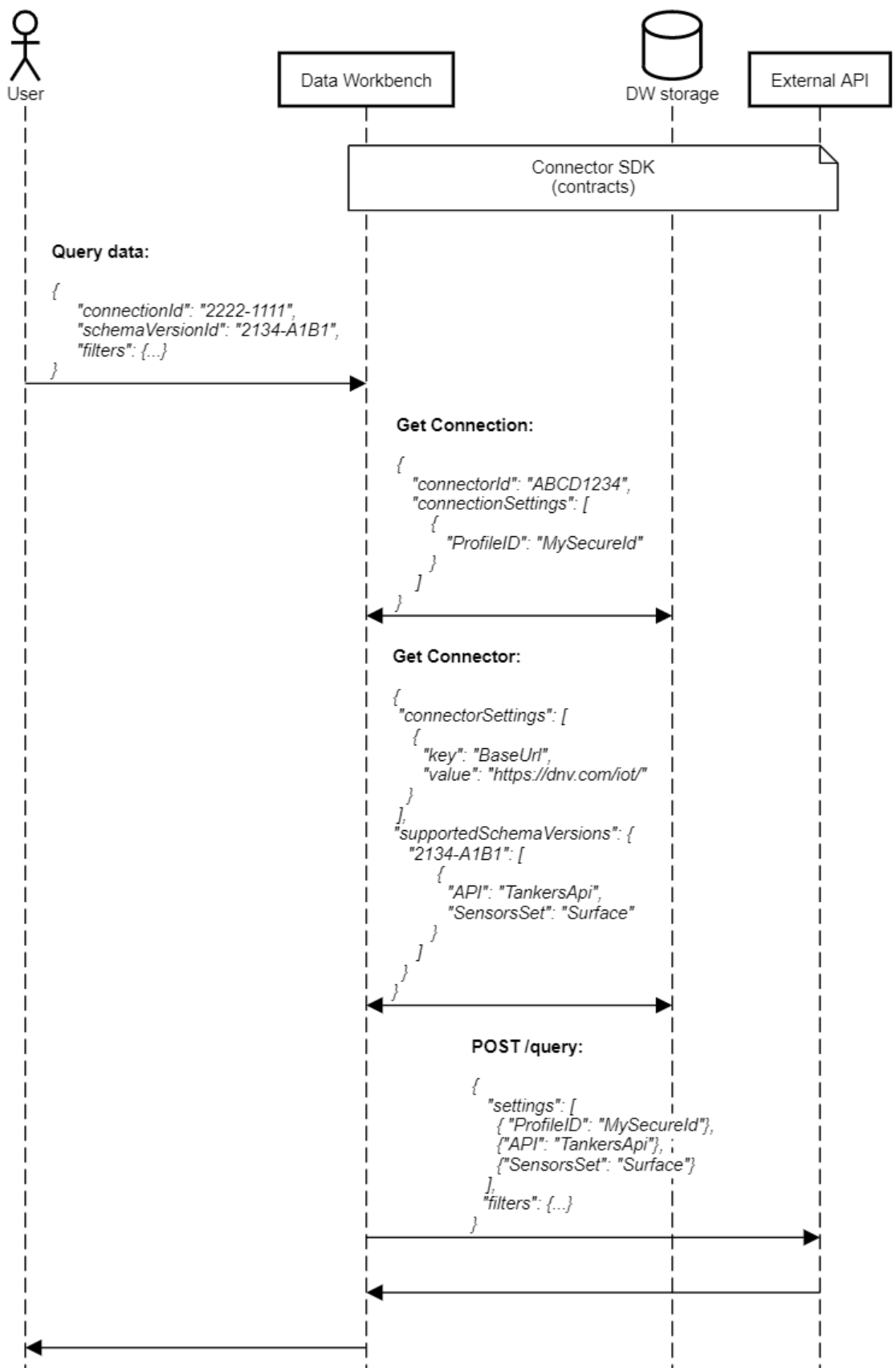
This endpoint is supposed to be called when data is queried.

Input:

* settings (an array of key-value pairs). These settings consist of connection settings (can be used for access control) and of settings specific to schema (can be used as a metadata specific to data source).
* columns (an array of strings). Specify which columns from a given data source should be available in a response.
* filters (an array of objects describing filtering condition per column). Specifies filters for data. For a given column it specifies logical operator which should be applied to values of this column as well as argument(s) used in a required logical operation.
* pagination parameters

Output:

* columns (an array of strings). An array of column names.
* rows (an array of arrays). Here relevant data satisfying the query is returned. It is represented by an array of rows. Each row in its turn is an array of data. Data in rows must be ordered according to the order of columns. Type of data in rows must match corresponding type in schema (e.g., numbers should be presented as numbers, not as strings).
* pagination information



### *Data Discovery*

The endpoint has the following path:

**POST /discovery**

This endpoint is supposed to be called for discovering which filter options are available in a data source.

Input:

* settings (an array of key-value pairs). These settings consist of connection settings (can be used for access control) and of settings specific to schema (can be used as a metadata specific to data source).
* columns (an array of strings). Specify which columns from a given data source should be available in a response.
* filters (an array of objects describing filtering condition per column). Specifies filters for data. For a given column it specifies logical operator which should be applied to values of this column as well as argument(s) used in a required logical operation.
* pagination parameters

Output:

* summary for found data. This consists of a time period covering available data, total number of assets and of records, data classification.
* available filtering options: array of objects per each column which can be used for filtering. Each object contains a summary for a given column: vessel name, time period, total number of records, data classification information.
* pagination information

### Optional API

### *File/Document Download*

The endpoint has the following path:

**POST /file/{fileId}**

This endpoint is supposed to be called for downloading a file. Not all data providers offer such functionality, hence this endpoint is optional.

Input:

* ID of the file to be downloaded. It is represented by string. But the value depends on a provider's way of identifying files (numerical IDs, GUIDs, etc.)
* settings (an array of key-value pairs). These settings consist of connection settings (can be used for access control) and of settings specific to schema (can be used as a metadata specific to data source).

Output:

* the file should be returned as a binary array
* a response should have a header ContentType (e.g. "application/pdf")
* a response should have a header ContentDisposition. The header must contain "FileName" key-value pair (e.g., Content-Disposition: attachment; filename="filename.pdf")

# Connector SDK

Connector SDK is distributed as a NuGet package.

## Contracts

A part of the SDK is contracts for External API. The contracts are defined as classes. For each endpoint there are contracts describing an input (request’s payload) and an output (response’s payload). Certain properties in contracts can be required while others are optional. This is documented directly in the classes’ declarations.

## API

All endpoints described above are presented in the SDK as interfaces. Each endpoint is defined in its own interface. These interfaces bind together input and output contracts. It is recommended to implement the API as implementations of these interfaces. Otherwise, they can be used as a reference.

# Authentication of the Data Workbench

Here we are talking about authentication of the Data Workbench as an application in the provider’s Web API application.

API in general have many different ways to secure access to itself from other API/applications. In the Data Workbench we are working on adding support for some of them. Currently the following options can be used.

### Firewall + Whitelisting

The External API is supposed to be protected from access from outer world by a firewall. And the firewall must have a so called ‘whitelist’ which allows access to the API for requests sent from a given list of IPs or DNS. When an IP/DNS of the Data Workbench is added to this list, the Data Workbench can send requests to the provider’s API.

This way the access is secured on hardware/infrastructure level, not on application level. With this option, Data Workbench doesn’t need to do anything: from its perspective the External API is open and doesn’t require any authentication.

### API Management + OAuth B2C

This is an authentication mechanism used in Microsoft Azure. Essentially, it consists of two mechanisms, as it is reflected in this section’s name.

*API management* part requires from each HTTP request to include a subscription key. It has some name and a value. Both can be configured when an instance of Connector is created/edited.

*OAuth B2C* is described here: <https://learn.microsoft.com/en-us/azure/api-management/howto-protect-backend-frontend-azure-ad-b2c>

In order to use this mechanism, a configuration of a Connector should include Client ID, Client Secret, Scope, Authority URL and Tenant ID