Watsonx.governance
Integration with external Systems

Lab Exercise Guide

(Version 1 | January, 2025)

Andreas Christian
Senior Technical Sales Specialist
achristian@de.ibm.com

IBM

Notices and disclaimers

U.S. Government Users Restricted Rights — use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM.

This document is current as of the initial date of publication and may be changed by IBM at any time. Not all offerings are available in every country in which IBM operates.
Information in these presentations (including information relating to products that have not yet been announced by IBM) has been reviewed for accuracy as of the date of initial publication and could include unintentional technical or typographical errors. IBM shall have no responsibility to update this information.

This document is distributed "as is" without any warranty, either express or implied. In no event, shall IBM be liable for any damage arising from the use of this information, including but not limited to, loss of data, business interruption, loss of profit or loss of opportunity. IBM products and services are warranted per the terms and conditions of the agreements under which they are provided. The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

IBM products are manufactured from new parts or new and used parts.
In some cases, a product may not be new and may have been previously installed. Regardless, our warranty terms apply."

Any statements regarding IBM's future direction, intent or product plans are subject to change or withdrawal without notice.

Performance data contained herein was generally obtained in a controlled,
isolated environments. Customer examples are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual performance, cost, savings or other results in other operating environments may vary.

References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business.

Workshops, sessions and associated materials may have been prepared by independent session speakers, and do not necessarily reflect the views of IBM. All materials and discussions are provided for informational purposes only, and are neither intended to, nor shall constitute legal or other guidance or advice to any individual participant or their specific situation.

It is the customer's responsibility to ensure its own compliance with legal requirements and to obtain advice of competent legal counsel as to the identification and interpretation of any relevant laws and regulatory requirements that may affect the customer's business and any actions the customer may need to take to comply with such laws. IBM does not provide legal advice or represent or warrant that its services or products will ensure that the customer follows any law.

Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products. IBM does not warrant the quality of any third-party products, or the ability of any such third-party products to interoperate with IBM's products. **IBM expressly disclaims all warranties, expressed or implied, including but not limited to, the implied warranties of merchantability and fitness for a purpose.**
The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents, copyrights, trademarks or other intellectual property right.

IBM, the IBM logo, and ibm.com are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at:
www.ibm.com/legal/copytrade.shtml.

Table of Contents

1 Introduction

Watsonx.governance™ helps customers to automate and accelerate responsible AI workflows to help save time, manage risks and comply with regulations. This workshop provides a high level introduction to the available APIs (REST API, Java API) of the *Governance Console* (a.k.a. *OpenPages*). It helps you to understand how these APIs can be used to integrate Watsonx.governance with external systems.

Note: The lab exercises walk you through some code examples which are easy to understand. You don't need programming skills to complete this lab.

The lab mainly summarizes the knowledge we gained during a project with one of the leading banks in Europe. The bank wanted to implement a *Model Risk Management* system both for ML models and also for classical risk models. A classical risk model may, for example, estimate the likelihood that a borrower defaults within a specific time frame.

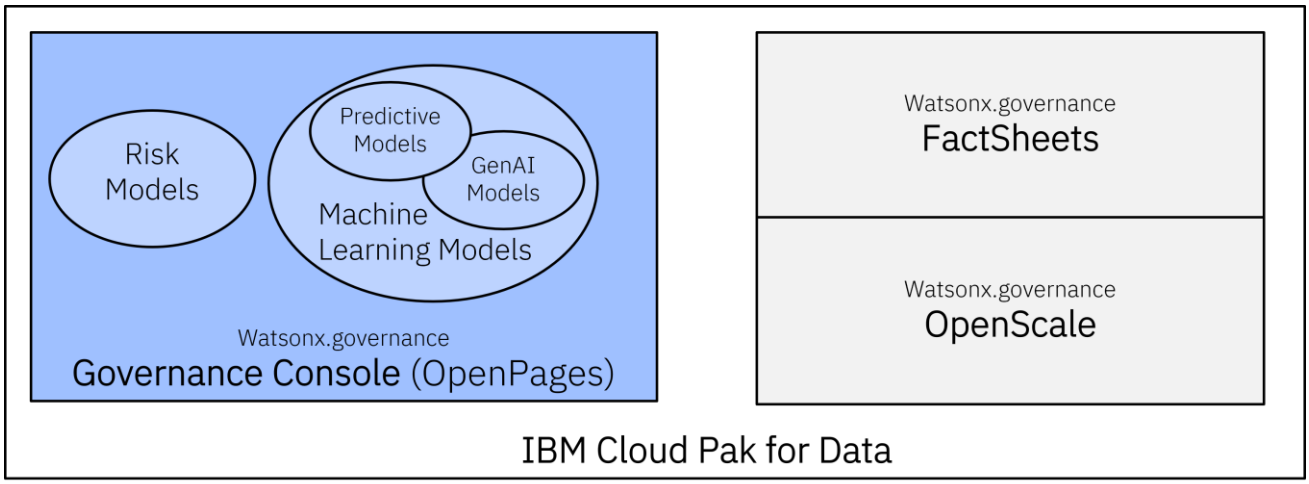


Figure 1: IBM solution components proposed to the bank are marked blue

The lab consists of two parts:

1. **Introduction to the REST API:** Among other topics, you learn how the API can be used by external systems to create/query objects (for example "model" and "use case") in the inventory of the Governance Console.
2. **Introduction to the Java API:** You learn, how custom workflow actions can be used to trigger tasks in other systems. For example, if a user approves a "model development request" as part of a workflow in the Governance Console, this step can trigger an activity in an external system.

Note: Throughout this document "WxGov" is used as abbreviation for IBM Watsonx.governance.

1.1 Typical Customer Requirements

Some of the main requirements we received from the bank are listed below together with our answers. They are all related to the ability of the Governance Console to be **integrated with external systems**. The lab exercises explain how these requirements can be implemented using the available APIs of the Governance Console.

Requirement	Answer
Connections from WxGov to the bank's databases or databases in a third-party cloud.	**1:** Java API: Can be used to connect to external databases and to execute for example SQL queries.
Connectivity to WxGov via API interfaces is possible for the bank's systems.	**2**: REST API: Can be used by external systems to create and retrieve Model Risk Governance objects stored in the inventory of the Governance Console.
Integration into the bank's ML Ops platform/ process: Content (e.g. model documentation) pushes out from the bank's Model Build Environment to WxGov. Workflow steps (e.g. approvals) in IBM WxGov can be used for the bank's deployment process as gate.	**3a**: REST API: Documents can be imported to the inventory of the Governance Console via the REST API. **3b**: Java API: workflow stages can trigger custom workflow actions. For example, when moving from stage „Proposed" to „Approved" custom Java code can execute a REST call to the external system. **3c**: REST API: The current status of Model Risk Governance (MRG) objects (e.g. approval status of models and use cases) can be checked via the REST API.
Connects with other tools of the bank: Connections via REST API to the bank's systems.	**4**: Java API: Custom Java code can execute REST calls to external systems.
Configuration via API	**5**: REST API: Provides some configuration capabilities.
Read / Write Data via API	**6**: REST API: REST calls can read/write MRG objects
Workflows in WxGov can be triggered from the bank's systems.	**7**: REST API: Can trigger workflows for the objects stored in the inventory.

Figure 2: Sample customer requirements regarding Watsonx.governance system integration capabilities

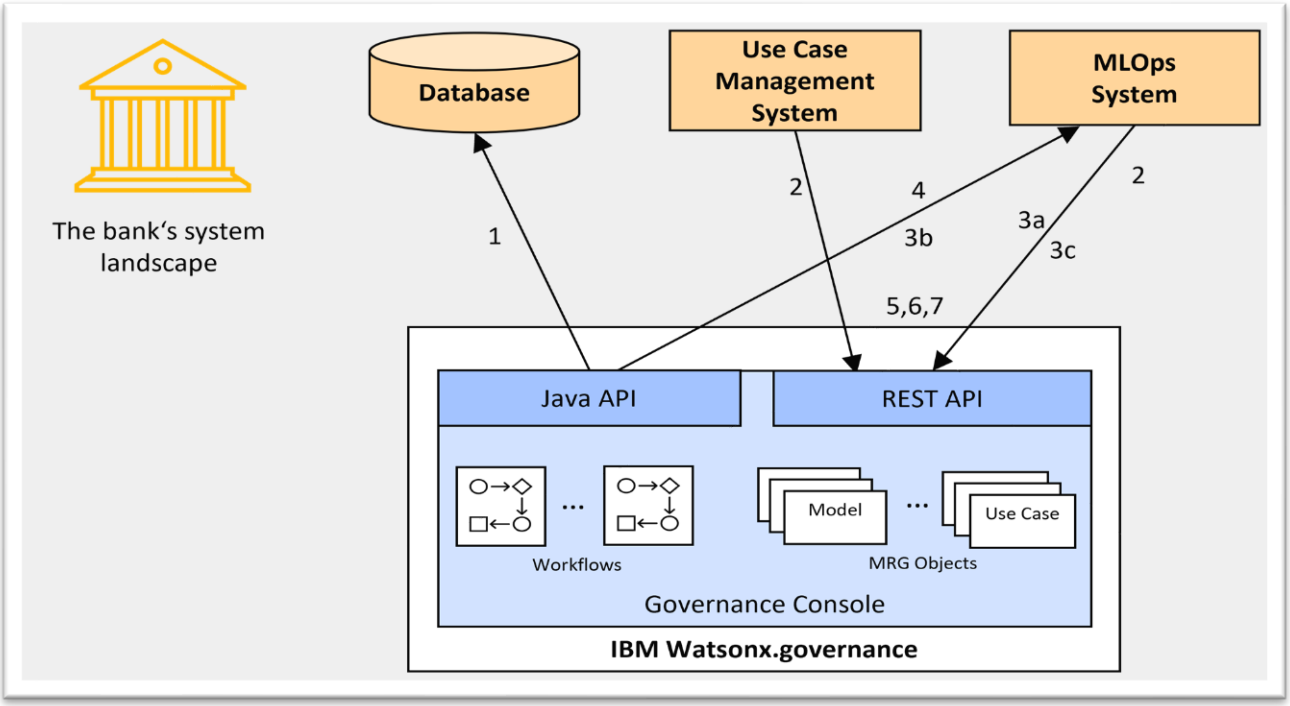


Figure 3: Some options to integrate Watsonx.governance with external systems

1.2 Lab Overview

The lab exercises consist of the following parts:

- **REST API lab**: This part of the lab is based on a Jupiter notebook (python code) which contains different types of REST calls to work with the different types of objects in the Governance Console. You will access Watson Studio via the IBM Cloud Pak for Data console using your local web browser. As part of the exercises you will query the objects in the Governance Console and you will create new objects like business entities, models and use cases together with their relationships.
- **Java API lab**: This part of the lab provides an introduction to custom workflow actions. You will create a small workflow, deploy a custom workflow action (Java program) and then execute that workflow. The custom workflow action performs a REST call to a web server that is running on the Virtual Machine (VM) of your lab environment.

The picture below shows the different components of the lab environment.

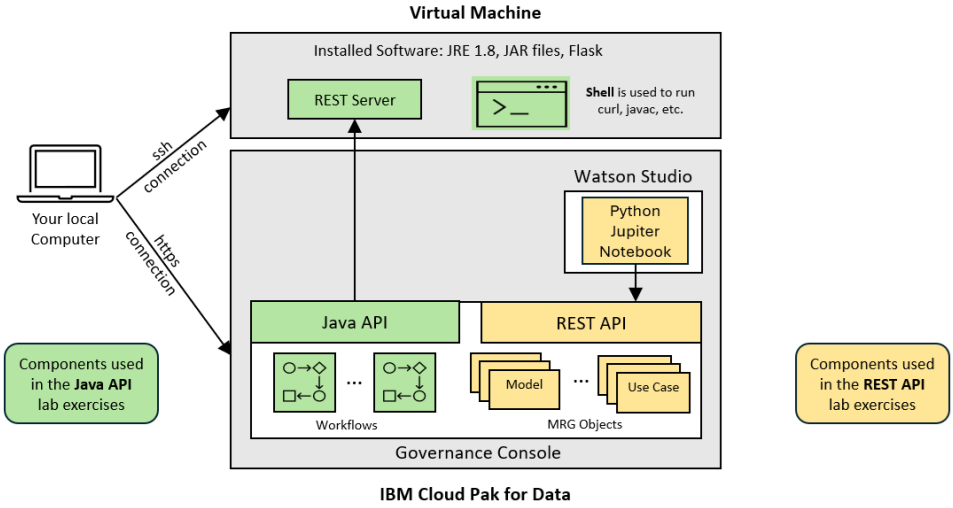


Figure 4: Lab environment components

Note: The lab environment provides the advantage that you can define the business logic of your custom workflow actions inside the REST server (that means outside of the Cloud Pak for Data cluster). This is shown in the picture below.

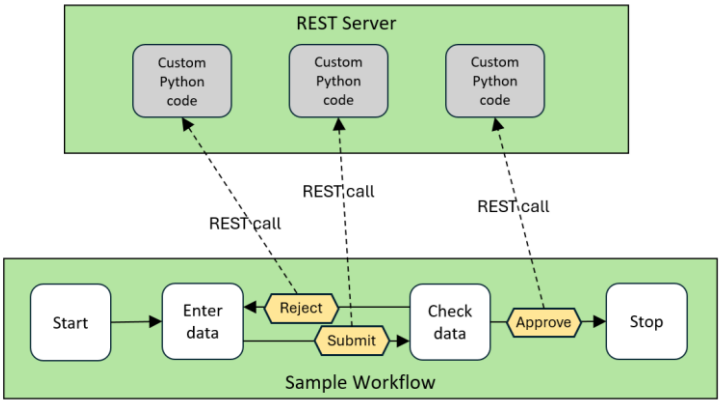


Figure 5: Sample custom workflow actions (yellow boxes)

This approach has several advantages:

- You can use Python code instead of Java code to implement and test the business logic of your custom workflow actions.
- There is no need to restart the Governance Console whenever you want to add new custom business logic to your workflows. Instead, you just attach a predefined custom workflow action to your workflows together with some properties (basically the names of the new Python functions that you have developed/deployed on the REST server).
- There is no need to setup a Java development environment and you don't need to compile or package Java code (Nevertheless, as part of the lab exercises you can optionally create your own custom Java code and attach it to a workflow. The required JDK files and libraries are installed on the virtual machine and the lab explains the required steps to compile and deploy your code in the Governance Console).
- This approach might also be useful for Proof of Concepts or MVP projects to reduce the development and testing effort for your custom code.

1.3 Demo content overview and generator script

The lab files include a python script that uses the REST API of the Governance Console to create some sample objects that are used in the lab exercises (business entities, use cases, models and relationships between these objects). This script may also be useful to quickly create Watsonx.governance demo content for customer briefings. You can customize this script to your own needs (the required steps to use the script are explained further down).

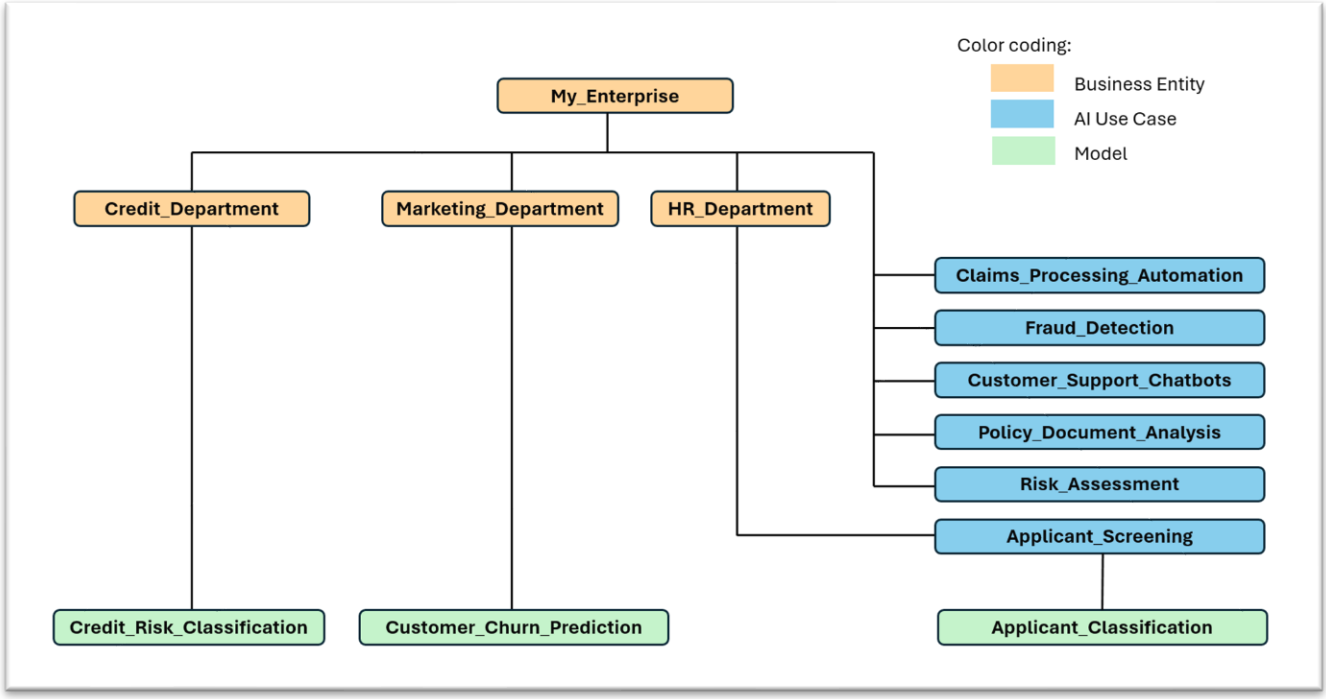


Figure 6: Demo content used in the lab exercises

1.4 Overview of the REST- and Java API

The **REST API** provides functions to create, retrieve and manage resources stored in the *Governance Console*. It is intended to be used by external systems that need to interact with the *Governance Console*. Some example of those resources are:

- Objects: In the lab exercises you will work with objects like **model** and **use case**. We explain how these objects can be created and pre-populated with data and how they can be queried and modified using the REST API.
- Files: Different types of files (Pdf documents,Word-documents, etc.) can be attached to models and use cases (e.g. development documentation, testing documentation, re-certification documentation, etc.)
- Workflows: There are functions available to retrieve information about workflow (e.g. information about the stages and activities). In the lab exercises, we explain how external systems can retrieve information about currently running workflows.
- Other resources: Other resources that can be managed via the REST API are e.g. system configuration information, users, groups and processes. In the lab exercises, you will find an example on how to retrieve configuration information.

The **Java API** provides a large set of functions to manage different areas in the Governance Console, for example:

- Workflows: The corresponding functions allow you to work with the native workflow system. In the lab exercises, we will explain how you can attach custom workflow actions to an activity in a workflow. These actions allow you to execute any kind of Java code as soon as a workflow progresses from one stage to another stage. In our example, we explain how you can execute REST calls to an external system via a custom workflow action.
- Resources: These are objects like “model”, “use case”, files (e.g. document attachments), folders, etc.
- Metadata: The main components of metadata are type definitions (the first level component for any OpenPages resource object template), field definitions and association definitions (relationships between type definitions).

Here is the link to the official reference documentation of the Java API and the REST API:
https://www.ibm.com/docs/en/openpages/9.0.0?topic=developer-guide

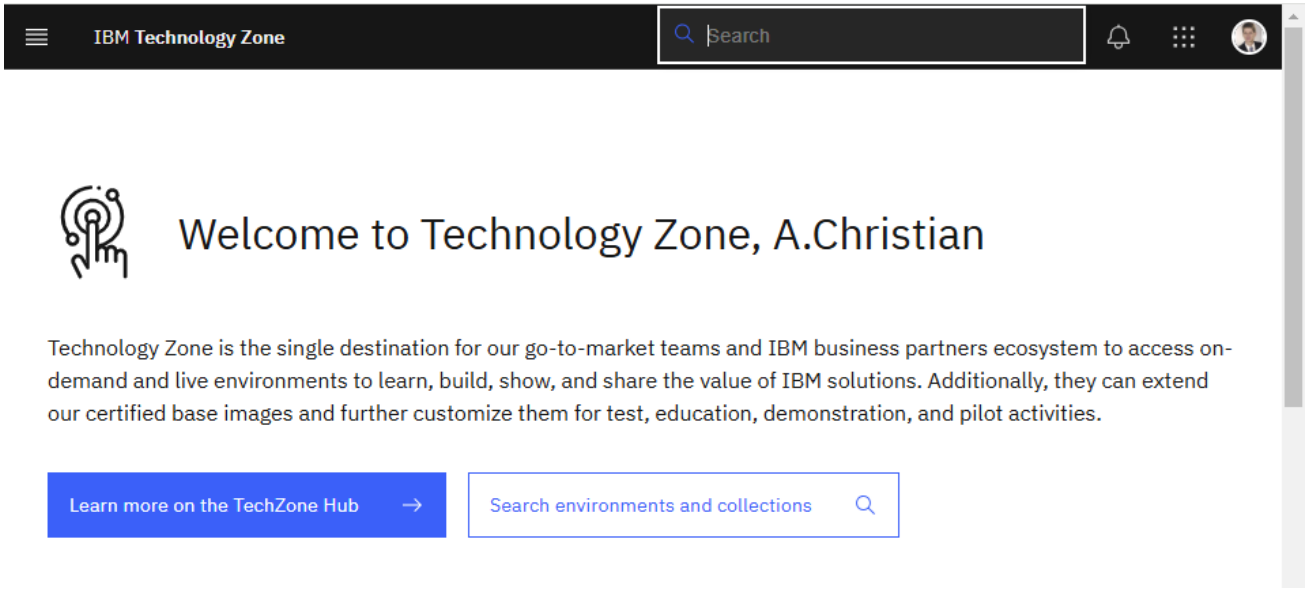
The official API reference documentation is not easy to understand if you have no prior experience with the API calls. Therefore, the lab exercises provide hands-on examples on how to use the various API functions and should provide you a quick and easy start with the API.

1.5 Preparational steps

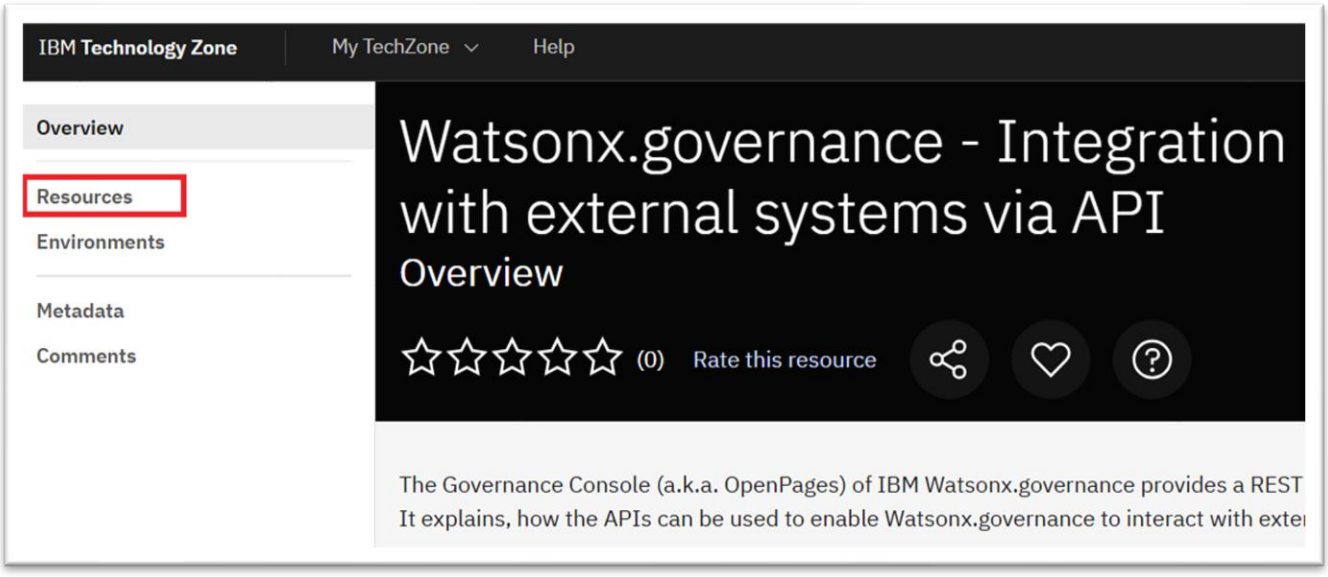
1.5.1 **Reserve the lab environments**

Reserve the the Watsonx.governance environment:

- Access *IBM Technology Zone* using your local web browser: https://techzone.ibm.com/home

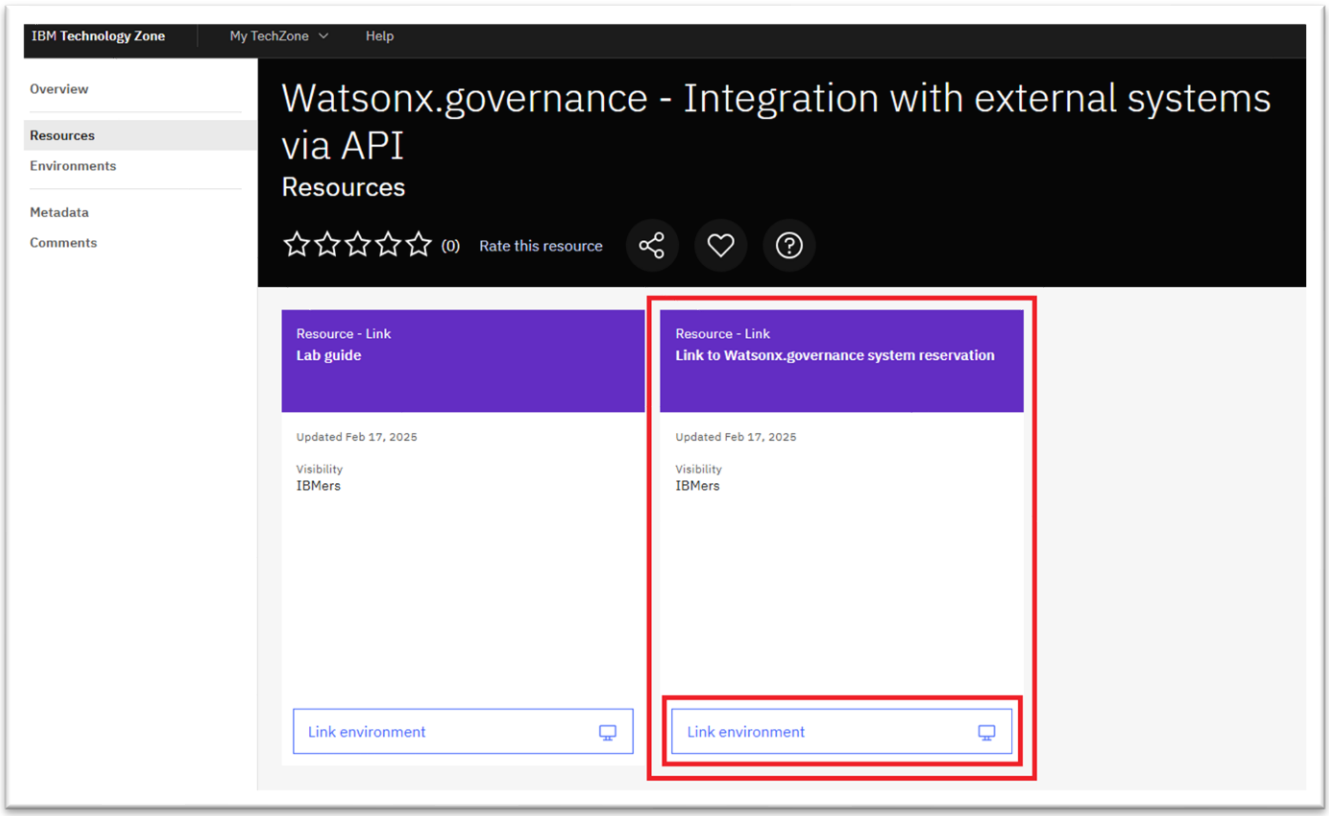


- Search and open the collection "**Watsonx.governance - Integration with external Systems via API".**
- Click "Resources":



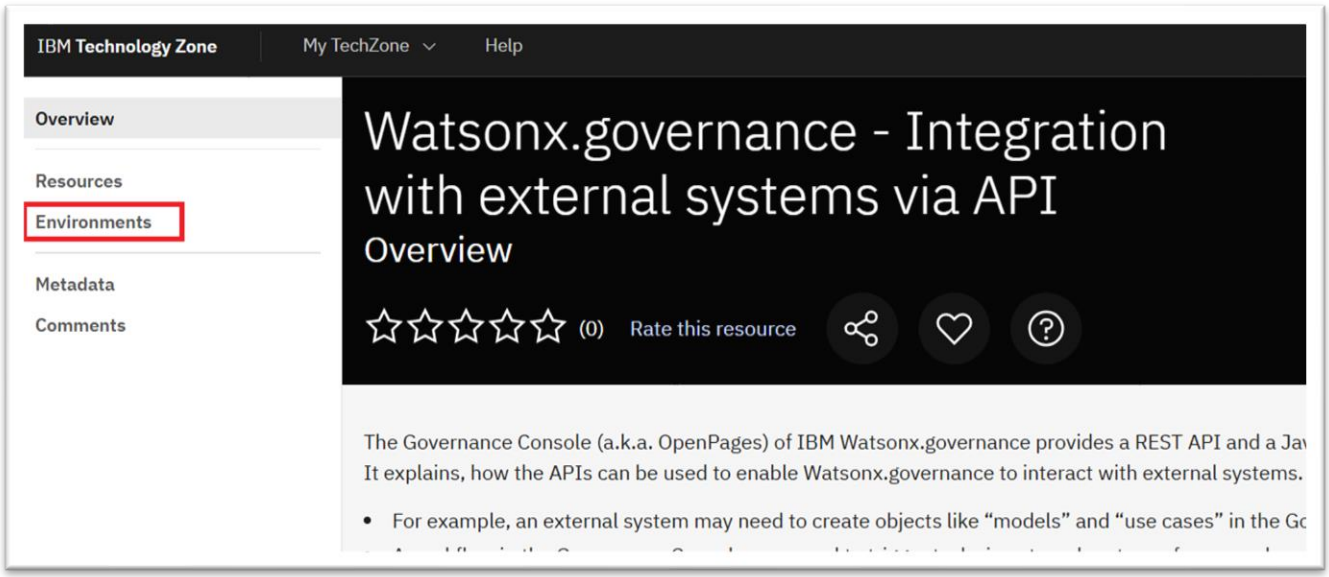
- On tile “Link to Watsonx.governance system reservation” click “Link environment”. This takes you to the reservation page for the Watsonx.governance environment that you will use in this lab. Fill out the reservation form and reserve a Watsonx.governance environment.

Note: The deployment of the system may take several hours to complete!

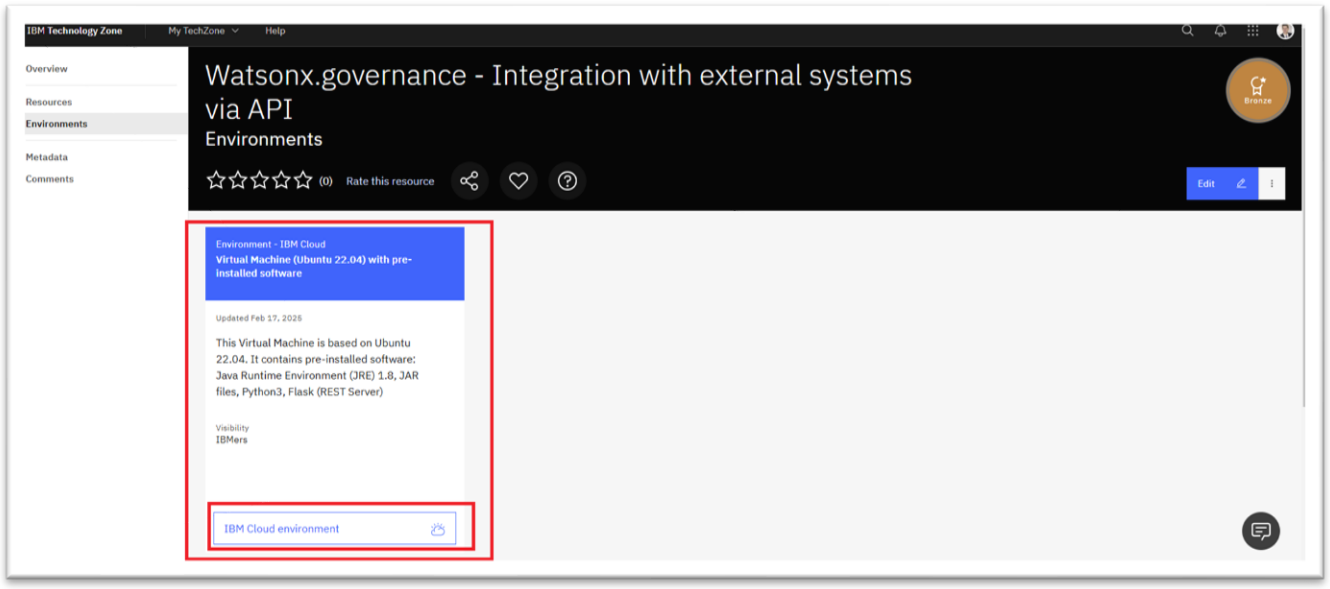


Reserve the Virtual Machine:

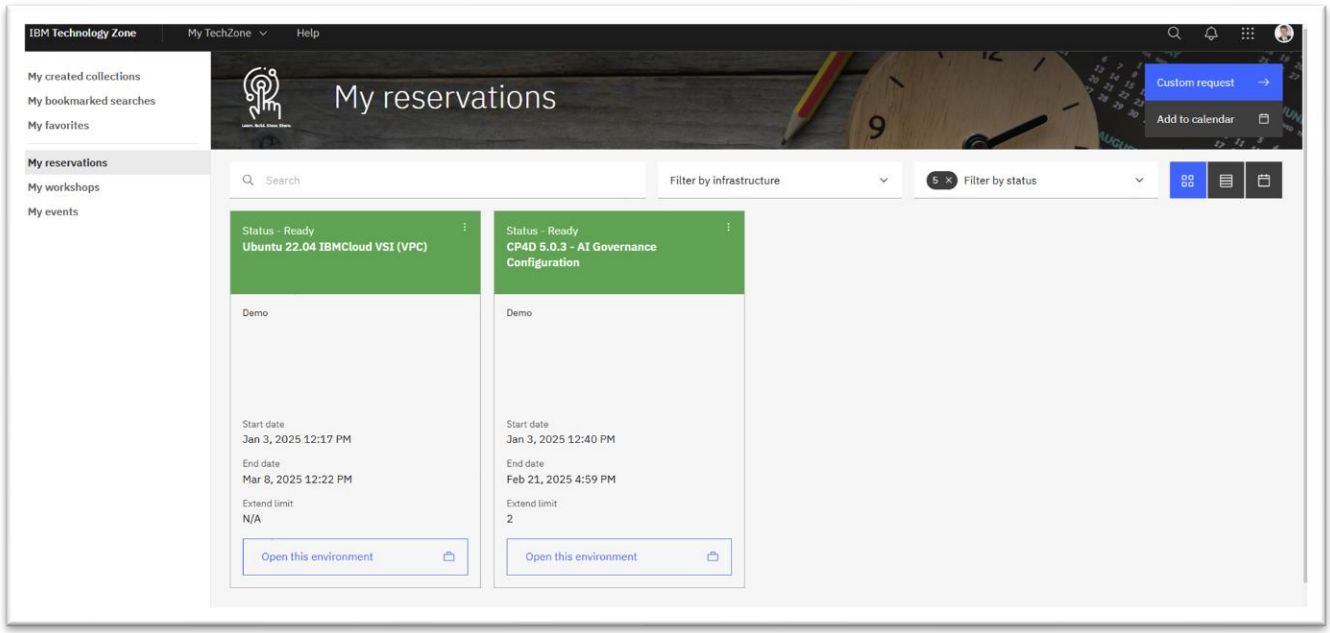
- Search and open the collection “**Watsonx.governance - Integration with external Systems via API”.**
- Click “Environments”:



- Click “IBM Cloud Environment”. This takes you to a reservation form that you need to fill out to reserve the virtual machine that is required for the lab. The deployment typically only takes a few minutes to complete.

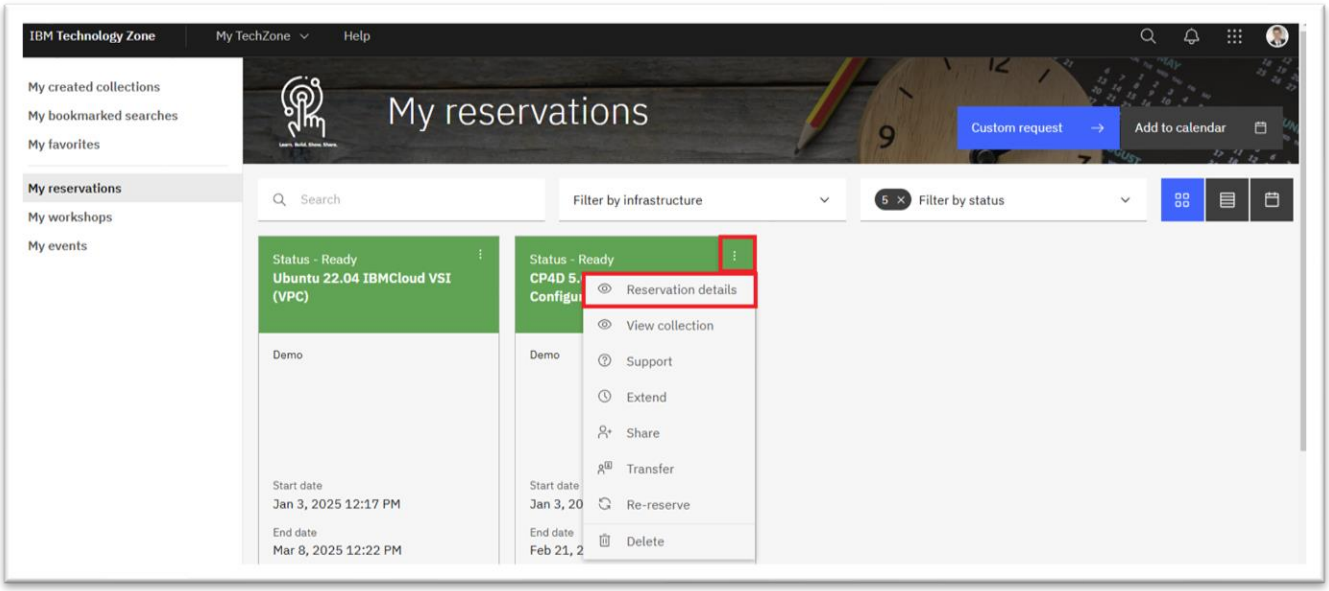


After you have successfully completed the reservation process you should see the two system reservations on IBM Technology Zone:

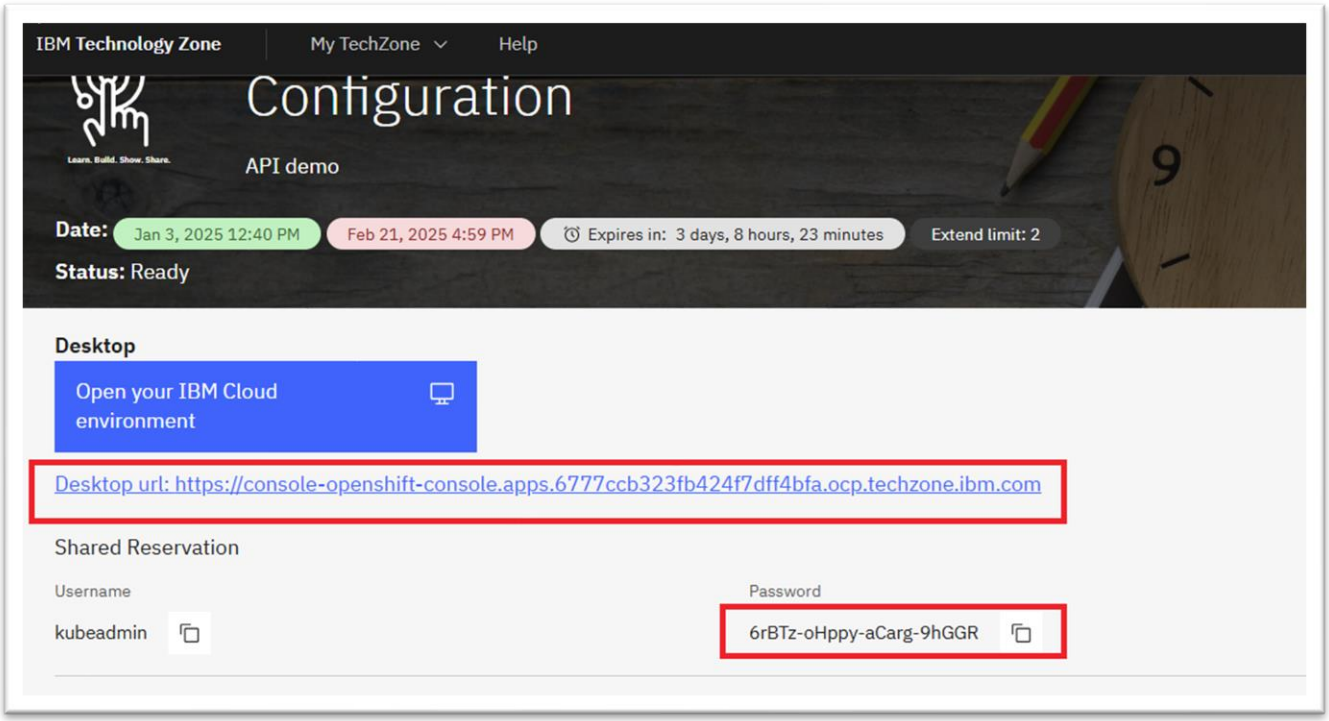


1.5.2 **Access the Watsonx.governance environment**

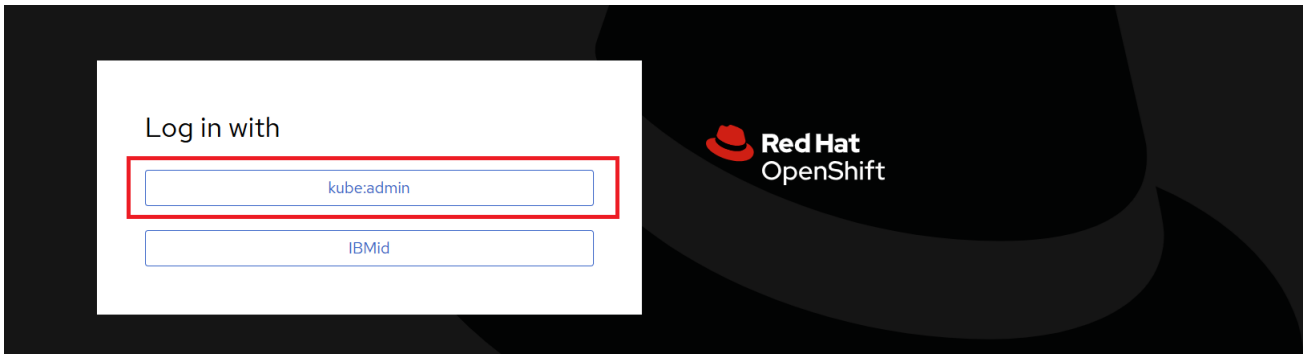
- On IBM Technology Zone open the “My reservations” page. Then open the popup menu of your CP4D environment and select “Reservation details”:



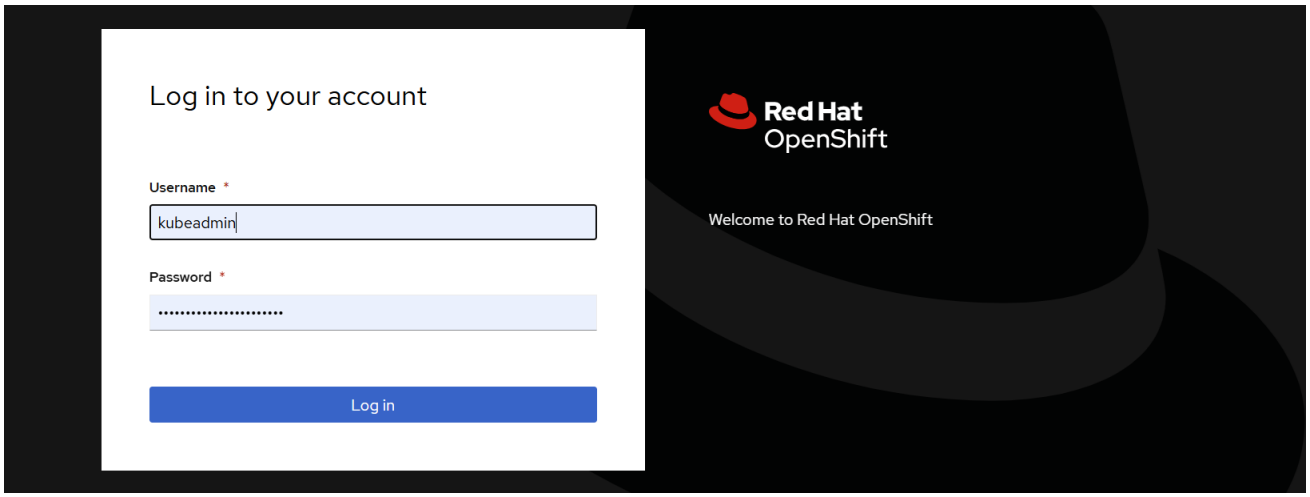
- Take a note of the password of user *kubeadmin*. Then click the *Desktop url* to open the *OpenShift console* window of the environment.



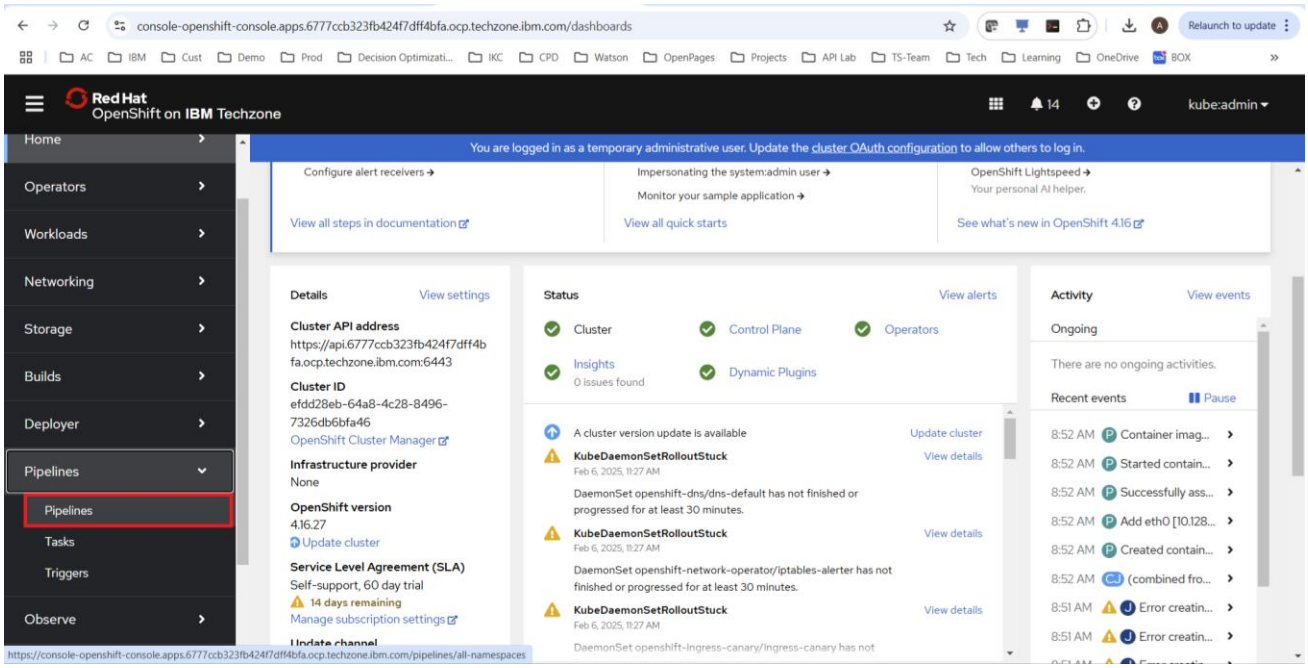
- Click *kube:admin*



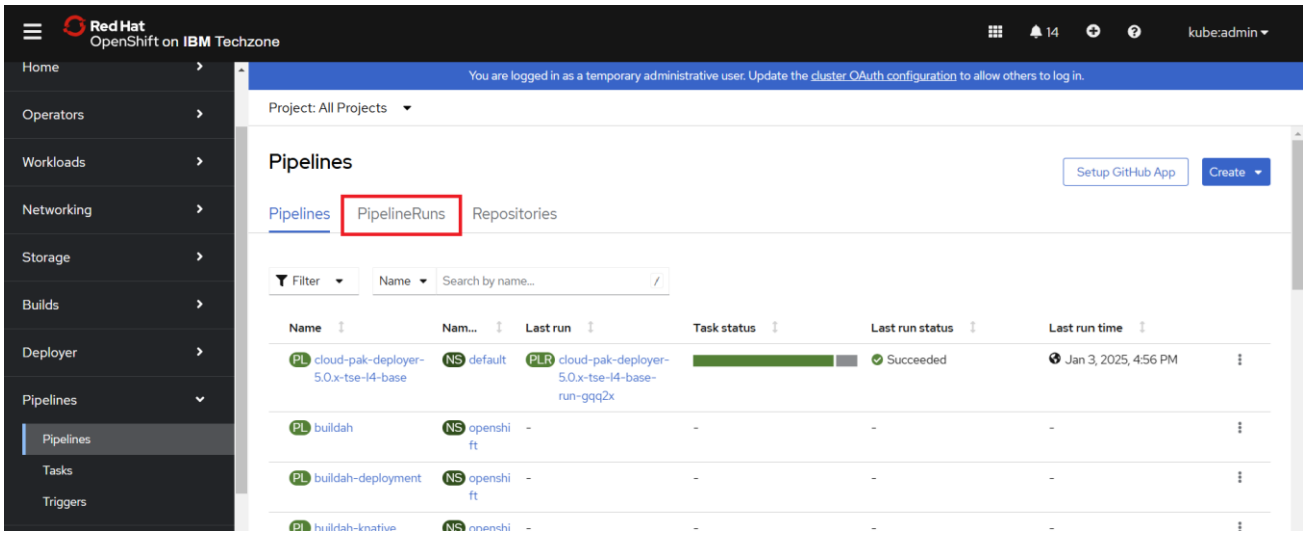
- Logon with the *kubeadmin* user and password:



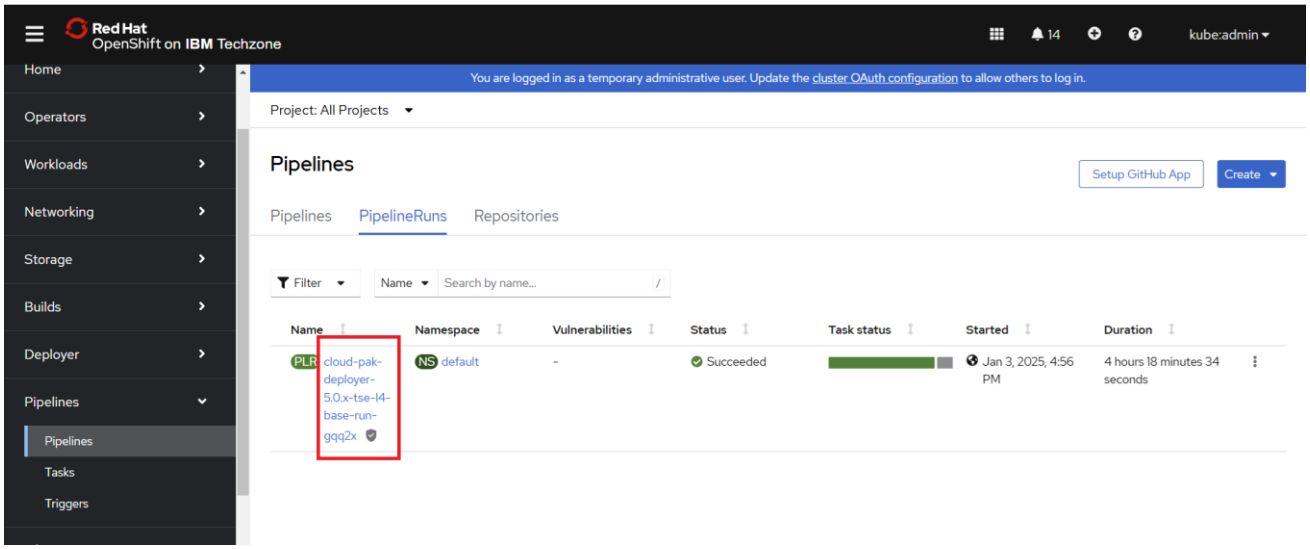
- Now you need to find the *Cloud Pak for Data* console link of your system. In section *Pipelines* click *Pipelines*:



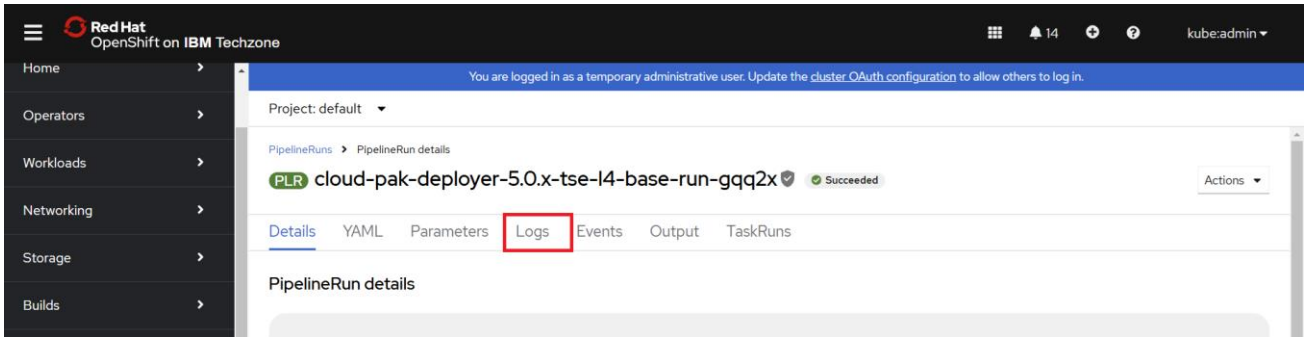
- Click *PipelineRuns*:



- There should be one *pipeline run* listed. Click the name of the *pipeline run*:

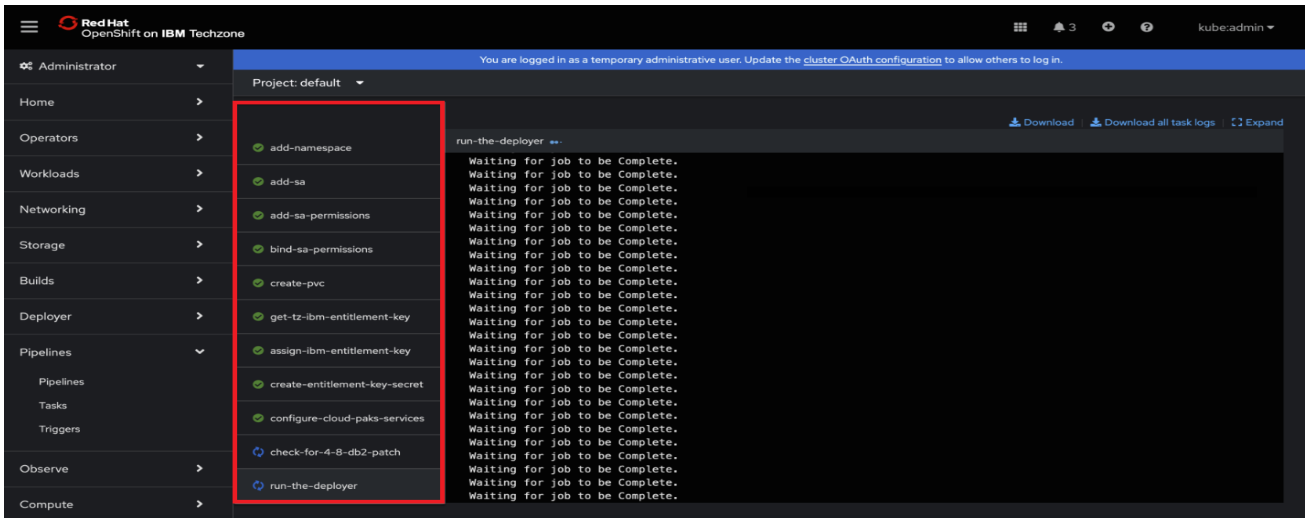


- Click *Logs*:

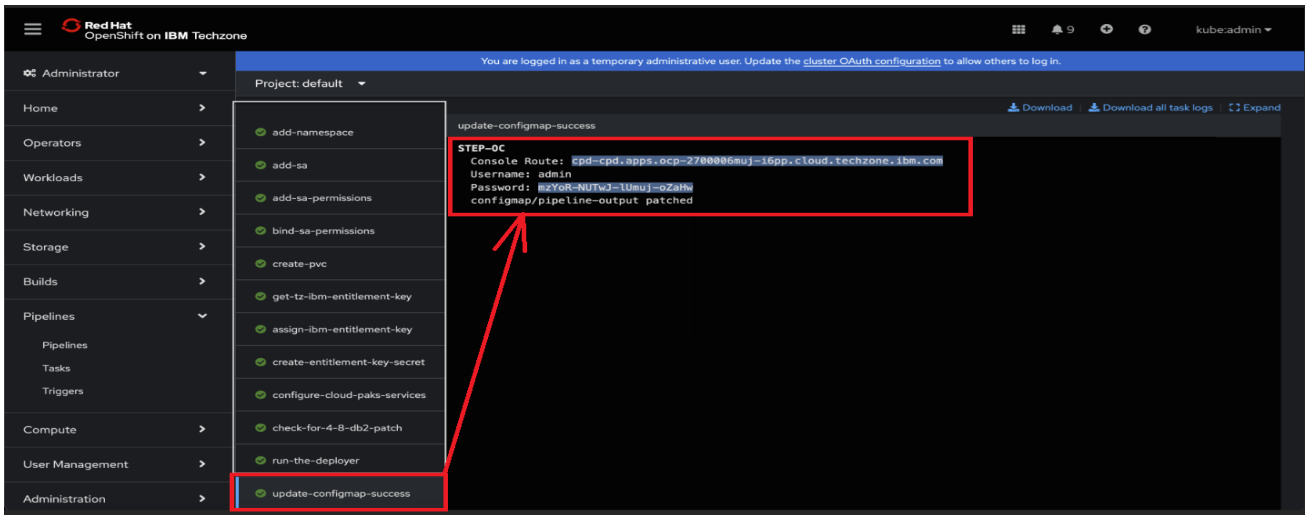


- You see a list of pipeline steps. All steps should be marked with a green checkmark. If the system deployment is still running, you have to wait until all steps are completed.

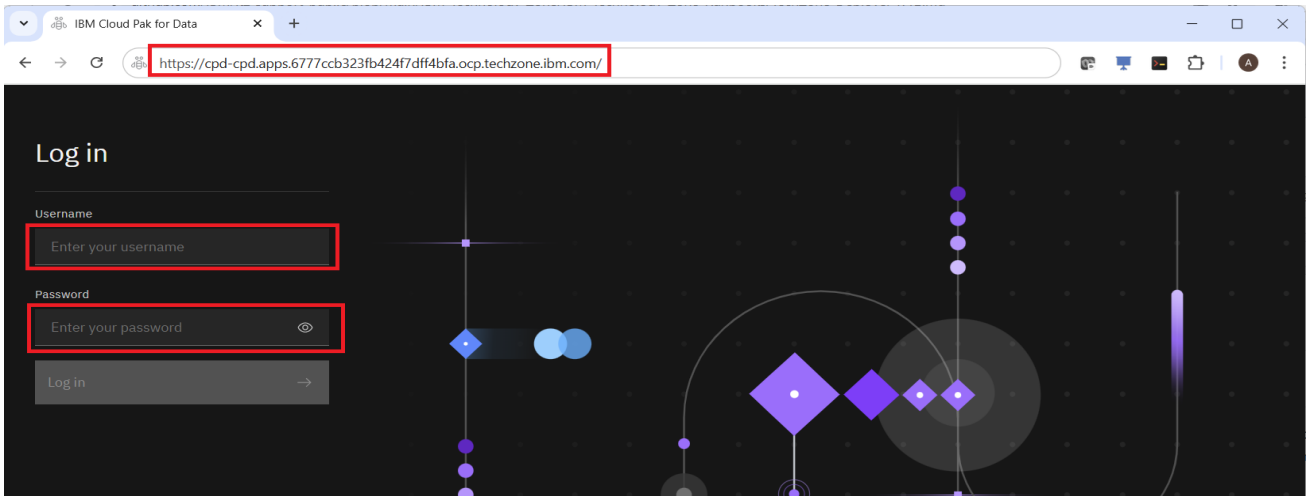
Note: The pipeline may run for several hours until all pipeline steps are completed!



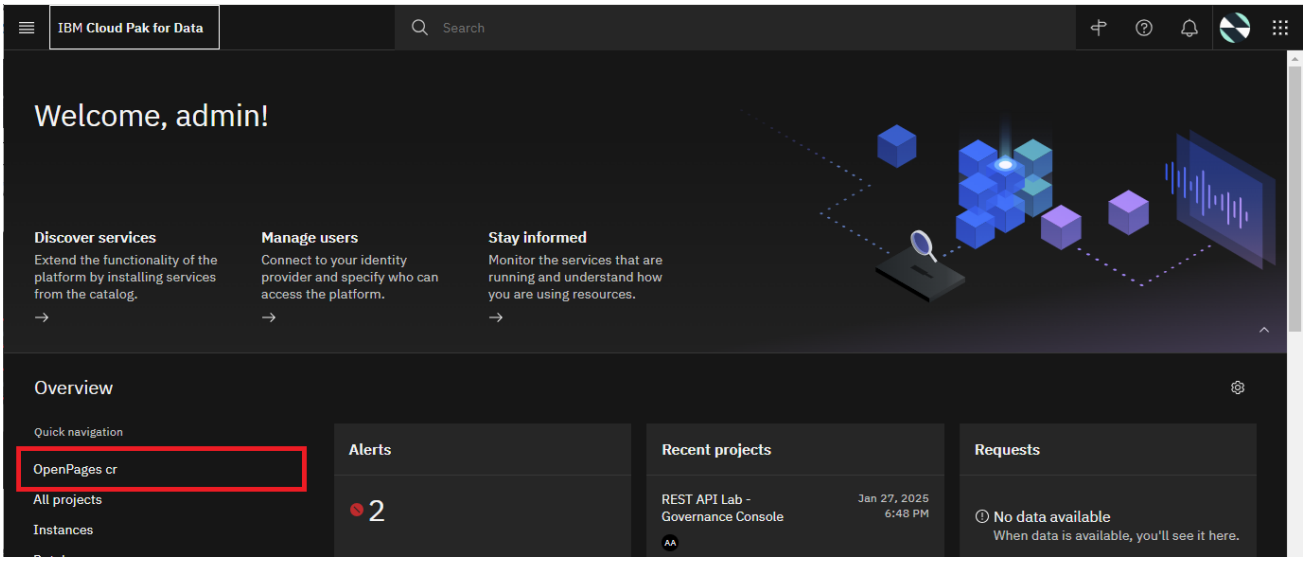
- After all stages of the pipeline have completed successfully, click “update-configmap-success” and take a note of the Cloud Pak for Data ***Console Route*** (server name) and ***Password*** of user ***admin***:



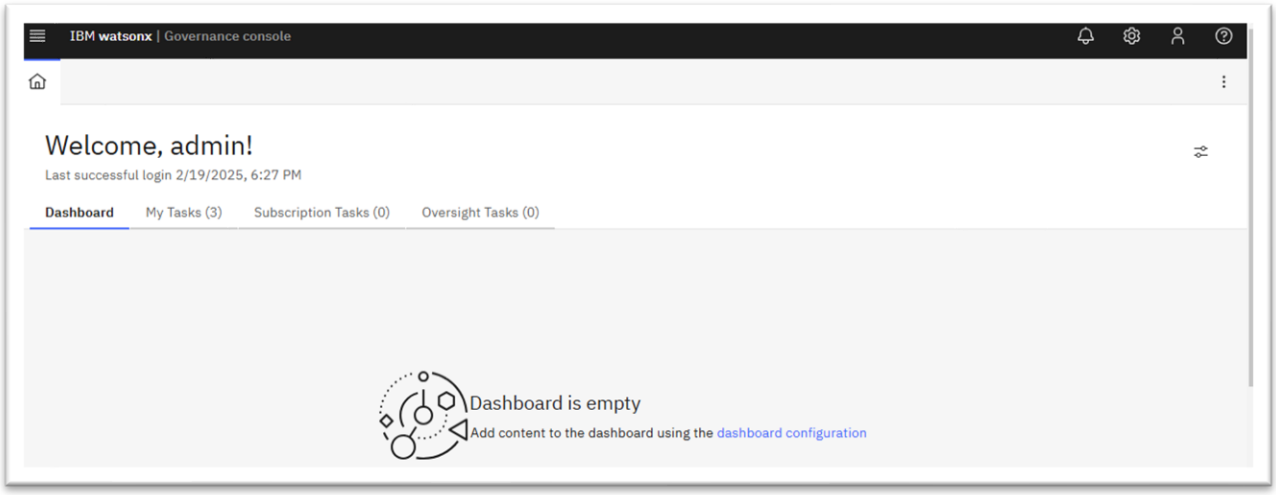
- Open a new web browser window and enter the Cloud Pak for Data *Console Route* to access the logon screen of the CP4D console. Now you can logon as user *admin*:



- After successful logon you see the home screen of the *IBM Cloud Pak for Data console.* To access the Governance Console, click *OpenPages cr*:

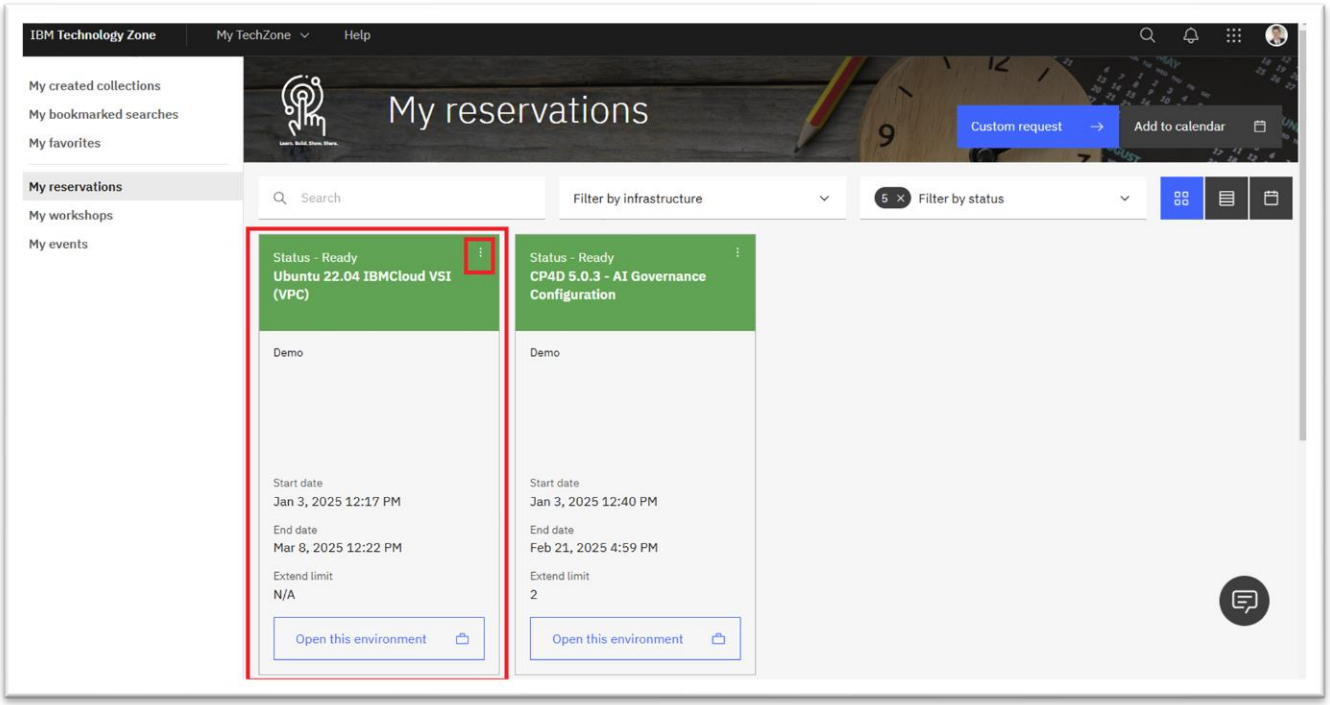


- This takes you to the home screen of the Governance Console:

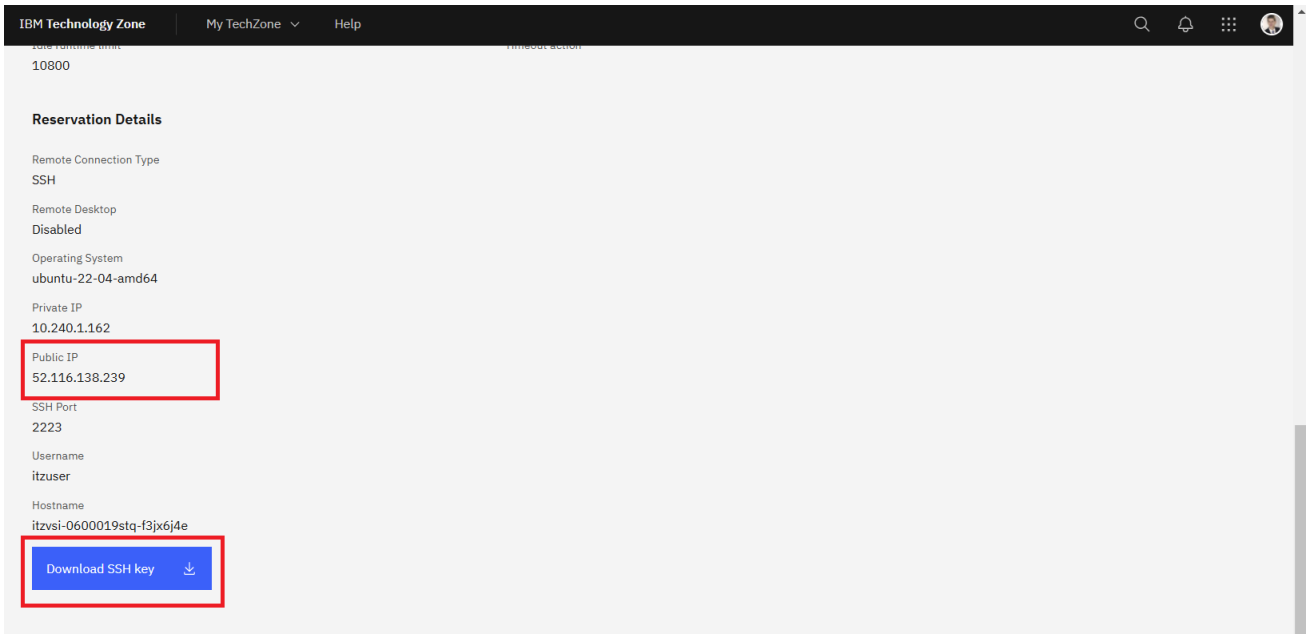


1.5.3 **Access the Virtual Machine environment**

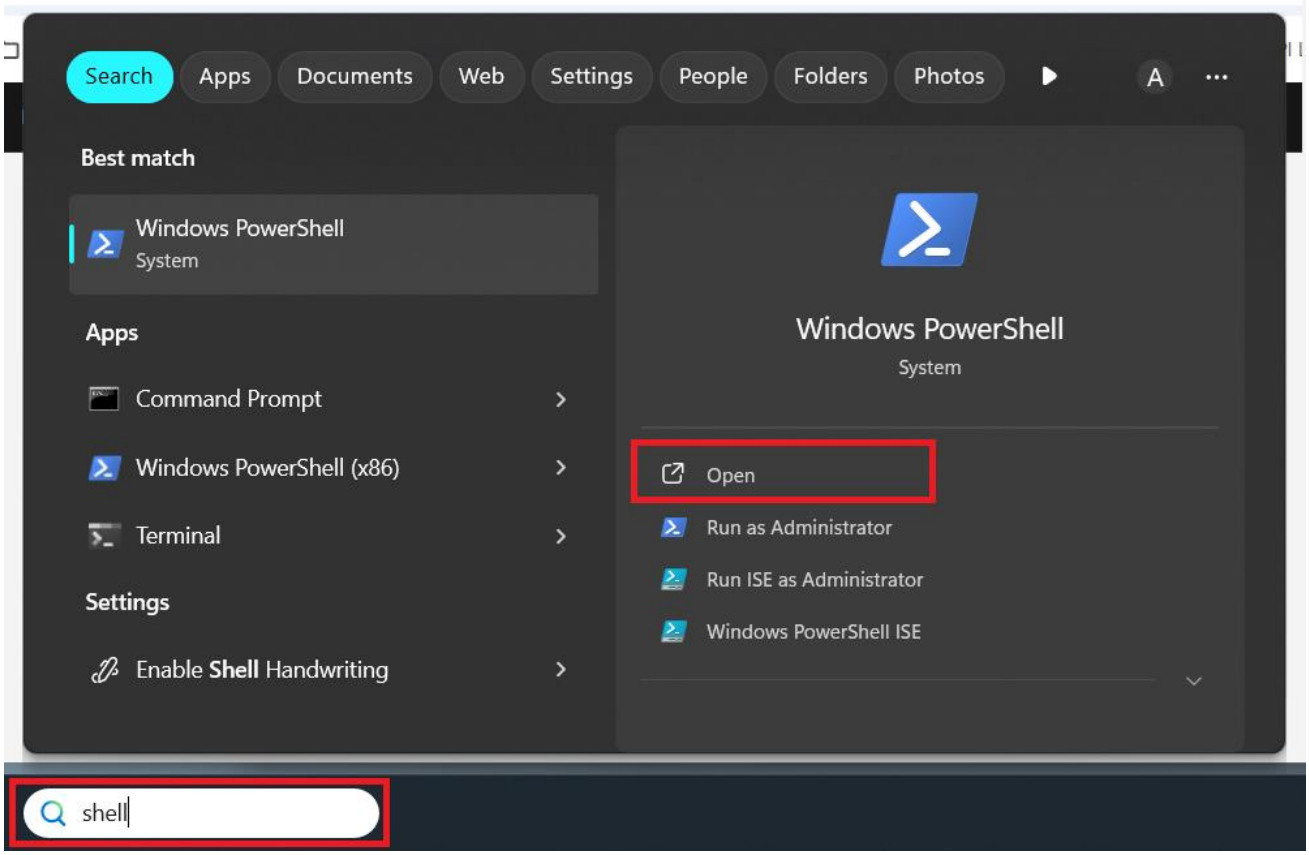
- On IBM Technology Zone open the “My reservations” page. Then open the popup menu of your reserved *virtual machine* environment and select “Reservation details”:



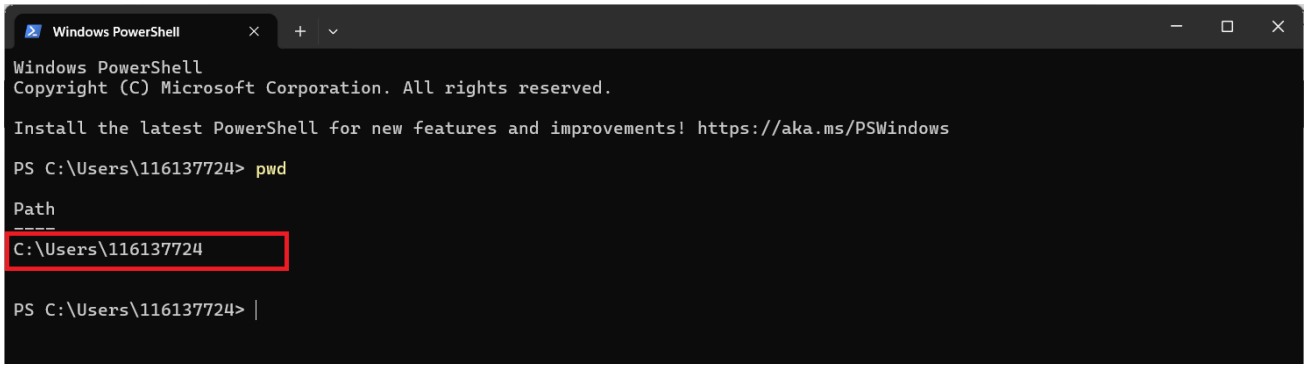
- Scroll down to section “Reservation Details”, take a note of the “Public IP” of your virtual machine and then click “Download SSH key”. This will download a file named “pem_ibmcloudvsi_download.pem”.



- Open a shell on your local computer. For example, if you use the *Windows* operating system you can open the Windows Power Shell (enter “shell” in the windows search bar and click “Open”):



- Type “pwd” to check the current directory of your shell (this command also works on Unix-like operating systems):



- Copy file *pem_ibmcloudvsi_download.pem* to the current directory of your shell.

  Note: If you use a Unix-like operating system, you have to adjust the access permissions of the pem-file. In this case, execute the following command:
  `chmod 600 pem_ibmcloudvsi_download.pem`

- Execute the following command to open an ssh connection to your virtual machine (replace ip_address with the public IP address of your virtual machine):
  `ssh -i pem_ibmcloudvsi_download.pem -p 2223 itzuser@ip_address`
  For example:
  `ssh -i pem_ibmcloudvsi_download.pem -p 2223 itzuser@52.116.138.239`

You are now connected to your virtual machine.

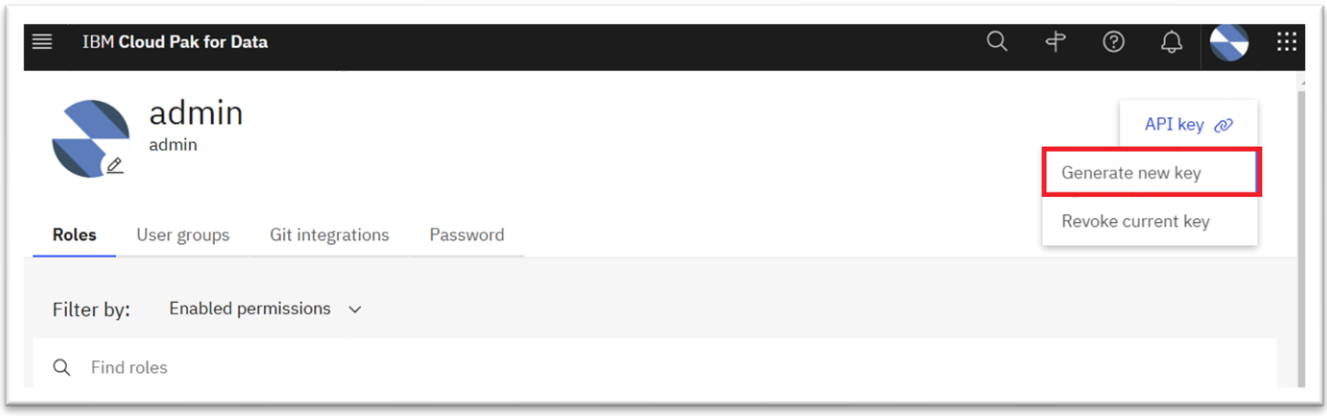
1.5.4 **Create an authorization token**

To be able to use the REST API, you require an authorization token. Whenever you execute a call to the REST API this token must be specified in the header of the REST call (see details further down).

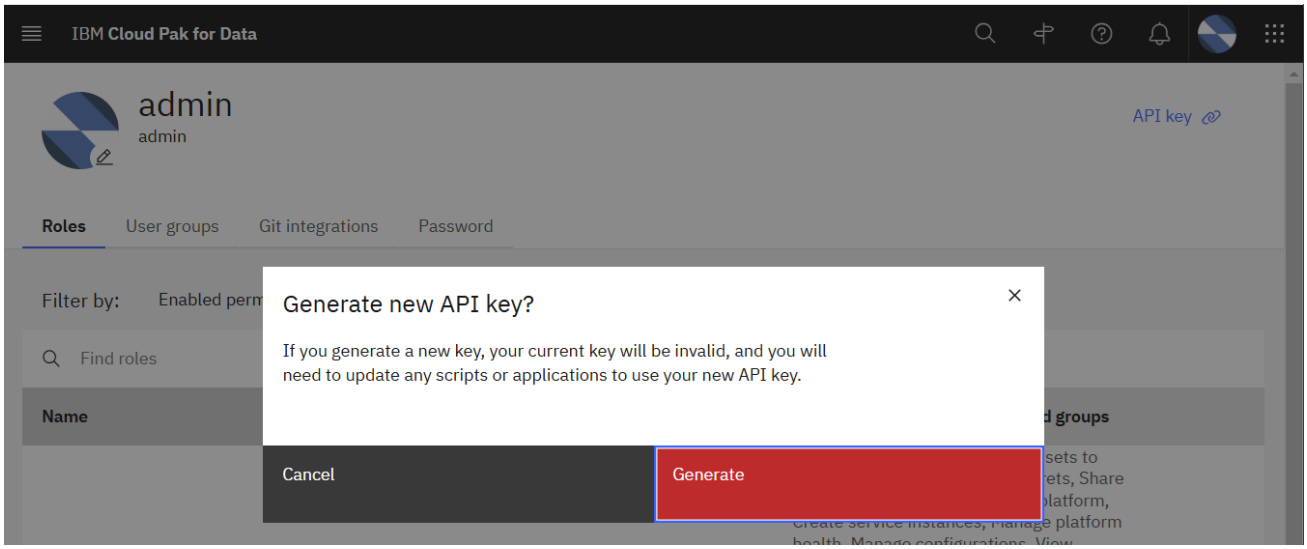
Perform the following steps to create an authorization token:

- Login to the Cloud Pak for Data console with your user id (for example user *admin*).
- Click the user profile icon in the upper right of the console and select *Profile and settings*:

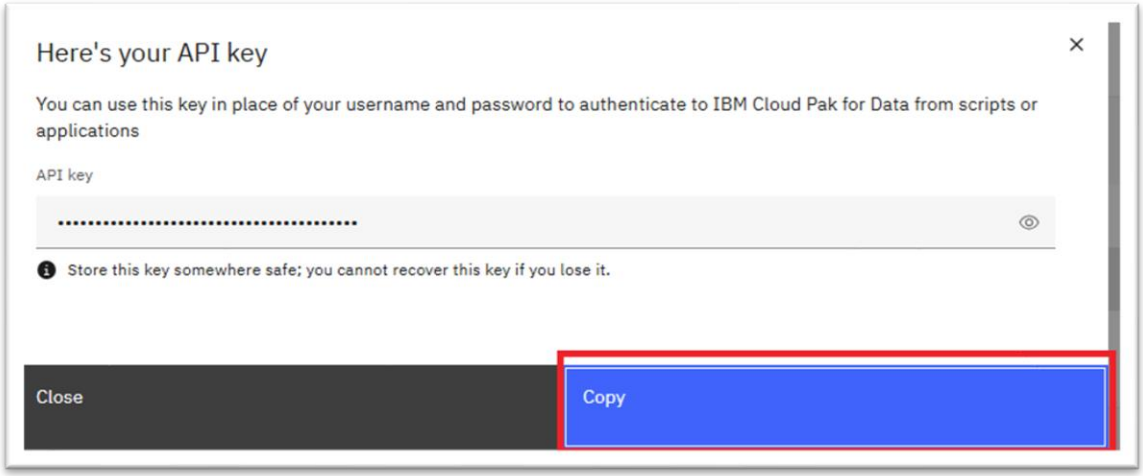
- Click *API key* and select *Generate new key*:



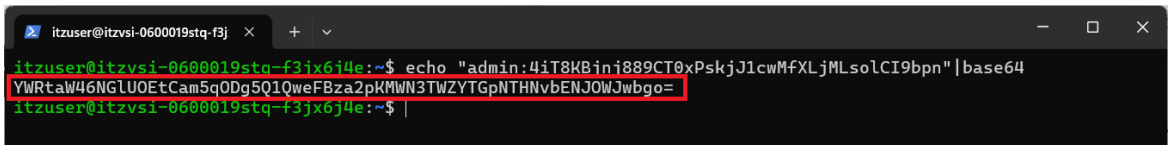
- Click *Generate*:



- Click *Copy* and store the copied **API key** in a text file on your local machine:



- Open a shell window on your local computer and connect via ssh to your *virtual machine* environment (see steps in previous chapter).
- In the shell window execute the following command:
  `echo "<username>:<api_key>" | base64`
  For example:



- Copy the resulting output (marked red) to a text file on your local machine. This is your *authorization token.*

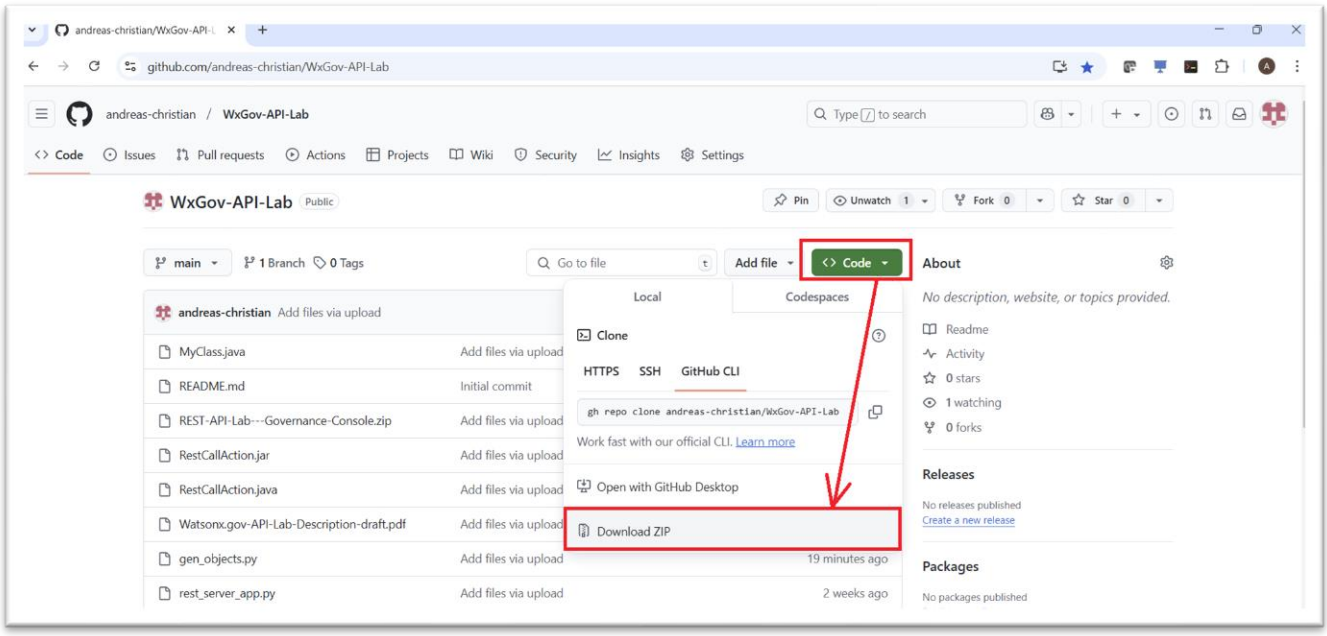
For your reference, here are the links to the official documentation on how to create an authorization token:
https://www.ibm.com/docs/en/cloud-paks/cp-data/5.0.x?topic=steps-generating-api-keys
https://www.ibm.com/docs/en/cloud-paks/cp-data/5.0.x?topic=apis-generating-api-auth-token

1.5.5 **Download git repository**

We have created a git repository with some files and scripts that are required for the lab exercises. You need to download the git repository to your local computer and also to your virtual machine environment.

To download the git repository to your local computer:

- Open a web browser on your local computer and open the following link: https://github.com/andreas-christian/WxGov-API-Lab
- Click *Code* and select *Download ZIP*:

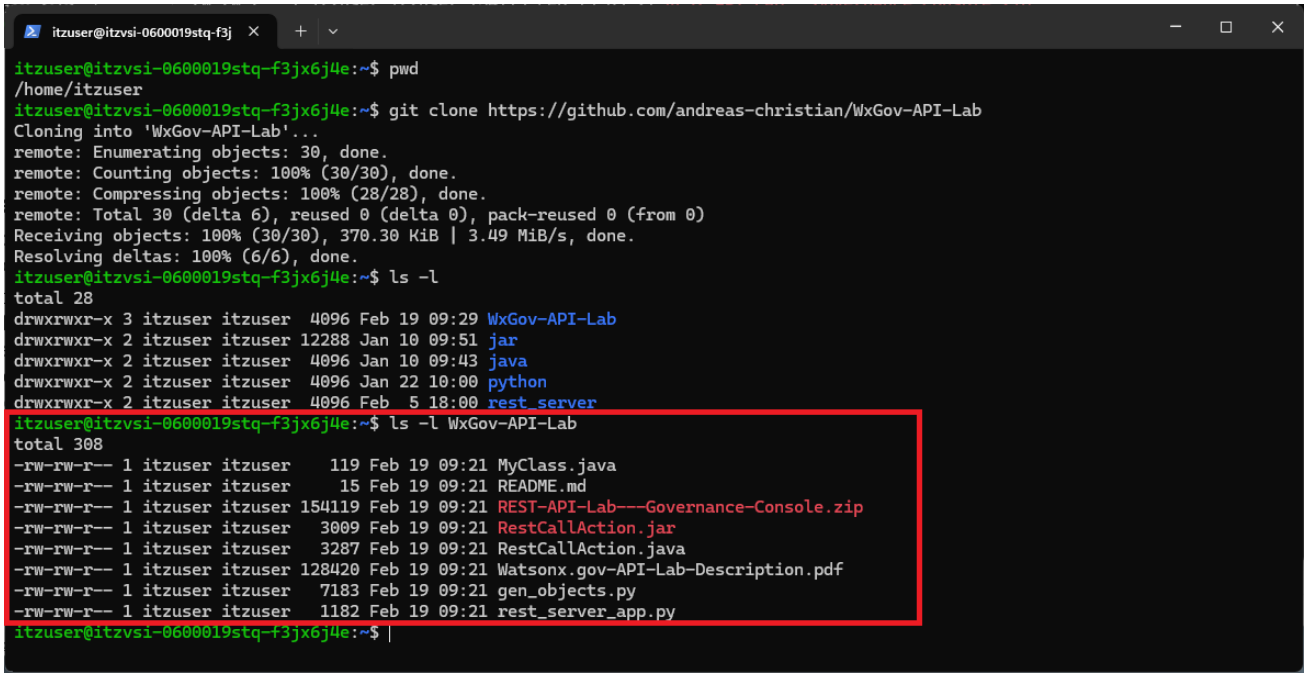


To download the git repository to your virtual machine environment:

- Connect to your virtual machine via ssh
- In the home directory of your user run the following command:
- git clone https://github.com/andreas-christian/WxGov-API-Lab

```
itzuser@itzvsi-0600019stq-f3j
itzuser@itzvsi-0600019stq-f3jx6j4e:~$ pwd
/home/itzuser
itzuser@itzvsi-0600019stq-f3jx6j4e:~$ git clone https://github.com/andreas-christian/WxGov-API-Lab
Cloning into 'WxGov-API-Lab'...
remote: Enumerating objects: 30, done.
remote: Counting objects: 100% (30/30), done.
remote: Compressing objects: 100% (28/28), done.
remote: Total 30 (delta 6), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (30/30), 370.30 KiB | 3.49 MiB/s, done.
Resolving deltas: 100% (6/6), done.
itzuser@itzvsi-0600019stq-f3jx6j4e:~$
```

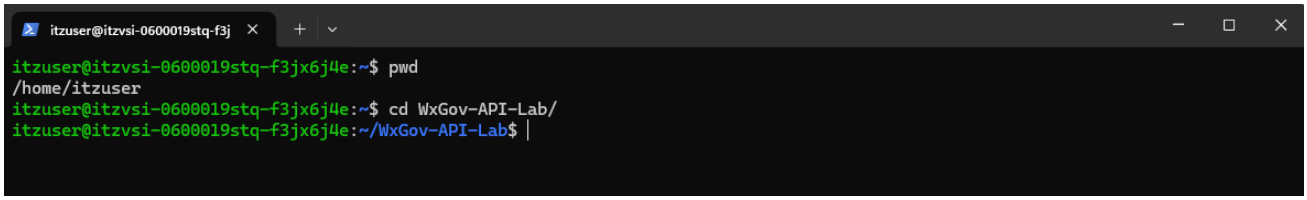
- The command generates subdirectory *WxGov-API-Lab* that contains all files of the git repository:



1.5.6 **Create demo content using a Python script**

To be able to demonstrate the REST API functions, we first need to create some objects (models, use cases, etc.) in the repository of the Governance Console:

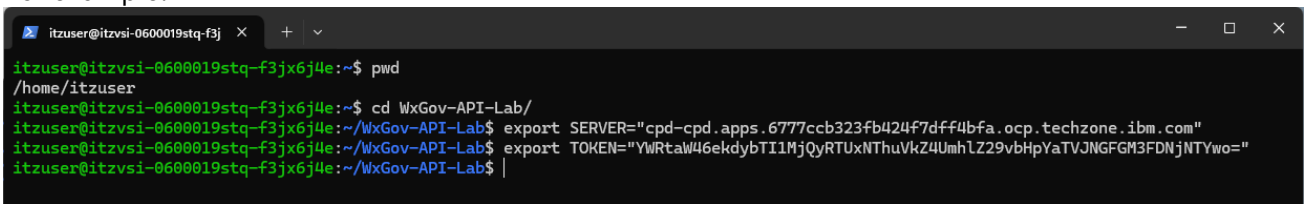
- In the home directory of your virtual machine environment go to subdirectory *WxGov-API-Lab:*



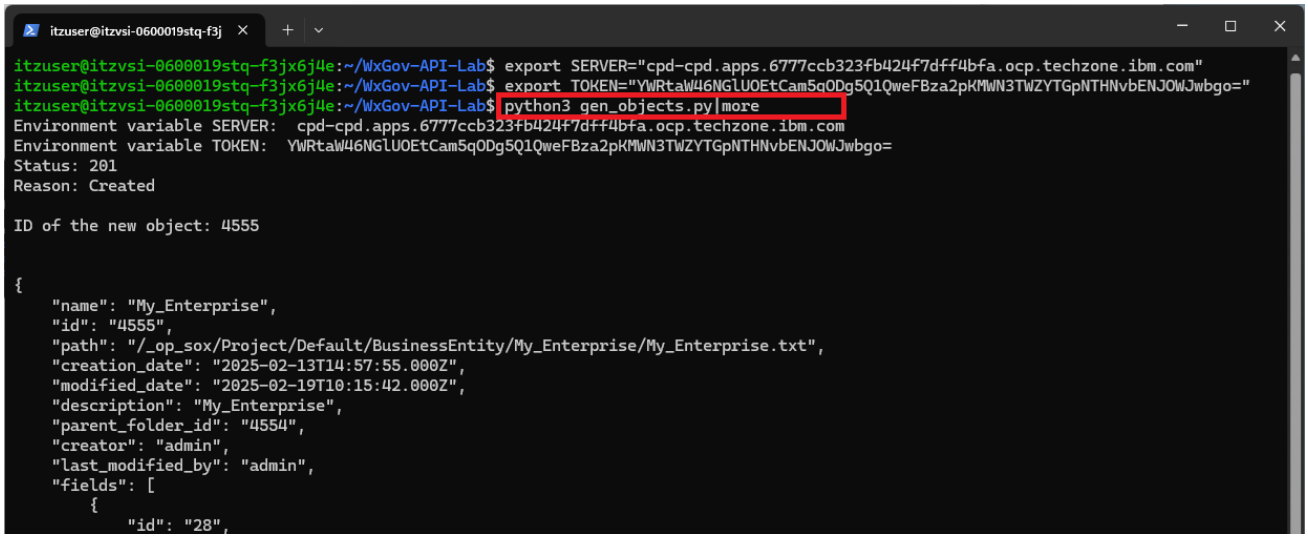
- Define environment variables SERVER and TOKEN with the following commands:
  export SERVER=<CP4D server name>
  export TOKEN=< authorization token>
  For example:

- Define the following environment variables (replace <server_name> with the name of your CP4D server and <authorization_token> with the authorization token that you generated in the previous steps)
  - export SERVER=”<server_name>”
  - export TOKEN=”<authorization token>”

  For example:



- Execute the python script to generate some objects in the Governance Console:
  `python3 gen_objects.py`



The script creates some sample objects in the repository of the Governance Console. These objects will be used throughout the REST API lab. In case you need to prepare customer demos, this script might also be usefull to quickly create the required content for the demos.

2 Lab exercises

2.1 REST API Exercises

Before you execute the Python code examples of this lab, it is recommended to read the introduction in the following sections. Check the official REST API documentation for a detailed description of all methods included in the API:
https://developer.ibm.com/apis/catalog/openpages--ibm-openpages-with-watson-rest-api-v2/Getting+Started

2.1.1 General introduction to the REST API

The REST API provides different REST method types:

- **GET**: Is used to **retrieve** data from a resource. You can, for example, retrieve all attribute values related to an *AI use case* or a *model*.
- **POST**: Can perform any type of operation. The REST API of the Governance Console uses POST requests to **create** new objects.
- **PUT**: The REST API of the Governance Console uses PUT requests to **update** existing objects.
- **DELETE**: Deletes the specified resource.

Note: The REST API implementation of the Governance Console uses POST requests (not PUT requests) to create new objects in the repository. These POST requests are idempotent: If you execute the same POST request multiple times with the exact same payload and parameters, you will only create one object (not multiple instances of that object). This is quite handy. For example, if you use a script to generate multiple objects and the script fails after creating some objects, you can just rerun it without the need to perform any clean-up of the objects that were already created during the previous run.

All REST calls must include a *Uniform Resource Locator* (URL) which specifies the server name, base path and the actual method that is supposed to be executed in the REST API. Here is a simple example of a URL that you can use to retrieve all tags that are defined in the Governance Console:

```
https://{server_name}/{base_path}/v2/configuration/tags
```

In addition, parameters can typically be specified as well. In the below example, they are marked yellow. Parameters are always introduced by a leading question mark and multiple parameters are separated by ampersand (&):

```
https://{server_name}/{base_path}/v2/query?q=REPLACE_THIS_VALUE&offset=REPLACE_THIS_VALUE&case_insensitive=REPLACE_THIS_VALUE&limit=REPLACE_THIS_VALUE&max_rows=REPLACE_THIS_VALUE&honor_primary=REPLACE_THIS_VALUE
```

The next example shows how to execute a REST call using the *curl* commad. In addition to the URL it also contains the method type (GET) and a header that includes an authorization token (server_name and base_path must be replaced according to your specific system environment):

```
curl --request GET \
  --url 'https://{server_name}/{base_path}/v2/configuration/tags' \
  --header 'Authorization: Basic REPLACE_AUTHORIZATION_TOKEN' \
  --header 'accept: application/json'
```

A REST call also returns data, for example a return code and some corresponding descriptive text:

```
Status ‘200’, Reason ‘OK’
```

There are different ways to execute a REST call (cURL command, Python code, etc.). Throughout the lab exercises you will use Python (Jupyter notebook) to execute REST calls.

2.1.2 How to query objects

The ***query service*** allows you to query the objects stored in the repository of the Governance Console. To write a query you use SQL-like syntax. In the following example the query text is marked yellow:

```
https://{server_name}/{basePath}/v2/query?q=SELECT+[Name]+FROM+[Register]
```

Field names must be embraced with square brackets. Special characters like space, single quote or percent sign are not allowed in URLs. For example, instead of spaces you have to use plus signs as shown above. To make the query examples more readable, the sample code (Jupyter notebook) uses spaces instead of plus signs to initially define the query text. It then replaces any special characters in the query text with those characters that are allowed in HTML URLs. Here is an example:

```
query = "SELECT [Name] FROM [Register]"
query = query.replace(" ","+").replace("%","%25").replace("'","%27")
```

More information about HTML URL encoding can be found in the following link:
https://www.w3schools.com/tags/ref_urlencode.ASP

Note: You can not use wild cards (*) in the select clause of a query. Instead, you have to specify the names of all fields that you want to retrieve.

For example, this query here will NOT work:

```
SELECT * FROM [Register]
```

The type of object you want to retrieve must be specified in the FROM clause of the query. In the appendix of this document you find a list of all supported *type names,* the corresponding *type id* and a short *type description*. You can use the following REST endpoint to retrieve all type definitions that are supported (an example how to retrieve all supported *type names* is also included in the Jupyter notebook):

```
https://{server_name}/{basePath}/v2/types
```

Here are some type names that are useful to know:

- **Model**
- **Register** (Type name for *use cases*)
- **ChangeRequest**
- **Employee**
- **SOXBusEntity** (Type name for *business entities*)
- **SOXIssue** (Type name for *issues*. An issue can be for example a *test failure* or a *review comment*)
- **SOXTask** (Type name for *action items*. Action items can be linked to issues)
- **SOXDocument** (Any kind of file, for example PDF- or Word-documents. Files can be attached to models and use cases)

To retrieve the names and object IDs of all *models*, *use cases* and *business entities* (e.g. business departments) you can use the following queries:

```
SELECT [Resource ID], [Name] FROM [Model]
SELECT [Resource ID], [Name] FROM [Register]
SELECT [Resource ID], [Name] FROM [SOXBusEntity]
```

There is also a REST endpoint available to retrieve all *field names* of a given object type. An example, how to retrieve the field names of a specific object type is included in the Jupyter notebook. For example, to retrieve all supported field names of object type *model* (type id = 58) you can use this endpoint here:

```
https://{server_name}/{basePath}/v2/types/58
```

Here is an example of a query that includes a WHERE-clause:

```
SELECT [Resource ID], [Name], [Description] FROM [Register] WHERE [Name] =
'Applicant_Screening'
```

The next query example includes a WHERE-clause with a LIKE predicate. The % character is used as wild card character. The query searches for all model that include the substring “Classification” in their name. The query also includes an order by clause. Hence the query result is sorted by model names:

```
SELECT [Resource ID], [Name], [Description] FROM [Model]
WHERE [Name] LIKE '%Classification%' ORDER BY [Name]
```

The data that is returned by the *query service* is encoded in JSON format. We use function *json.loads()* to convert the data into a compound Python data structure:

```
my_dict = json.loads(data)
```

This data structure has the following general format:

```
{
      "definitions": [column1, … , columnN ]
      "rows":
          [row1]
          [row2]
           …
          [rowN]
}
```

Curly brackets indicate a Python dictionary. Square brackets indicate an array. That means the query result is stored in a Python dictionary that contains two elements:

- **definitions:** This element is a Python *array*. The array elements describe each column of the query result. The column descriptions (e.g. column1) are of type Python *dictionary*. They contain the *id*, *name* and *type* of the corresponding column.
- **rows:** This element is also a Python *array*. It contains the actual rows that are returned by a query. Each row is of type Python *dictionary.*

You will get a better understand of this data structure when you execute the code examples in the Jupyter notebook.

2.1.3 **How to retrieve the entire content of an object**

Finally, besides the *query service* you can use the following endpoint to retrieve the entire contents of an object including all of its fields. This method requires the object’s ID as input and you need to execute it with a REST call of type GET:

```
https://{server_name}/{basePath}/v2/contents/{id}
```

Here is an example ():

```
curl --request GET \
  --url 'https://<SERVER_NAME>/<BASEPATH>/v2/contents/4033' \
  --header 'Authorization: Basic REPLACE_AUTHORIZATION_TOKEN' \
  --header 'accept: application/json'
```

2.1.4 **How to update an object**

To *update* fields of an existing object you can use the same REST endpoint as in the previous example. But in this case you need to use a REST call of type **PUT**:

```
https://{server_name}/{basePath}/v2/contents/{id}
```

Let's assume you want to update the following object:

- Object ID: *4033*
- Object type: *use case* (type_definition_id=74)
- Object name: *Applicant_Screening*

You can use the following curl command (REQUEST_BODY must be replaced with a string in JSON format):

```
curl --request PUT \
  --url 'https://<SERVER_NAME>/<BASEPATH>/v2/contents/4033' \
  --header 'Authorization: Basic REPLACE_AUTHORIZATION_TOKEN' \
  --header 'accept: application/json'
  --header 'content-type: application/json' \
  --data REQUEST_BODY
```

Before you can execute the above command you need to assign a JSON formatted string to variable REQUEST_BODY. Here is an example of a valid request body in JSON format:

```
{
    "type_definition_id": "74",
    "name": "Applicant_Screening",
    "description": "My new use case description",
    "fields":[
                {
            "name": "MRG-AIFacts-ModelUseCase:Purpose",
            "value": "My new purpose description"
        }
    ]
}
```

Note: The base fields *type_definition_id* and *name* (marked yellow above) must always be specified when you update objects!

With the above request body, the field values (marked green) of the following fields will be updated:

- description
- MRG-AIFacts-ModelUseCase:Purpose

Note: In case you specify an object name that is different from the current name ("Applicant_Screening") the object name would also be updated!

2.1.5 **How to create a new object**

To create new objects you use POST requests. The next example is a POST request that creates a new object. The data related to that object is specified in the request body (marked yellow). *Content-type application/json* indicates that the request body is specified in JSON format:

```
curl --request POST \
  --url 'https://replace_server_name_variable/REPLACE_BASEPATH_VARIABLE/v2/contents \
  --header 'Authorization: Basic REPLACE_BASIC_AUTH' \
  --header 'accept: application/json' \
  --header 'content-type: application/json' \
  --data REQUEST_BODY
```

Before you can execute the above command you need to assign a JSON formatted string to variable REQUEST_BODY. Below is an example of a valid request body. The *type_definition_id* specifies the type of object that you want to create (type_definition_id *58* is related to objects of type *model*). Primary_parent_id specifies the id of the parent object (this could be for example a *use case* or a *business entity*):

```
{
    'name': 'Customer_Segmentation',
    'description': 'Some ML model description',
    'type_definition_id': '58',
    'primary_parent_id': '4577'
}
```

When you execute the above REST call it returns a long list of all field values (attribute values) of the new object. Among these values is the **object ID** (marked yellow below). You can use this ID to access this object later on (using REST calls or using the Governance Console). Here is an example of the data returned by the above REST call (the data was truncated to fit on this page!):

```
{
    "name": "Customer_Segmentation",
    "id": "4628",
    "path":
    "/_op_sox/Project/Default/ICDocumentation/Models/Your_Enterprise/Customer_Segmentation.txt",
    "creation_date": "2025-02-20T15:12:51.000Z",
    "modified_date": "2025-02-20T15:12:51.000Z",
    "description": "Customer_Segmentation",
    "parent_folder_id": "4578",
    "creator": "admin",
    "last_modified_by": "admin",
    "fields": [
        {
            "id": "28",
            "data_type": "ID_TYPE",
            "name": "Resource ID",
            "has_changed": false,
            "value": "4628"
        },
        {
            "id": "65",
            "data_type": "STRING_TYPE",
            "name": "Comment",
            "has_changed": false
        …
```

The Jupyter notebook provides some example of how to process the data returned by the REST calls. For example, it shows how to extract the ID of the object that was created by a REST call.

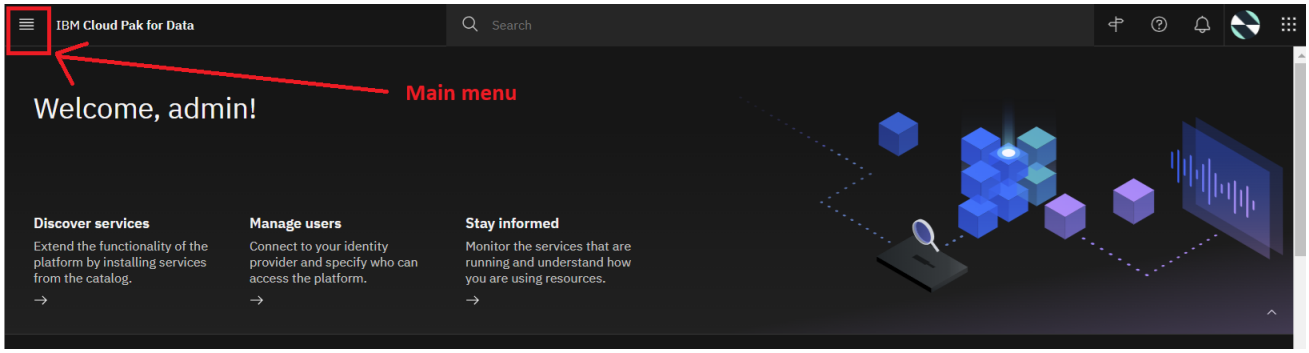
2.1.6 **Execute the Python code examples (Jupyter notebook)**

The Jupyter notebook is included in the Git repository. Check section 1.5.5 on how to download the repository to you local computer (ZIP file). Extract the contents of the ZIP file. Among other files it includes the following project file:

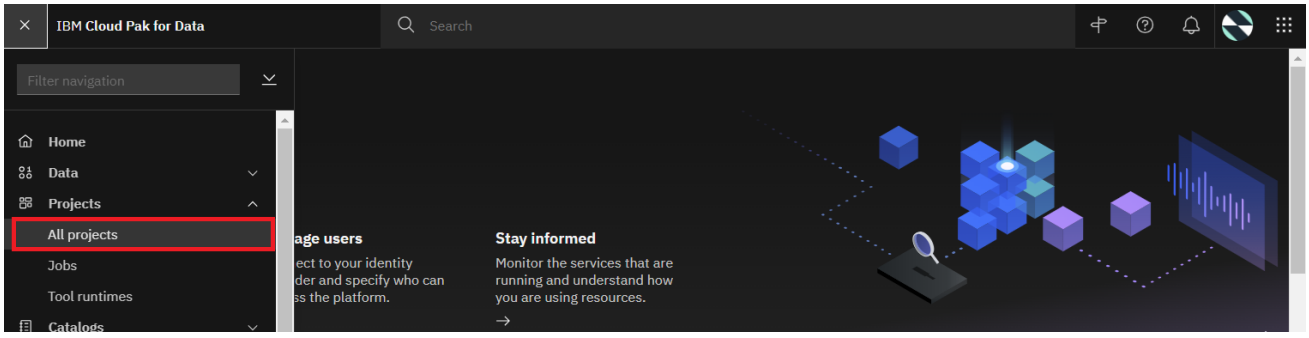
```
REST-API-Lab---Governance-Console.zip
```

After you have extracted the project file, perform the following steps:

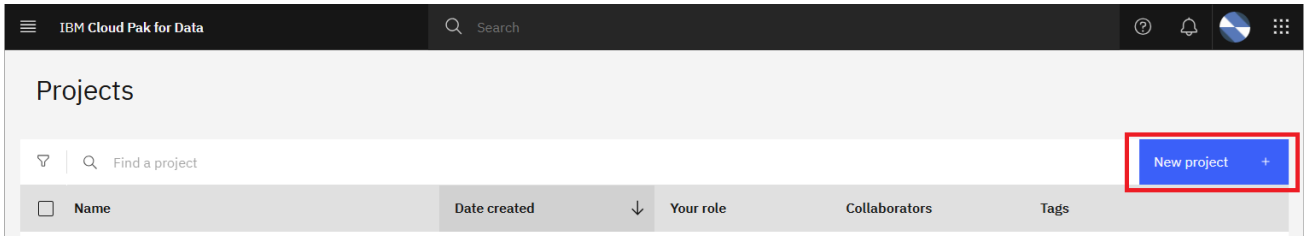
- Logon to the *Cloud Pak for Data console* of your lab environment (see description in section 1.5.2). This takes you to the CP4D home screen. Open the main menu (click the icon marked red):



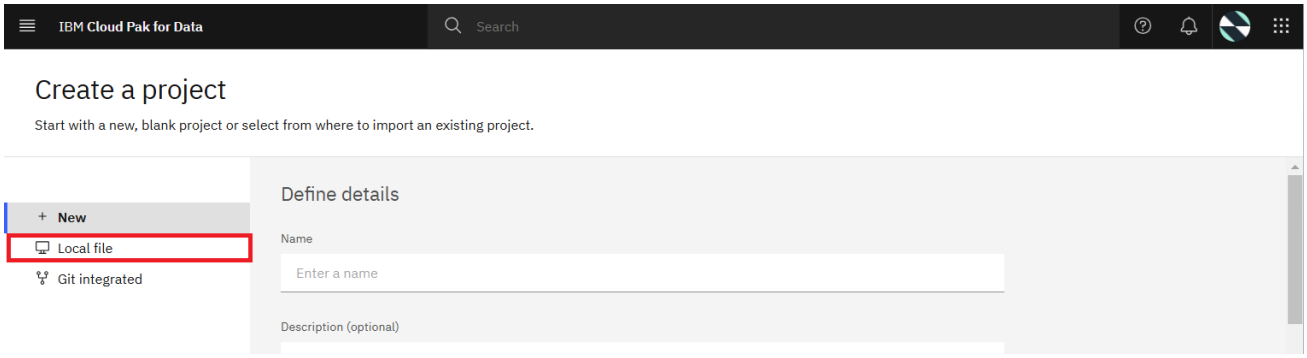
- Open the main menu and click



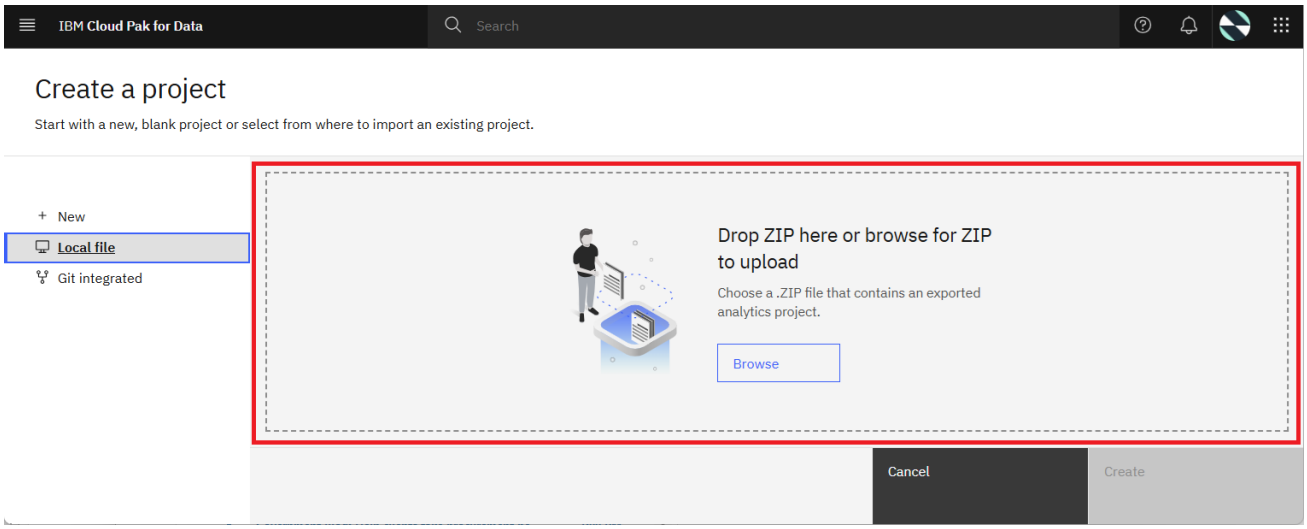
- Click “New project”:



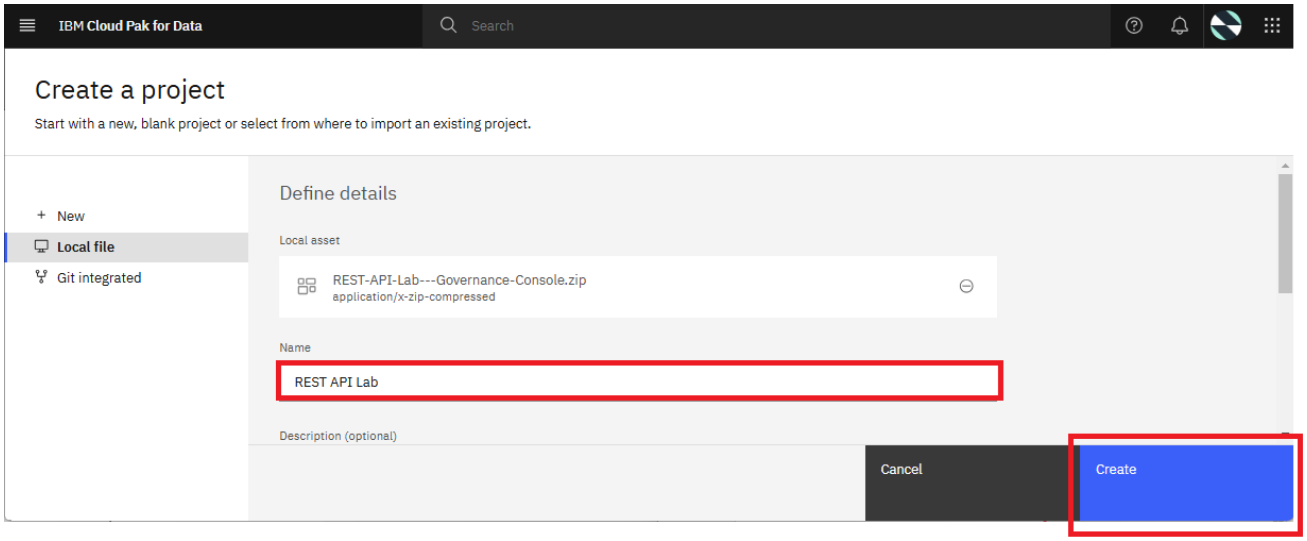
- Click “Local file”:



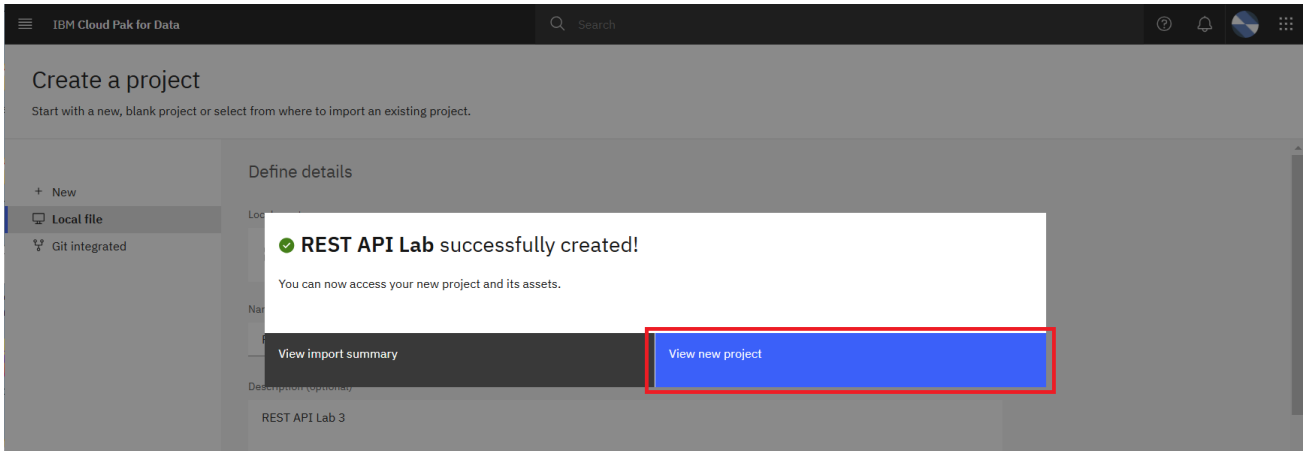
- Drag & drop the project file into the area marked red:



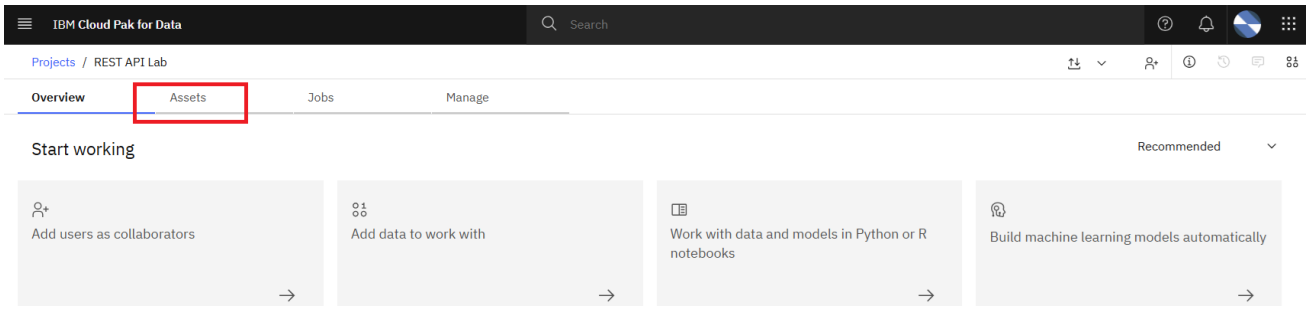
- Enter a project name (for example “REST API Lab”) and click *Create*:



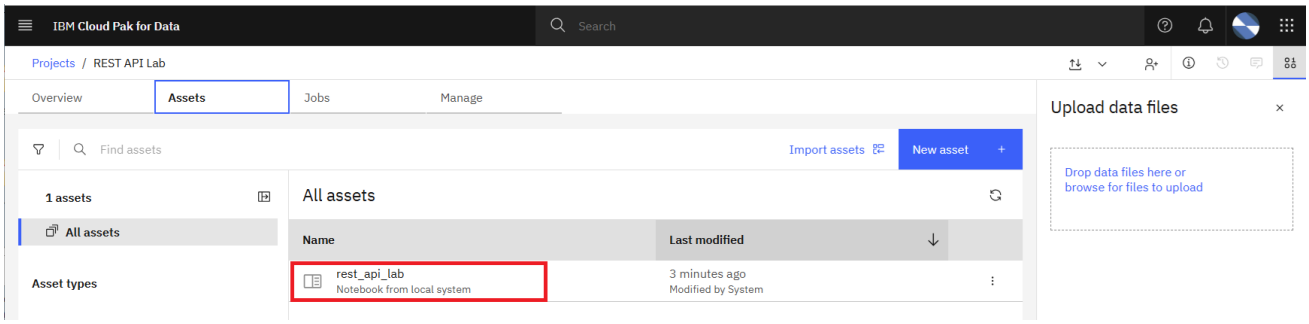
- Click *View new project*:



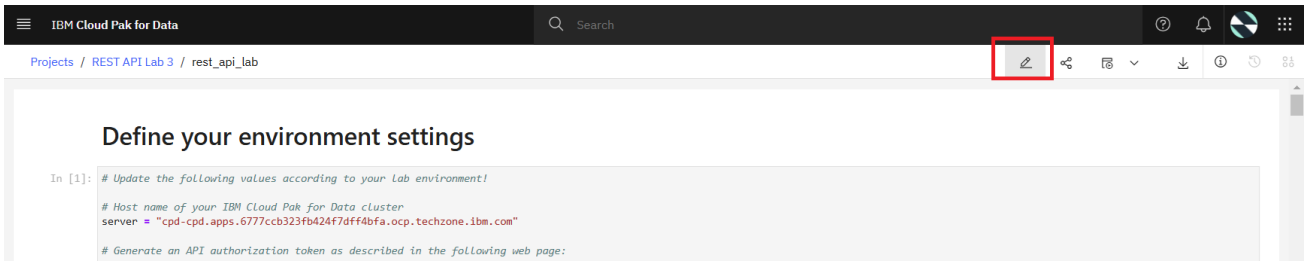
- Click *Assets*:



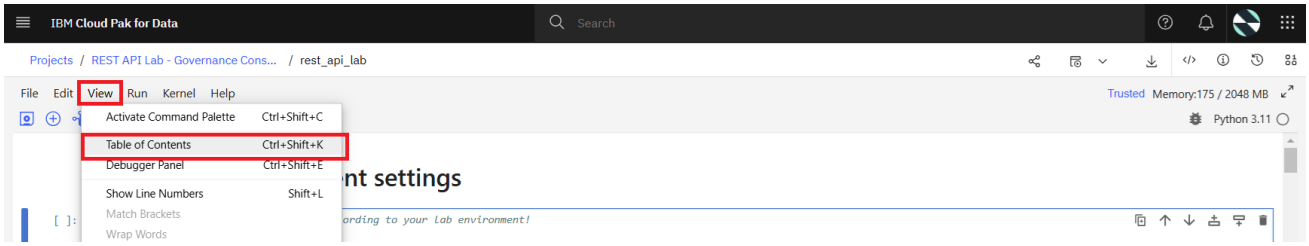
- To open the Jupyter notebook click *rest_api_lab*:



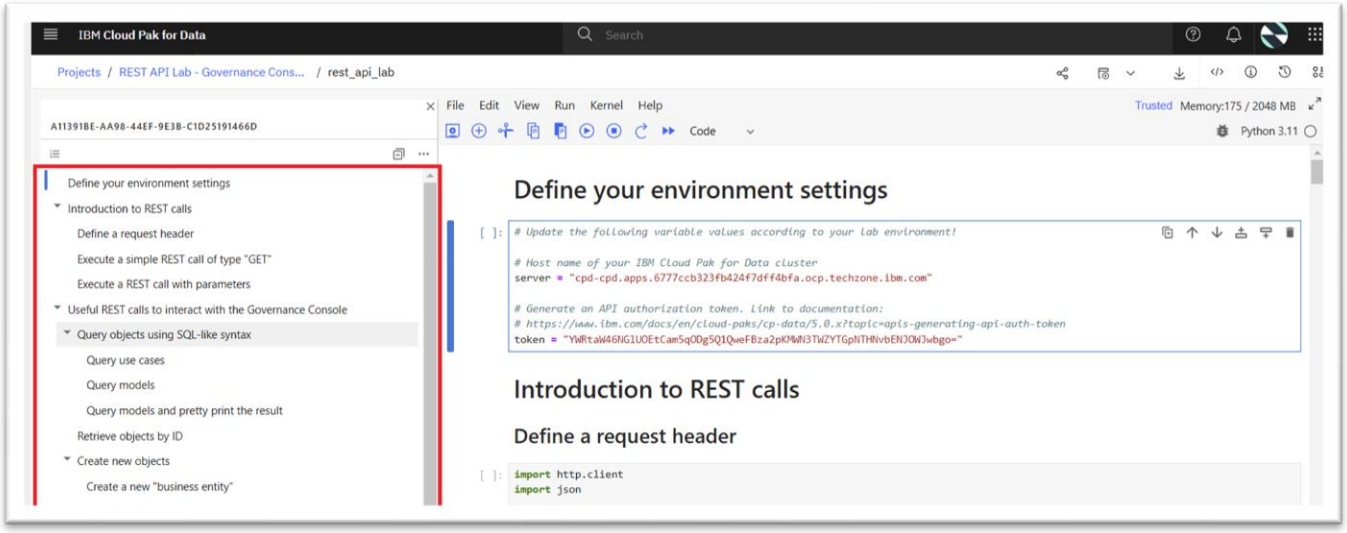
- Click the pencil icon in the upper right corner:



- Click *View* and the click *Table of Contents*:

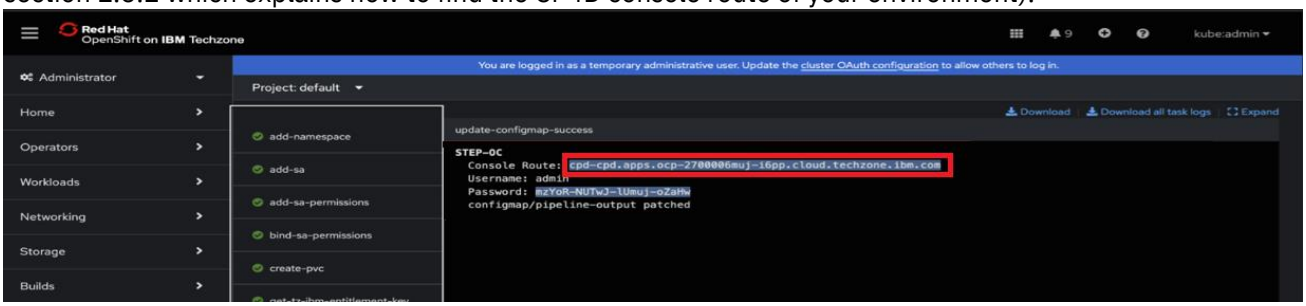


- This opens the table of contents. Here you can click a section headline to jump to the corresponding section in the notebook:



Note: In the first code cell of the notebook you need to update the values of the variables ***server*** and ***token***.

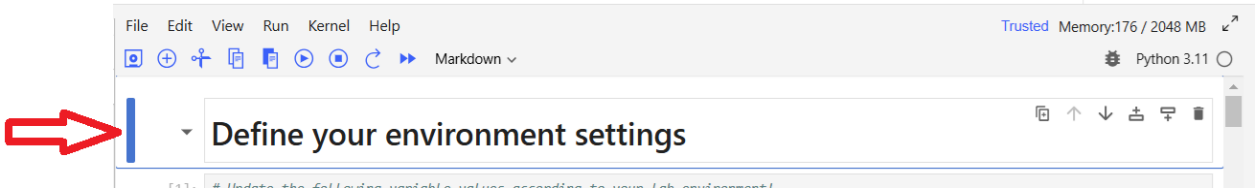
- The value of *server* must be set to the *Console Route* of your Cloud Pak for Data environment (see section 1.5.2 which explains how to find the CP4D console route of your environment):



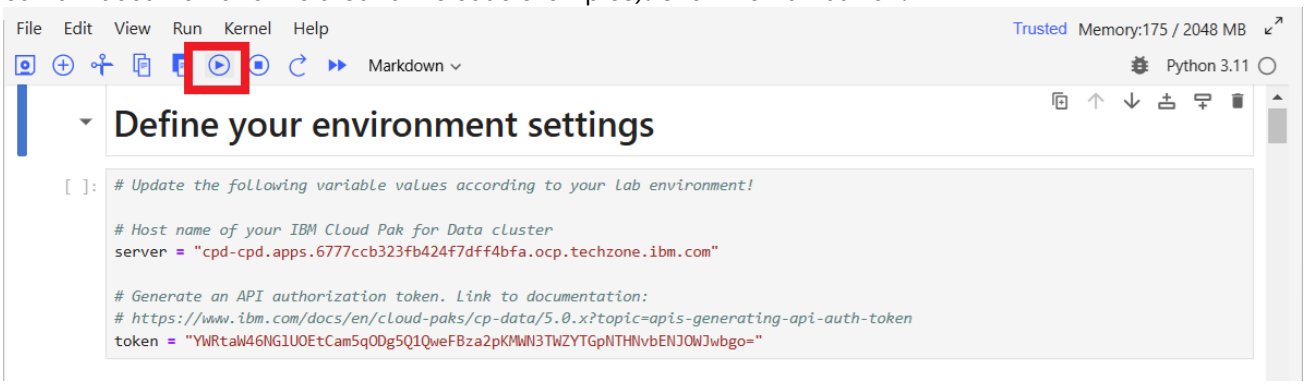
  The value of variable *token* must be replaced with your ***authorization token*** (see section 1.5.4 on how to create your authorization token).

Now you can start to execute the code examples in the Jupyter notebook from top to bottom:

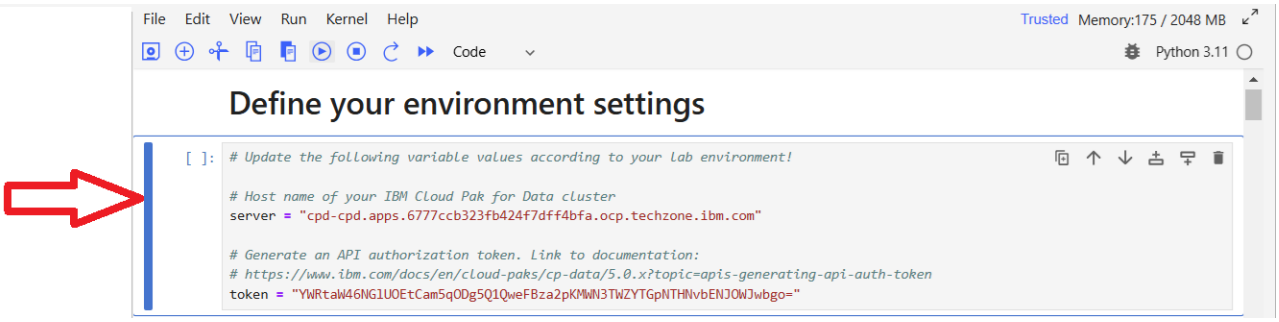
- The blue bar on the left always indicates the currently selected cell:



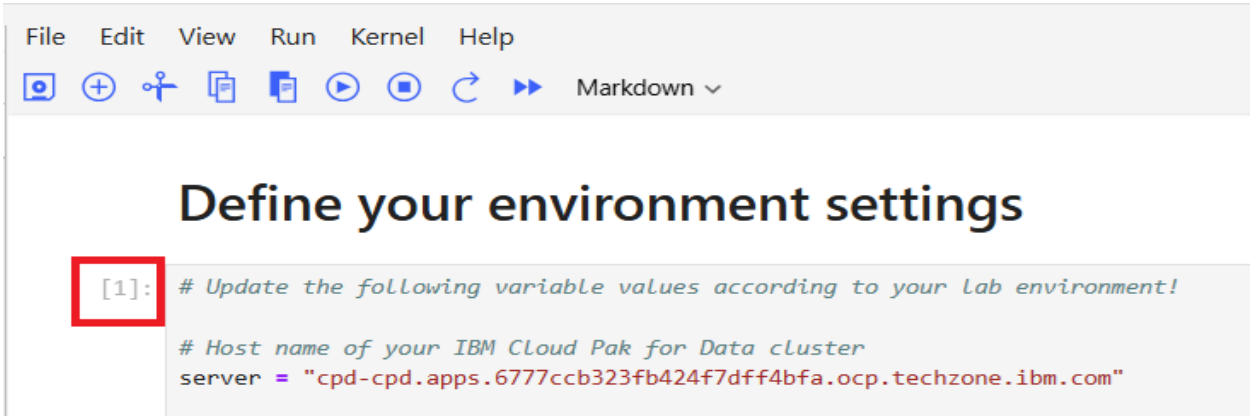
- There are two types of cells: *code cells* (which contain Python program code) and *Markdown cells* (which contain documentation related to the code examples). Click the Run button:



- If the currently selected cell is a Markdown cell, then the next cell is selected. If the currently selected cell is a code cell, the code is executed and then the next cell is selected:



  - After a code cell was executed a number appears between the square brackets on the left of the code cell. This indicates, that the cell was executed:



  - You can now continue to click the run button to walk though all of the Python code examples.

2.2 Java API Exercises

2.2.1 Lab overview

This part of the lab provides an introduction to *custom workflow actions*. Custom workflow actions can be attached to the actions of a workflow. They contain custom Java code that can perform any kind of activity.
Since Java code development is not an easy task, we have prepared a generic custom workflow action for you. That means you don't need to perform any Java programming, compilation and packaging. You just use the generic workflow action that is part of the Git repository (*RestCallAction.jar*). This generic workflow action performs a REST call to the REST server on your virtual machine environment. On the virtual machine, you can implement any custom code in *Python* instead of Java. This is much easier to develop, to deploy and to test.

As part of the lab you will perform the following tasks:

- Copy the custom workflow action to your Cloud Pak for Data lab environment
- Create a simple workflow
- Attach the custom workflow action to some actions (submit, approve) of the workflow
- Start the REST server: the REST server will receive a REST call from the custom workflow action whenever the corresponding workflow action is executed by a user
- Start the workflow and execute some of the workflow actions: As a result, you will see some messages popping up on the REST server. This proves, that the REST calls were received by the REST server.
- Optionally: Compile and package your own custom workflow action (the virtual machine contains a Java Development Kit and the required libraries).
- Optionally: Implement your own Python code on the REST server to perform any kind of tasks related to the workflow actions:
    - For example, if a user approves a new "AI use case" your Python code could automatically send a notification to a user.
    - Or if someone approves a "model" for deployment your custom Python code could trigger the actual deployment process on an external ModelOps system.

2.2.2 Copy the custom workflow action to the CP4D cluster

Part of the Java API lab is the deployment of a custom workflow action (JAR file). For the deployment (see lab exercises further down) you need to copy the OpenShift login command of your CP4D lab environment.

2.2.3 Create a simple workflow in the Governance Console

2.2.4 Attach the custom workflow action to the workflow

2.2.5 Start the REST server

Execute the workflow and check the results

2.2.6 **Run the custom workflow actions and check the results**

Overview of this exercise:
Steps on the Virtual Machine:

1. Update the web server
2. Start the web server

Steps on your local computer:

3. Test the web server by executing a REST call from you local web browser

Steps in the Governance Console:

4. Adjust workflow “Sample workflow” (define proper IP address)
5. Save and publish the modified workflow
6. Create a new use case -> *This starts the “Sample Workflow”*
7. Open the new use case and select action “Submit_For_Review” -> this will trigger the custom workflow action which executes a REST call. You will see the incoming REST call on the web server that is running on your virtual machine.

Update the web server:
git clone https://github.com/andreas-christian/WxGov-API-Lab

Modify the property “ip_address” with the Public IP address of your VM. This needs to be changed in the following actions of the workflow:

- Submit_For_Review
- Return_For_Update
- Approve

Name	RestCallAction
Class Name	com.ibm.openpages.ext.custom.workflow.action.RestCallAction

Properties of the custom workflow actions:

username	itzuser
ip_address	<Public IP adress of your VM>
endpoint	/Submit_For_Review

username	itzuser
ip_address	<Public IP adress of your VM>
endpoint	/Return_For_Update

username	itzuser
ip_address	<Public IP adress of your VM>
endpoint	/Approve

2.2.7 **Optional: Implement your own custom actions using Python**

2.2.8 **Optional: Implement your own custom actions using Java**

3 Appendix

3.1 Supported *Type IDs* and *Type Names*

Type ID	Type Name	Description
23	Assertion	Assertion
24	Attestation	Attestation
27	AuditPhase	Audit Section
25	AuditProgram	Audit
28	AuditableEntity	Auditable Entity
29	Auditor	Auditor
108	BCBusinessImpactAnalysis	Business Impact Analysis
109	BCEvent	Business Continuity Event
110	BCPlan	Business Continuity Plan
111	BCTest	Business Continuity Test Plan
112	BCTestResult	Business Continuity Test Result
115	BusService	Business Service
116	BusServiceEval	Business Service Eval
31	Campaign	Campaign
32	Challenge	Challenge
33	ChangeRequest	Change Request
34	Committee	Committee
35	CompliancePlan	Compliance Plan
117	CompliancePlanEval	Compliance Plan Eval
67	ComplianceReviewComment	Compliance Review Comment
36	ComplianceTheme	Compliance Theme
118	ComplianceThemeEval	Compliance Theme Eval
37	Contract	Contract
42	CostCenter	Cost Center
39	CtlEval	Control Eval
123	DisclosureStatement	Disclosure Statement
43	Employee	Employee
44	Engagement	Engagement
16	ExporterXML	ExporterXML
46	FIRSTLoss	FIRST Loss
45	Finding	Finding
47	Incident	Incident
48	KeyPerfIndicator	KPI
49	KeyPerfIndicatorValue	KPI Value
50	KeyRiskIndicator	KRI
51	KeyRiskIndicatorValue	KRI Value
113	Location	Location
52	LossEvent	Loss Event
53	LossImpact	Loss Impact
54	LossRecovery	Loss Recovery
55	Mandate	Mandate

56	Metric	Metric
57	MetricValue	Metric Value
17	MigrationJAR	MigrationJAR
58	Model	Model
106	ModelAttestation	Model Attestation
59	ModelInput	Model Input
60	ModelLink	Model Link
61	ModelOutput	Model Output
107	ModelScorecard	Model Risk Scorecard
119	ModelVersion	Model Version
62	ORICLoss	ORIC Loss
63	ORXLoss	ORX Loss
125	Objective	Objective
121	OblEval	Obligation Evaluation
122	OblEvalValue	Obligation Evaluation Value
120	Obligation	Obligation
64	Plan	Plan
65	Policy	Policy
66	PolicyReviewComment	Policy Review Comment
69	PrefGrp	Preference Group
68	Preference	Preference
70	Procedure	Procedure
21	ProcessDiagram	ProcessDiagram
72	ProcessEval	Process Eval
124	Product	Product
15	Program	Program
73	Project	Project
13	QuestionTemplate	Question Template
14	QuestionnaireAssessment	Questionnaire Assessment
10	QuestionnaireTemplate	Questionnaire Template
91	RAEval	Risk Assessment Eval
87	RICat	RI Component
88	RIReq	RI Sub-Component
76	RegApp	Regulation Applicability
79	RegChange	Regulatory Change
78	RegInt	Regulator Interaction
80	RegTask	Regulatory Task
74	Register	Use Case
77	Regulator	Regulator
128	RegulatoryEvent	Regulatory Event
75	RegulatoryInitiative	Regulatory Initiative
3	Report	Report
82	ReqEval	Requirement Evaluation
83	ReqEvalValue	Requirement Evaluation Value
81	Requirement	Requirement
84	Resource	Asset

85	ResourceLink	Asset Link
86	Review	Review
26	ReviewComment	Audit Review Comment
90	RiskAssessment	Risk Assessment
41	RiskEntity	System
92	RiskEval	Risk Eval
30	RiskSubEntity	Baseline
22	SOXAccount	Account
5	SOXBusEntity	Business Entity
38	SOXControl	Control
40	SOXControlObjective	Control Objective
4	SOXDocument	File
8	SOXExternalDocument	Link
6	SOXIssue	Issue
71	SOXProcess	Process
2	SOXProject	SOXProject
89	SOXRisk	Risk
9	SOXSignature	Signature
95	SOXSubaccount	Sub-Account
97	SOXSubprocess	Sub-Process
7	SOXTask	Action Item
98	SOXTest	Test Plan
99	SOXTestResult	Test Result
93	ScenarioAnalysis	Scenario Analysis
94	ScenarioResult	Scenario Result
11	SectionTemplate	Section Template
12	SubSectionTemplate	SubSection Template
130	Subcontractor	Subcontractor
96	Submandate	Sub-Mandate
126	SummaryAuditPlan	Summary Audit Plan
1	SysXMLDocument	SysXMLDocument
114	Team	Team
127	Threat	Threat
100	Timesheet	Timesheet
101	Usage	Model Deployment
141	UseCaseReview	Use Case Review
102	Vendor	Vendor
129	VendorSubsidiary	Vendor Subsidiary
103	Vulnerability	Vulnerability
104	Waiver	Waiver
105	Workpaper	Workpaper

3.2 Software included in the VM and related installation steps

t.b.d