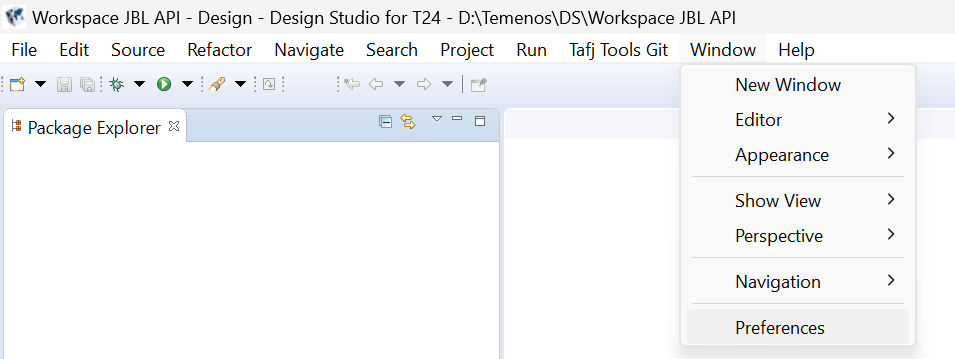
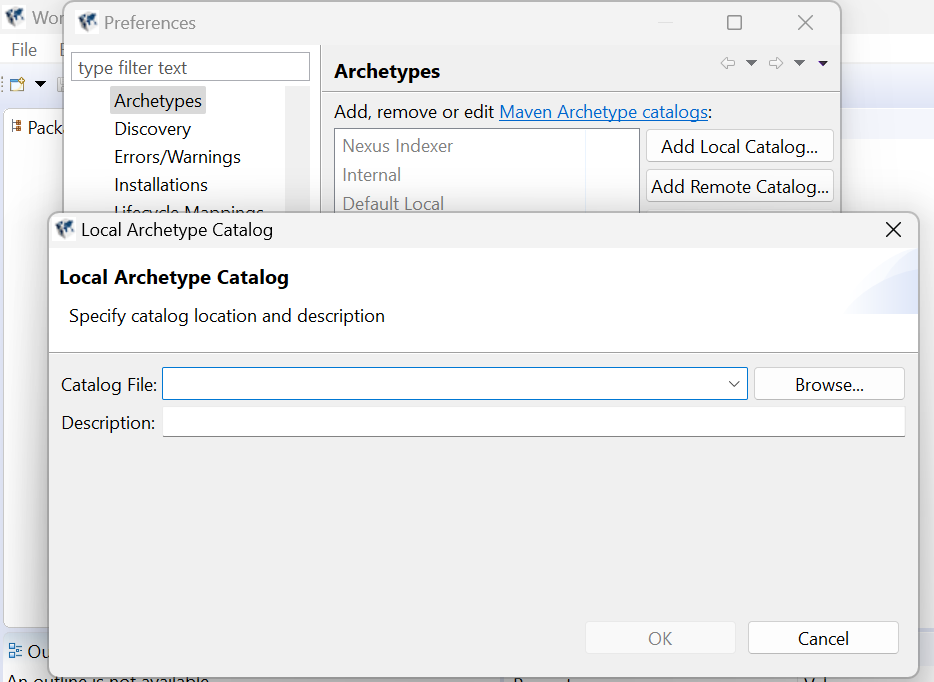
**IRIS API**

**Adding Archetype in Design Studio:**

****

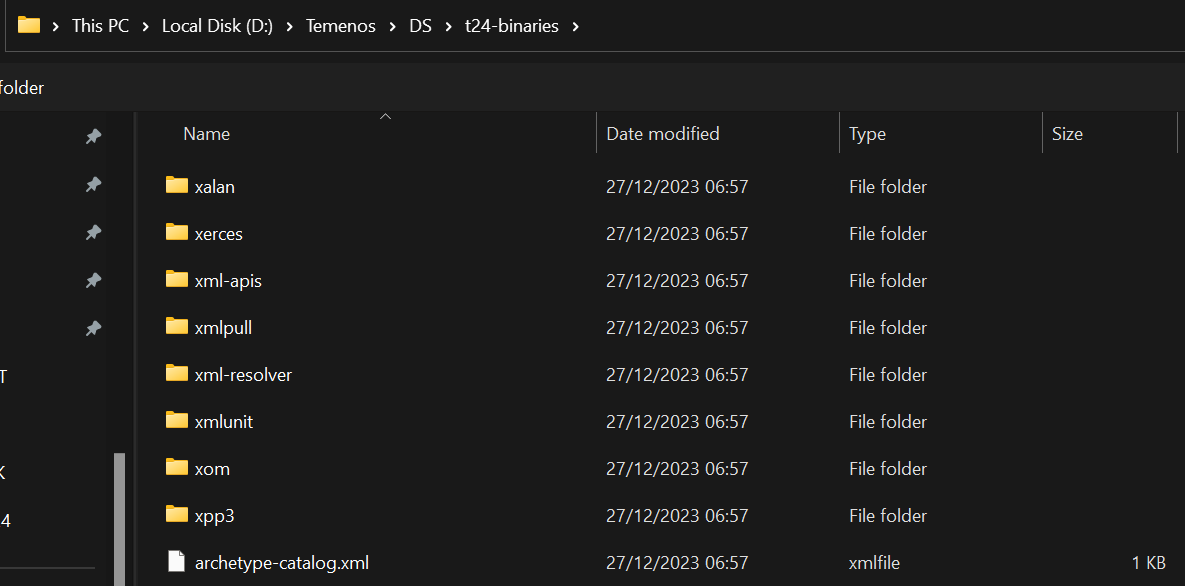


**Window > Preferences > Archetypes > Add Local Catalog**

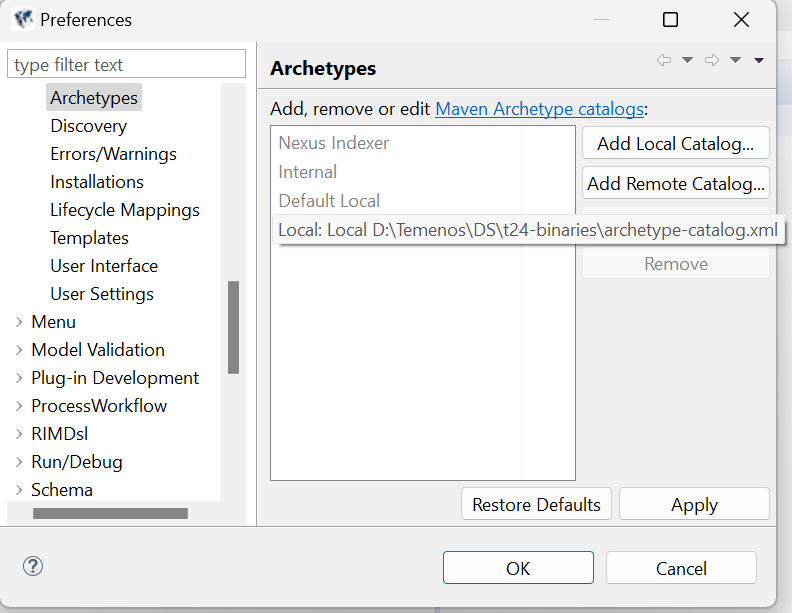
****



**T24-binaries > D:\Temenos\DS\t24-binaries -- archetype-catalog.xml**

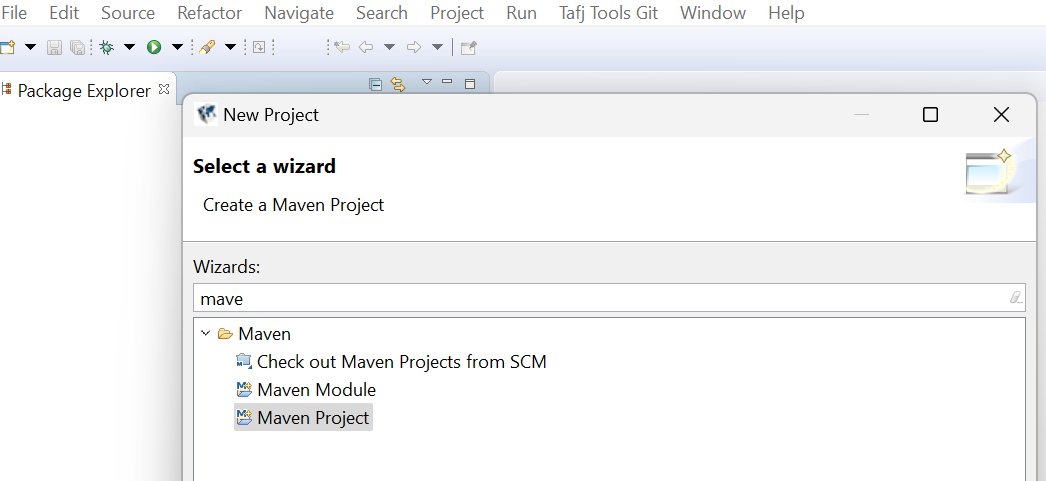
****



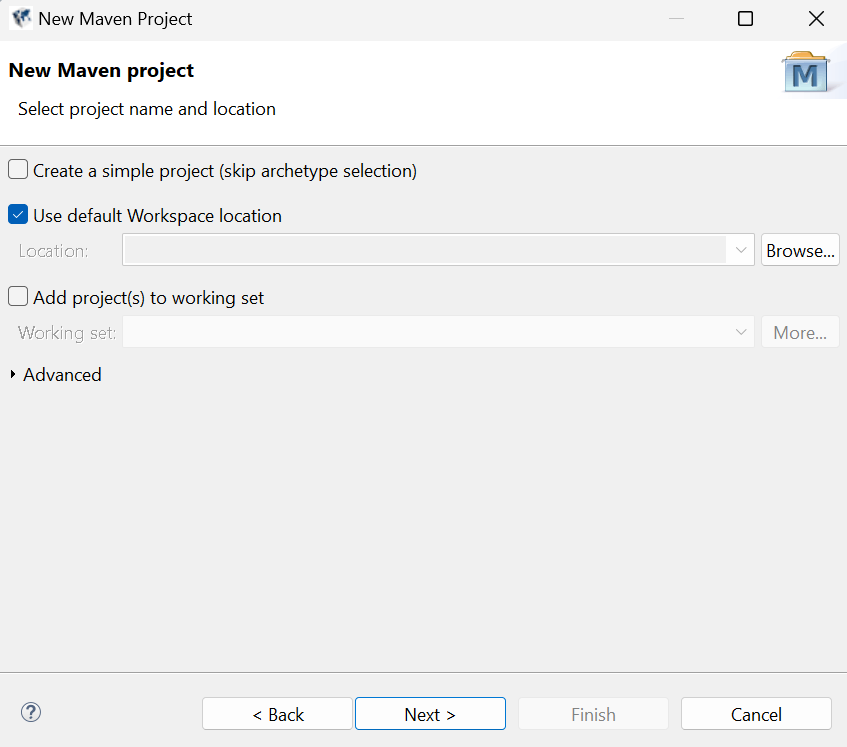
****



**New Mavel Project from DS:**

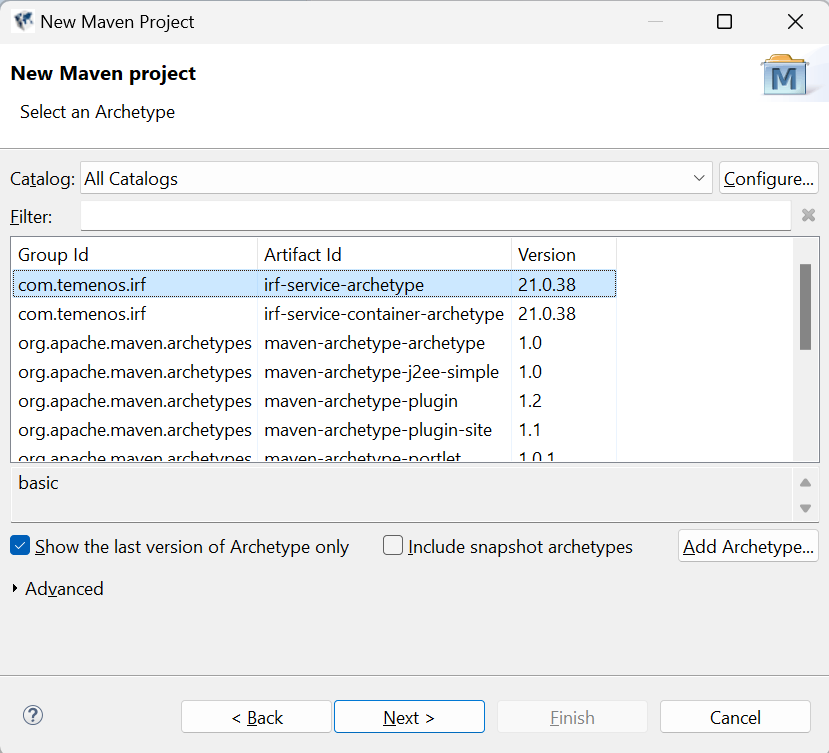
****



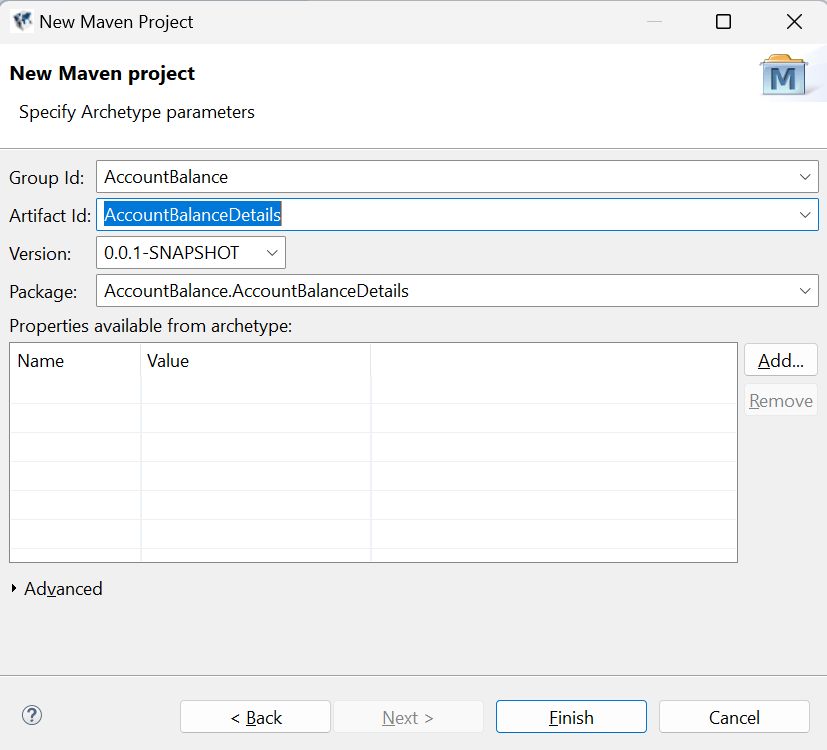
****



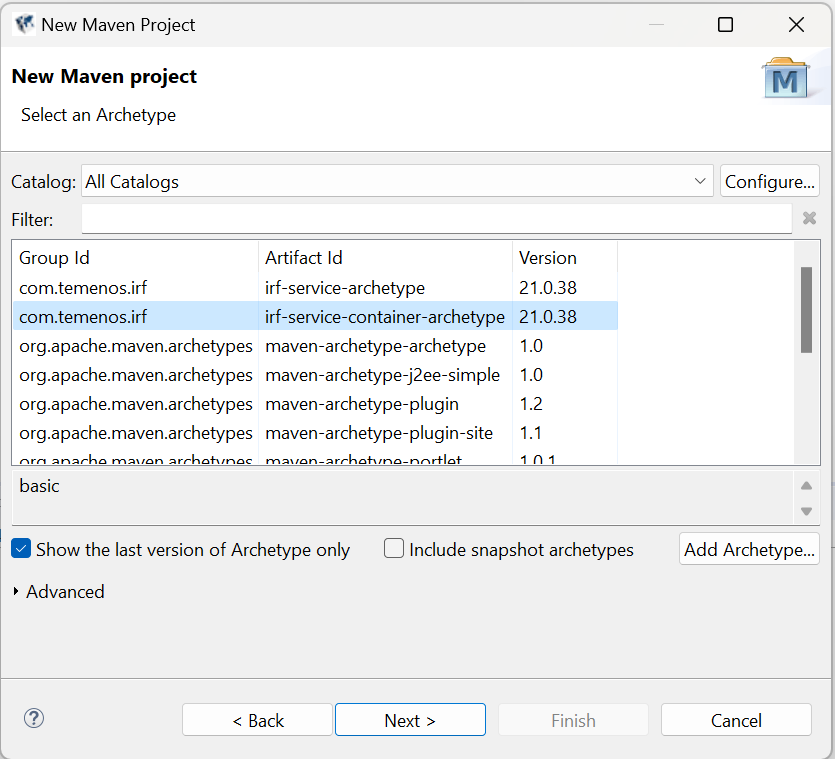
There must be two **Maven** Projects (irf-service-archetype & irf-service-container -archetype)

****

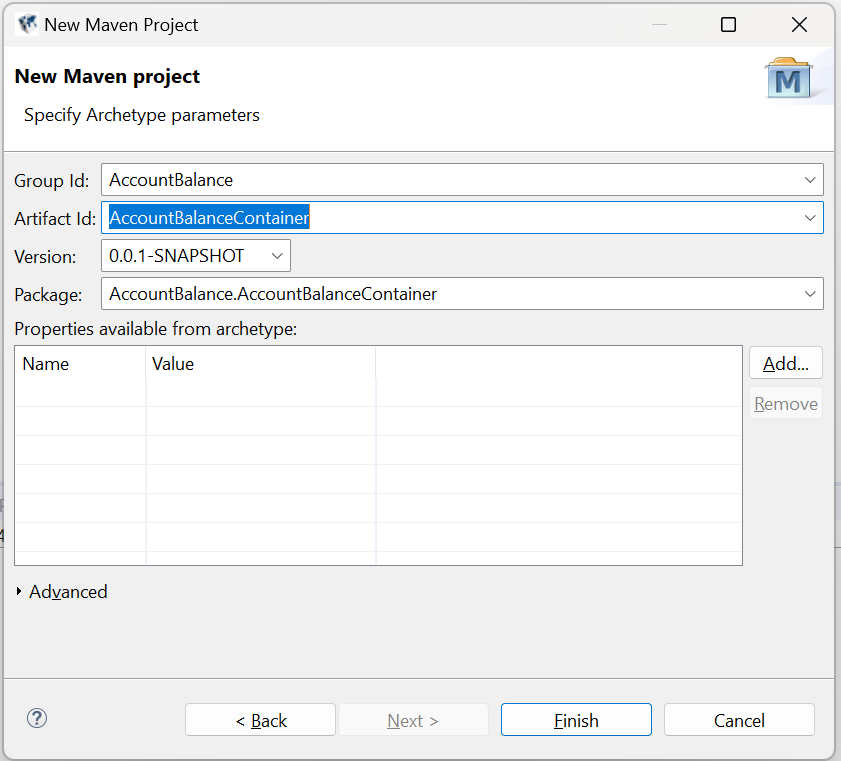


****



****

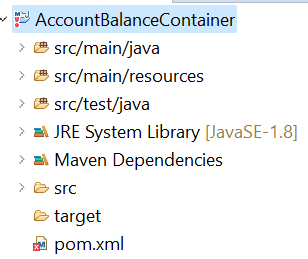


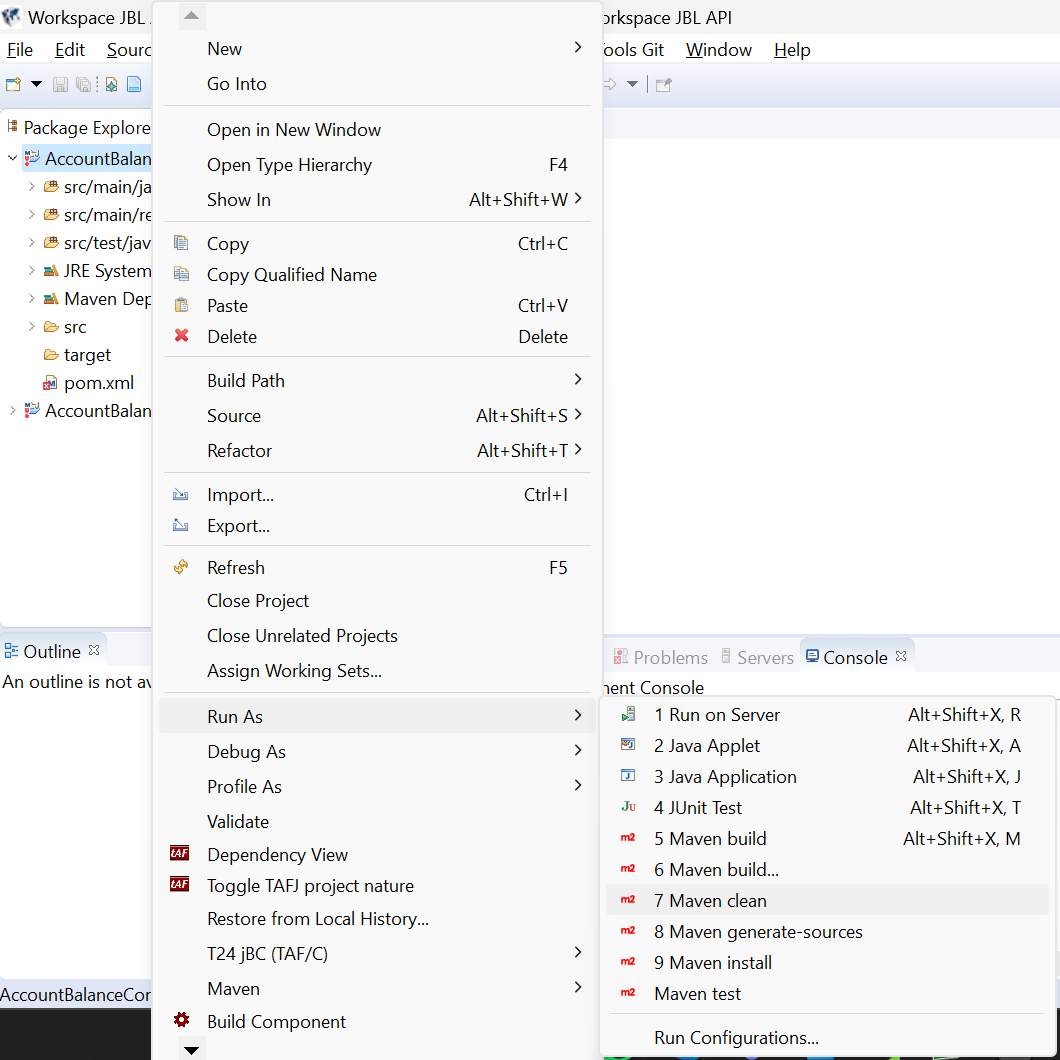
****

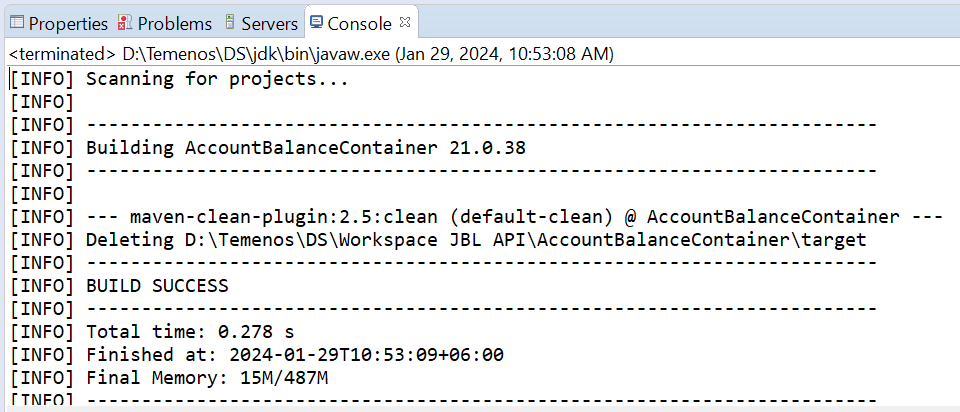


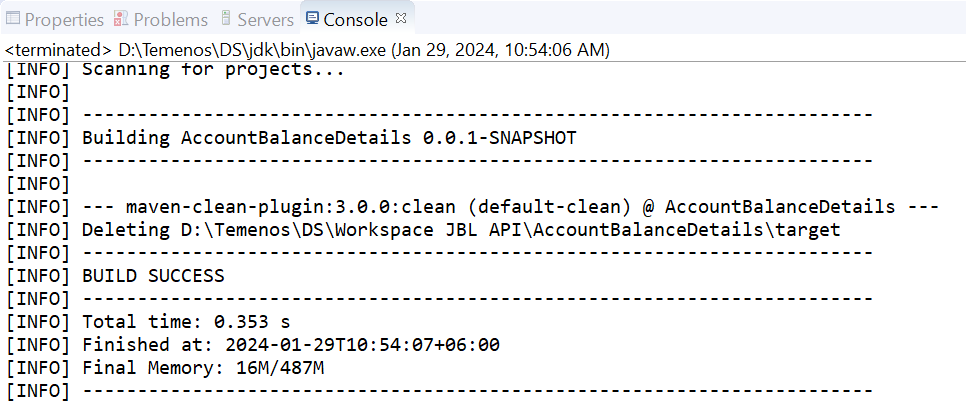
**Clean the both container and Service project by Selecting,**

**Right click on Project → Run As → Maven Clean.**

****

****

****

****

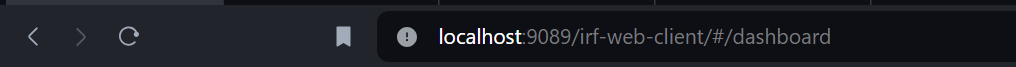
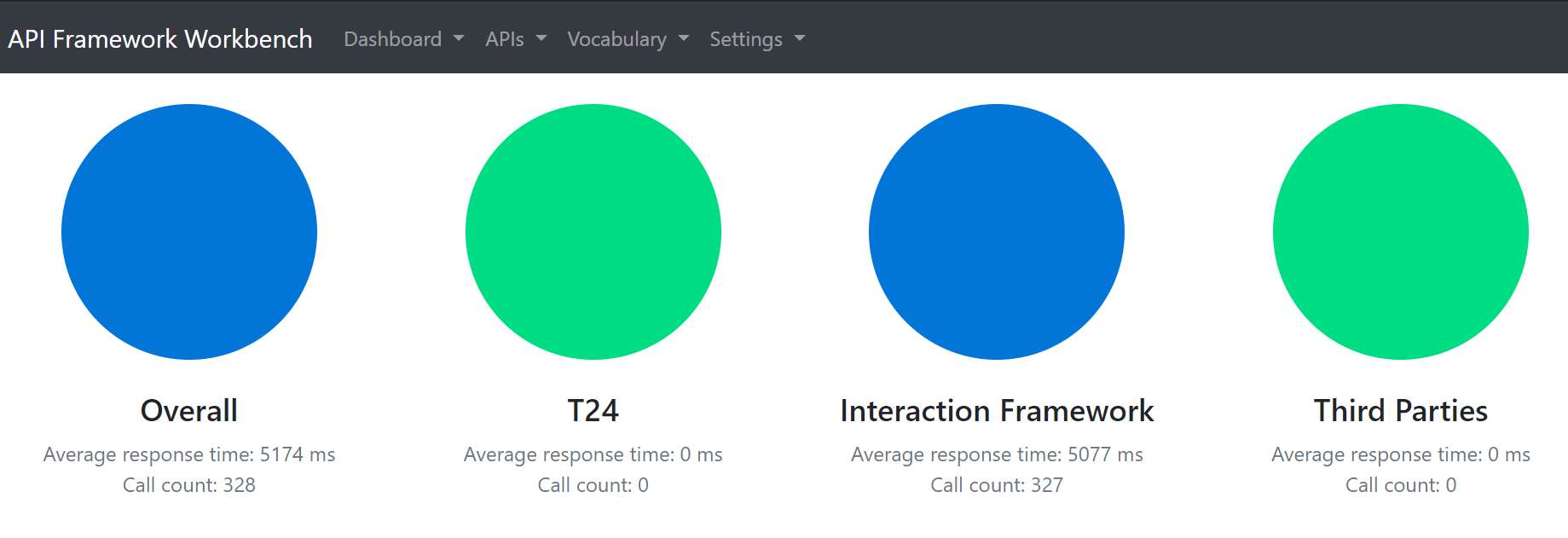
***IRIS 2.0 war File Generation***

Deploy the 2 war files which is given by temenos in your JBoss

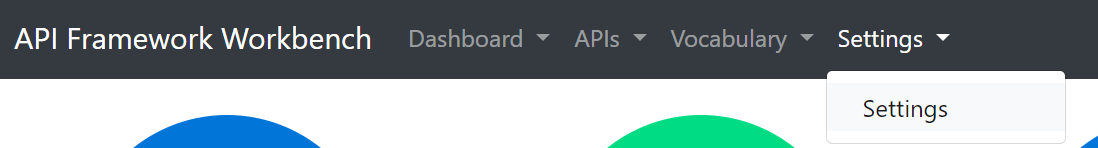
**irf-web-client - Workbench**

**irf-test-web - To test the IRIS Services**

Give the base URL in the browser-> Workbench gets opened…

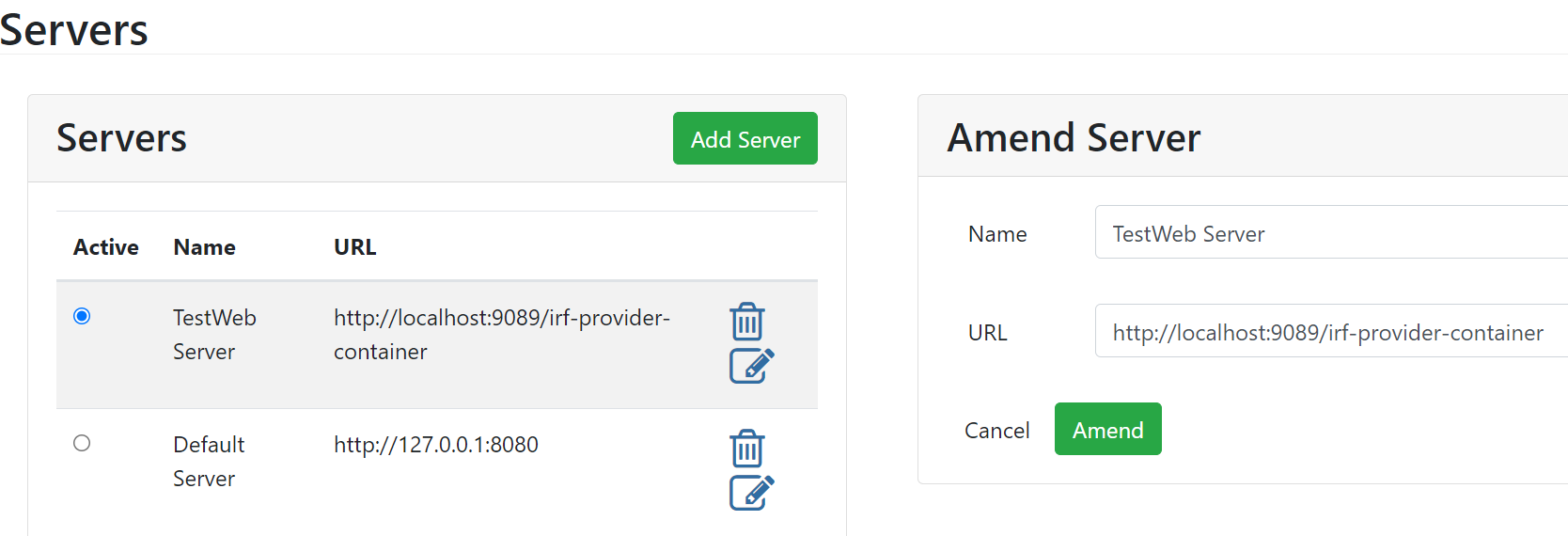
 







**Select settings tab → click settings → Then the servers page gets opened.**

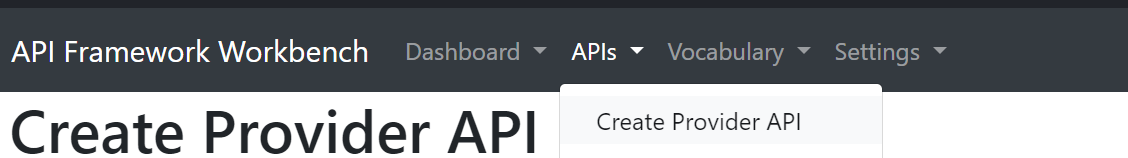
****



**API Creation from Workbench**

**API Creation:**   
First We should create a Version or Enquiry in below format.  
**Product.API.verb.version** -----EX: ST.API.CUSTOMERS.1.0.0

go to localhost:9089/irf-web-client  
Click API>create provider API.

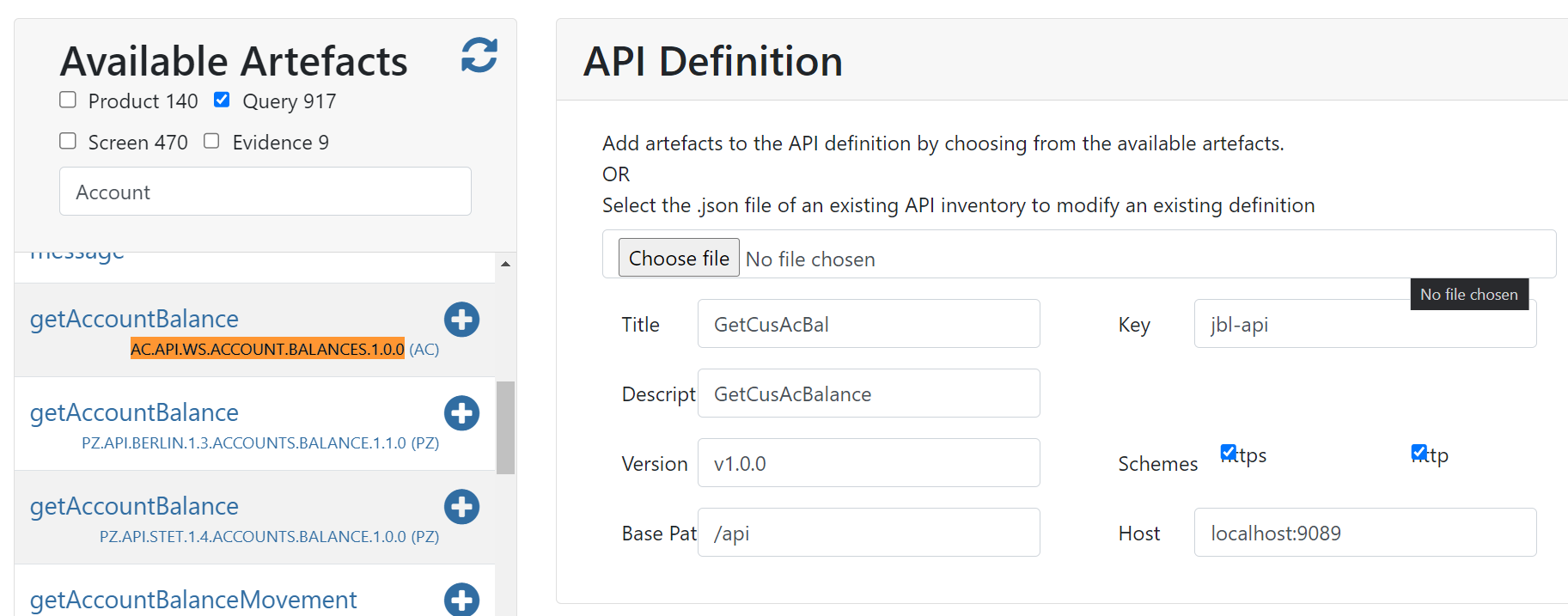




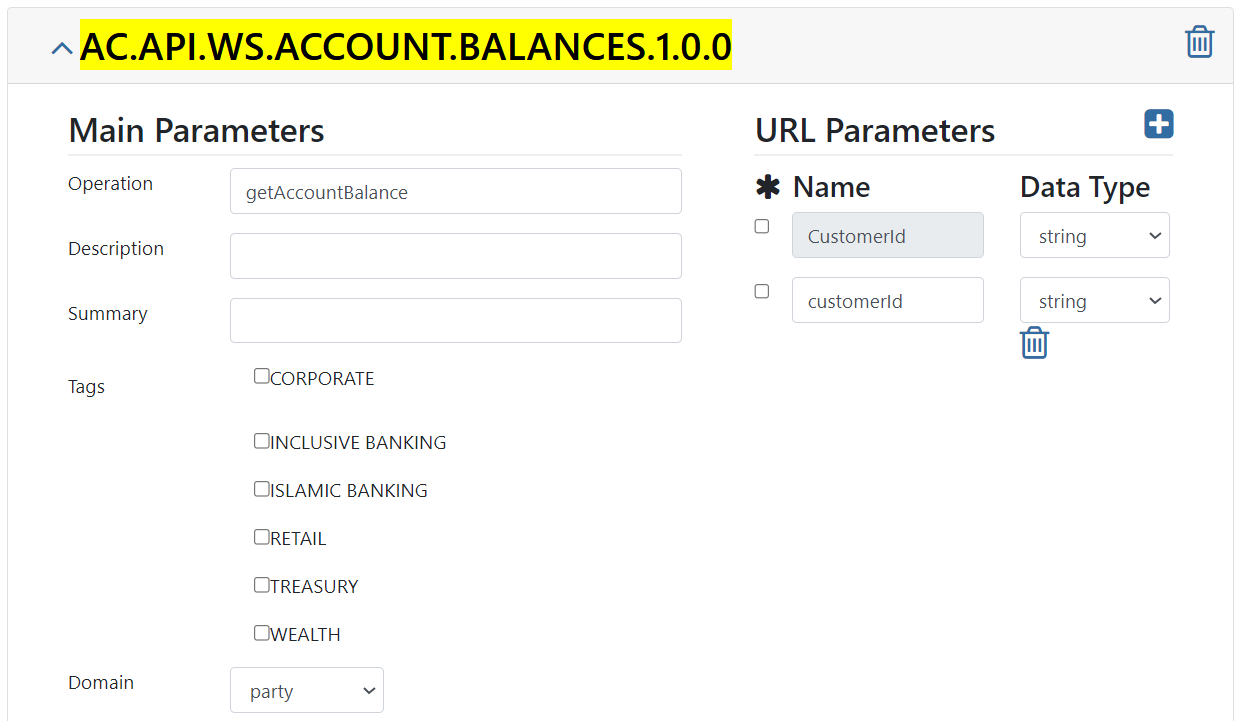
The Available Artefacts can be displayed. The Artefacts can be Versions(Screen), Enquiries(Query),AA Product(Product) and so on.

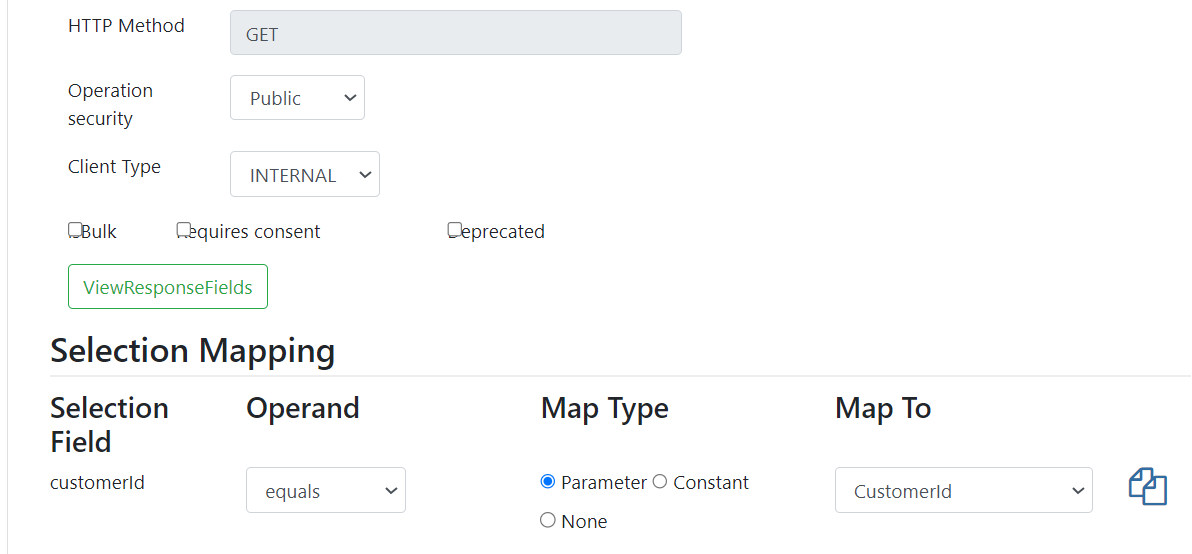
***Select the data’s which you want to import.***

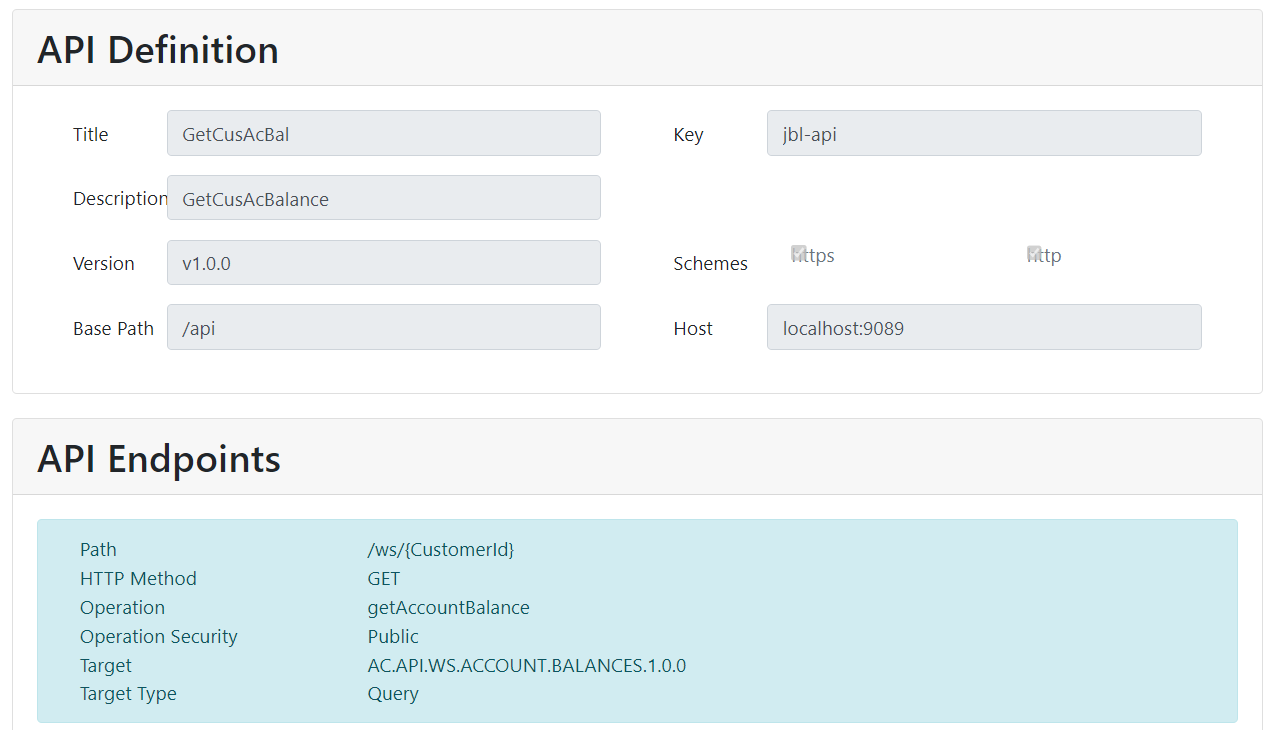
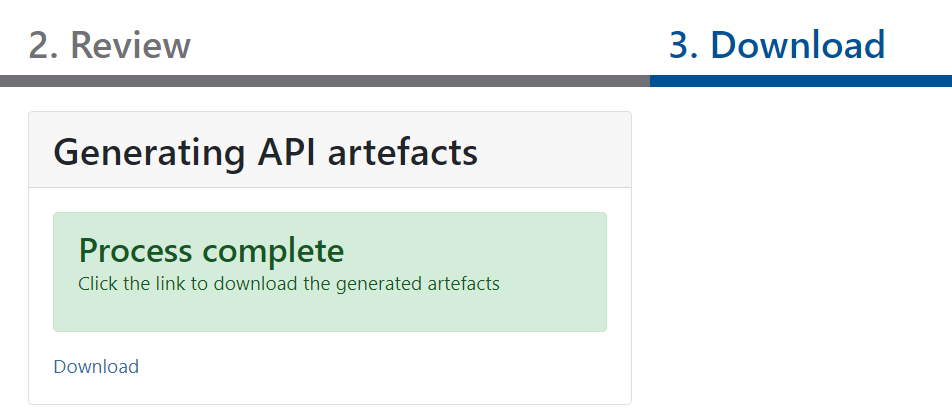
***The API definition page will be displayed.***  
Select the required artefacts from available artefacts.

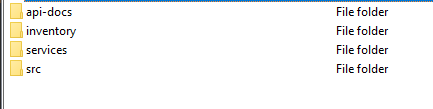
******

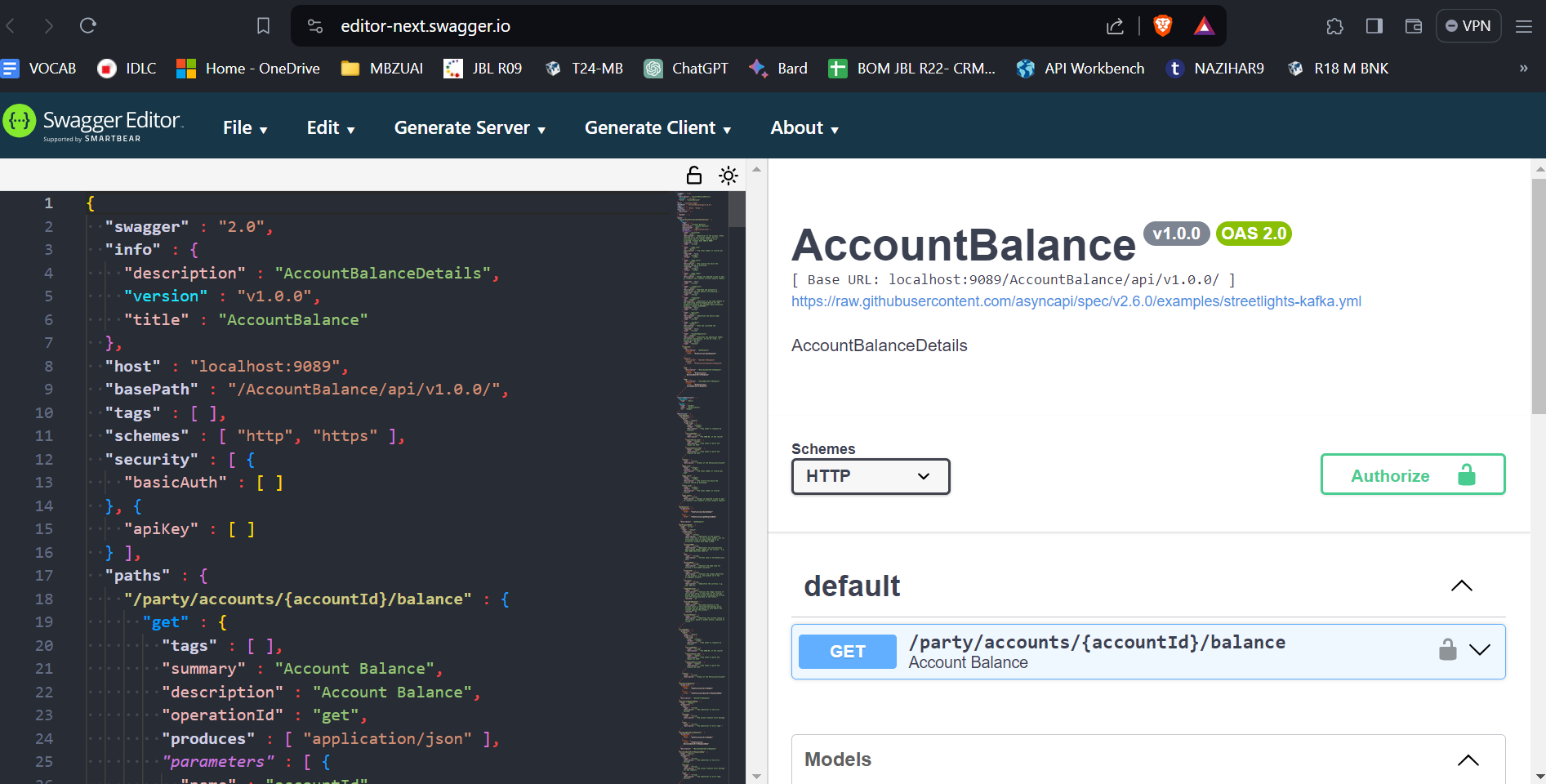
Provide key , url, summary, description. (**give request and response payload versions If it is AA API**)



  
click **NEXT** on top of the screen>**FINISH**

  
  
 IRIS Workbench will generate a zip based on selected transact artefacts. Download it.  
That zip contains 3 folders. **i.e. api-docs, inventory, services**

After creating a **Maven** project (**AccountBalanceDetails**),  
Delete all files in src/main/resources  
  
**api-docs** – This folder contains swagger which will show the specification of our API  
**Ex**:

****

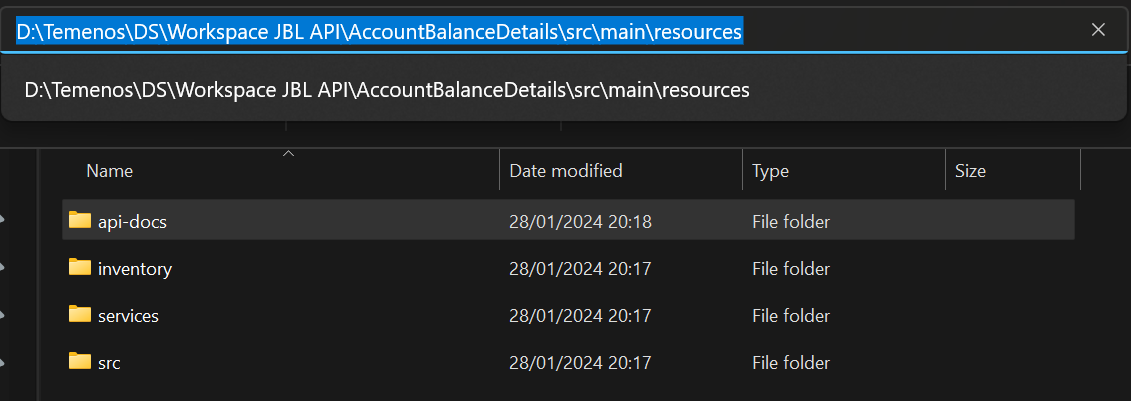
**Inventory** – This folder contains details of the generated artefacts/APIs. Which is useful to load existing APIs to the workbench rather than selecting artefacts from scratch.

**Services**- This folder contains the xml routes for routing the API to appropriate transact artefact.

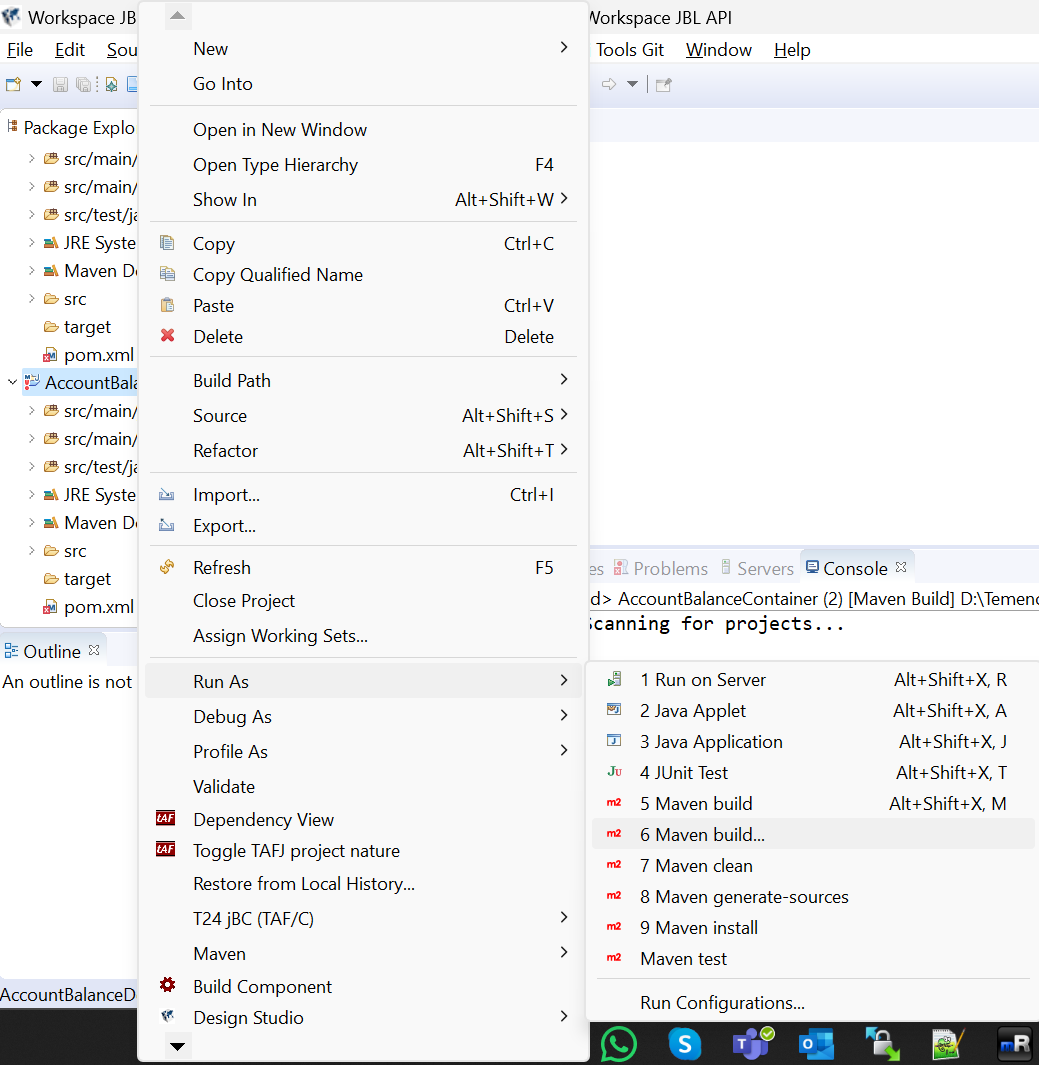
This file alone is fine to deploy in a container to get our API work.

Paste the **Downloaded** files from the **Workbench** in the Below path,

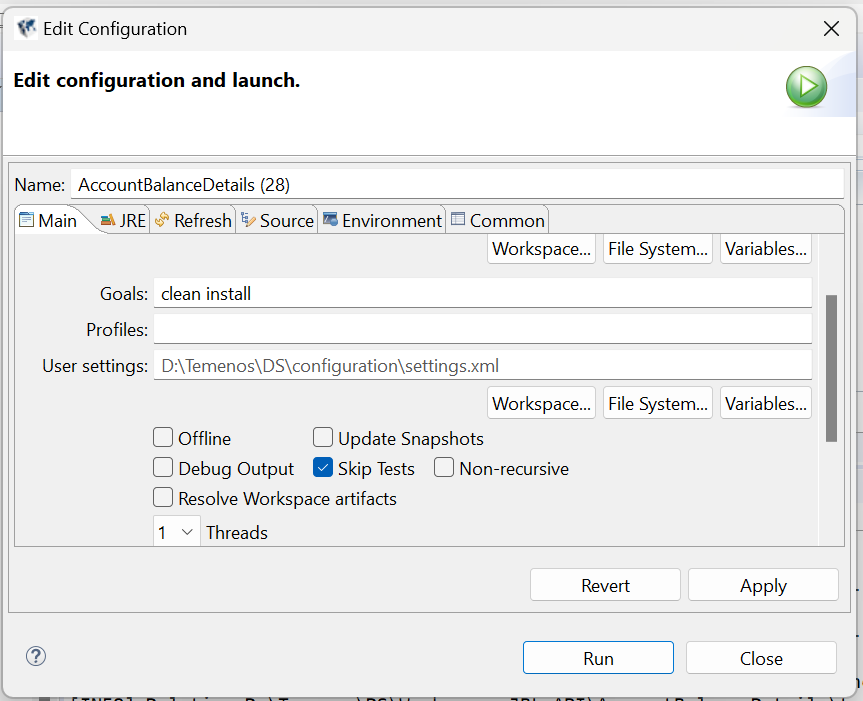
**D:\Temenos\DS\Workspace JBL API\AccountBalanceDetails\src\main\resources**

****

Right click on the main project(**AccountBalanceDetails**)>Run as> **Maven build**…>clean install> run (Musk check **Skip Tests**)







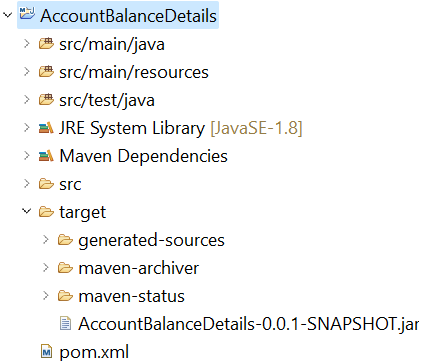


If the BUILD is FAILED then need to download the .jar  file and paste it inside the **t24 binaries** directory like this,

**D:\Temenos\DS\t24-binaries\org\json\json\20170516**

We also need to replace each http to https in **archetype-catalog.xml** file.

If the **BUILD** is **SUCCESSFUL** then the **jar** file will be created here.

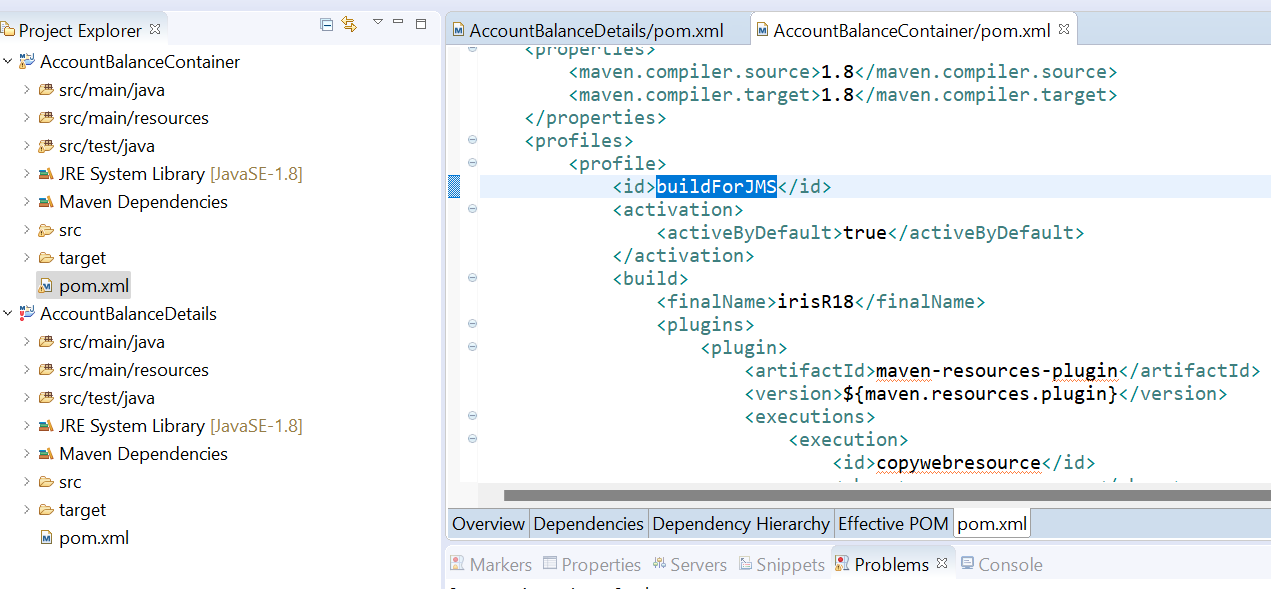




Go to the **container** project >pom.xml  
Go to **dependencies** tab and add your jars (created in main project)

Go to **pom.xml** tab

buildforJMS > true ; Not Required/Optional for Model Bank.



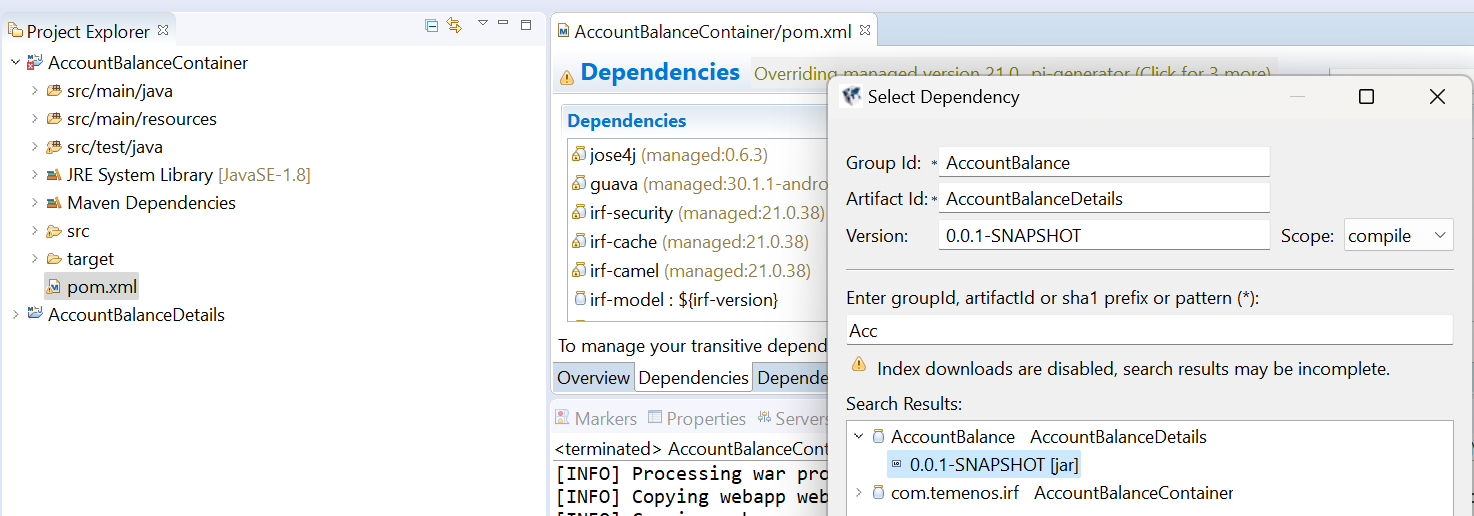
**Adding Dependencies:**

**Special Note:** We must build the main project first then the container project.

Double click on pom.xml → select Dependencies tab → click Add.

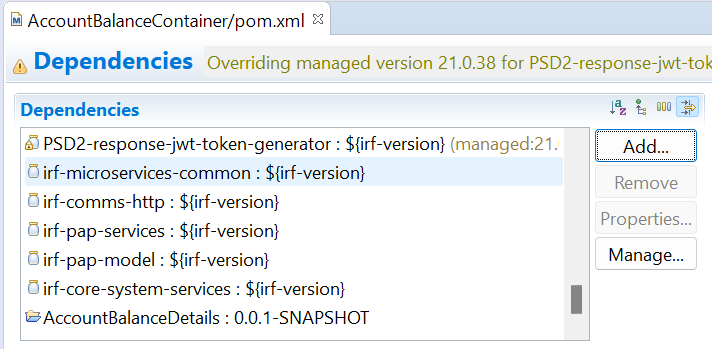
The **container** will have the main project **jar file** as a Dependency.

Search your service name in Enter **groupId** field → select your service → Click **OK**.





Then **Save** it. → Ctrl +S



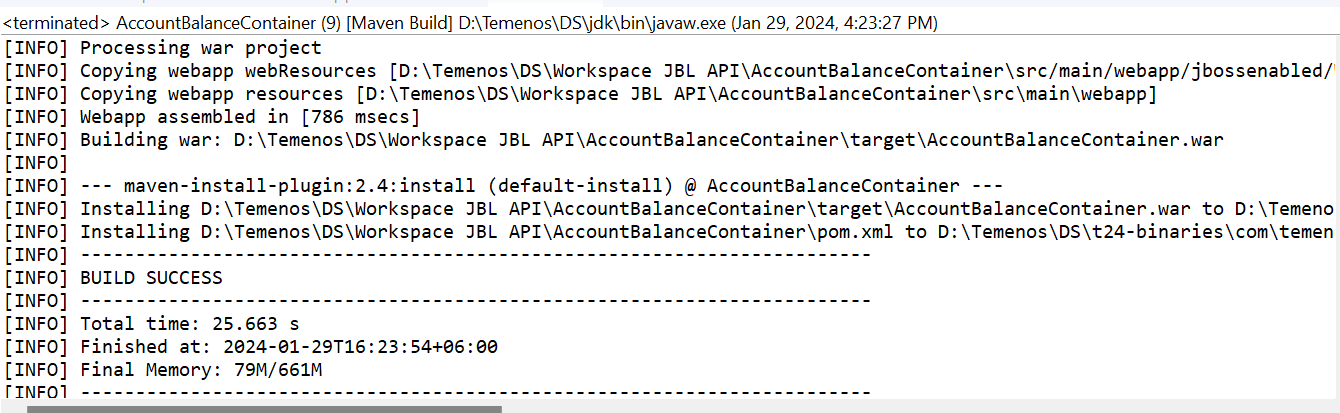


If the jboss is a remote server then,

Provide the jboss environment: username and password.

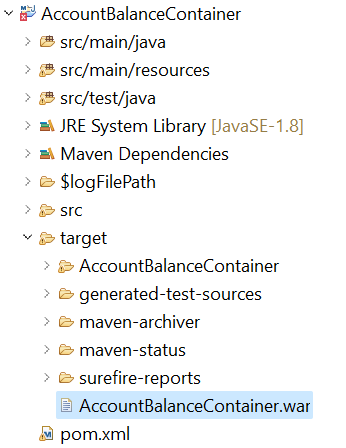
Java.naming.security.principal = username

Java.naming.security.credentials = Password



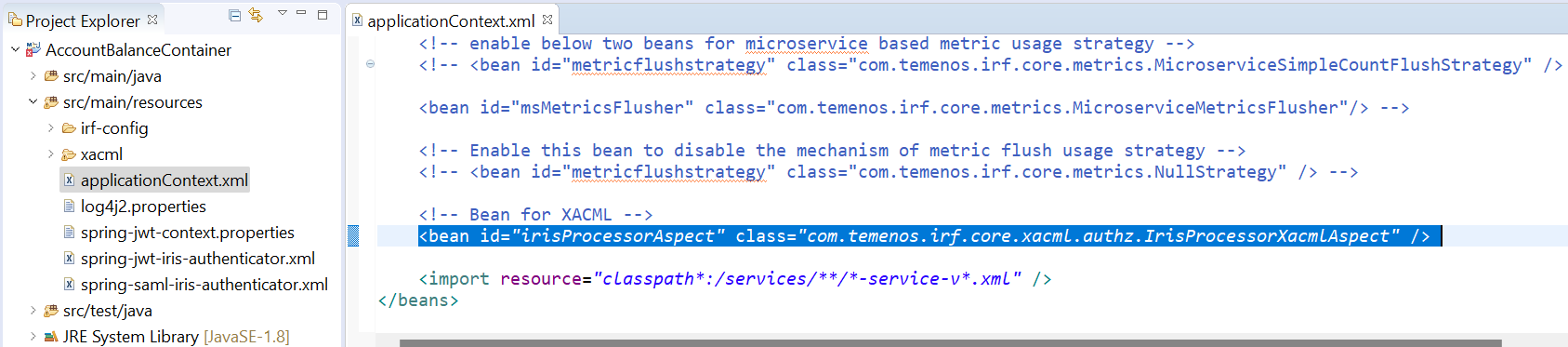


**D:\Temenos\DS\Workspace JBL API\AccountBalanceContainer\target**



**To Enable/disable xacml:**

**Comment** out below line to disable xacml. (OR) **Uncomment** to enable xacml.

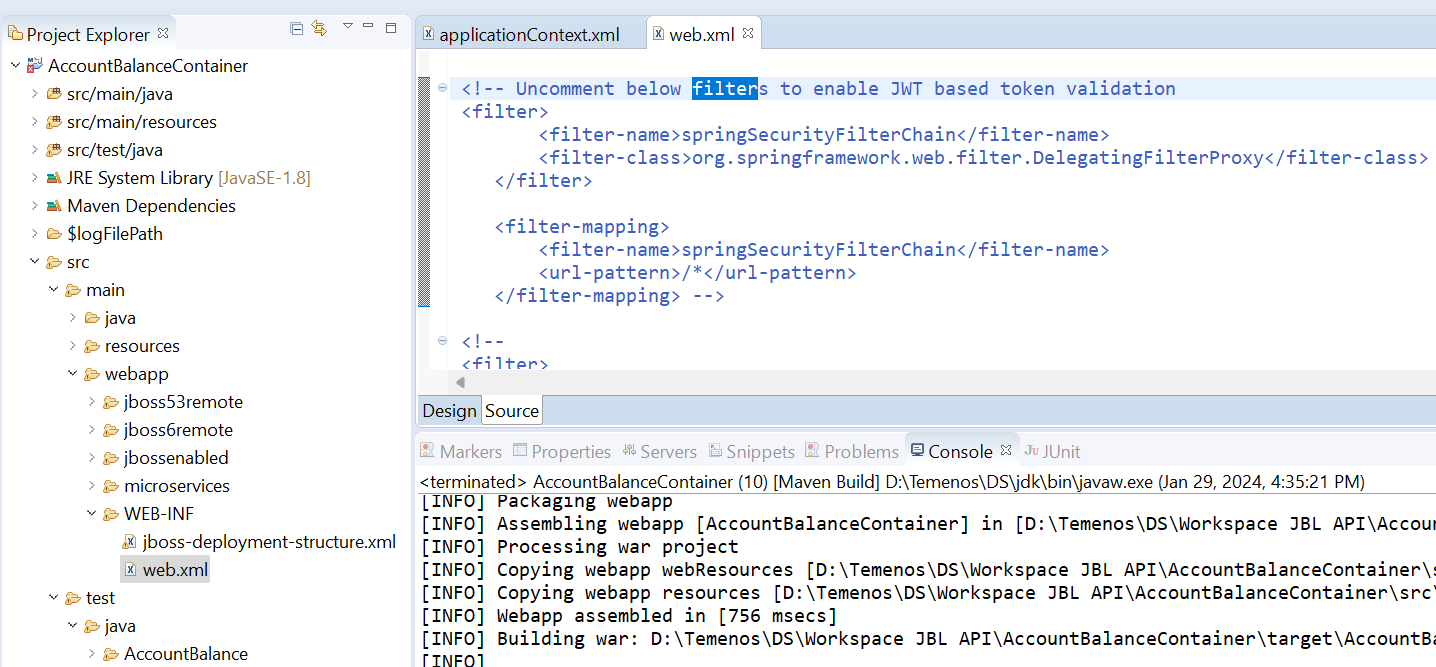




**To Enable/disable JWT:**

**Comment** out below set of lines to disable JWT (OR)

**UnComment** below set of lines to enable JWT.



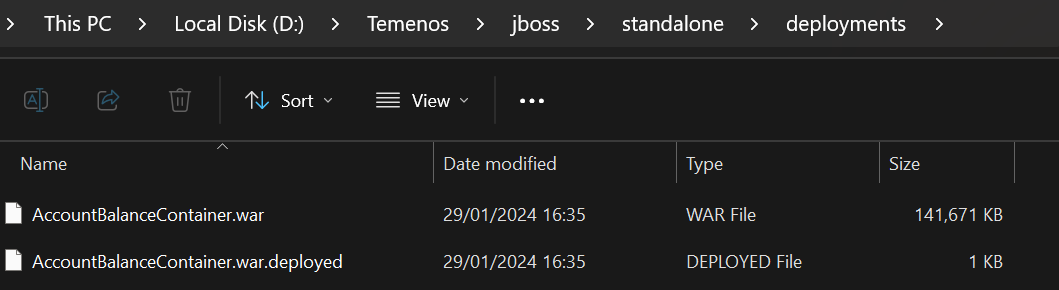


If **jms.properties** is required to change from the war file,



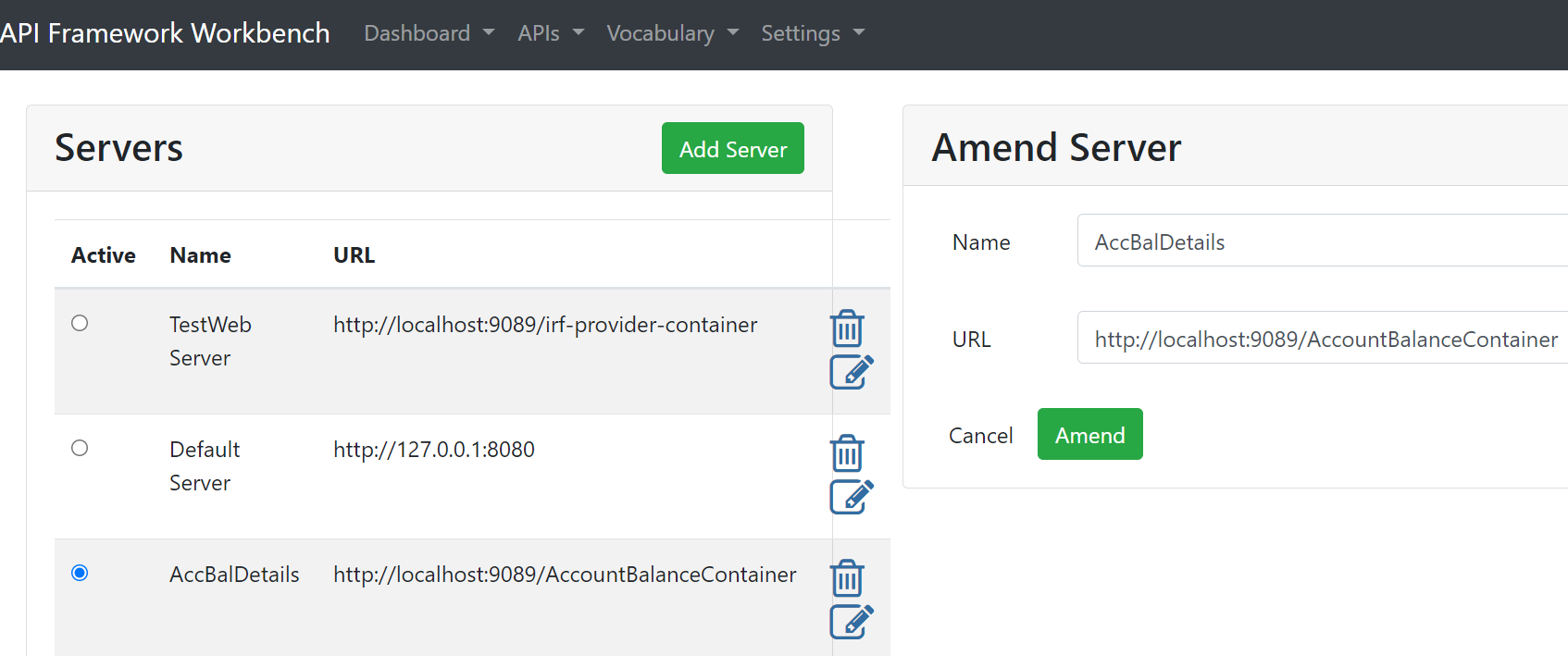
**Deployment *:***

Deploy the .war file in your jboss. **D:\Temenos\jboss\standalone\deployments**

****

**Adding server to view the API :**

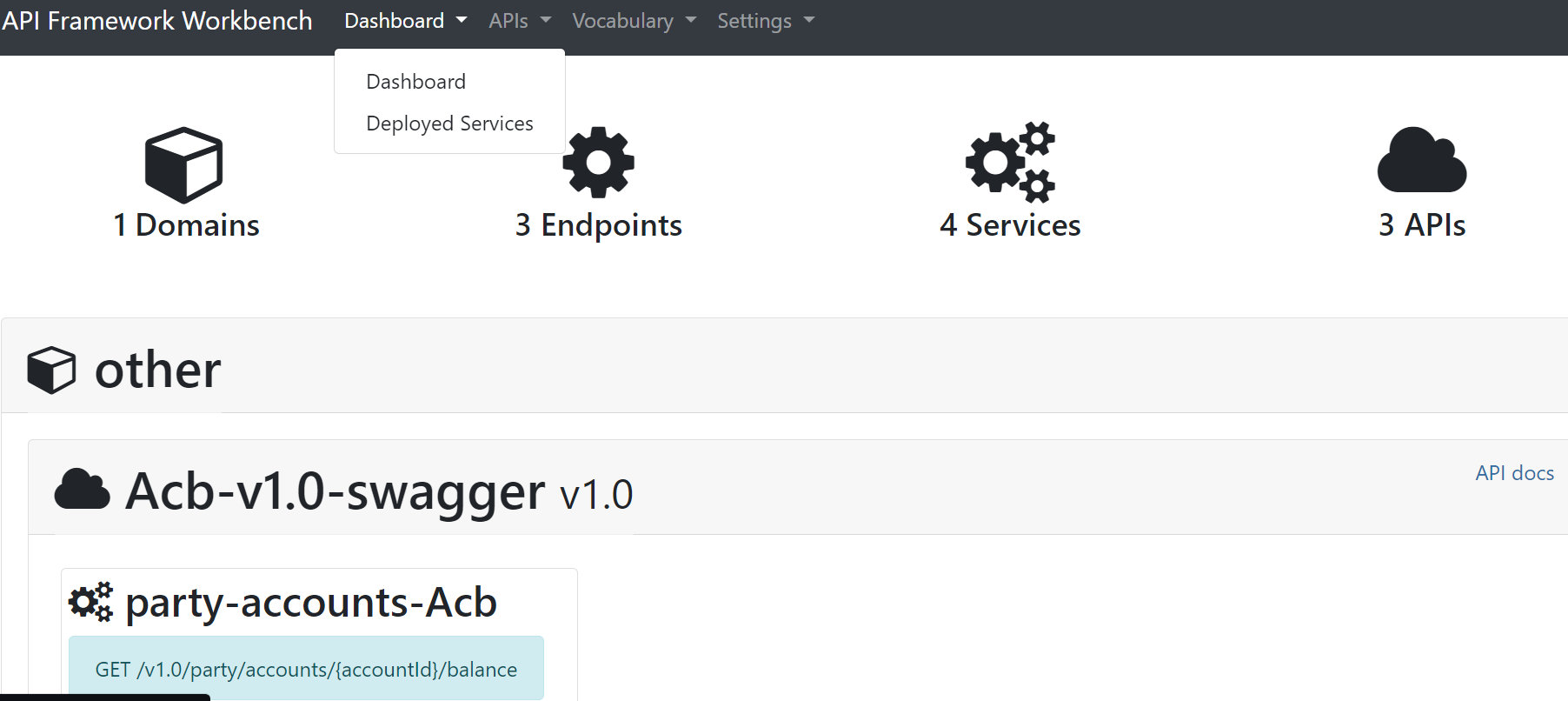
**Open workbench → click on Settings tab → click Add Server button.**

****



**Checking the Deployed Services**



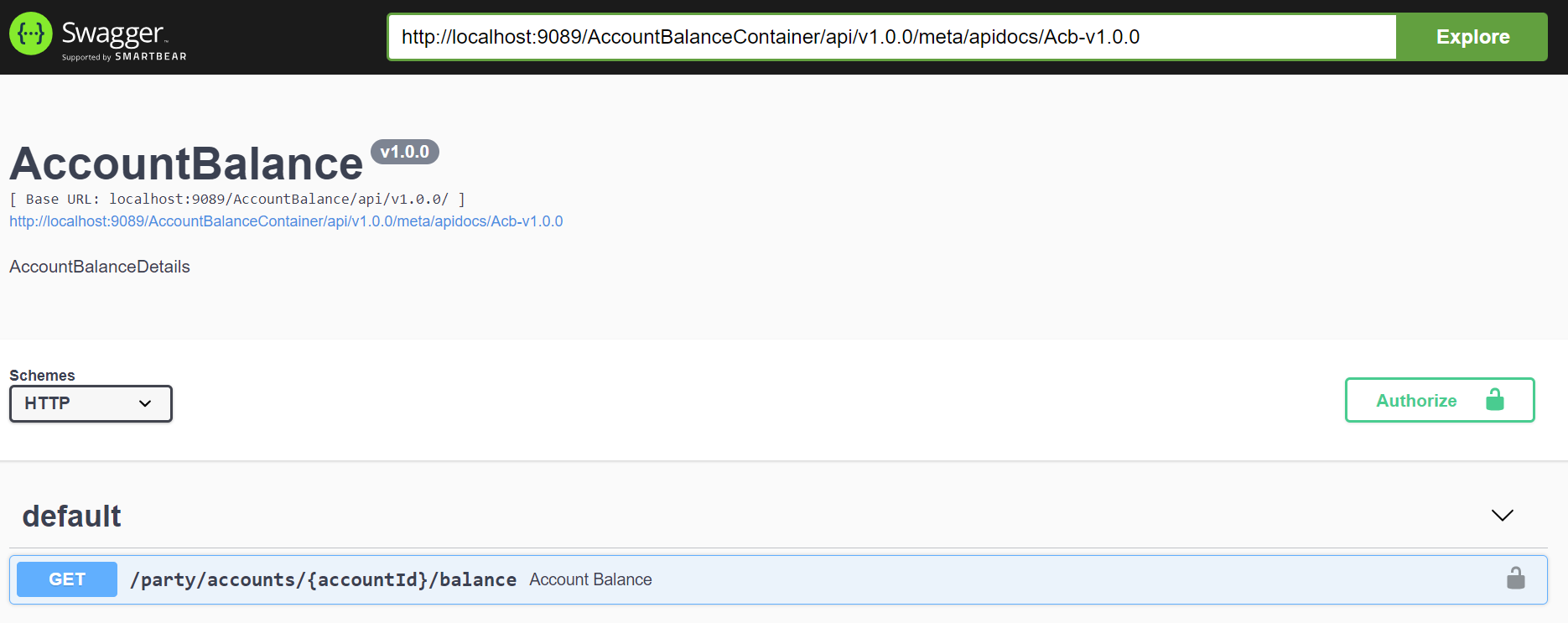
****

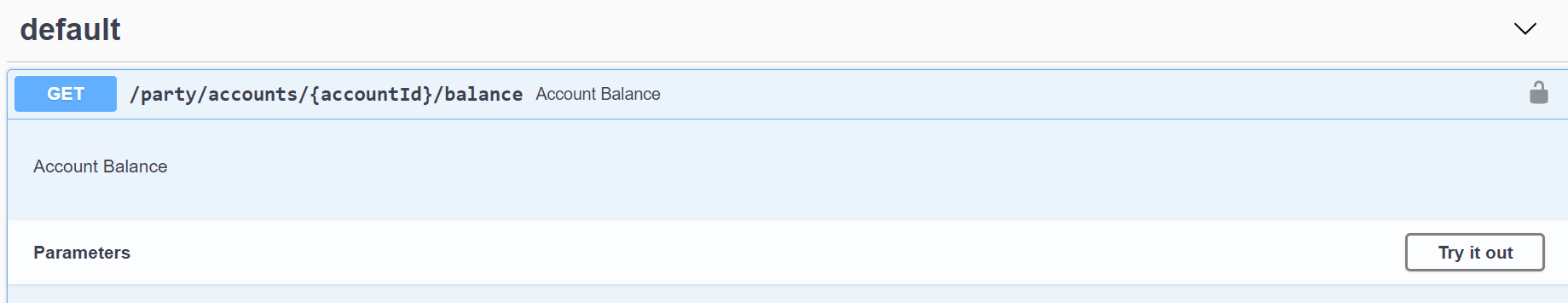


**CLICK - API docs**

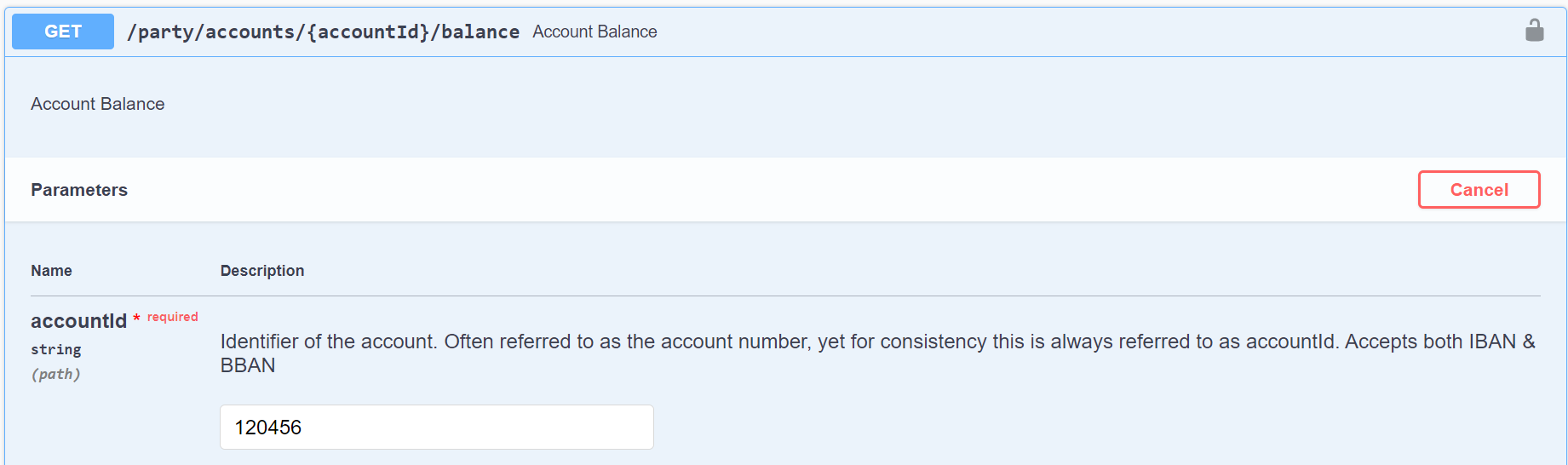
****



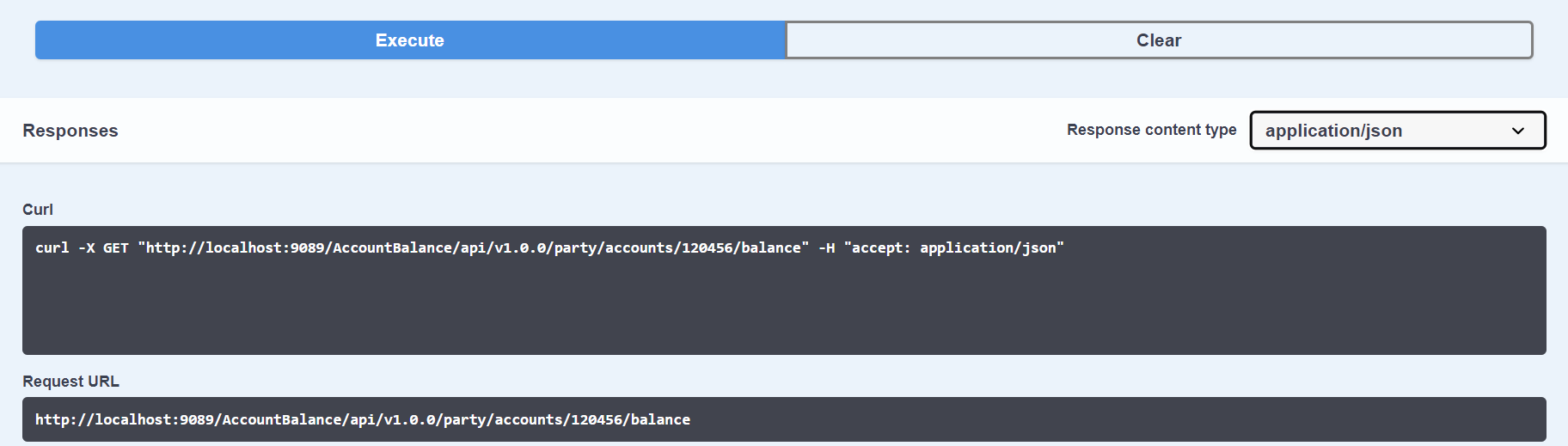
****

****



****



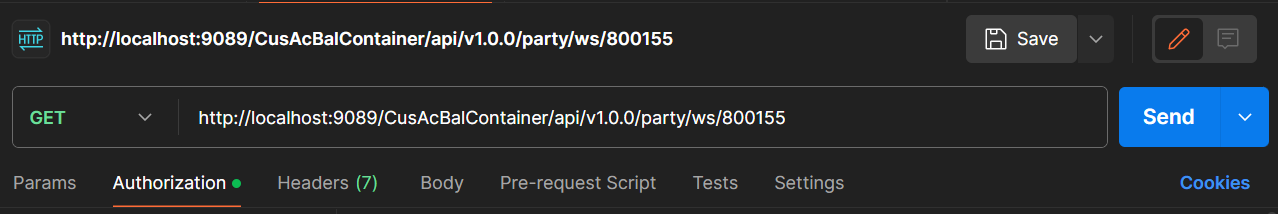
****



**Checking API in Postman:**

**Run postman → click ‘+’ button → paste your API link and add War filename in the link.**

**Click Send → Response will be shown.**

****



If authentication(Basic Auth) error is found then **applicationContext.xml file** need to be changed.

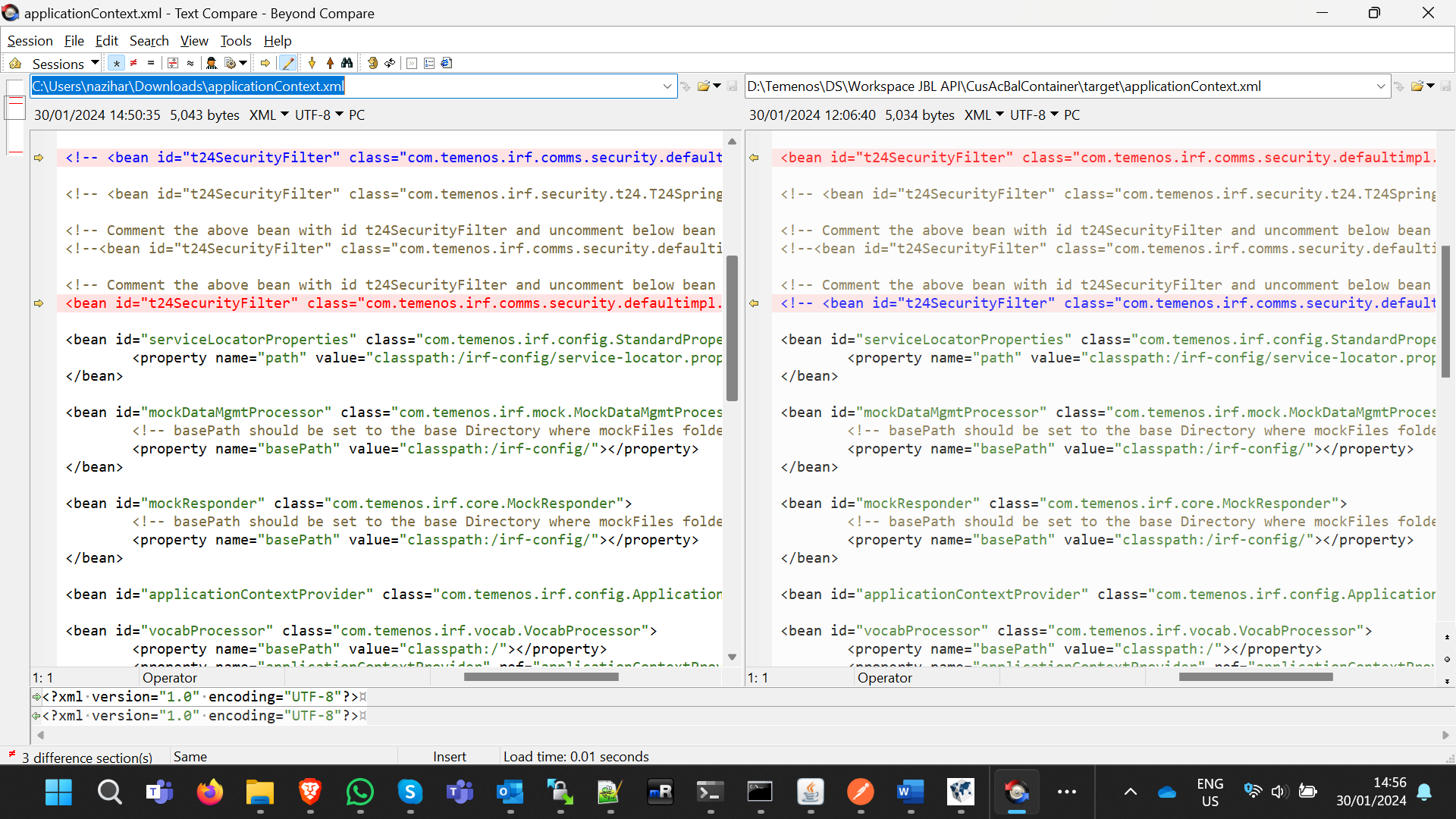
**Commend out** these lines:

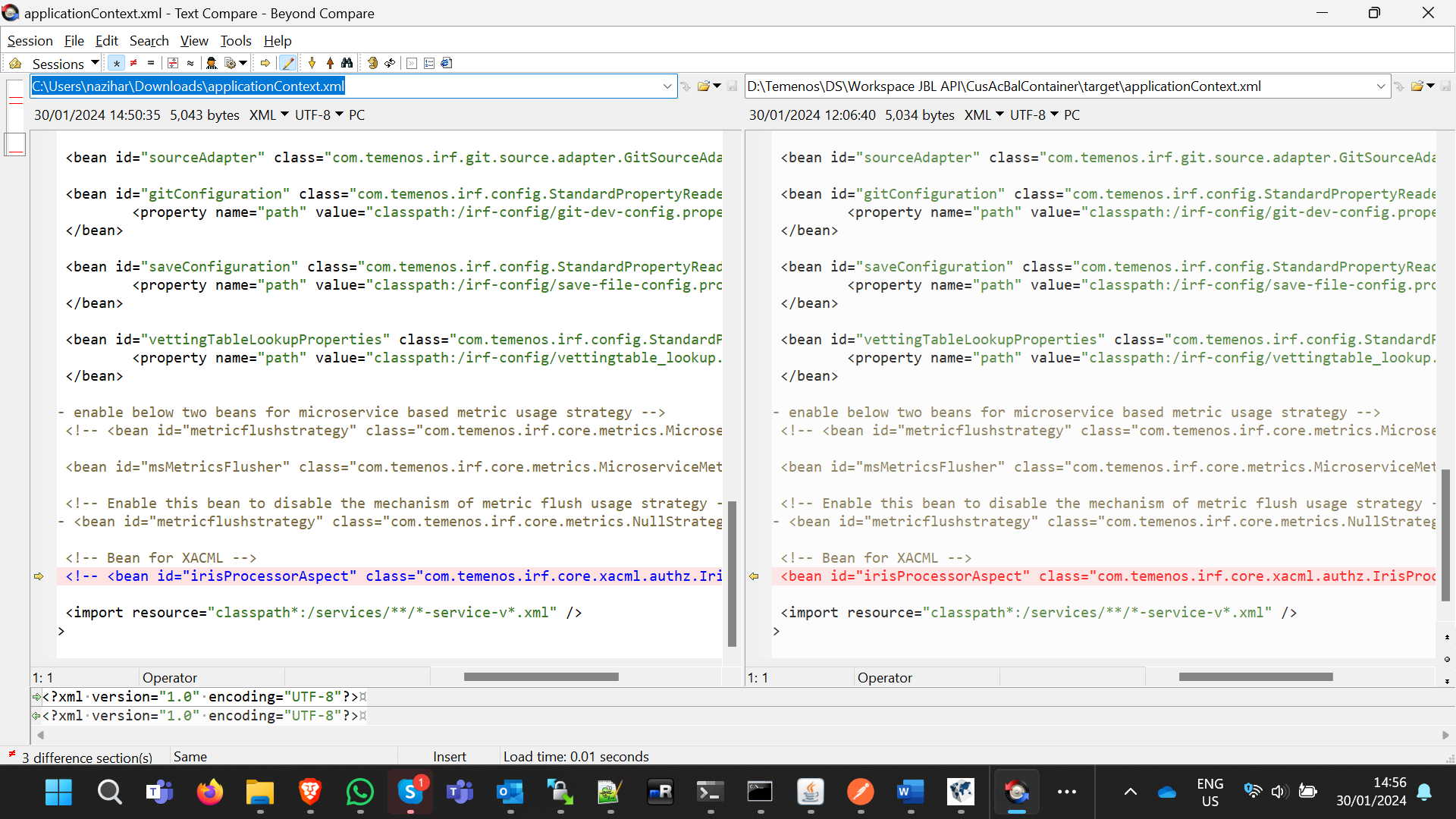
<!-- <bean id="t24SecurityFilter" class="com.temenos.irf.comms.security.defaultimpl.NullBean" /> -->

<!-- <bean id="irisProcessorAspect" class="com.temenos.irf.core.xacml.authz.IrisProcessorXacmlAspect" /> -->

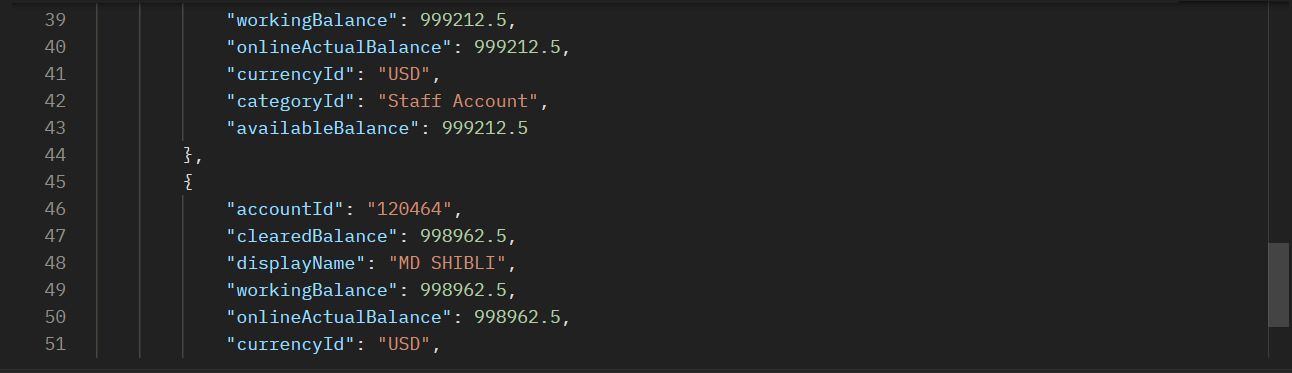
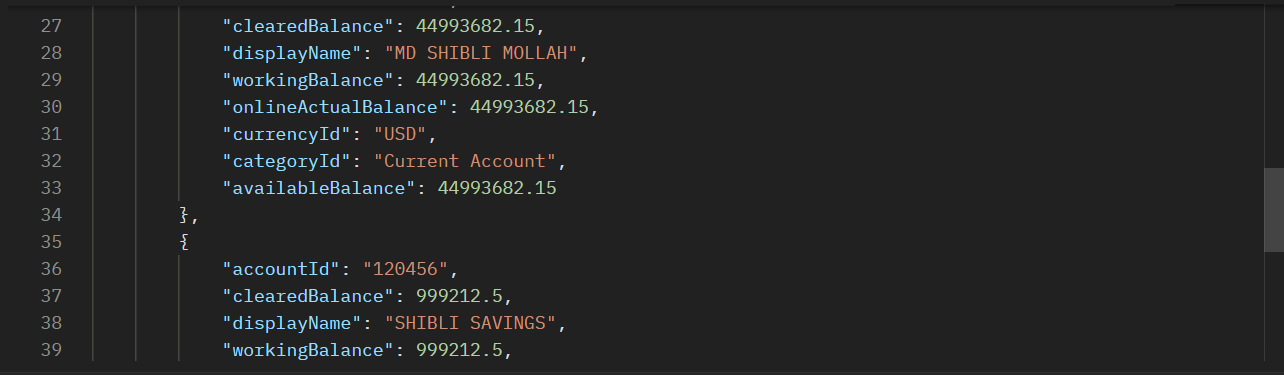
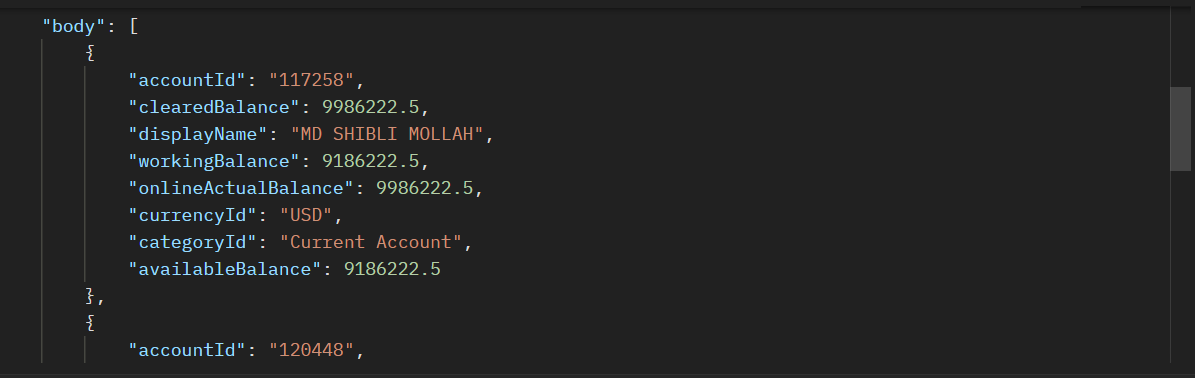
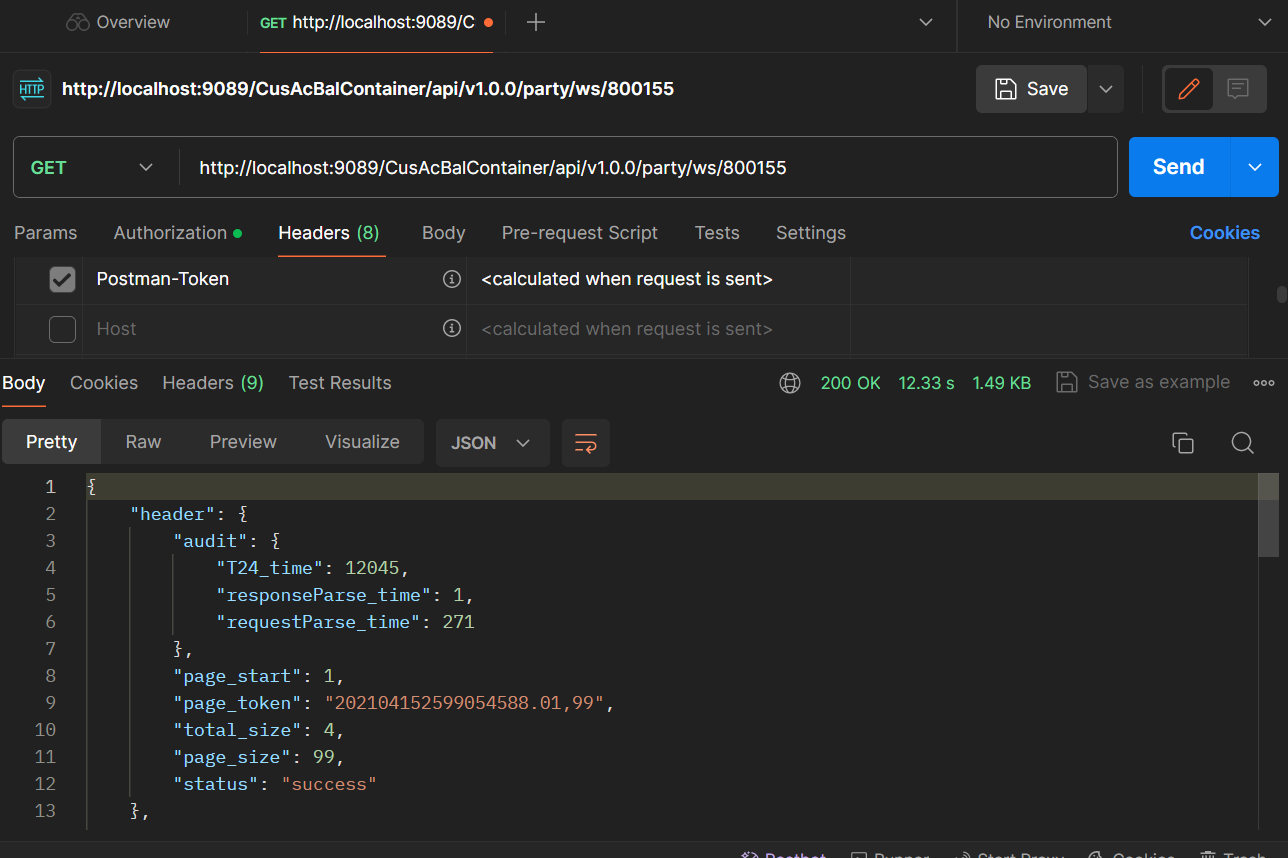
**Uncomment**:

<bean id="t24SecurityFilter" class="com.temenos.irf.comms.security.defaultimpl.T24BasicAuthenticationCheck" />

****

****

After changing and replacing the file in the war file it works!

****

**Response:**

{

"header": {

"audit": {

"T24\_time": 12045,

"responseParse\_time": 1,

"requestParse\_time": 271

},

"page\_start": 1,

"page\_token": "202104152599054588.01,99",

"total\_size": 4,

"page\_size": 99,

"status": "success"

},

"body": [

{

"accountId": "117258",

"clearedBalance": 9986222.5,

"displayName": "MD SHIBLI MOLLAH",

"workingBalance": 9186222.5,

"onlineActualBalance": 9986222.5,

"currencyId": "USD",

"categoryId": "Current Account",

"availableBalance": 9186222.5

},

{

"accountId": "120448",

"clearedBalance": 44993682.15,

"displayName": "MD SHIBLI MOLLAH",

"workingBalance": 44993682.15,

"onlineActualBalance": 44993682.15,

"currencyId": "USD",

"categoryId": "Current Account",

"availableBalance": 44993682.15

},

{

"accountId": "120456",

"clearedBalance": 999212.5,

"displayName": "SHIBLI SAVINGS",

"workingBalance": 999212.5,

"onlineActualBalance": 999212.5,

"currencyId": "USD",

"categoryId": "Staff Account",

"availableBalance": 999212.5

},

{

"accountId": "120464",

"clearedBalance": 998962.5,

"displayName": "MD SHIBLI",

"workingBalance": 998962.5,

"onlineActualBalance": 998962.5,

"currencyId": "USD",

"categoryId": "Savings Acct",

"availableBalance": 998962.5

}

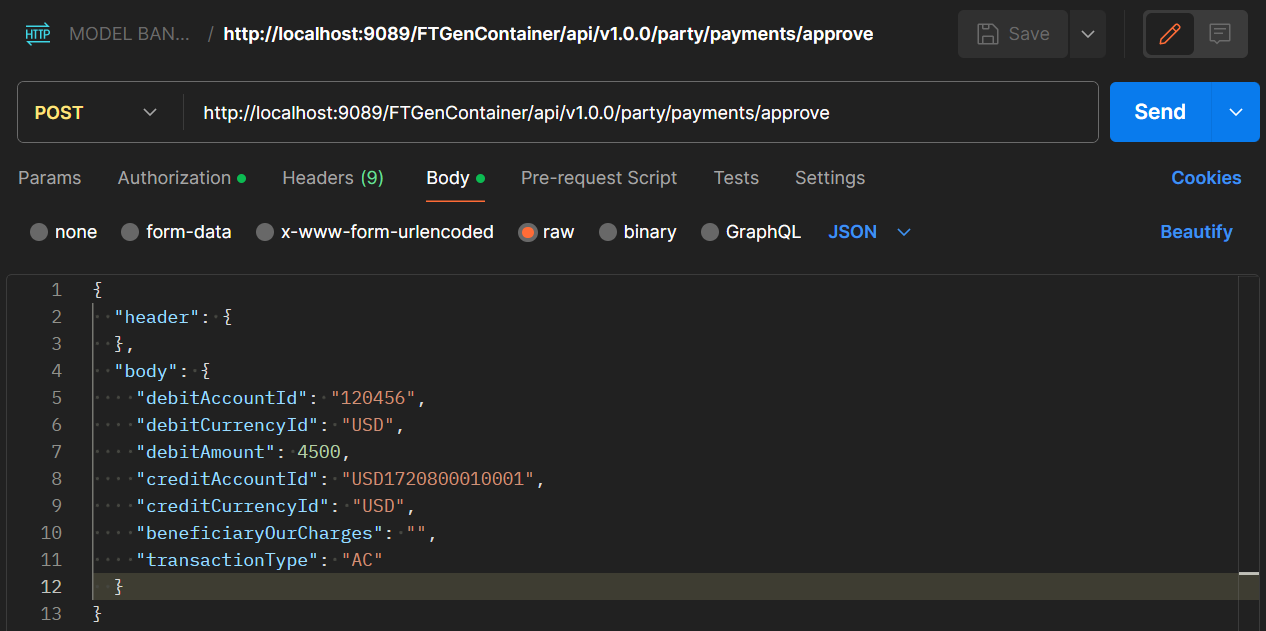
]

}

**VERSION - POST**

Api creation and war file creation is almost same as the process for ENQUIRY.

POST request format:



Body:

{

  "header": {

  },

  "body": {

    "debitAccountId": "120456",

    "debitCurrencyId": "USD",

    "debitAmount": 4500,

    "creditAccountId": "USD1720800010001",

    "creditCurrencyId": "USD",

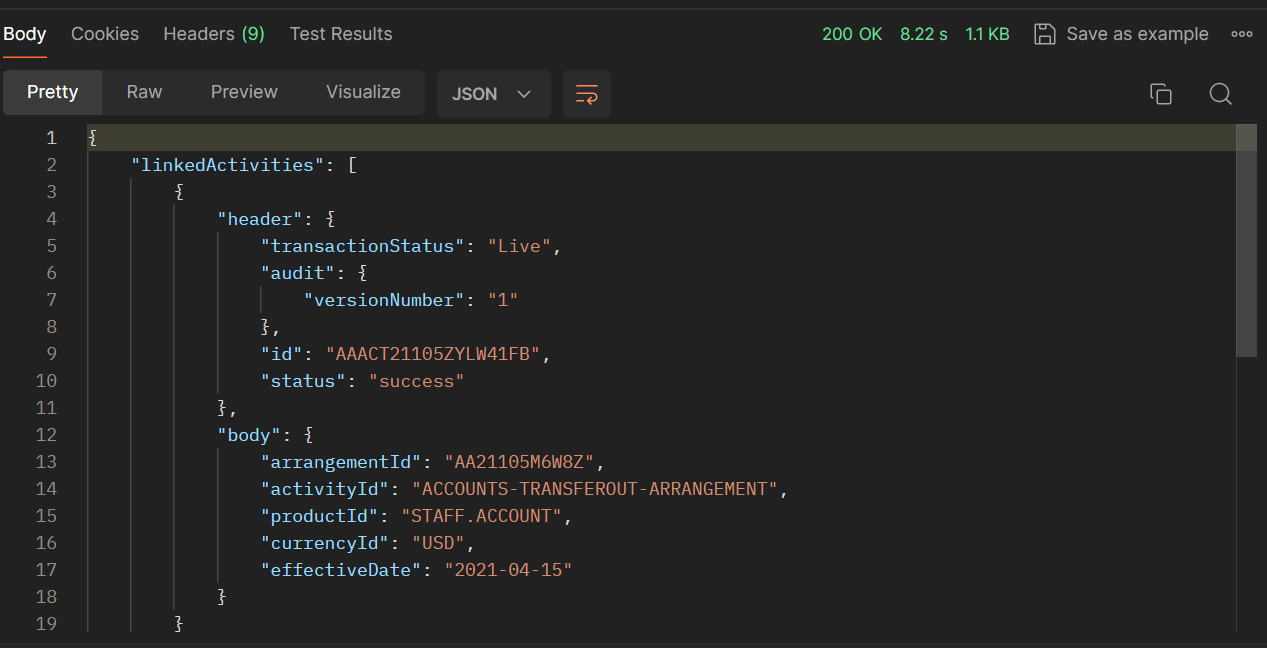
    "beneficiaryOurCharges": "",

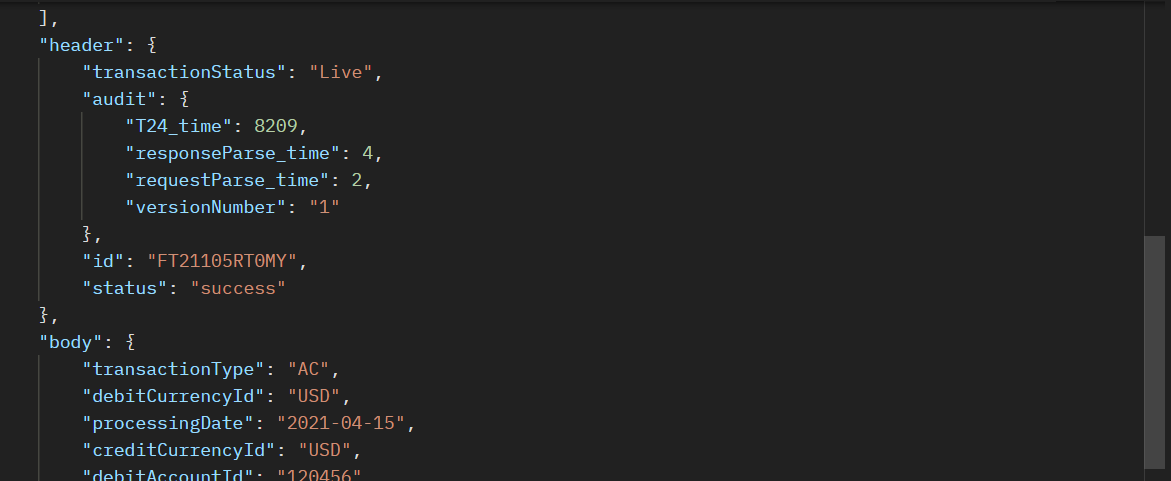
    "transactionType": "AC"

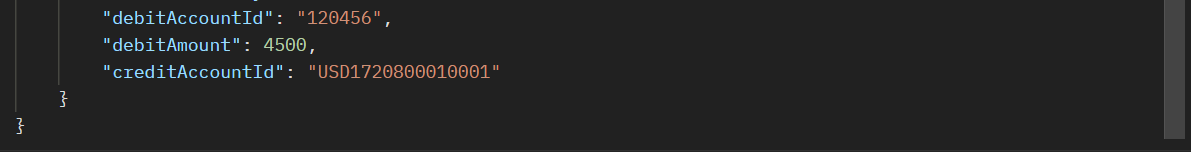
  }

}

**Response:**

****

****

****

***Orchestration***

***Question***: Creating orchestration for FT and Account

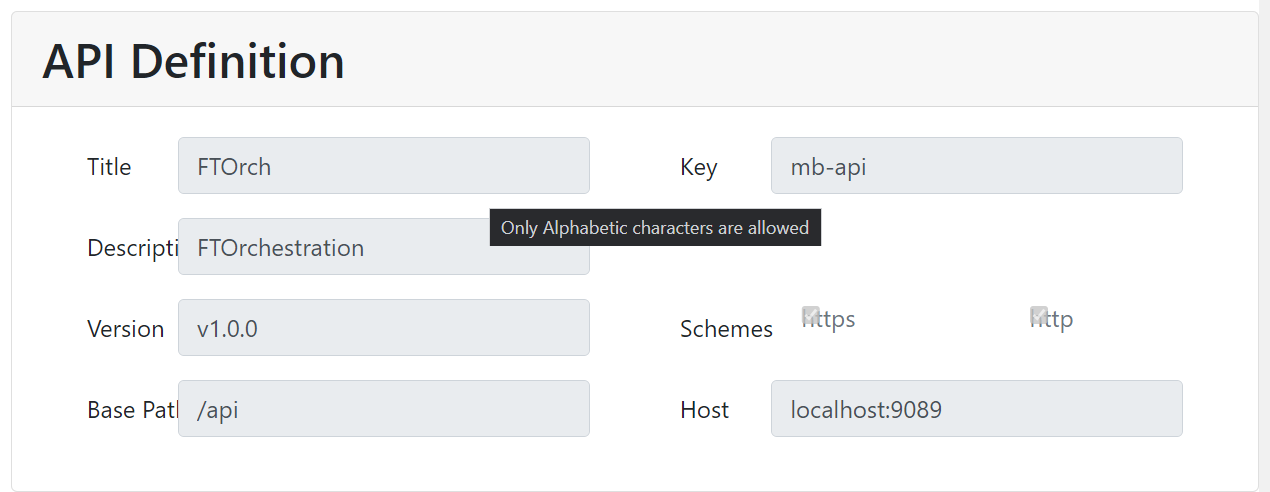
***Requirement :*** After Authorization of FT(version), The enquiry which shows the debit Account Id’s Available balance

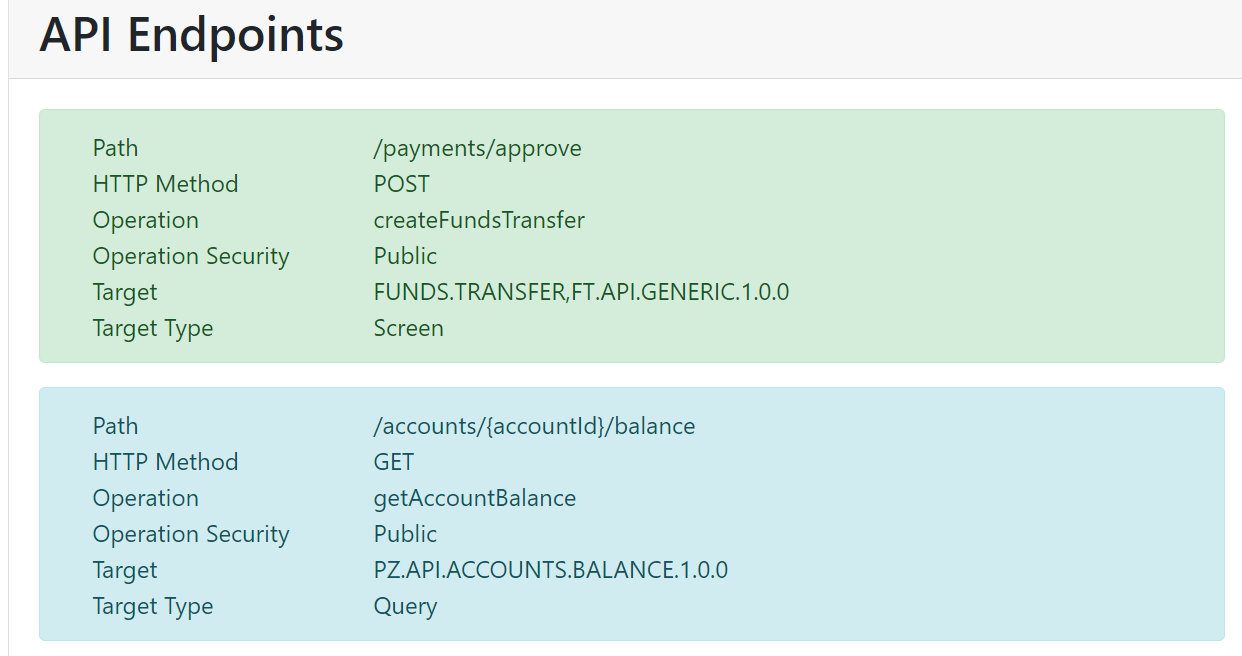
***Step 1:***

Choose the artefacts in the workbench and generate the zip file.

**Enquiry :** FUNDS.TRANSFER,FT.API.GENERIC.1.0.0

**Version :** PZ.API.ACCOUNTS.BALANCE.1.0.0

****

****

## Inventory

{

"paths": [

{

"method": "POST",

"url": "/party/payments/approve",

"function": "input",

"tags": [],

"operationId": "createFundsTransfer",

"operationSecurity": "Public",

"resources": [

{

"key": "FUNDS.TRANSFER,FT.API.GENERIC.1.0.0",

"resourceType": "Screen",

"consentManaged": false

}

],

"properties": {

"clientType": "INTERNAL",

"isBulk": false,

"deprecated": false

}

},

{

"method": "GET",

"url": "/party/accounts/{accountId}/balance",

"tags": [],

"operationId": "getAccountBalance",

"operationSecurity": "Public",

"resources": [

{

"key": "PZ.API.ACCOUNTS.BALANCE.1.0.0",

"resourceType": "Query",

"consentManaged": false,

"selections": [

{

"field": "ACCOUNTREFERENCE",

"param": "accountId",

"operand": "EQ",

"required": "",

"type": "string"

}

]

}

],

"properties": {

"clientType": "INTERNAL",

"isBulk": false,

"deprecated": false

}

}

],

"version": "v1.0.0",

"title": "FTOrch",

"description": "FTOrchestration",

"key": "mb-api",

"schemes": [

"http",

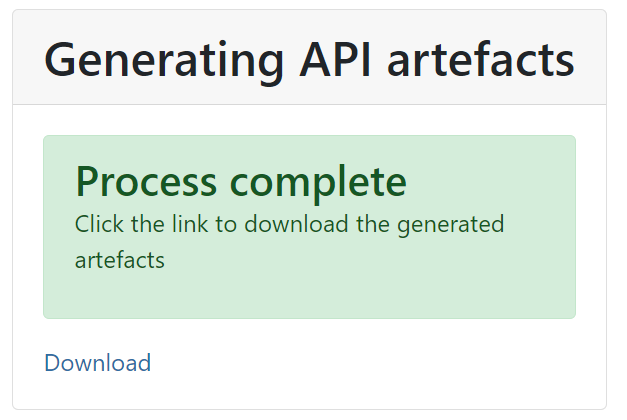
"https"

],

"basepath": "/api",

"host": "localhost:9089"

}

****

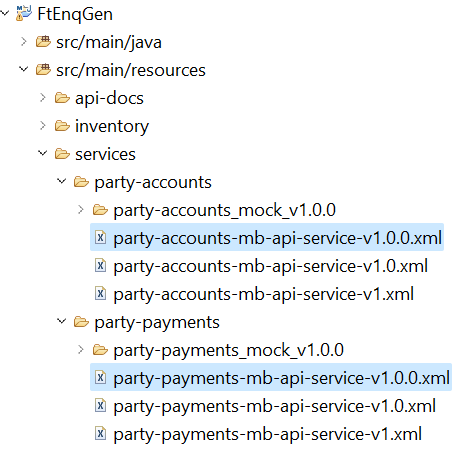
**Step 2:**

Extract the zip file and place it in the new service project.

src -> main -> resources

**Step 3:**

Open the service xml file in the project.



**Step 4:**

Open the **xml file** and write the below code under the **t24 VERSION** processor line in the xml file.

<choice>

<when>

<simple>${headers.CamelHttpResponseCode} == 200</simple>

<setHeader headerName = *"debitAccountId"* >

<camel:jsonpath>$.body.debitAccountId</camel:jsonpath>

</setHeader>

<to uri=*"* *direct-vm:party-accounts.v1.0.0.getAccountBalance"* />

</when>

</choice>

Above xml file is for **VERSION**. Here I am passing debitAccountId field to the **ENQUIRY** as selection field.

**Open the ENQUIRY** **xml file:**

Here we are Mapping the **debitAccountId** to the selection field. (Here we passing **headerName** which is written in the version xml file)

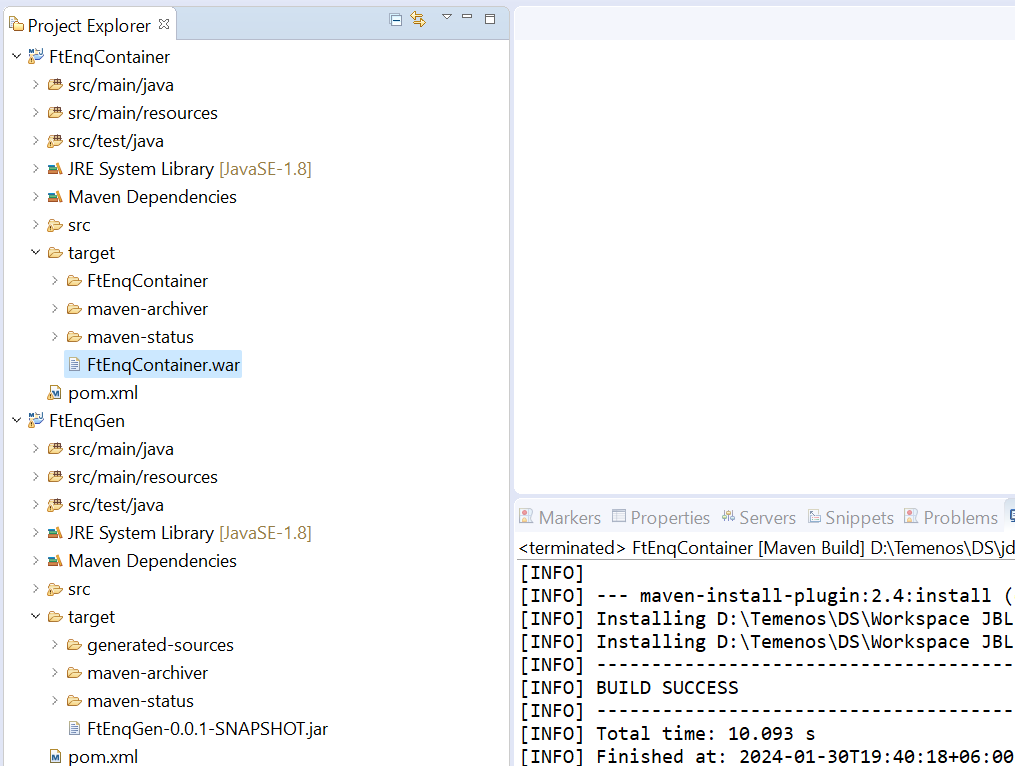
**Edit this line:** <constant>ACCOUNTREFERENCE EQ {debitAccountId}</constant>



**Step 5:**

Right click on both project and Container **RunAs**-> Maven clean & Install.

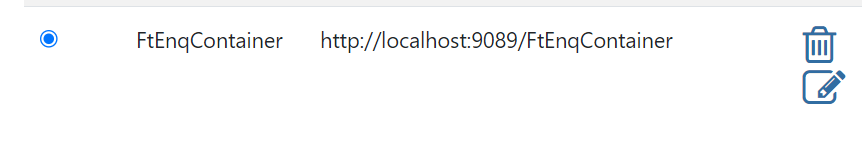
**Jar file** is created in the **service project** & **war file** is created in the **container project**.

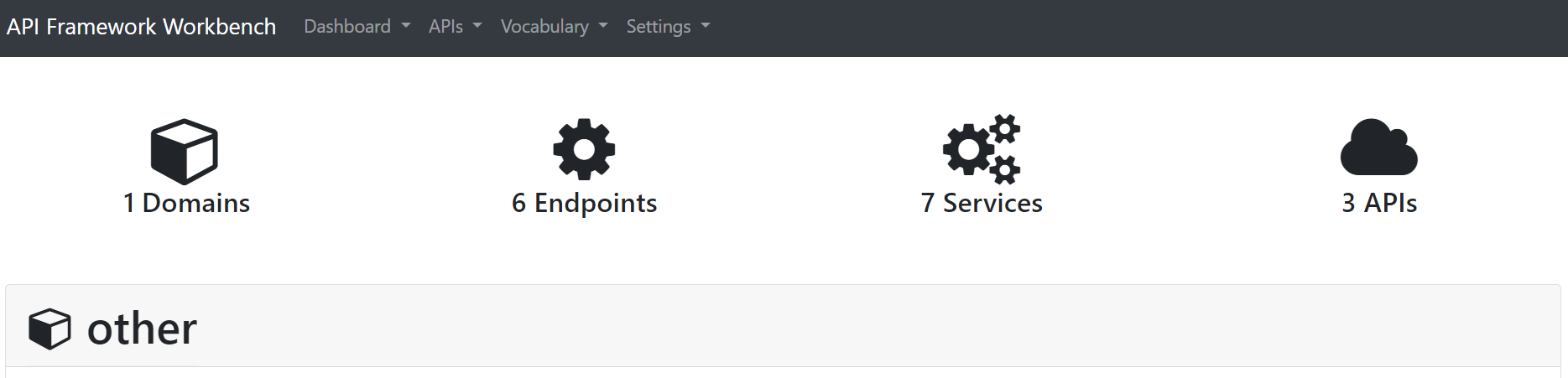
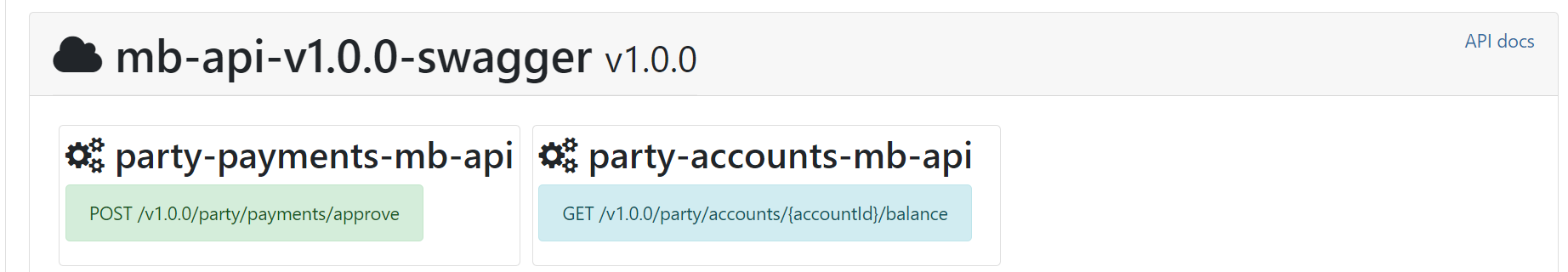


***Step 6:***

Deploy the war file in JBoss.

Add the server in the workbench.



<http://localhost:9089/FtEnqContainer/api/v1.0.0/party/payments/approve>

**Special Note:** If Authorization issue occurs then we need to update the service xml

<setProperty propertyName="validate\_only">

<header>true</header>

</setProperty>

<setProperty propertyName="userDetails">

<header>Authorization</header>

</setProperty>

<setProperty propertyName="function">

<constant>validate</constant>

</setProperty>

<process ref="t24VersionProcessor"/>

<choice>

<when>

<simple>${headers.CamelHttpResponseCode} == 200</simple>

<setHeader headerName="accountId">

<camel:jsonpath suppressExceptions="true">$.body.debitAccountId</camel:jsonpath>

</setHeader>

<setHeader headerName="Authorization">

<camel:exchangeProperty>userDetails</camel:exchangeProperty>

</setHeader>

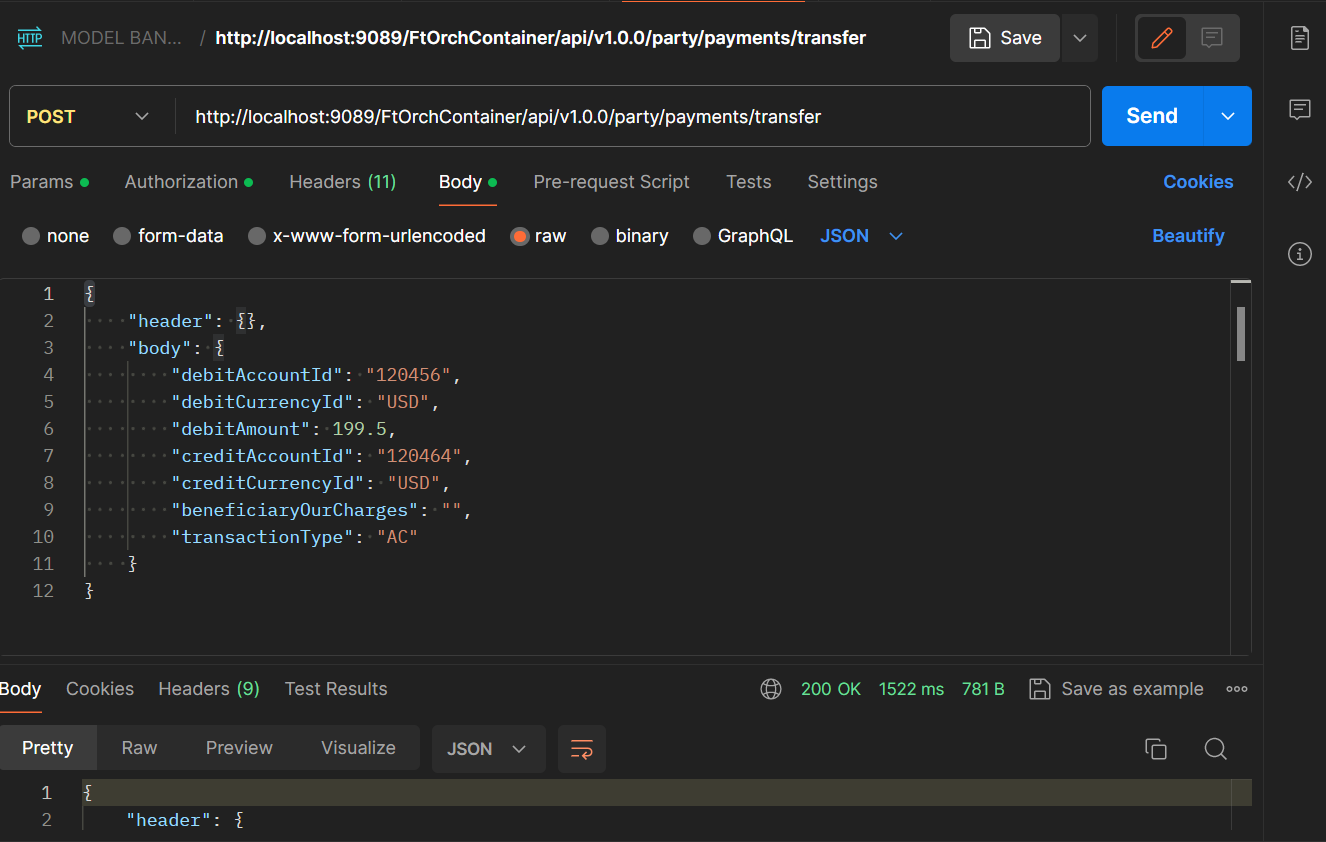
<to uri="direct-vm:party-accounts.v1.0.0.getAcBalance" />

</when>

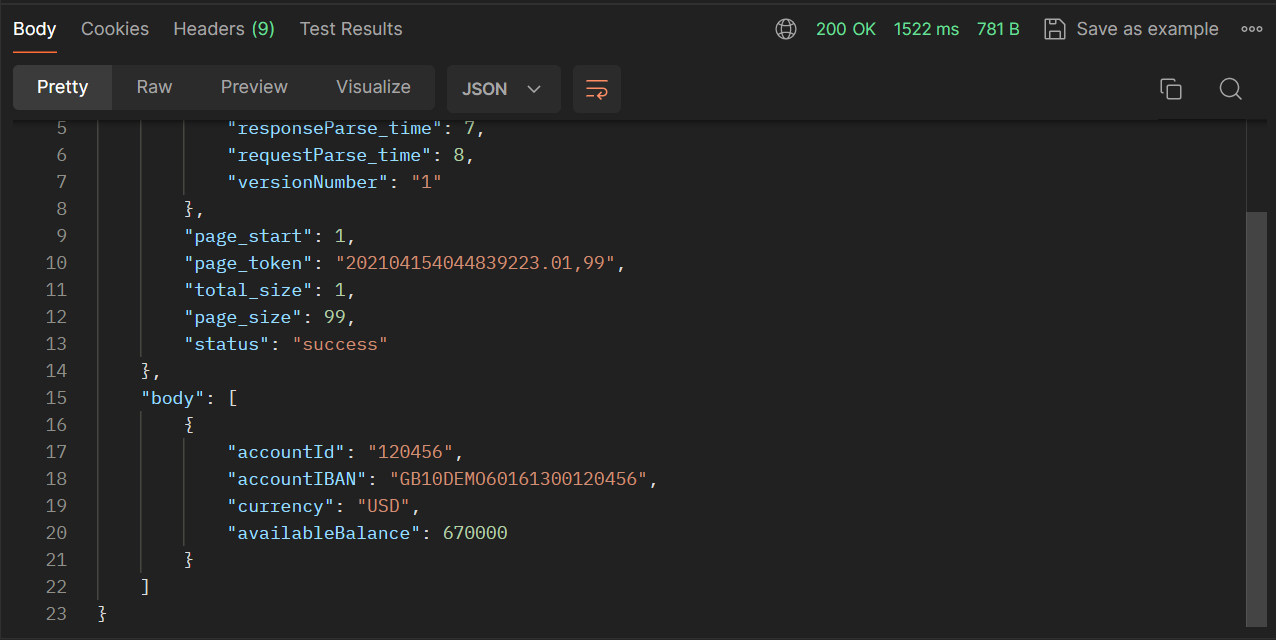
</choice>

</route>

***Step 7: Check it in POSTMAN***



**Response:**

****

**XACML/ JWT Token**

**What is Xacml?**

Extensible Access Control Markup Language – Which is used for attribute based authorization.

For example, An API request must contain “**roleId**” as “**ADMIN**”. This **roleId** we can pass through headers or **JWT Token**.

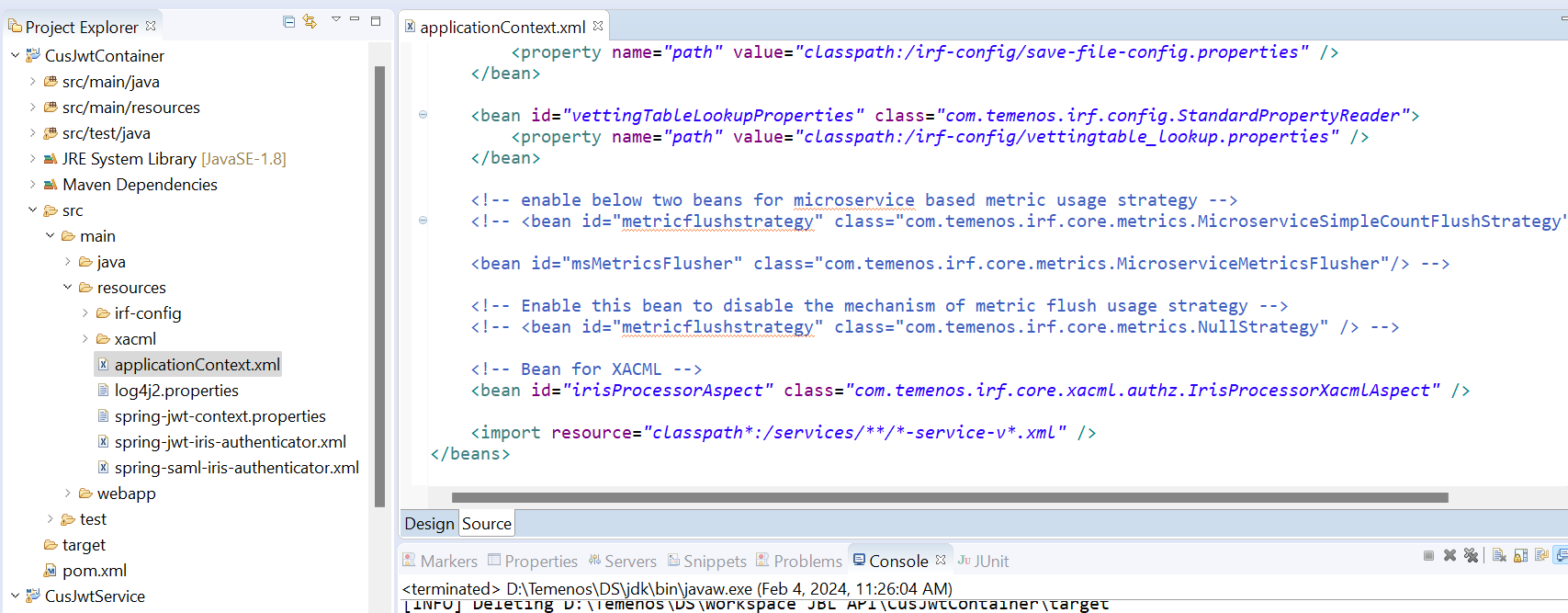
**Xacml policy file will be available inside the container.**

**C:\Users\user\Downloads\irf-provider-container.war\WEB-INF\classes\xacml\**

**To Enable xacml: NO NEED TO ENABLE XACML if it’s JWT Only.** *Keep the* ***applicationContext.xml*** *as it is.*

Uncomment below line to enable **xacml**.







**IRIS Auth token Generation(JWT Only)**

The functionality required in IRIS is to provide capability in IRIS to **validate the credentials** passed in **header**(**Basic Authentication**) against T24 and send back the JWT token for successful authentication.

**Deploy** the war file **irf-auth-token-generation-container.war** in JBoss.

Check it in the postman by the below URL

**http://<host>:<port>/IrisAuthTokenGenerator/api/v1.0.0/generateauthtoken**

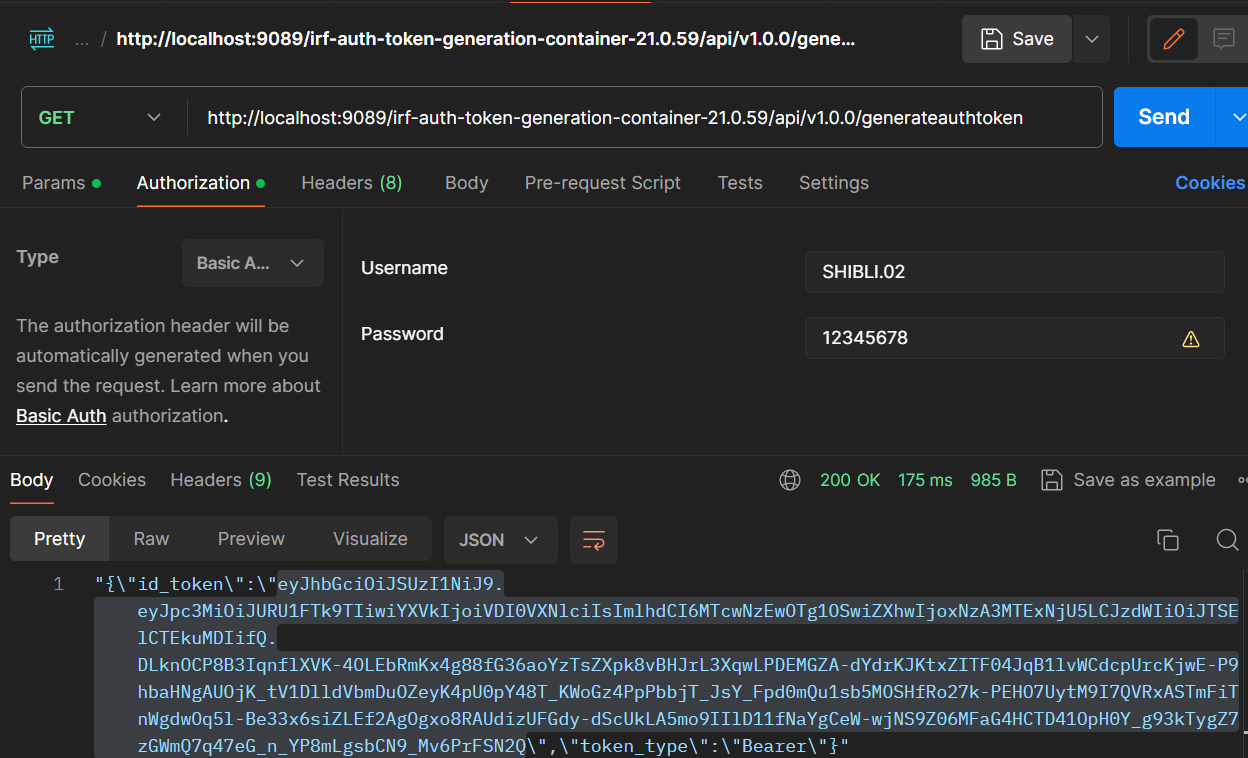
**irf-auth-token-generation-container-21.0.59.war**

http://<**host**>:<**port**>/**IrisAuthTokenGenerator**/api/v1.0.0/generateauthtoken

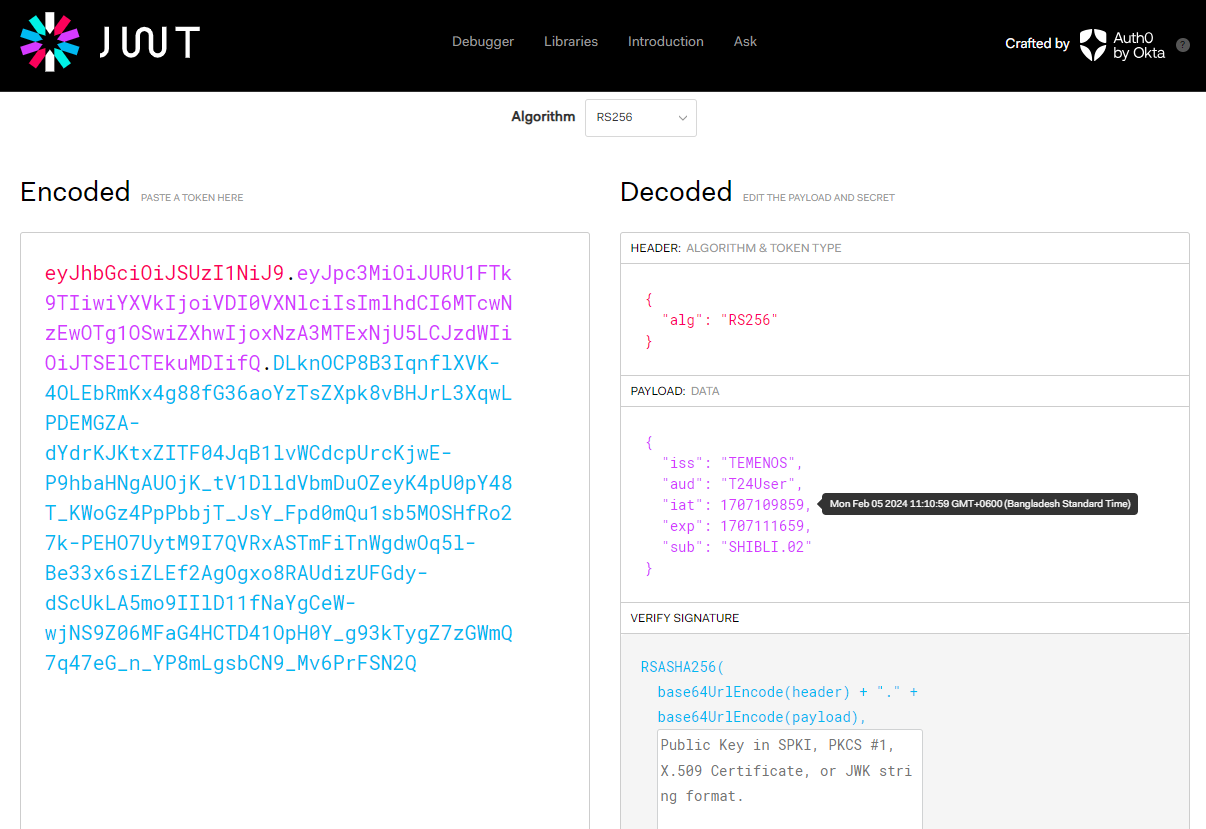
<http://localhost:9089/irf-auth-token-generation-container-21.0.59/api/v1.0.0/generateauthtoken>

**pass with credentials,**

**The Auth token has been generated.**

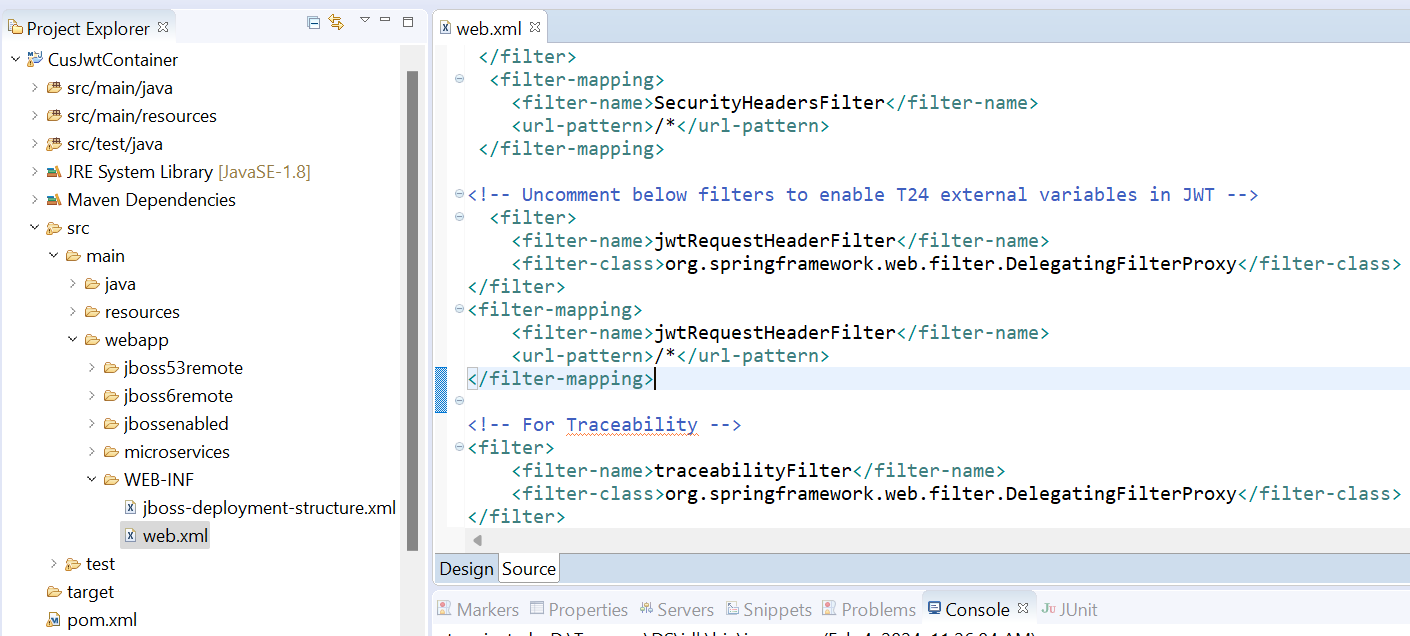


[**https://jwt.io/**](https://jwt.io/)



**To Enable JWT:**

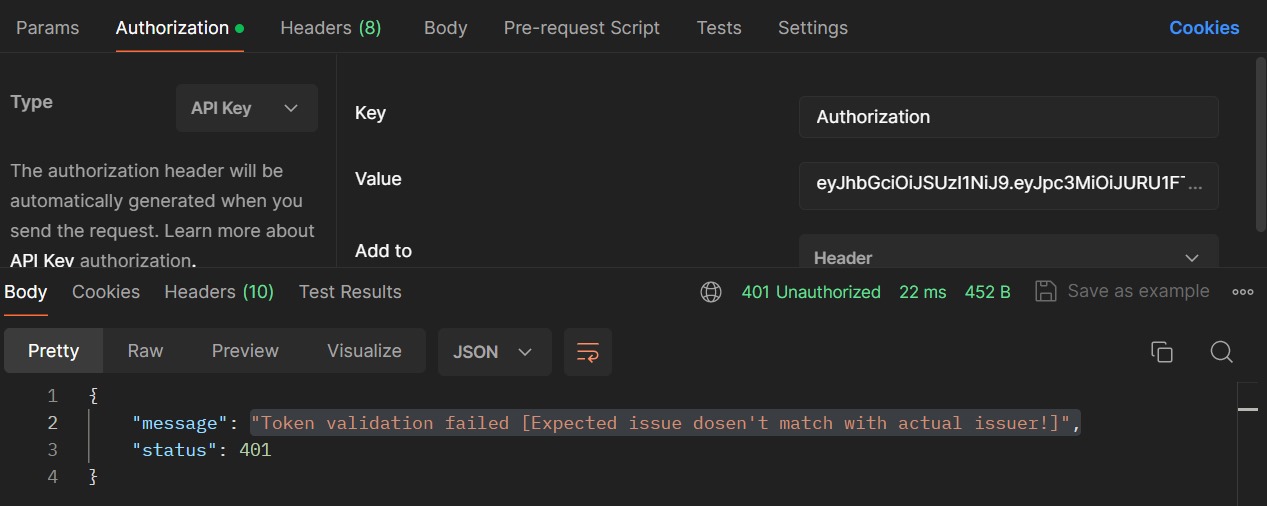
UnComment below set of lines to enable JWT.

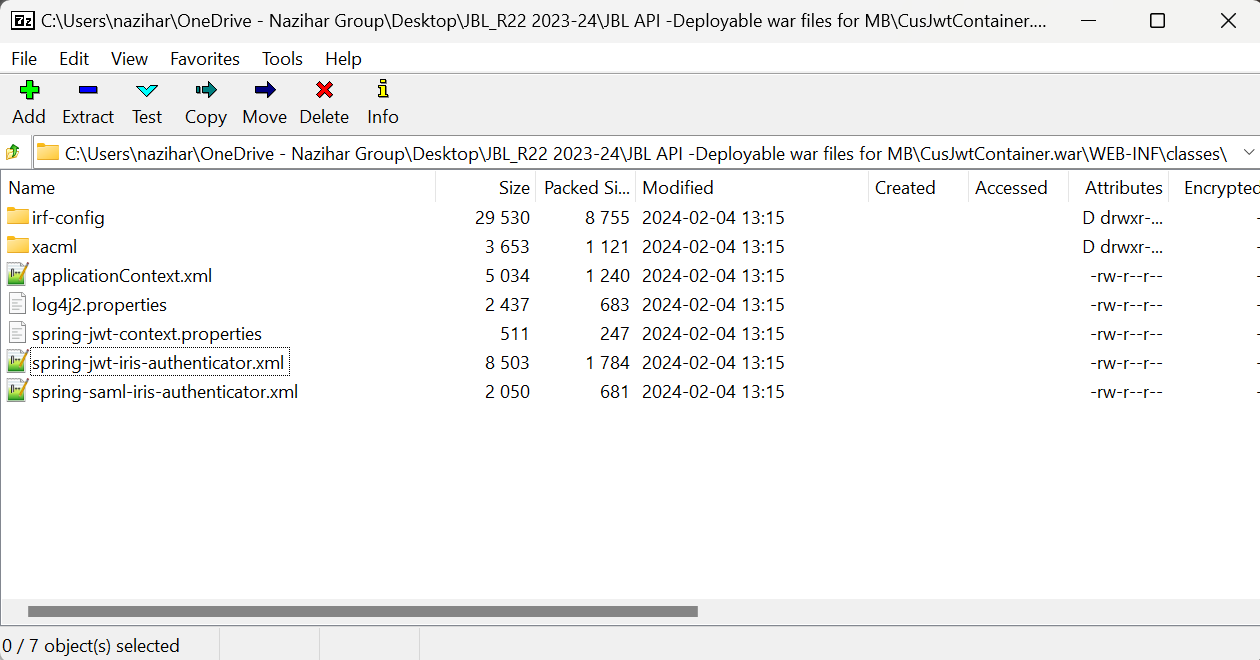




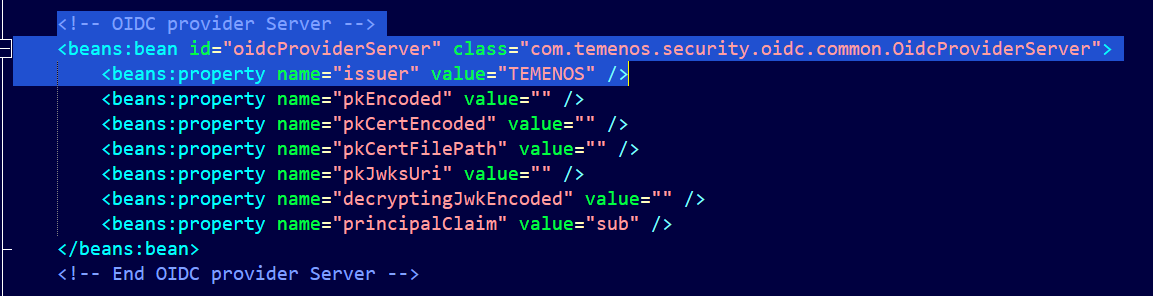
**Change spring-jwt-iris-authenticator.xml:**

Changed the OIDC provider issuer to TEMENOS if there is an issue with the provider like the SS below.

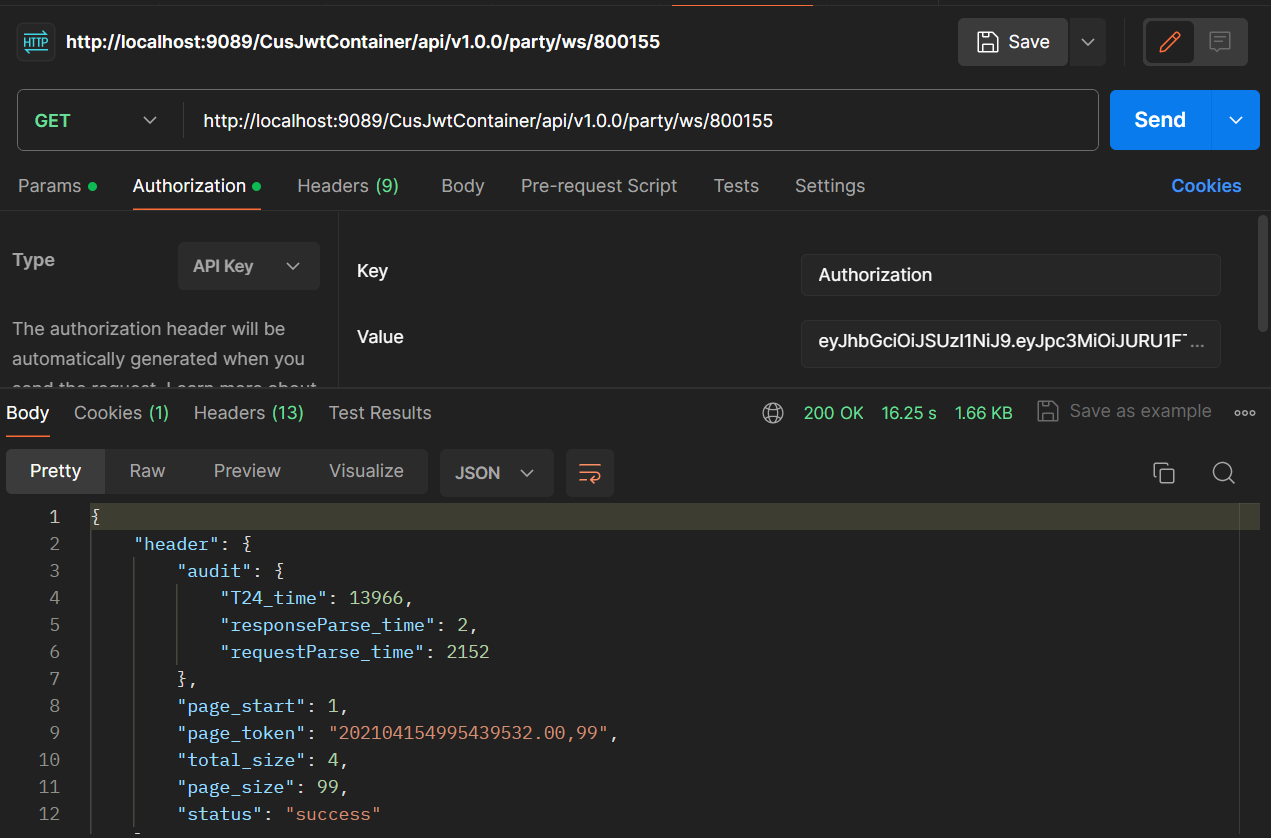




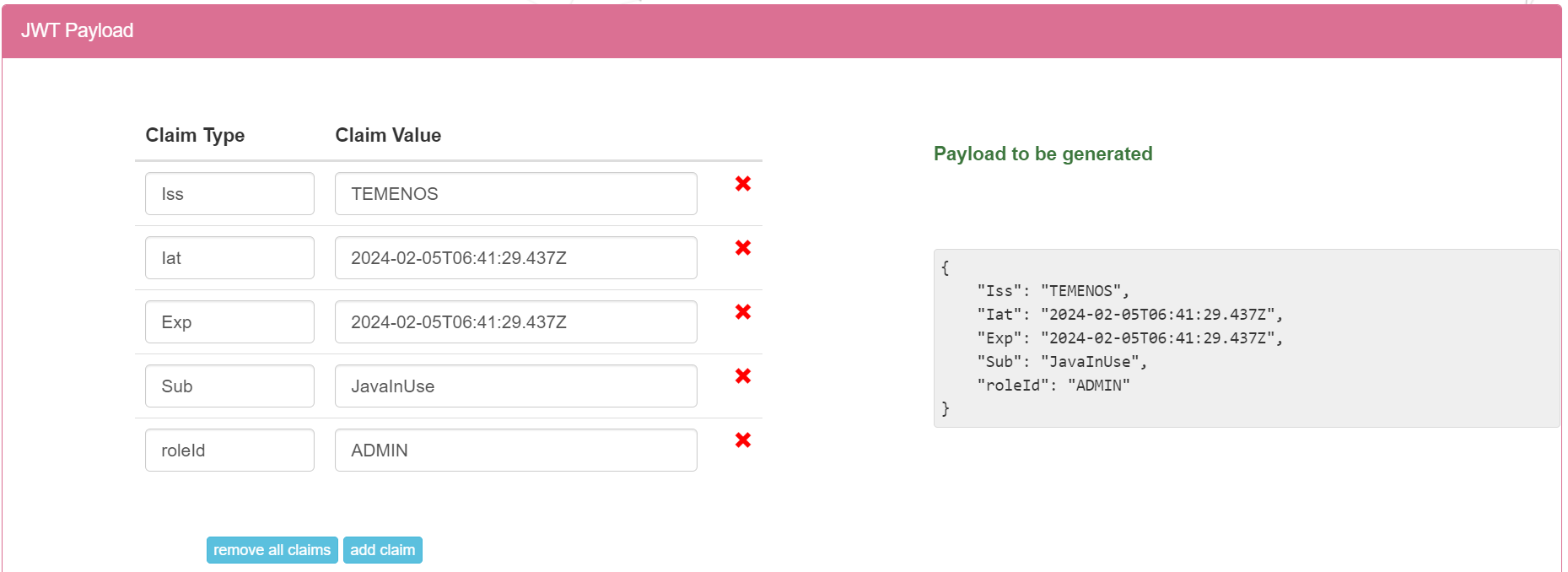
After changing the xml file it will look like this:



Now using **API Key,** the response is **OK** now.



**Create JWT using Web tool(**<https://www.javainuse.com/jwtgenerator>**)**





**CREATING PUBLISHER API**

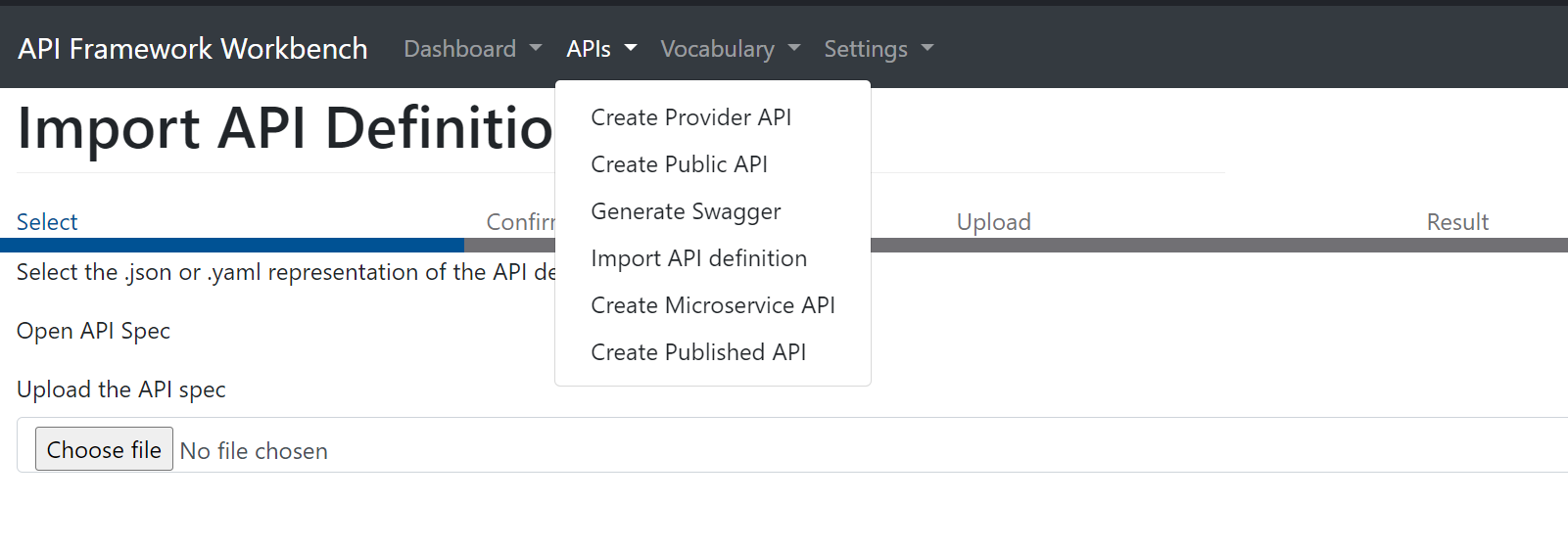
**Publisher APIs** are built over **PROVIDER APIs** so that the **JSON** structure and field values of **PROVIDER** **API** request and response can be modified as per the end user requirement at **IRIS layer**.

1. From the provider APIs **api-docs**, take the **swagger JSON**, modify as per your requirement of JSON structure and **save it.**

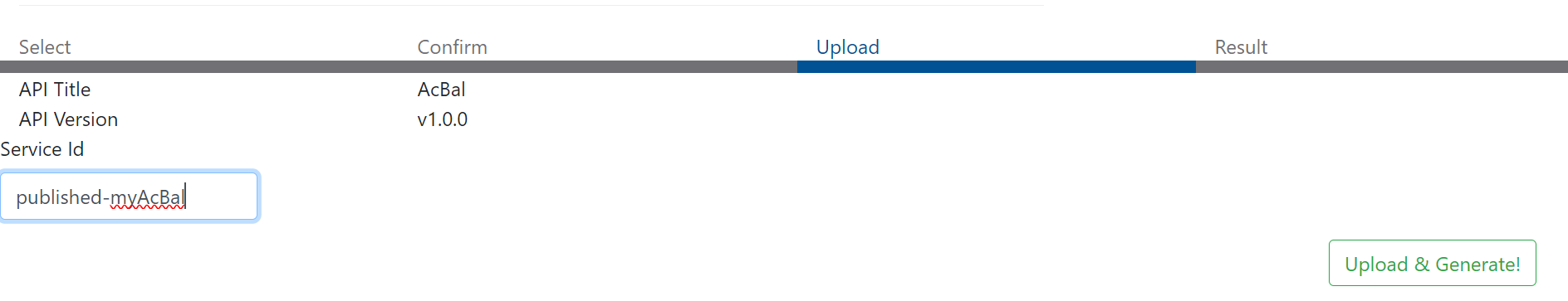
2. Check the correctness of that **swagger JSON** via the open-source swagger editor, the **URL** of which is given below. If any structural errors are thrown, resolve it.

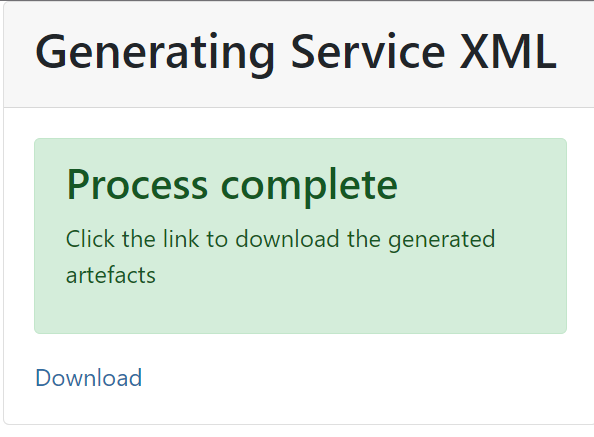
<https://editor-next.swagger.io/>

3. Import this swagger definition in workbench using the **“Import API definition”** option and download the publisher zip file.



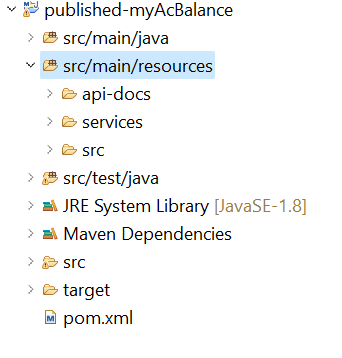
 





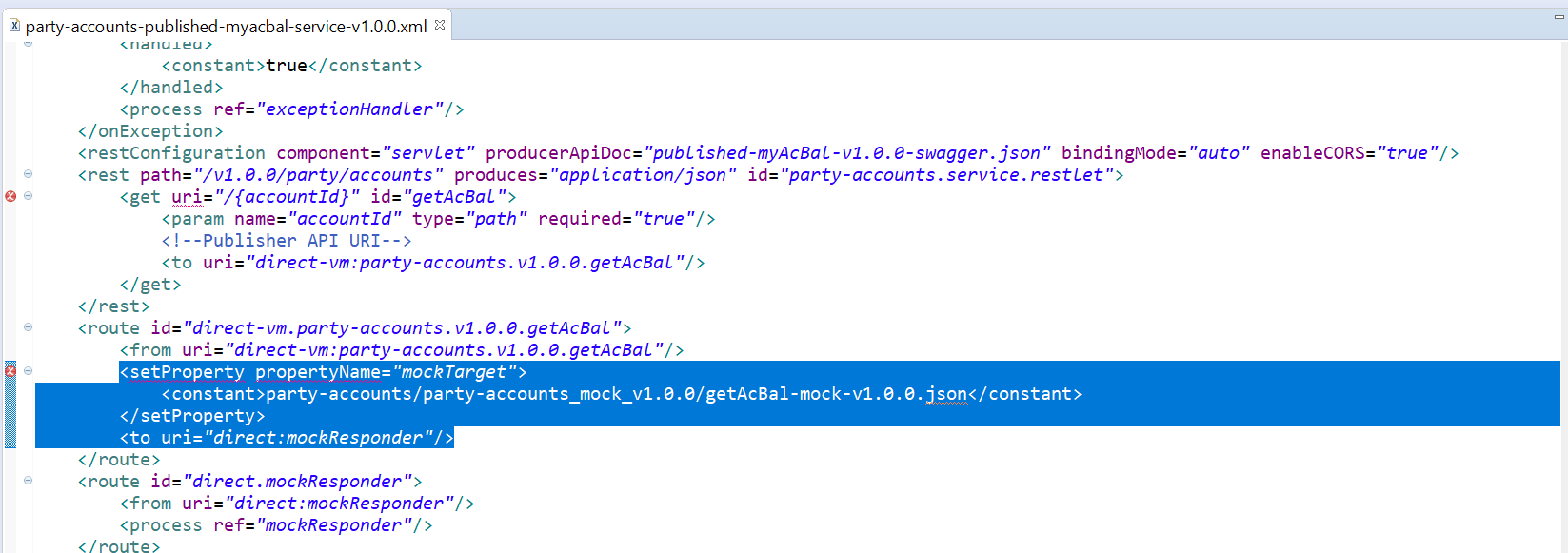
Download the zip file.

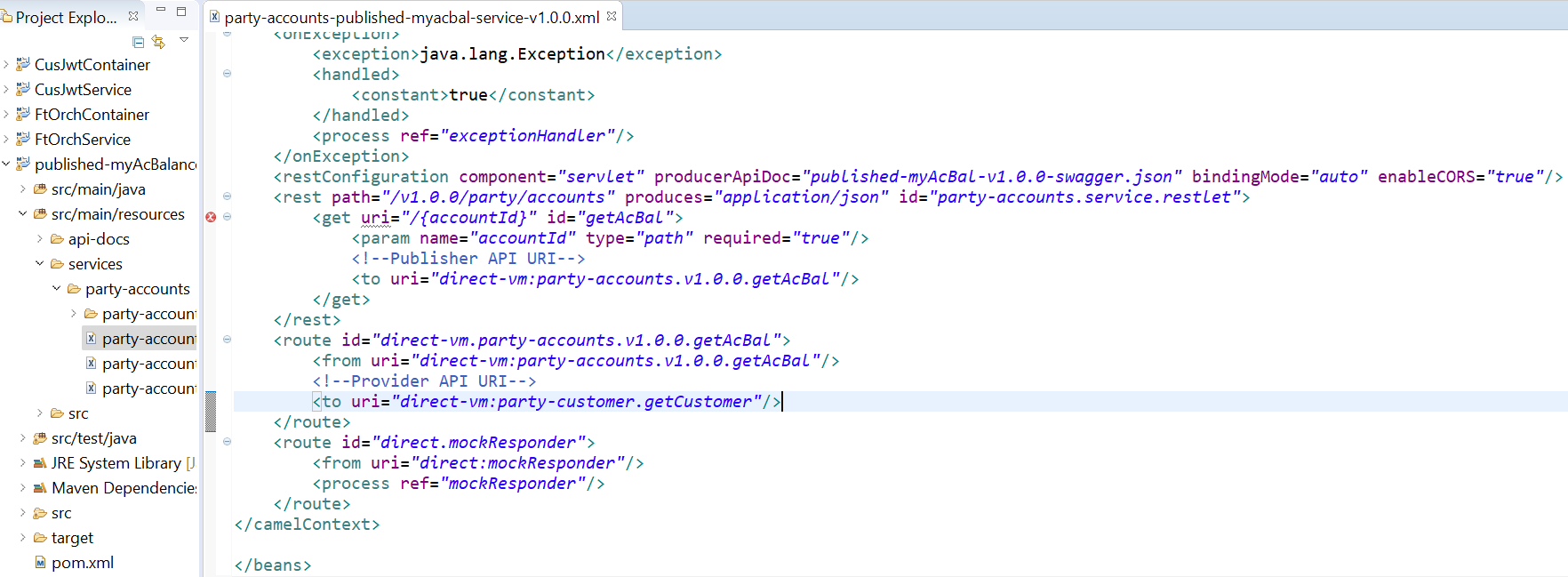
4. Create a maven project with **archetype-service catalog** and import the zip file downloaded in the previous step.



5. Edit the **service.xml** of **published API** such that the **URI** is redirected to the **provider API**.

**Replace** these lines with <to uri="direct-vm:party-customer.getCustomer"/>



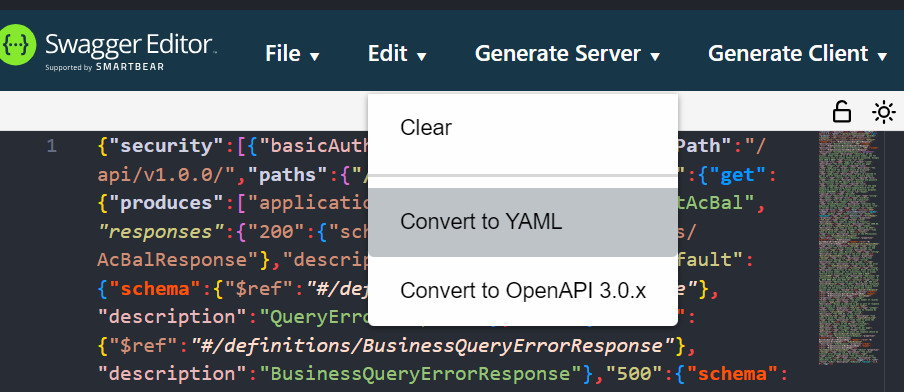


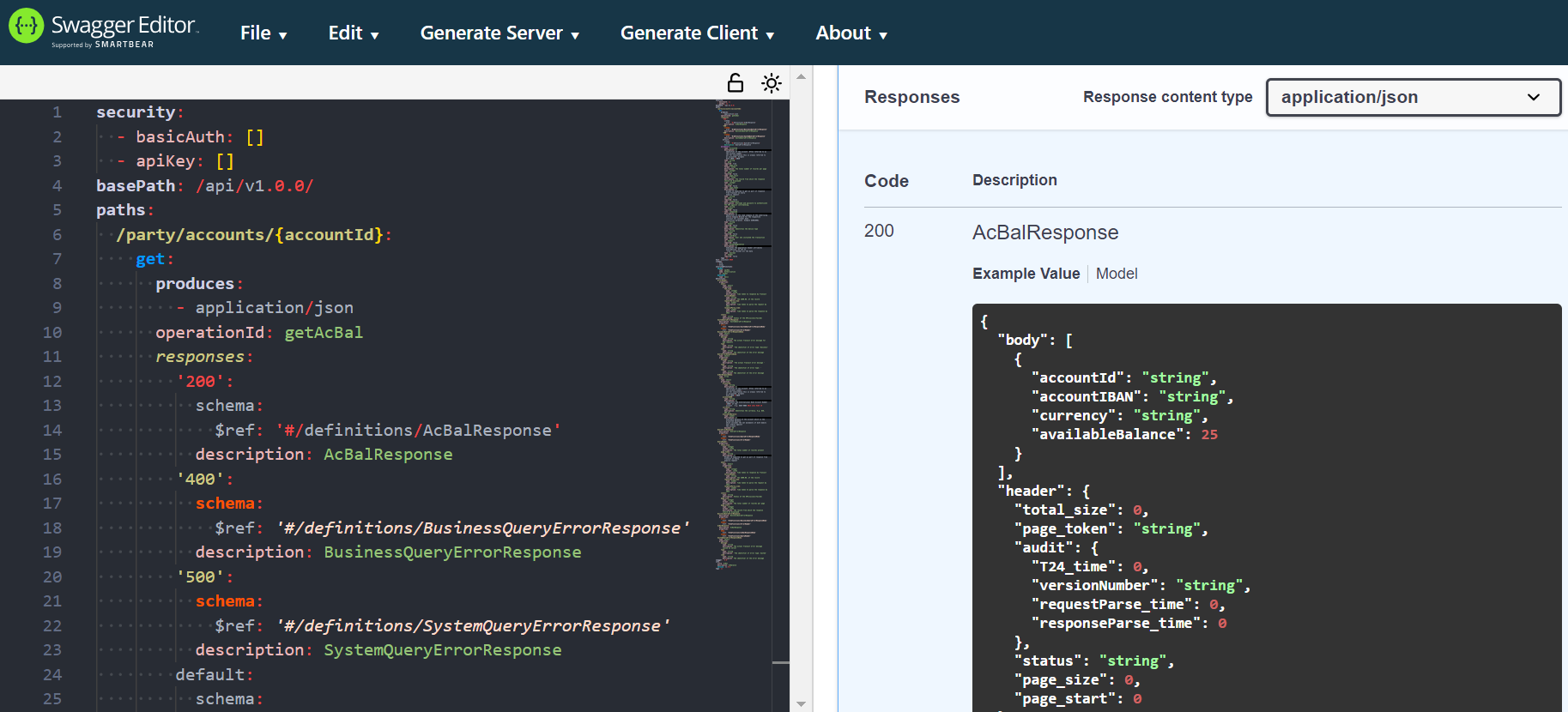


6. Generate code for the **published API swagger JSON**. To generate code, do the following steps.

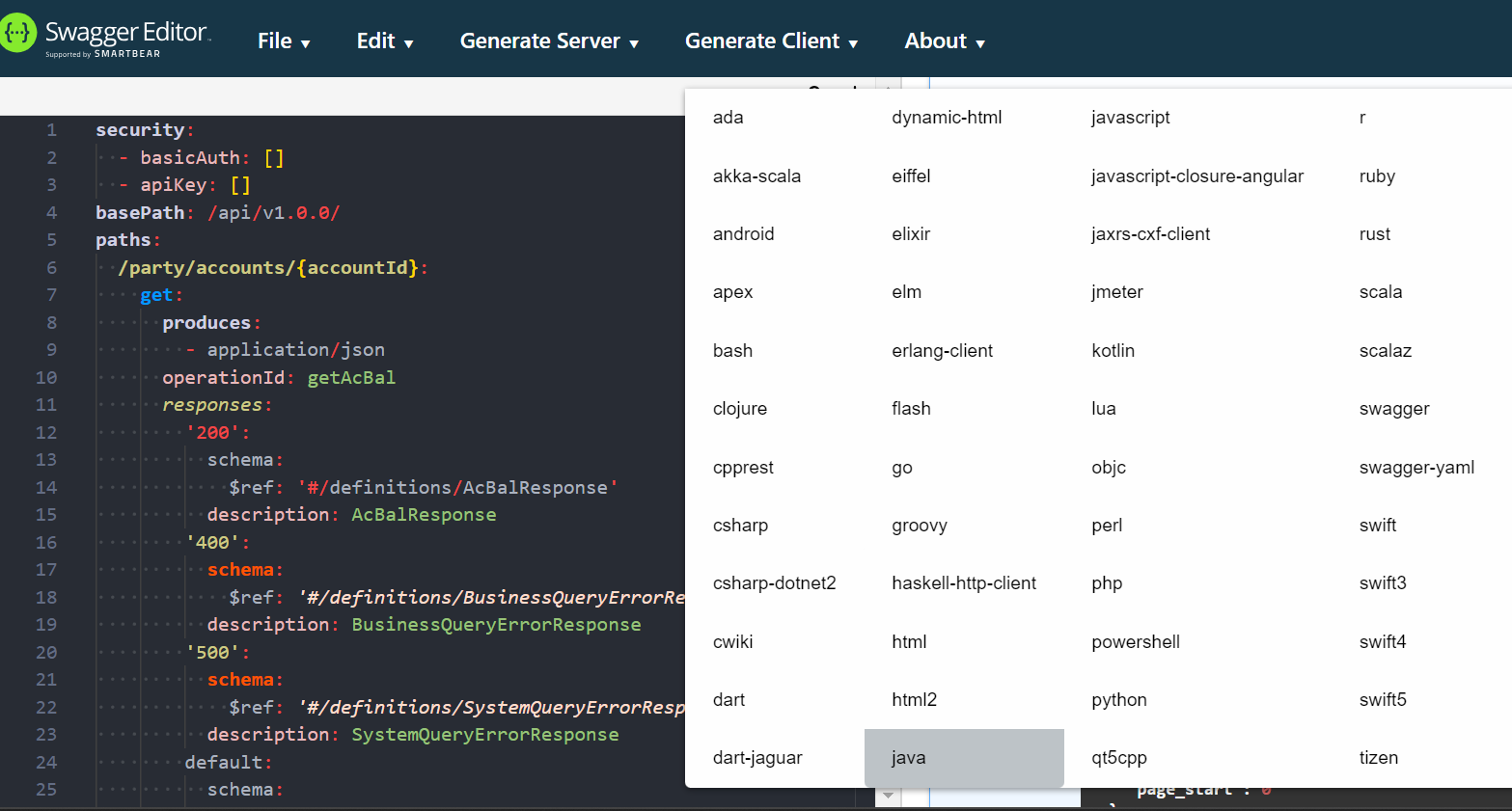
a. Launch <https://editor.swagger.io/> in browser.

b. Paste the content of the published API swagger JSON file. (It will be converted to YAML). Ensure no errors are thrown.





c. Click the option “Generate Client” and select “java” as shown in the below image.



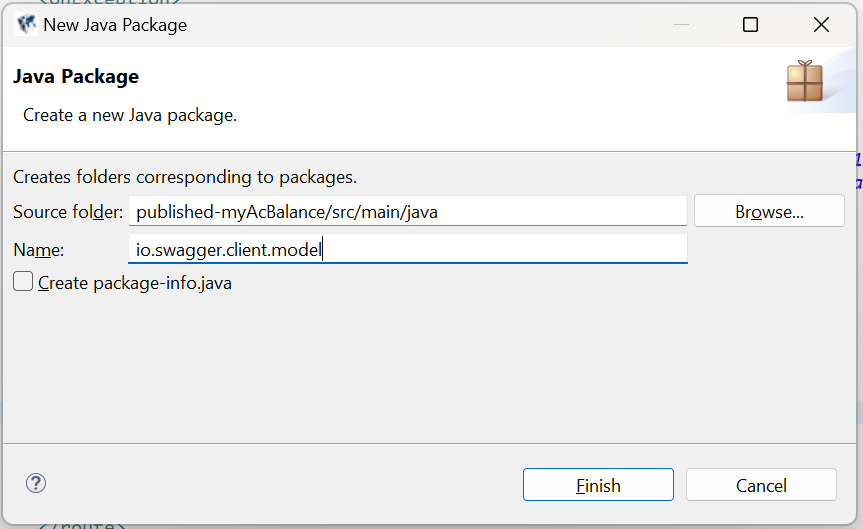
d. A file named **java-client-generated.zip** will be downloaded. **Extract** the contents of this file to a folder.

e. Under **<<PublishedApi\_project>>\src\main\ java** directory, create a **java package** named **io.swagger.client.model**



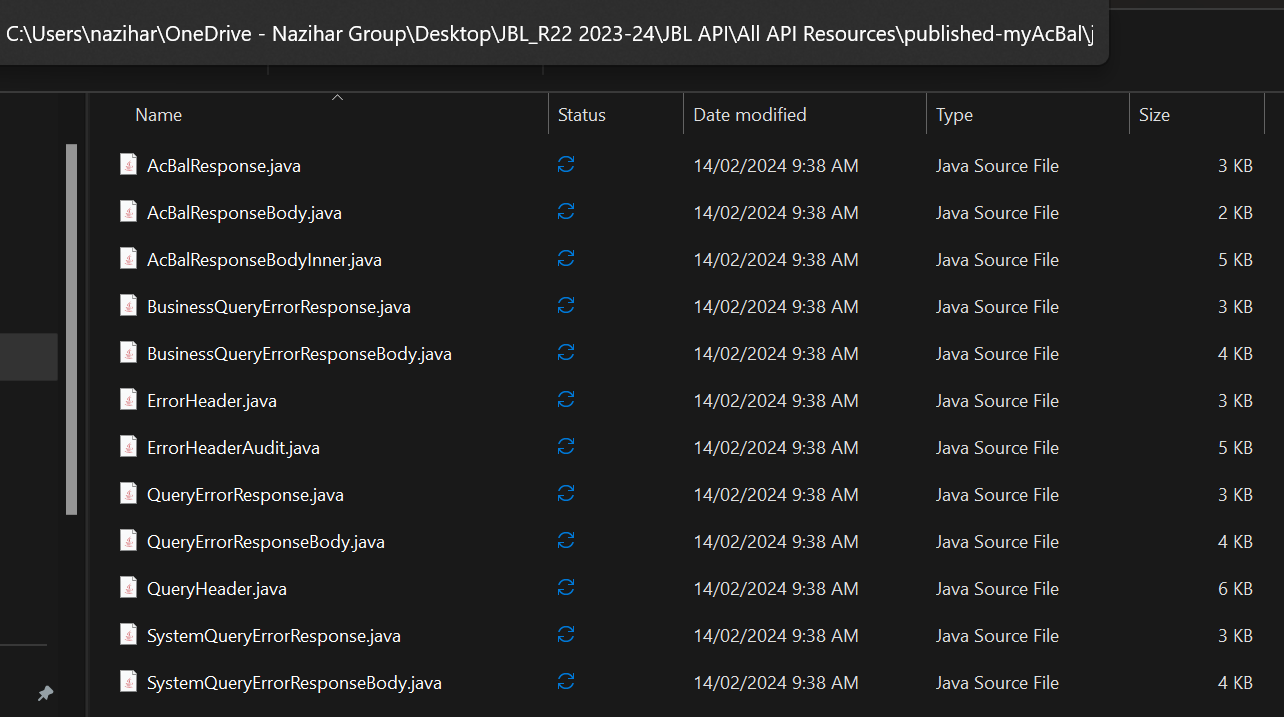


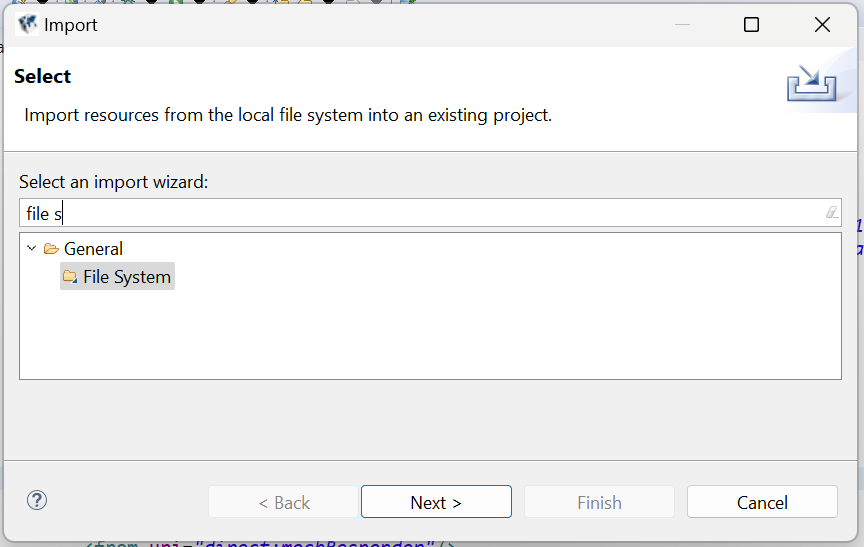




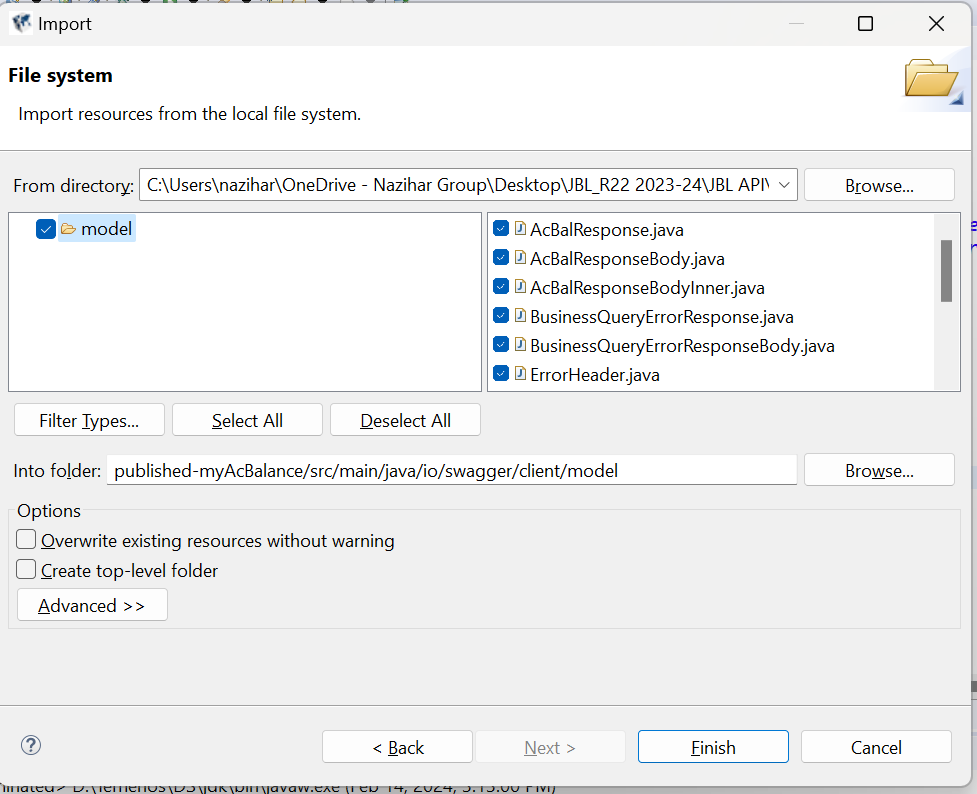
f. Right click src\main\java\io.swagger.client.model directory and select Import > File System -> navigate to the extracted zip file path\io\swagger\client\model and select this folder.

C:\Users\nazihar\OneDrive - Nazihar Group\Desktop\JBL\_R22 2023-24\JBL API\All API Resources\published-myAcBal\java-client-generated\java-client\src\main\java\io\swagger\client\model

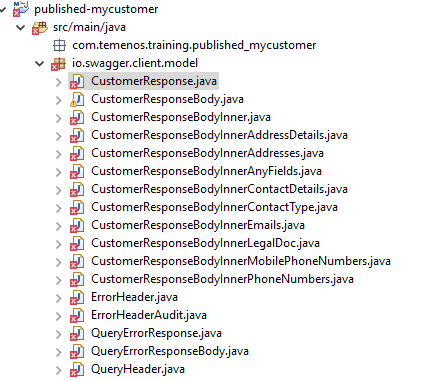




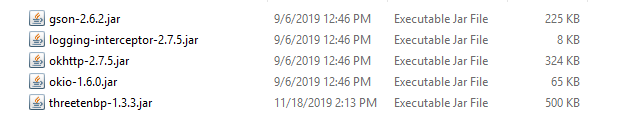
g. Click the “Select All” button and click Finish.

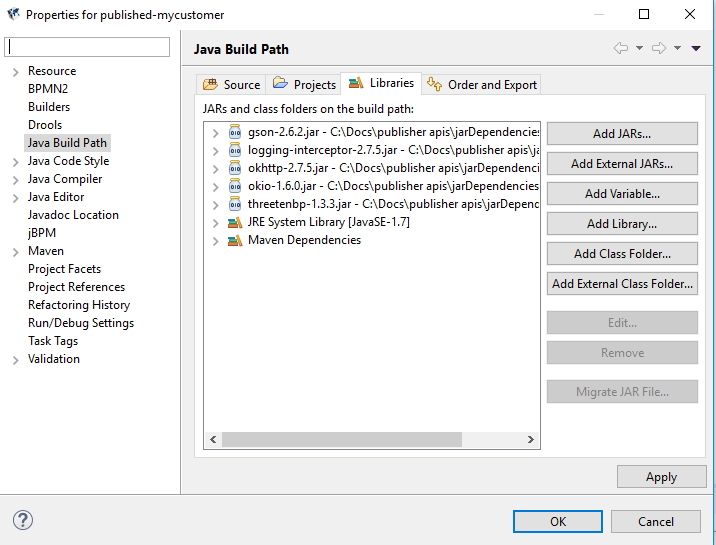


7. The imported java classes will throw error citing missing dependency jars. (DEPENDS on the SET-UP) – **May not Occur in NEW MB.**



1. To resolve these errors, download the below mentioned open-source jars (You can download it from any open-source jar download sites) and add it to the Build Path of the publisher API project (by right clicking the project > Configure Build Path > Libraries tab > Add External JARS > browse for the jars downloaded and select). **May not Occur in NEW MB.**





1. Edit the **pom.xml** of the publisher API project by adding the following dependencies.

-------------------------------------------------------------------------------------------------------------------------------------

<dependency>

<groupId>com.squareup.okhttp</groupId>

<artifactId>okhttp</artifactId>

<version>2.7.5</version>

</dependency>

<dependency>

<groupId>com.squareup.okhttp</groupId>

<artifactId>logging-interceptor</artifactId>

<version>2.7.5</version>

</dependency>

<dependency>

<groupId>com.google.code.gson</groupId>

<artifactId>gson</artifactId>

<version>2.6.2</version>

</dependency>

<dependency>

<groupId>org.threeten</groupId>

<artifactId>threetenbp</artifactId>

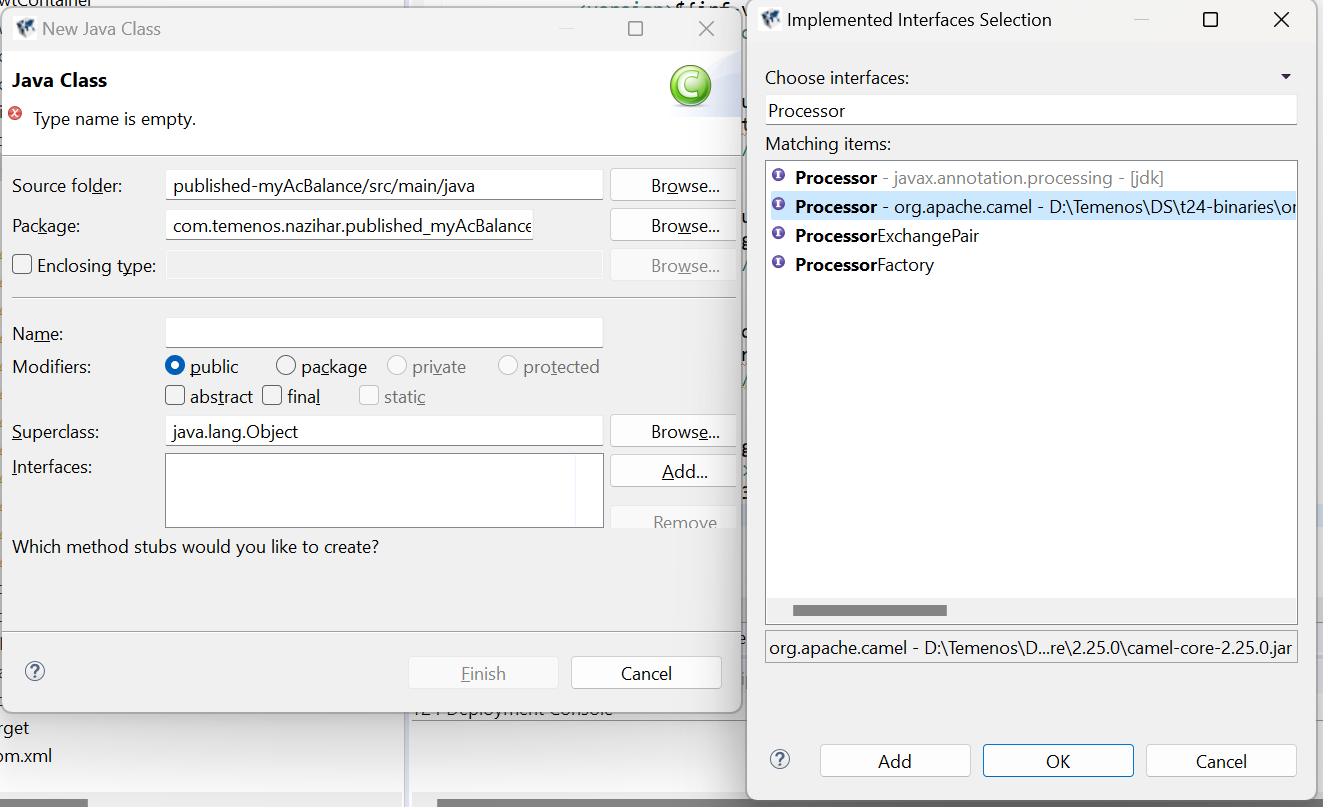
<version>1.3.3</version>

</dependency>

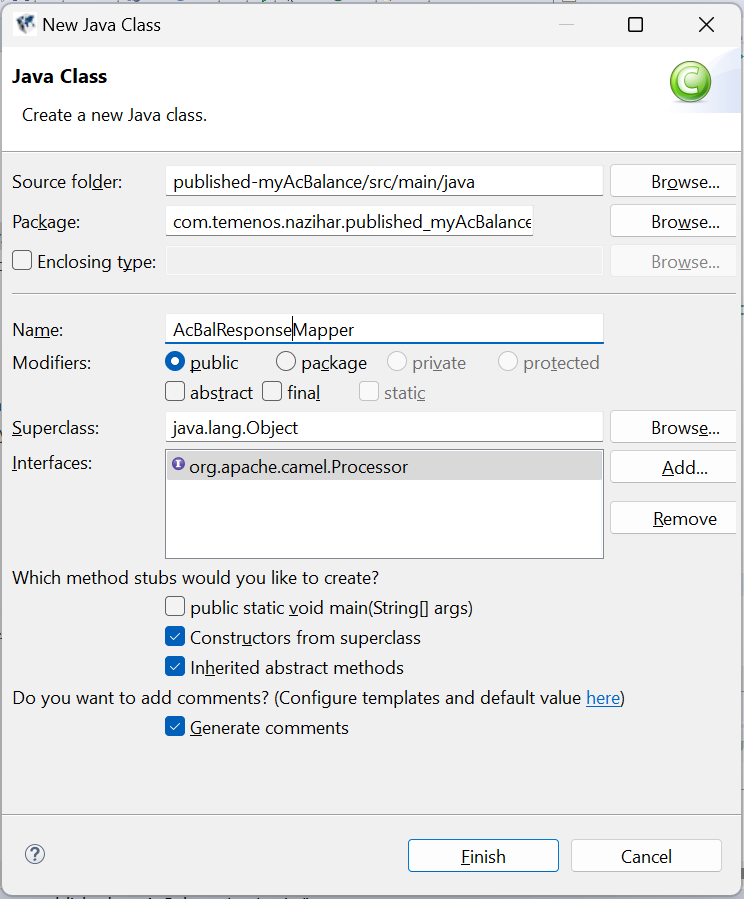
--------------------------------------------------------------------------------------------------------------------------------------

1. Navigate to **publisherProject - published\_myAcBalance**

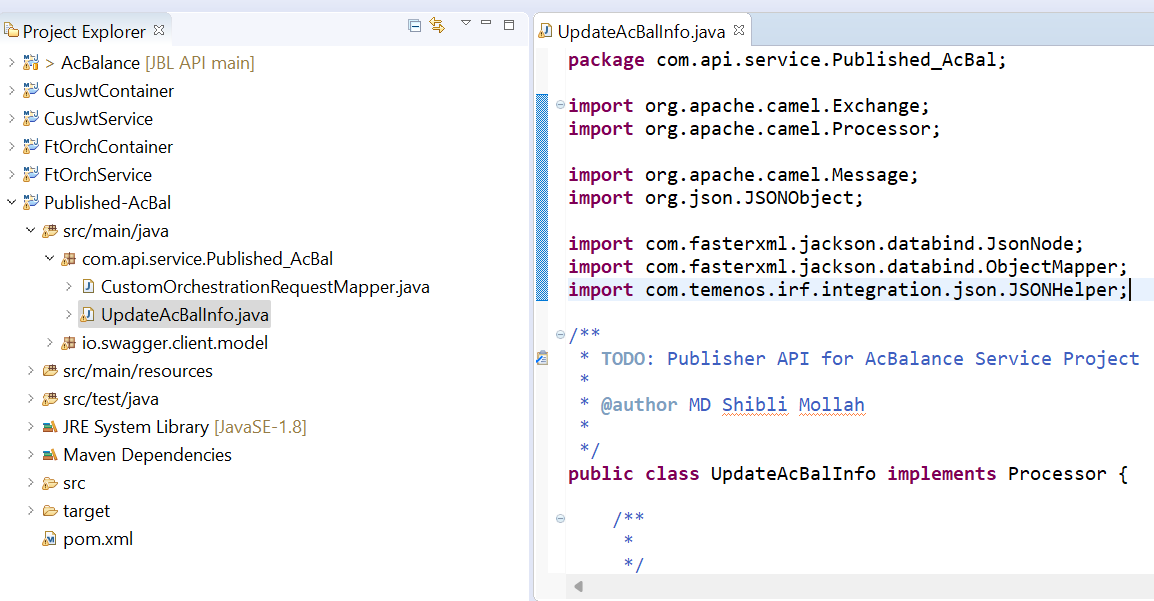
**<<publisherProject>>\src\main\java\** **com.temenos.nazihar. <<publisherProject>>** package directory, right click and select **New -> Java -> Class** and enter the class name as <<AnyRelevantName>>Mapper and add the interface named “Processor” under Interfaces.





****

11. The mapper class will be generated as shown below.

****

The process() method in the mapper must have the code logic to transform the provider API to publisher API and vice versa.

The code logic must be in such a way that the fields present in the **JSON response** body of the **provider** API are mapped to the fields defined in the swagger of **published** API using the **published** API swagger classes generated.



In the above illustration, we have generated **published** **API** for **ENQUIRY**. We will be explaining the step-by-step logic of mapper for both **ENQUIRY** based and **VERSION** based APIs below.

**Mapper class for ENQUIRY based API:**

For **ENQUIRY** API, we will require only a **response** mapper since we do not send a request body.

Below are the steps to create a Response Mapper for an ENQUIRY API based on ACCOUNT application.

**a.** As mentioned in **step 10**, create a **mapper class** with a relevant name (**AcBalResponseMapper**) and Processor interface. In the generated mapper java file, comment the constructor (if not required). Only the **process** method will be used.

**b.** **To map the fields of the provider API**, we must first receive the header and JSON body content (containing header and body section of JSON response) of the provider API after ensuring the http response code is **200**. Below is the corresponding code line. On confirming it is a successful response, we obtain the JSON response – header and body section.

**public** **void** process(Exchange exchange) **throws** Exception {

// **TODO** Auto-generated method stub

**if**((**int**)exchange.getIn().getHeader("CamelHttpResponseCode") == 200)

{

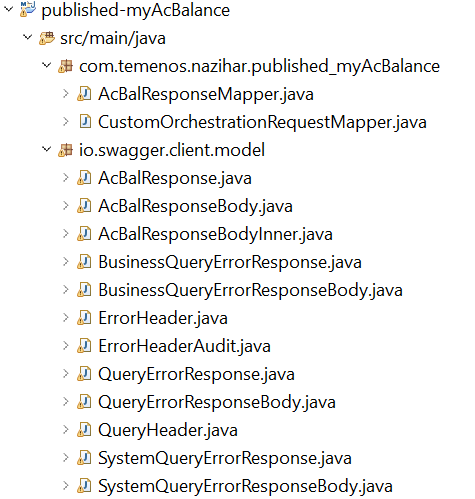
ObjectNode camelResponse = (ObjectNode)exchange.getIn().getBody(ObjectNode.**class**);

}

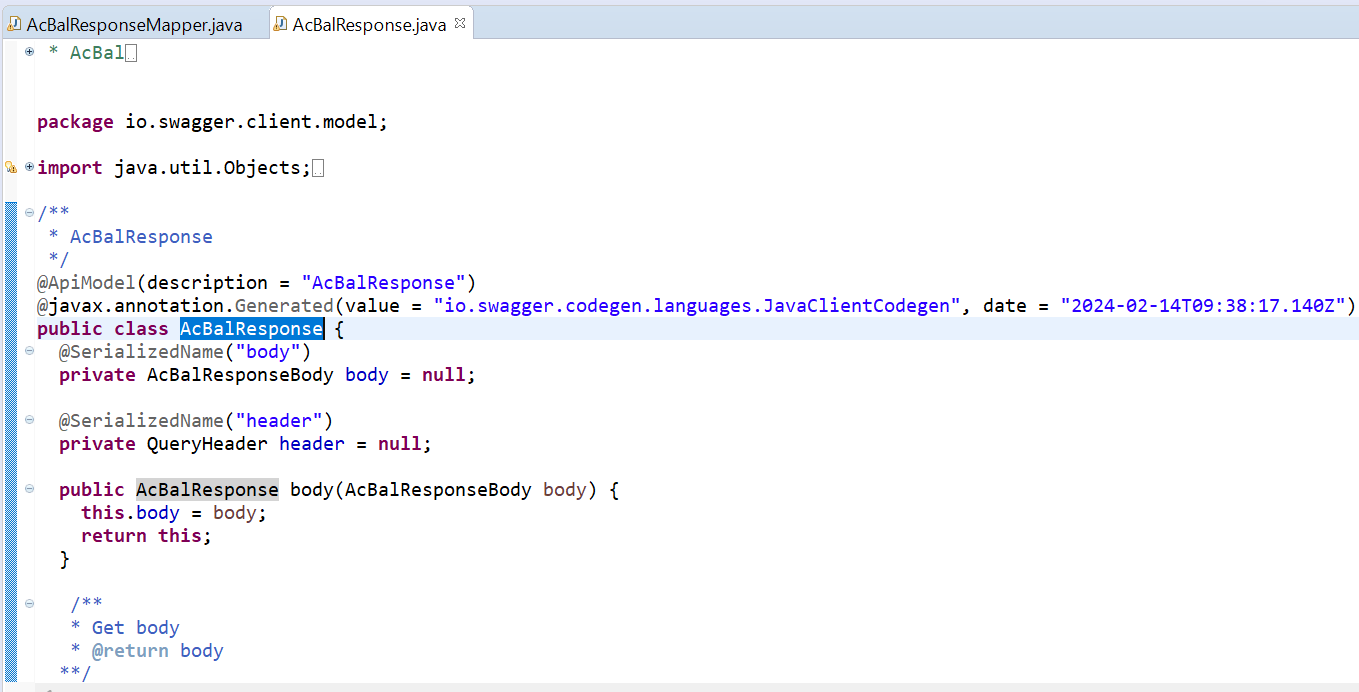
1. Receive the header and body content of provider JSON response.

|  |
| --- |
| ArrayNode body = (ArrayNode)camelResponse.get("body");  ObjectNode header = (ObjectNode)camelResponse.get("header"); |

d. Now, check the swagger classes which are generated under **io.swagger.client.model** package. Using the classes generated in that, we have to form the response body. Below are the classes generated for the **ACCOUNT ENQUIRY published API.**



e. Identify the parent response java class (it is **AcBalResponse.java** in the example) which will have the “header” and “data” (body) definition.



From the header and body data received from exchange object, we need to form the **new** **published API JSON** using the above swagger class and the other swagger classes this parent class depends on. In the above example, we must build the **objects** of **AcBalResponseBody** and **QueryHeader** and form the **AcBalResponse** which is illustrated below.

**public** QueryHeader formHeader(ObjectNode header)

{

QueryHeader qHeader = **new** QueryHeader();

JSONObject obj = **new** JSONObject(header.toString());

**if**(header.get("audit")!=**null**)

{

ErrorHeaderAudit eha = **new** ErrorHeaderAudit();

eha.setParseTime(obj.optInt("parse\_time"));

eha.setT24Time(obj.optInt("T24\_time"));

eha.setVersionNumber(obj.optString("versionNumber"));

qHeader.setAudit(eha);

}

**if**(header.get("page\_size")!=**null**)

qHeader.setPageSize(obj.optInt("page\_size"));

**if**(header.get("page\_start")!=**null**)

qHeader.setPageStart(obj.optInt("page\_start"));

**if**(header.get("page\_token")!=**null**)

qHeader.setPageToken(obj.optString("page\_token"));

**if**(header.get("total\_size")!=**null**)

qHeader.setTotalSize(obj.optInt("total\_size"));

**return** qHeader;

}

Enquiry response body will have response in form of multiple rows. We need to fetch the data field-wise for every row from the response of the provider API and form the body object of the publisher API. In the below snippet, code is written to include every row wise data to main publisher API body. Below is the corresponding code snippet.

**public** AcBalResponseBody formbody(ArrayNode body)

{

AcBalResponseBody acBalBody = **new** AcBalResponseBody();

**for**(**int** i=0; i<body.size(); i++)

{

AcBalResponseBodyInner acBalResp = **this**.formBodyContent(body.get(i));

acBalBody.add(acBalResp);

} **return** acBalBody;

}

Every row data must be formed by receiving the value of field present in provider JSON and setting it to the field of publisher JSON. If the fields are multi value fields or are part of an associated multi value set, then for every such case, separate class would have been created under io.swagger package. We need to make use of those classes to form the list of fields of a multi value set or an associated multi value set and add it to the row. Below is a small code snippet.

**public** AcBalResponseBodyInner formBodyContent(JsonNode object)

{

AcBalResponseBodyInner acBalResp = **new** AcBalResponseBodyInner();

JSONObject obj = **new** JSONObject(object.toString());

**if**(object.get("accountId")!=**null**)

acBalResp.setAccountId(obj.optString("accountId"));

**if**(object.get("accountIBAN")!=**null**)

acBalResp.setAccountIBAN(obj.optString("accountIBAN"));

**if**(object.get("accountIBAN")!=**null**)

acBalResp.setAccountIBAN(obj.optString("accountIBAN"));

For example, let us consider the line

acBalResp.setAccountId(obj.optString("accountId"));

Here, the “accountId” is the field display name of accountId in the provider API response and is set to the accountId field of publisher api using **setAccountId** method.

**Setting header and body to form complete response:**

AcBalResponse ar = **new** AcBalResponse();

AcBalResponseBody arb = **this**.formbody(body);

ar.setHeader(**this**.formHeader(header));

ar.setData(arb);

**Form the new json and set it to exchange variable.**

**final** ObjectMapper om = JSONHelper.createObjectMapper();

Gson gson = **new** GsonBuilder().setPrettyPrinting().create();

String gjsonStr = gson.toJson(ar);

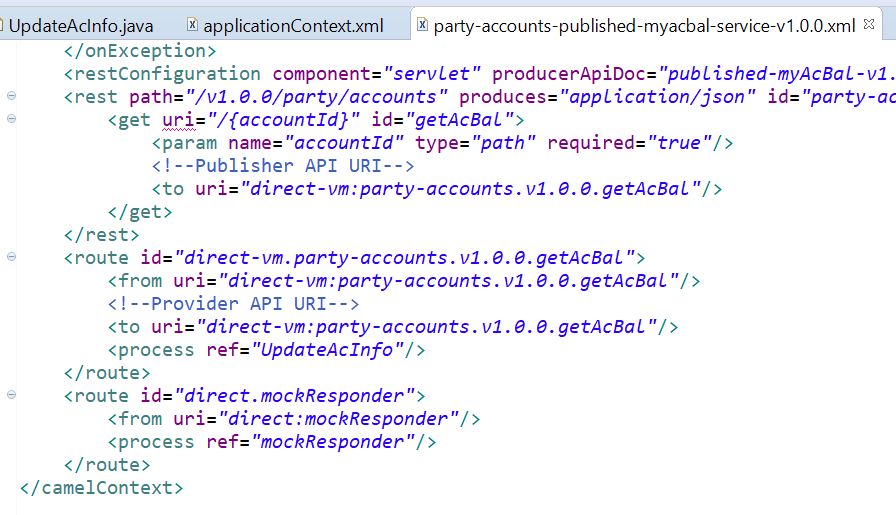
// logger.info("gson json string : "+gjsonStr);

JsonNode actualObj =om.readTree(gjsonStr);

exchange.getOut().setBody(actualObj);

**Save it.**

g. Edit the **service.xml** of the publisher API by adding this **ResponseMapper(***UpdateAcInfo***)** as a process after redirecting it to the provider API.



12. Maven Install the **publisher API** project(s).

13. Include the **publisher API** and **provider API** project(s) dependency in the **container** project’s pom.xml and perform maven install.



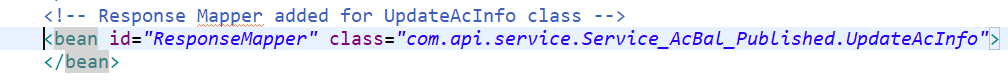




14. Create a new bean with any relevant name for “id” attribute in **applicationContext.xml** of the container project for the mapper class created.

<bean id="ResponseMapper" class="com.api.service.Service\_AcBal\_Published.UpdateAcInfo">

</bean>



**SAVE it.**

14. Deploy war in application server (or) jetty server.

15. APIs can now be accessed with structure of published api json and the corresponding response is now formed as per requirement.

Below screenshots show the JSON requests and responses for the examples explained in this document.