Oracle Banking Digital Experience

Custom Service Development Guide

July 2017

Custom Service Development Guide

July 2017

Table of Contents

1. Creating your own REST API with DIGX Framework

DIGX Framework overview:

Let's go through the building blocks of DIGX framework. To build a REST API, each of these framework components (as mentioned below) needs to be addressed and that's why it becomes important to have a holistic idea about each of them. The arrangement of all of these framework components can be clearly understood in the following diagram:

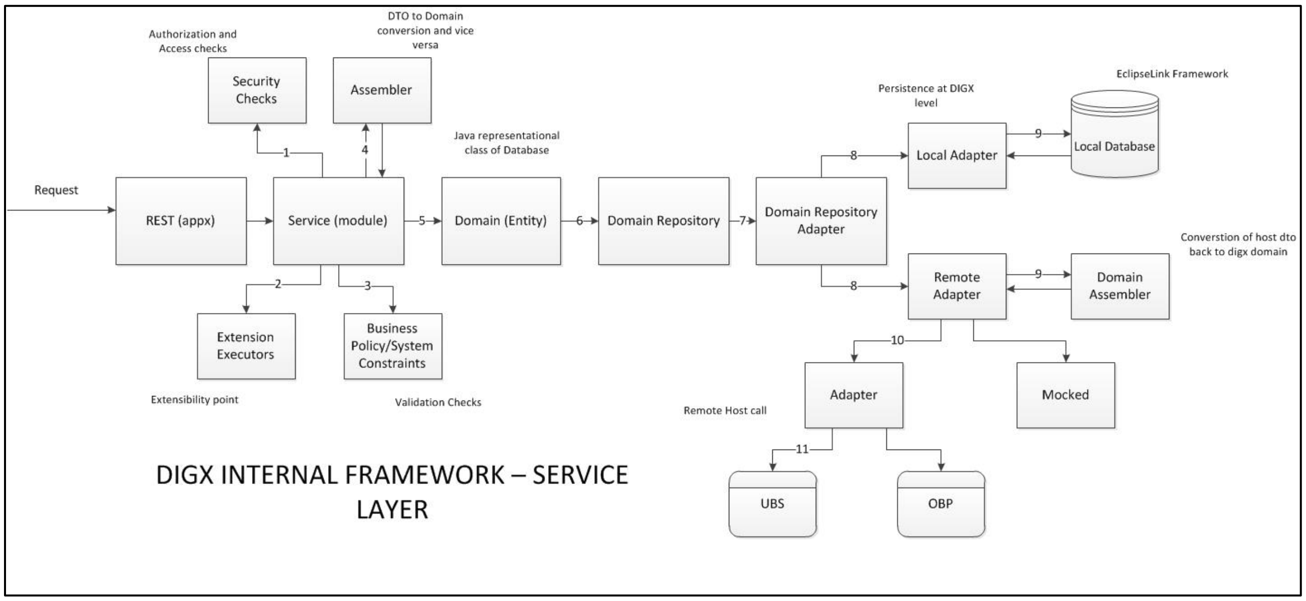

DTO to Domain conversion and vice versa
Authorization and Access checks
Security Checks
Assembler
EclipseLink Framework
Persistence at DIGX level
Local Adapter
Local Database
Java representational class of Database
Request
REST (appx)
Service (module)
Domain (Entity)
Domain Repository
Domain Repository Adapter
Converstion of host dto back to digx domain
Remote Adapter
Domain Assembler
Extension Executors
Business Policy/System Constraints
Extensibility point
Validation Checks
Adapter
Mocked
Remote Host call
DIGX INTERNAL FRAMEWORK – SERVICE LAYER
UBS
OBP

Figure 1 DIGX Service Layer

1. **REST:** The endpoint layer which gets invoked whenever a request URI is called. Also known as the layer which contains REST annotations and path to resources or sub-resources of an application
2. **Service:** Also called as module layer of the framework. Generally, the core modules of DIGX application will have their own service implementation classes responsible for implementing core business logic, validation and security checks
3. **Assemblers:** These are the mapping classes which convert data object containing request or response parameters into domain or database compatible form. These classes help us to get the required domain objects which can be further used in object-relational mapping
4. **Business Policy/ System Constraints:** Before letting the query data read or persisted in the core application, certain business policies need to be validated. This separate layer of constraints check let the application behave as per the policies configured
5. **Domain/Entity:** Represents the Java Object form of Database. This domain layer also contains data to be persisted or query response fetched through Object relational mapping
6. **Domain Repository:** The term 'repository' denotes any data storage component. Each module of the application will have its own repository to manage its CRUD operations and that can be easily done using this component of the DIGX framework

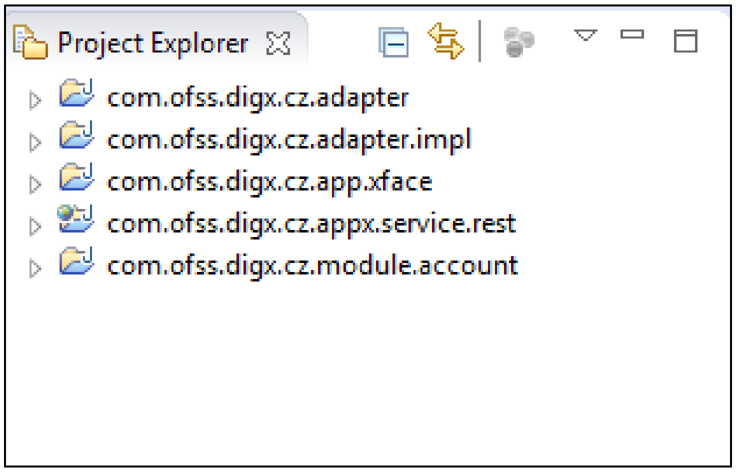
7. **Domain Repository Adapter:** Adapters are the connecting points to some external system and as the name suggests, this part of the framework contacts two kinds of repositories of DIGX application – Local Repository and Remote Repository. Eventually, the configured one out of these two will be invoked
8. **Adapters:** Finally these are the adapter classes that can call either Local Database (DIGX specific tables) or Remote Repository (external system). Remote adapters can further be mocked if required
9. **External System/ Host:** The core banking application such as UBS or OBP or any third-party application which operates final banking transactions.

Projects description:

In order to implement the DIGX architecture, we will create separate projects for different framework components in Eclipse (with JDK 8)

Why separate projects? : Ensures high extensibility and loose coupling between different components of the system. Also, in later stages, sustenance becomes easier and it helps developers also to effectively maintain the ever-growing code.

Moving on, let's create the following projects as shown in the Figure 2:



com.ofss.digx.cz.adapter
com.ofss.digx.cz.adapter.impl
com.ofss.digx.cz.app.xface
com.ofss.digx.cz.appx.service.rest
com.ofss.digx.cz.module.account

Figure 2 Project structure

How to create project packages and classes?

Project 1: com.ofss.digx.cz.appx.service.rest (REST project)

Create a **Dynamic Web Project** with the project name '**com.ofss.digx.cz.appx.service.rest**'. Now create a package in that project with the name **'com.ofss.digx.appx.<module_name>'** .

This package contains the endpoint class to take requests and send responses back to server. The REST classes usually contain JAX-RS annotations and URIs which help them to locate themselves whenever a request is made through a REST call.

All the classes in this project extend **AbstractRESTApplication** and also have a default interface class prefixed with 'I' before the name of the corresponding REST implementation class. For instance, Account REST class will have IAccount Interface class in the same the package of this project.

One more important thing about this project is that this gets deployed as an application and rest of the projects as libraries. Having said that, this project should therefore be created as **Dynamic Web Project** in the New option of Eclipse IDE.

Project dependencies for REST project:

Add the following projects (to be created later) in the Java build path of this project:

com.ofss.digx.cz.app.xface, com.ofss.digx.cz.module.<module_name>

Classpath variables:

Create the following classpath variables (left column) and extend them to the jars (names on the right side column). **Note** that these libraries can be found in the OBDX Installer folder and the exact location of each jar can be found in the section 8 (last part) of this document.

Classpath name	**Jars to extend**
DIGX_LIB	com.ofss.digx.appcore.jar, com.ofss.digx.common.jar, com.ofss.digx.infra.jar, com.ofss.digx.infra.audit.jar, AbstractRESTApplication.jar, com.ofss.digx.datatype.jar, com.ofss.digx.enumeration.jar, com.ofss.digx.module.common.jar, com.ofss.digx.module.origination.jar, com.ofss.digx.module.security.jar, com.ofss.digx.appcore.dto.jar
EXT_LIB	All jersey2 libraries found in OBDX installer folder
OBP_LIB	com.ofss.fc.infra.jar, com.ofss.fc.appcore.dto.jar, com.ofss.fc.appcore.jar, com.ofss.fc.enumeration.jar

Project 2: com.ofss.digx.cz.app.xface (DTO/xface project)

Create a Java Project with the project name '**com.ofss.digx.cz.app.xface'**. Now create a package in that project with the name **'com.ofss.digx.app.<module_name>.dto' .**

This package consists of Data Transfer Object classes, also referred as Plain Old Java Objects or POJO. All the request as well as response DTO classes are created under this project. The Request DTO classes in this project extend **DataTransferObject** present in OBP libraries whereas the Response DTO classes extend **BaseResponseObject**.

Also, every Request DTO classes will have a separate **Canonicalizer** class responsible for reducing possibly encoded string parameters into its simplest form. This is therefore a very important part that should not be missed while writing any DTO classes. Similarly, for the Response DTO classes, we will have to write a separate **Encoder** class responsible for encoding the string back to its original form for HTML pages. The naming convention for Canonicalizer and Encoder classes will be - <RequestDTO_classname>Canonicalizer and <ResponseDTO_classname>Encoder respectively.

Project dependencies for xface project:

Not needed.

Classpath variables:

Extend the following classpath variables to the jars found in OBDX_Installer:

Classpath name	**Jars to extend**
DIGX_LIB	com.ofss.digx.appcore.dto.jar com.ofss.digx.infra.jar,

EXT_LIB	No jars to extend
OBP_LIB	com.ofss.fc.appcore.dto.jar, com.ofss.fc.infra.jar,

Project 3: com.ofss.digx.cz.module.<module_name> (module project)

Create a new Java project with this name. This project contains the vital business logic, extension points, constraints, security checks like authorization and access control. The following packages need to be created inside this project:

1. **com.ofss.digx.app.<module_name>.service :** Add the Service Interface and Implementation class in this package. The name of Service class should be same as the name of the REST class created in the REST project. For instance, this package will have classes named IAccount.java and Account.java which are same as the REST class name for Account. This service class extends **AbstractApplication** of the DIGX framework.
2. **com.ofss.digx.app.<module_name>.service.ext:** Contains classes for extensions and executors. Each Service classes will have their own extension points. Refer mock workspace for more detail.
3. **com.ofss.digx.app.<module_name>.assembler:** Create <module_name>Assembler class inside this package. All Assembler classes extends **AbstractAssembler** .
4. **com.ofss.digx.domain.<module_name>.entity:** This package should include Entity class for the module. The name of entity class to be created should be same as REST as well as Service class names. For instance, it will have Account.java entity class for Account service and REST classes. Also known as Domain classes, they extend **AbstractDomainObject** taken from OBP libraries.
5. **com.ofss.digx.domain.<module_name>.entity.assembler:** Add a domain assembler class with the name <module_name>DomainAssembler in this package.
6. **com.ofss.digx.domain.<module_name>.entity.policy:** Add the business policy classes in this package to ensure the validation of business constraints added in these classes. Refer workspace attached with this document for more detail. Classes in this project are again one of the must-haves as far as enforcement of any system validation is concerned.
7. **com.ofss.digx.domain.<module_name>.entity.repository:** Contains repository class (<module_name>Repository.java) which invokes Repository adapter classes described in the next point. This class extends **AbstractDomainObjectRepository** of DIGX framework.
8. **com.ofss.digx.domain.account.entity.repository.adapter:** Add repository adapter interfaces, Local and Remote Repository Adapter classes in this project. If you are writing for the Account service, the naming convention of these classes should be I<module_name>RepositoryAdapter, Local<module_name>RepositoryAdapter, Remote<module_name>RepositoryAdapter respectively.

With this ends the package structure for service classes. The implementation of this project takes maximum time and involves majority of the DIGX service layer handling. It is therefore a very crucial part to look for while developing a REST API in DIGX.

Project dependencies for module project:

Add the following projects in the Java build path of this project:

com.ofss.digx.cz.adapter, com.ofss.digx.cz.app.xface

Classpath variables:

Extend the following classpath variables to the jars found in OBDX_Installer:

Classpath variable name	**Jars to extend**
DIGX_LIB	com.ofss.digx.framework.domain.jar, com.ofss.digx.infra.jar , com.ofss.digx.appcore.jar, com.ofss.digx.common.jar, com.ofss.digx.datatype.jar, com.ofss.digx.adapter.jar, com.ofss.digx.module.alerts.jar, com.ofss.digx.module.approval.jar, com.ofss.digx.module.party.jar, com.ofss.digx.enumeration.jar, com.ofss.digx.module.common.jar, com.ofss.digx.appcore.dto.jar
EXT_LIB	No jars to extend
OBP_LIB	com.ofss.fc.framework.domain.jar, com.ofss.fc.enumeration.jar , com.ofss.fc.datatype.jar , com.ofss.obp.patch.jar, com.ofss.fc.infra.jar, com.ofss.fc.appcore.dto.jar, com.ofss.fc.appcore.jar, com.ofss.fc.framework.dto.jar,

Project 4: com.ofss.digx.cz.adapter (adapter project)

Create a Java Project with this name which contains all the Adapter Interfaces in this project. Within this project create a package with the name **com.ofss.digx.app.<module_name>.adapter.** Now include the adapter interface for the adapter implementation class for your module. For example, in case of Account module, name of the interface created should be IAccountAdapter.

Project dependencies for adapter project:

Add the following projects in the Java build path of this project:

com.ofss.digx.cz.app.xface

Classpath variables:

Extend the following classpath variables to the jars found in OBDX_Installer:

Classpath name	**Jars to extend**
DIGX_LIB	com.ofss.digx.infra.jar

EXT_LIB	No jars to extend
OBP_LIB	No jars to extend

Project 5: com.ofss.digx.cz.adapter.impl (adapter impl project)

This project contains implementation classes for all the adapter interfaces created in the **com.ofss.digx.cz.adapter** project. Create a package named **com.ofss.digx.app.<module_name>.adapter.impl** and add the following classes:

1. **<module_name>AdapterFactory.java :** Factory class to generate Adapter instances for every getAdapter request call. Returns either mock adapter or adapter to call host interface
2. **<module_name>Adapter.java:** A very essential Adapter class for a specific module which is entitled to call external host system
3. **<module_name>MockAdapter.java:** In case a call to host system needs to be skipped and local mocked data needs to be fetched, this adapter class can be used

Project dependencies for adapter impl project:

Add the following projects in the Java build path of this project:

com.ofss.digx.cz.adapter , com.ofss.digx.cz.app.xface, com.ofss.digx.cz.module.<module_name>

Classpath variables:

Extend the following classpath variables to the jars found in OBDX_Installer:

Classpath name	**Jars to extend**
DIGX_LIB	com.ofss.digx.appcore.dto.jar, com.ofss.digx.infra.jar, com.ofss.digx.adapter.jar, com.ofss.digx.datatype.jar,
EXT_LIB	No jars to extend
OBP_LIB	com.ofss.fc.framework.dto.jar

Implementing Classes:

Refer the mock classes in the workspace attached with this project. The self-explanatory documentation should be able to guide you towards creating the classes specified in the above projects.

Database Scripts to be added:

There are few places where we decide which classes to be invoked in runtime. These are the possible database configurations done in an ideal case. Please add the following entries in the DIGX_FW_CONFIG_ALL_B table: (Account Service taken as an example and in accordance with the workspace example)

1. INSERT INTO digx_fw_config_all_b (PROP_ID, CATEGORY_ID, PROP_VALUE, FACTORY_SHIPPED_FLAG, PROP_COMMENTS, SUMMARY_TEXT, CREATED_BY, CREATION_DATE, LAST_UPDATED_BY, LAST_UPDATED_DATE, OBJECT_STATUS, OBJECT_VERSION_NUMBER) VALUES ('ACCOUNT_CZ_REPOSITORY_ADAPTER', 'repositoryadapterconfig', 'com.ofss.digx.domain.account.entity.repository.adapter.RemoteAccountRepositoryAdapter', 'N', null, 'Adapter repository adapter class', 'ofssuser', sysdate, 'ofssuser', sysdate, 'Y', 1);
2. INSERT INTO digx_fw_config_all_b (PROP_ID, CATEGORY_ID, PROP_VALUE, FACTORY_SHIPPED_FLAG, PROP_COMMENTS, SUMMARY_TEXT, CREATED_BY, CREATION_DATE, LAST_UPDATED_BY, LAST_UPDATED_DATE, OBJECT_STATUS, OBJECT_VERSION_NUMBER) VALUES ('ACCOUNT_CZ_ADAPTER_FACTORY', 'adapterfactoryconfig', 'com.ofss.digx.app.account.adapter.impl.AccountAdapterFactory', 'N', null, 'adapter factory class', 'ofssuser', sysdate, 'ofssuser', sysdate, 'Y', 1);
3. INSERT INTO digx_fw_config_all_b (PROP_ID, CATEGORY_ID, PROP_VALUE, FACTORY_SHIPPED_FLAG, PROP_COMMENTS, SUMMARY_TEXT, CREATED_BY, CREATION_DATE, LAST_UPDATED_BY, LAST_UPDATED_DATE, OBJECT_STATUS, OBJECT_VERSION_NUMBER) VALUES ('ACCOUNT_CZ_ADAPTER_MOCKED', 'adapterfactoryconfig', 'false', 'N', null, 'Flag to decide to go to Mocked adapter or Remote', 'ofssuser', sysdate, 'ofssuser', sysdate, 'Y', 1);

Configuring newly created services:

a. Task Registration

Every new service to be integrated as a part of **OBDX** needs to provide a task code. This task code is required while integrating the
service with various infrastructural aspects applicable to the service. Few examples of infrastructural aspects or cross cutting
concerns provided out of the box with **OBDX** are:

- **Limits**
- **Approvals**
- **Two Factor Authentication**
- **Transaction Blackout**

- **Working Window**

Guidelines for formulating a task code are as follows.

A task code should ideally comprise of 3 parts:

1. **Module Name** : The first 2 alphabets representing the module to which the service in question belongs. eg TD represents Term Deposits
   module.
2. **Task Type(type of service) :** OBDX supports the following 6 types of services.
   a. **FINANCIAL_TRANSACTION(F) :** Any transaction as a result of which there is a change in the status of the finances of accounts of
   the participating parties. In general any transaction that involves monetary transfer between parties via their accounts. Few
   examples include Self transfer, New deposit(Open term deposit), Bill payment etc.
   b. **NONFINANCIAL_TRANSACTION(N) :** Any transaction that pertains to an account but there is no monetary payment or transfer involved
   in it. For example Cheque book request.
   c. **INQUIRY(I) :** Any read only transaction supported in OBDX that does not manipulate any business domain of the financial
   institution. For example list debit cards, read loan repayment details, fetch term deposit penalties etc.
   d. **ADMINISTRATION(A) :** Transactions performed by bank admins and corporate admins for a party come under this category. Few examples of
   such transactions include limit definition, limit package definition, user creation, rule creation and various others.
   e. **MAINTENANCE(M) :** Maintenances done by a party for itself fall under this category. Maintenance transactions performed by a
   non admin user which does not involve any account or monetary transaction comprise of this transaction type. Example add biller.
   f. **COMMON(C)** : Common transactions include transactions which do not fall under any of the above mentioned categorization. Example login.
   So 1 alphbet F,N,I,A,M or C for each of the above mentioned task types respectively forms the second part of the task code.
3. **Abbreviation for service name :** A 3 to 10 lettered abbreviation for the service name. Example OTD for Open Term Deposit.
   All the above mentioned 3 parts are delimited by an underscore character.
   Example : TD_F_OTD where TD represents module name. F represents that its a financial transaction i.e. task type and OTD is the abbreviated
   form of the transaction(service) name.

Steps to register a task with OBDX:

The task code needs to be configured in the database table DIGX_CM_TASK. For example if we consider Open Term Deposit then the below
query fulfills the requirement mentioned in this step.

Insert into DIGX_CM_TASK (ID, NAME, PARENT_ID, EXECUTABLE, APPROVAL_SUPPORTED, LIMIT_REQUIRED, TASK_TYPE, MODULE_TYPE, CREATED_BY, CREATION_DATE, LAST_UPDATED_BY, LAST_UPDATED_DATE, OBJECT_STATUS, OBJECT_VERSION_NUMBER, WORKING_WINDOW_SUPPORTED, TFA_REQUIRED, BLACKOUT_SUPPORTED) values ('TD_F_OTD', 'New Deposit', 'TD_F', 'Y', 'Y', 'N', 'FINANCIAL_TRANSACTION', 'TD', 'ofssuser', sysdate, 'ofssuser', sysdate, null, 1, 'Y', 'N', 'Y');

As evident from the above query example Tasks have a hierarchy. Every task might have a parent task denoted by the task code value held
by the PARENT_ID column of DIGX_CM_TASK. In most of the cases its a 3 level hierarchy.

- Leaf level tasks to which services are mapped at the lowest level
- Task representing the module to which the service belongs at the mid level
- Task representing the task type at the root level

For instance consider the task code AP_N_CUG which represents the Usergroup creation service under module approvals(AP). So the
PARENT_ID column of task AP_N_CUG(leaf level task) has task code as AP(mid level task). If we look at the entry for task code
AP(mid level task) then the value in the PARENT_ID column of DIGX_CM_TASK has MT(root level task) which is the task code representing
task type ADMINISTRATION. The leaf level task has 'Y' as the value in its EXECUTABLE column. The mid level and root level tasks have 'N' as the value in its EXECUTABLE column.

- **Step 2.** Register the newly created service against this task.
  For this step firstly, you need to get the service id for your service(transaction). Service id is the fully qualified name of the
  class appended by the dot character (.) and the method name. For example taking open term deposit into consideration, the business
  logic for the service is encapsulated in the method named create of the service class com.ofss.digx.app.td.service.account.core.TermDeposit.
  Hence the service id is derived as
  : com.ofss.digx.app.td.service.account.core.TermDeposit.create
  Secondly the below query fulfills the requirement mentioned in this step.

  insert into DIGX_CM_RESOURCE_TASK_REL (ID, RESOURCE_NAME, TASK_ID, CREATED_BY, CREATION_DATE, LAST_UPDATED_BY, LAST_UPDATED_DATE, OBJECT_STATUS, OBJECT_VERSION_NUMBER) values ('1', 'com.ofss.digx.app.td.service.account.core.TermDeposit.create', 'TD_F_OTD', 'ofssuser', sysdate, 'ofssuser', sysdate, null,1);

The aforesaid procedure enrolls your newly created service as a task in OBDX.

Limit Configuration

The below procedure describes the steps required to enable Limits for a newly developed service.

A prerequisite to this configuration is that this newly developed service should be registered as a task in OBDX. Refer “Task Registration” section for further details.**The types of Limits supported by the system are:**

- Periodic Limit(Cumulative) : Limits that get reset after the expiration of a period. Example Daily-limits.
- Duration Limit(Cooling Period) : Limits that get applicable after the occurence of an event, for instance payee creation, and
  then are applicable for the specified duration after commencement of the event.
- Transaction Limit : Limits applicable to each invocation of a transaction. Holds minimum and maximum amount that can be transacted in a single transaction invocation.

Limits are applicable to targets. The types of targets supported by OBDX are Task and Payee.

1. Task : Any service developed as a part of OBDX and registered as a task as mentioned here TASK REGISTRATION
2. Payee : A payee resource created via Payee creation transaction in OBDX.

To enable limits for a service, rather for a task mapped to the service to be precise, we need to follow the below mentioned steps:

- Ensure that the LIMIT_REQUIRED column of the DIGX_CM_TASK table is updated as 'Y' for your task id.
- **Step 2.** Register taskEvaluatorFactory for your task code.
  This needs an insert in **DIGX_FW_CONFIG_ALL_B** table under the category_id '**taskEvaluatorFactories**' as shown below

  ```
  Insert into DIGX_FW_CONFIG_ALL_B (PROP_ID, CATEGORY_ID, PROP_VALUE,
  FACTORY_SHIPPED_FLAG, PROP_COMMENTS, SUMMARY_TEXT, CREATED_BY,
  CREATION_DATE, LAST_UPDATED_BY, LAST_UPDATED_DATE, OBJECT_STATUS,
  OBJECT_VERSION_NUMBER) values (<<taskcode>>, 'taskEvaluatorFactories',
  'com.ofss.digx.framework.task.evaluator.DefaultTaskEvaluatorFactory',  'N',
  null, 'Task Evaluator Factory for Mixed FT', 'ofssuser', sysdate, 'ofssuser', sysdate,
  'Y', 1);
  ```

3. Code a LimitDataEvaluator for the task. LimitDataEvaluator is a class that implements ILimitDataEvaluator interface present in com.ofss.digx.finlimit.core jar. This is a functional interface with a single method having signature as shown below :

```
/**
* provide {@link LimitData} of currently executing task.
*
* @param serviceInputs
* the service inputs
* @return {@link LimitData} required for limit utilization and validation
* @throws Exception
*/
public LimitData evaluate(List<Object> serviceInputs) throws Exception;
```

This method recieves a List<Object> as an input. This list has all the arguments that were passed to the newly coded service for
which limits needs to be enabled. For instance consider the service to open a termed deposit. Signature of the service is as
shown below.

public TermDepositAccountResponseDTO create(SessionContext sessionContext, TermDepositAccountDTO termDepositAccountDTO) throws Exception

In this case when the LimitDataEvaluator coded for open term deposit task i.e. TD_F_OTD is invoked by the OBDX framework, the serviceInputs argument of evaluate method will contain 2 objects in the list namely SessionContext and TermDepositAccountDTO. The return type of evaluate method is LimitData. The state of a LimitData object comprises of three variables:

- **currencyAmount :** an Object of type CurrencyAmount which represents the monetary amount involved in the ongoing transaction along with the currency in the transfer or payment is made.
- **payee** : An object of type PayeeDTO. Needs to be populated in case a payee is involved in the transaction.
- **limitTypesToBeValidated** : A list of LimitTypes. For all unexceptional practical purposes this needs to be populated as shown below:

  limitTypesToBeValidated = new ArrayList<LimitType>(Arrays.asList(LimitType.PERIODIC,LimitType.DURATION,LimitType.TRANSACTION));

  *These 3 fields in case applicable needs to be derived from the argument serviceInputs and populated in the returned LimitData
  object.*

- Register the LimitDataEvaluator coded in Step 3.

  This needs an insert in ***DIGX_FW_CONFIG_ALL_B*** *table under the category_id '**limitDataEvaluator**' as shown below* Insert into DIGX_FW_CONFIG_ALL_B (PROP_ID, CATEGORY_ID,PROP_VALUE, FACTORY_SHIPPED_FLAG, PROP_COMMENTS, SUMMARY_TEXT, CREATED_BY, CREATION_DATE,
  LAST_UPDATED_BY, LAST_UPDATED_DATE, OBJECT_STATUS, OBJECT_VERSION_NUMBER)
  values (<<task code>>, 'limitDataEvaluator', <<limitDataEvaluator>>, 'N',
  'Limit data evaluator for <<service name>> service', null, 'ofssuser', sysdate, 'ofssuser', sysdate, 'A', 1);
  In the above query <<task code>> *is the task code for the service,* <limitDataEvaluator>> *is the fully qualified name of the class coded in Step 3.* <<service name>> *is a descriptive name for the service.*

- **Step 5.** Code a TargetEvaluator for your task.

Note : This step is needed only if your task requires limits involving Payees. Example Duration Limits and payee limits.

Payee limits are Periodic and Transactional limits applied on a Payee.

TargetEvaluator is a class that implements ITargetEvaluator interface. ITargetEvaluator is a functional interface that has only 1 method as shown below :

```
/**
* Evaluates the Target details for the given evaluated task code and service inputs in the form of
* {@link TargetDTO}.
*
* @param evaluatedTaskCode
* the given evaluated task code
* @param serviceInputs
* inputs of the service using this evaluator
* @return target details of the target for this task code and service inputs in the form of {@link
TargetDTO}.
* @throws Exception
* exception while evaluating {@link TargetDTO}
*/
public TargetDTO evaluate(String evaluatedTaskCode, List<Object> serviceInputs) throws
Exception;
```

This method accepts the task code and serviceInputs in case something needs to be derived from the arguments passed to the
service.

It returns a TargetDTO. TargetDTO has an id, name, value and TargetTypeDTO. TargetType tells whether the target is of type task or payee. If the TargetType is TASK then the variable value of TargetDTO holds the task code for the service.

If the TargetType is PAYEE then the variable value of TargetDTO holds the payeeId of the payee involved in the service.

As this step is required only for limits pertaining to payees so TargetType will be PAYEE and targetDTO's value will be payeeId.

Register the TargetEvaluator coded in Step 5.

Note: This step is needed only if your task requires limits involving Payees. Example Duration Limits and payee limts.

Payee limits are Periodic and Transactional limits applied on a Payee.

This needs an insert in **DIGX_FL_TARGET_EVALUATOR** table as shown below:

Insert into DIGX_FL_TARGET_EVALUATOR (TASK_CODE, TARGET_TYPE, EVALUATOR, PROP_COMMENTS, SUMMARY_TEXT, CREATED_BY, CREATION_DATE, LAST_UPDATED_BY, LAST_UPDATED_DATE, OBJECT_STATUS, OBJECT_VERSION_NUMBER) values (<<task code>>, 'PAYEE', <<TargetEvaluator>>, null, 'target evaluator for <<service name>> service', 'ofssuser', sysdate, 'ofssuser', sysdate, 'Y', 1);

In the above query <<*task code*>> is the task code for the service, <<*TargetEvaluator*>> is the fully qualified name of the
class coded in Step 5. <<*service name*>> is a descriptive name for the service.

The aforesaid procedure enables limits for a task in OBDX.

Approval Configuration

The below procedure describes the steps required to enable Approvals for a newly developed service.

A prequisite to this configuration is that this newly developed service should be registered as a task in OBDX. Refer "Task Registration" section for further details.

To enable approvals for a service, rather for a task mapped to the service to be precise, we need to follow the below mentioned steps:

- Ensure that the APPROVAL_SUPPORTED column of the DIGX_CM_TASK table is updated as 'Y' for your task id.
- Note : If the newly created task is of type ADMINISTRATION and the maintenance is not specific to a party then this step is not
  required. Examples of such transaction are 2 Factor Authentication maintenance, limit maintenance and limit package maintenance.
  Tasks of type ADMINISTRATION which are specific to a party like Rule management tasks, workflow management tasks etc require this step.
  Tasks of type FINANCIAL_TRANSACTION,NONFINANCIAL_TRANSACTION,MAINTENANCE,INQUIRY and COMMON require this step.

  Code an approval assembler for the new task. An approval assembler is a class that extends AbstractApprovalAssembler.

  There are 4 methods in abstract approval assembler out of which the one with the below signature:

  protected T toDomainObject(D requestDTO, T transaction) throws Exception;
  will encapsulate the logic required to populate Transaction domain which is used by approvals framework.

  Rest of the methods need to be overridden with empty or null implementations.
  As evident from the signature quoted above this method accepts a requestDTO(an object that **IS A** DataTransferObject) and a transaction(an object that **IS A** Transaction).
  requestDTO is the same DataTransferObject that was passed to your newly created service.
  For instance consider the service to open a termed deposit. Signature of the service is as shown below.

  public TermDepositAccountResponseDTO create(SessionContext sessionContext, TermDepositAccountDTO termDepositAccountDTO) throws Exception

  In this case when the ApprovalAssembler coded for open term deposit task i.e. TD_F_OTD is invoked by the OBDX framework, the requestDTO argument of toDomainObject method will be the same as termDepositAccountDTO.

This method populates the transaction object on the basis of the requestDTO and returns the transaction domain. The guidelines
to override this method are as follows:-

- **Instantiation :**
  The transaction object passed will be null and needs to be instantiated. If the task type of the newly created service is FINANCIAL_TRANSACTION then the transaction needs to be instantiated as an object of AmountAccountTransaction.

  transaction = new AmountAccountTransaction();

  If the task type of the newly created service is NONFINANCIAL_TRANSACTION then the transaction needs to be instantiated as an
  object of AccountTransaction.

  transaction = new AccountTransaction();

  If the task type of the newly created service is MAINTENANCE then the transaction needs to be instantiated as an object of PartyTransaction.

  transaction = new PartyTransaction();

  If the task is of type ADMINISTRATION and the maintenance is specific to a party then the transaction needs to be instantiated as an object of PartyTransaction.

  transaction = new PartyTransaction();

  If the task is of type ADMINISTRATION and the maintenance is not specific to a party then the transaction needs to be instantiated
  as an object of Transaction.

  transaction = new Transaction();

- **Call to AbstractApprovalAssembler :**

  Call
  transaction = super.toDomainObject(requestDTO, transaction);

  This populates the generic state of transaction domain which does not change with the task for which approvals is being configured.
  c. Populate the state of the transaction domain which is specific to the task for which approvals is being configured. Cast the

requestDTO to the type being accepted by the service. For example cast it to TermDepositAccountDTO as per the aforesaid example.
Use this DTO to populate the service specific state of the transaction domain like amount, account etc.

- **Step 3.** Register an approval assembler for your service or task.
  To register an approval assembler for your service an entry needs to be made in the database table *DIGX_FW_CONFIG_ALL_*B with
  the value of column *CATEGORY_ID* as 'approval_assembler'.

  If the newly created task is of type *ADMINISTRATION* and the maintenance is not specific to a party then the approval assembler to be registered against your service is
  om.ofss.digx.framework.domain.transaction.assembler.GenericDTOTransactionAssembler
  2 Factor Authentication Maintenance is a fine example of such transactions. The service id for this transaction is com.ofss.digx.app.security.service.authentication.maintenance. AuthenticationMaintenance.

  create
  The below query fulfills the requirement of this step:

```
Insert into DIGX_FW_CONFIG_ALL_B
(PROP_ID,
CATEGORY_ID,
PROP_VALUE,
FACTORY_SHIPPED_FLAG,
PROP_COMMENTS,
SUMMARY_TEXT,
CREATED_BY,
CREATION_DATE,
LAST_UPDATED_BY,
LAST_UPDATED_DATE,
OBJECT_STATUS,
OBJECT_VERSION_NUMBER)
values
('com.ofss.digx.app.security.service.authentication.maintenance.AuthenticationMaintenance.
create',
'approval_assembler',
'com.ofss.digx.framework.domain.transaction.assembler.GenericDTOTransactionAssembler'
,
'N',
'assembler class for conversion from UserSegmentTFAMaintenanceDTO to Transaction
domain',
'assembler class for conversion from UserSegmentTFAMaintenanceDTO to Transaction
domain',
'ofssuser',
sysdate,
'ofssuser',
sysdate,
'A',
1);
```

  In all other cases where you have implemented a custom approval assembler as per the guidelines in step 2, the fully qualified

class name of that approval assembler will be registered against your service. The below query fulfills the requirement of
this step:

```
Insert into DIGX_FW_CONFIG_ALL_B
(PROP_ID,
CATEGORY_ID,
PROP_VALUE,
FACTORY_SHIPPED_FLAG,
PROP_COMMENTS,
SUMMARY_TEXT,
CREATED_BY,
CREATION_DATE,
LAST_UPDATED_BY,
LAST_UPDATED_DATE,
OBJECT_STATUS,
OBJECT_VERSION_NUMBER)
values
(<<service id>>,
'approval_assembler',
<<ApprovalAssembler>>,
'N',
'assembler class for conversion from DataTransferObject to Transaction domain',
'assembler class for conversion from DataTransferObject to Transaction domain',
'ofssuser',
sysdate,
'ofssuser',
sysdate,
'A',
1);
```

In the above query <<service id>> is the fully qualified name of the class appended by the dot character (.) and the method
name. <<ApprovalAssembler>> denotes the fully qualified class name of the approval assembler coded in Step 2.

The aforesaid procedure enables approvals for a task in OBDX.

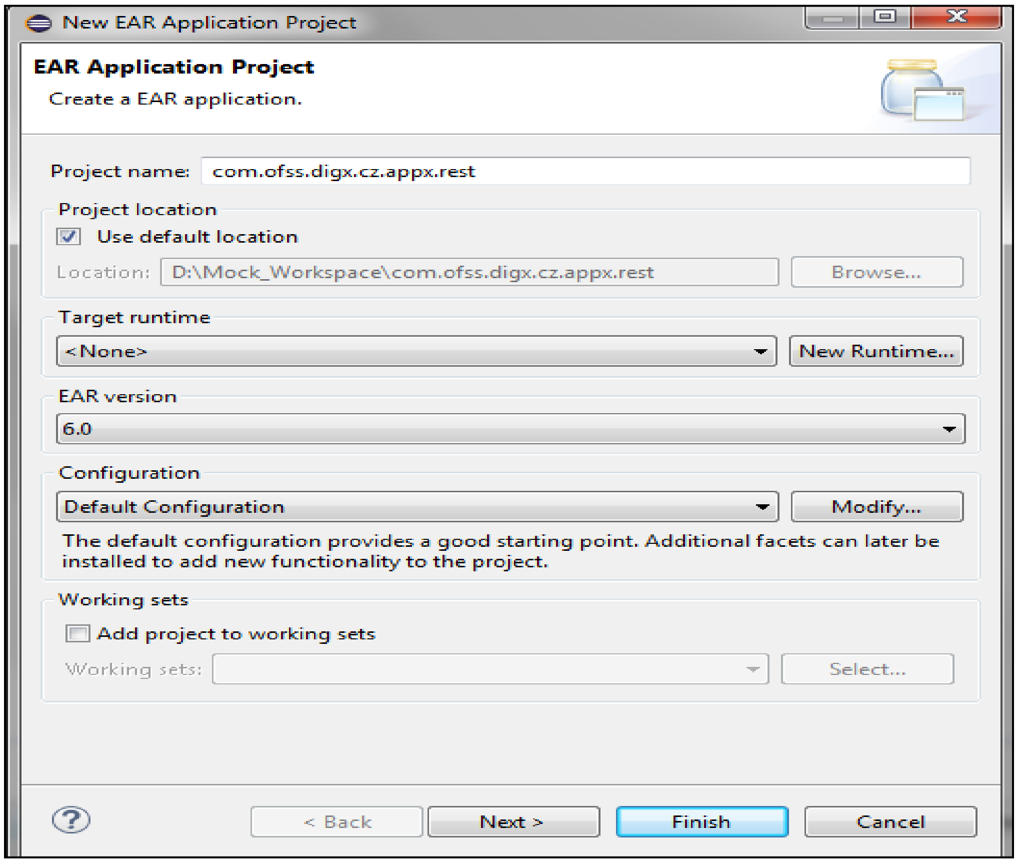
Creating EARs:

Once all the classes are created and implemented, generate the required ear deployments. The following two EARs need to be created: **com.ofss.digx.cz.appx.service.rest.ear** and **obdx.app.cz.domain.ear**.

To generate an EAR in eclipse, we need to create an Enterprise Application Project and include the required project/s during the creation. I am adding screenshot for creating **com.ofss.digx.cz.appx.service.rest.ear** , similar changes can be made for second EAR as well.

Steps to generate EAR in Eclipse:

Create an Enterprise Application Project (EAP) from the “New” option. Give details as mentioned in the Figure 3 and click on Next button


New EAR Application Project

EAR Application Project

Create a EAR application.

Project name: com.ofss.digx.cz.appx.rest

Project location

☑ Use default location

Location: D:\Mock_Workspace\com.ofss.digx.cz.appx.rest

Target runtime: <None>

EAR version: 6.0

Configuration: Default Configuration

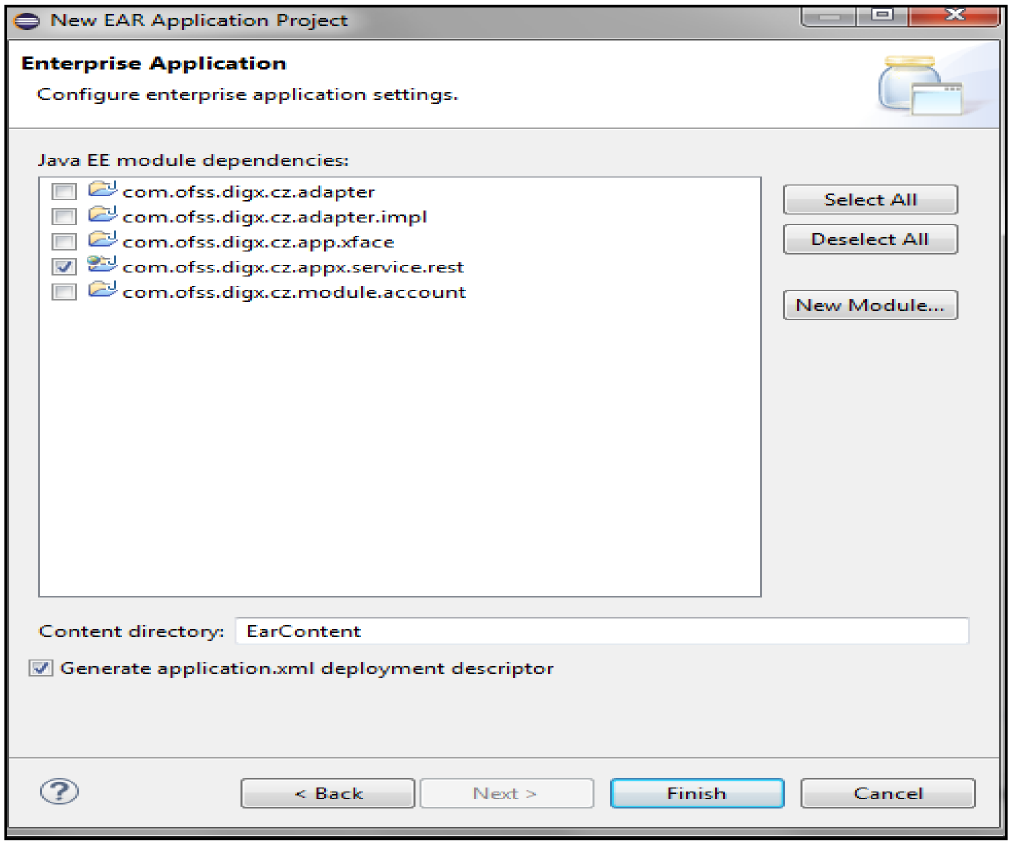
The default configuration provides a good starting point. Additional facets can later be installed to add new functionality to the project.

Working sets

☐ Add project to working sets

Working sets:

Figure 3 Creating new EAR Application Project


New EAR Application Project

Enterprise Application

Configure enterprise application settings.

Java EE module dependencies:

- [ ] com.ofss.digx.cz.adapter
- [ ] com.ofss.digx.cz.adapter.impl
- [ ] com.ofss.digx.cz.app.xface
- [✓] com.ofss.digx.cz.appx.service.rest
- [ ] com.ofss.digx.cz.module.account

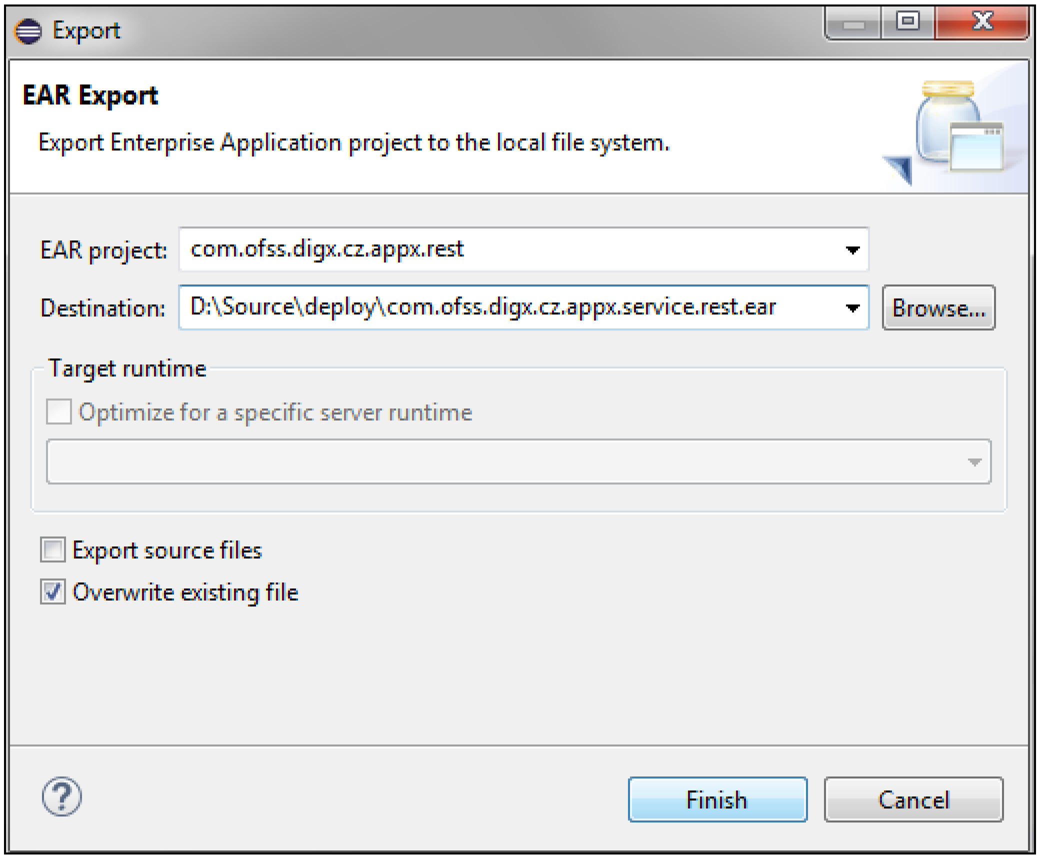
Content directory: EarContent

[✓] Generate application.xml deployment descriptor

Figure 4 Creating new EAR Application Project

ii. On the next screen, select **com.ofss.digx.cz.appx.service.rest project** and check on Generate application.xml deployment option as shown in the Figure 4. Click on Finish button after that.

Once the EAP is created, right click on that project and select export EAR option in that. Give the EAR name as ‘**com.ofss.digx.cz.appx.service.rest.ear’** with the complete destination path. Please refer the Figure 5 for more details:



Export Enterprise Application project to the local file system.

EAR project: com.ofss.digx.cz.appx.rest

Destination: D:\Source\deploy\com.ofss.digx.cz.appx.service.rest.ear

Target runtime

☐ Optimize for a specific server runtime

☐ Export source files

☑ Overwrite existing file

Figure 5 Exporting EAR to the destination path

The EAR will be successfully created at the mentioned destination and similar steps can be performed to create the second EAR - **obdx.app.cz.domain.ear.**

Note: The only difference will be the projects to be included (in this case, include the rest of the Java projects - which were not selected while creating the first EAR in the Figure 4) and keep the checkbox unchecked for Generate application.xml deployment option in the Figure 4. Once these two EARs are generated, go through the following changes in the web.xml, weblogic.xml, application.xml, and weblogic-application.xml files of the EARs

Adding web.xml:

Add the web.xml file from the **com.ofss.digx.appx.service.rest.ear** to the custom REST ear **com.ofss.digx.cz.appx.service.rest.ear.** This xml file can be found at the same location: com.ofss.digx.cz.appx.service.rest.ear / com.ofss.digx.cz.appx.service.rest.war / WEB-INF/

No changes need to be made in this file.

Changes in weblogic-application.xml:

Please copy weblogic-application.xml file from the **com.ofss.digx.appx.service.rest.ear** present in the OBDX_Installer folder to the custom REST EAR's META-INF folder. This file can be found at the location: com.ofss.digx.appx.service.rest.ear / META-INF of the **com.ofss.digx.appx.service.rest.ear**

Once it is copied, make the following changes in the weblogic-application.xml file:

Add <wls:library-ref><wls:library-name>obdx.app.cz.domain</wls:library-name></wls:library-ref> entry after the library reference of obdx.app.core.domain.

This ensures that along with the existing libraries, the newly created custom domain ear **obdx.app.cz.domain.ear** can be referenced by this custom REST application EAR.

Changes in weblogic.xml:

Please Copy weblogic.xml file from the **com.ofss.digx.appx.service.rest.ear** present in the OBDX_Installer folder to the newly generated custom REST EAR's WEB-INF folder. The exact path of weblogic.xml file is:

com.ofss.digx.cz.appx.service.rest.ear / com.ofss.digx.cz.appx.service.rest.war / WEB-INF

Now open weblogic.xml and in the <wls:context-root> option, update the value from digx to 'digx/cz'. The context root is changed and therefore the prefix url to all REST resources in this EAR will become 'digx/cz/v1/'

Changes in application.xml file:

Update application.xml file present in **com.ofss.digx.cz.appx.service.rest.ear**. Make changes in the <contex-root> entry and update it to 'digx/cz'.

Please copy application.xml file from the **obdx.app.domain.ear** present in the OBDX_Installer folder to the **obdx.app.cz.domain.ear** META-INF folder. Also, you will need to copy empty.jar present in the **obdx.app.domain.ear.** Now open this application.xml recently copied in the custom domain ear and change the <display-name> entry to 'obdx.app.cz.domain' .

Note: AbstractRESTApplication.jar needs to be shipped along with the supporting deployable components of OBDX application. Once it is found in OBDX_Installer folder, add this jar to the following path of custom REST ear:

com.ofss.digx.cz.appx.service.rest.ear/com.ofss.digx.cz.appx.service.rest.war/WEB-INF/lib/

After this step, both the EARs are ready to be deployed on the Weblogic server.

1. **Deploying application in Weblogic:**

The two EARs created in the previous step should be deployed in the existing deployment setup. Please deploy the **com.ofss.digx.cz.appx.service.rest.ear** as an application and **obdx.app.cz.domain.ear** as library.

2. **Test the application:**

Once the application is up, please go to the deployments section of the Weblogic Server. In the control option, you'll find the option to test the application. Just to verify, check whether the context-root of the custom application is changed to digx/cz. The request URL for testing this application will be –s

http://<hostname>:<port>/digx/cz/v1/application.wadl

Location for libraries in the installer folder:

Root_Location: OBDX_17.1.0.0.0/OBDX_Installer/installables/app/components/obdx/deploy/

DIGX_LIB	Root_Location/obdx.app.domain.ear/APP-INF/lib/
EXT_LIB	Root_Location/obdx.thirdparty.app.domain.ear/APP-INF/lib/
OBP_LIB	Root_Location/obdx.app.core.domain.ear/APP-INF/lib/