SADAD EBPP Portal Project Revamp

Project Setup and Development Document

Table of Contents

[Portal Project Introduction 6](#_Toc7960220)

[SADAD Portlet Project 6](#_Toc7960221)

[EBPP Portlets 6](#_Toc7960222)

[SADAD Admin Portlets 6](#_Toc7960223)

[SADAD Service Client Project 7](#_Toc7960224)

[Project SVN Location 7](#_Toc7960225)

[Project Setup 7](#_Toc7960226)

[Assumption 7](#_Toc7960227)

[Setup Maven Portal Dependencies & Development Environment. 7](#_Toc7960228)

[POM Files for Maven Portal Setup 7](#_Toc7960229)

[Project Setup in RAD 9](#_Toc7960230)

[Maven Build and Deployment using RAD 11](#_Toc7960231)

[Project Development 12](#_Toc7960232)

[Used Technologies, Libraries and Framework 12](#_Toc7960233)

[Common Architecture and Pattern 12](#_Toc7960234)

[Developing Portlet 13](#_Toc7960235)

[JSP Screen Formation 13](#_Toc7960236)

[HTTP Data Flow from Front-end to Back-end 15](#_Toc7960237)

[Java Portlet Classes 16](#_Toc7960238)

[Sequence Flow of Events and Portal Response 19](#_Toc7960239)

[Loading Static Resource Dynamically 20](#_Toc7960240)

[JSP Declaration Explanation 20](#_Toc7960241)

[JSP Load Static Resources 22](#_Toc7960242)

[Java Portlet and Custom Class Code Explanation 24](#_Toc7960243)

[Case of Render Request 24](#_Toc7960244)

[Case of Action Request 24](#_Toc7960245)

[Case of Resource Request 25](#_Toc7960246)

[Populating Request Parameters in Java Bean Dynamically 26](#_Toc7960247)

[Calling Backend Service Operation Dynamically 27](#_Toc7960248)

[Frontend Screen Navigation 27](#_Toc7960249)

[Pushing and Popping Screens into ArrayDeque 28](#_Toc7960250)

[JavaScript Function Explanation 29](#_Toc7960251)

[HTML Form Submission via JS Function 29](#_Toc7960252)

[HTTP Post Call via JS Function 30](#_Toc7960253)

[Application Level Cache 31](#_Toc7960254)

[Developing Web Service Client 32](#_Toc7960255)

[Generate Client using JAX-WS 32](#_Toc7960256)

[Custom Code for Web Service Client Project 34](#_Toc7960257)

[Service Endpoint Definitions 37](#_Toc7960258)

[Reference 39](#_Toc7960259)

List of Figures

[Figure 1 – RAD Maven Configuration for WAS Plugin 8](#_Toc7960173)

[Figure 2 – RAD Maven Configuration for Portal Plugin 8](#_Toc7960174)

[Figure 3 – RAD SVN Repository Perspective 9](#_Toc7960175)

[Figure 4 – RAD Java Configure Maven Project 9](#_Toc7960176)

[Figure 5 – RAD Project Facets for SADAD Portlet Project 10](#_Toc7960177)

[Figure 6 – RAD New Project Wizard 10](#_Toc7960178)

[Figure 7 – RAD Maven Build Configuration Dialog 11](#_Toc7960179)

[Figure 8 – Common Architecture of Project 12](#_Toc7960180)

[Figure 9 – Query Account Portlet Screen, show container 1, 2 and message container 13](#_Toc7960181)

[Figure 10 – JSP INDEX Page Formation 14](#_Toc7960182)

[Figure 11 – INDEX.JSP Source Code 14](#_Toc7960183)

[Figure 12 – High Level Request Data Flow 15](#_Toc7960184)

[Figure 13 – Data / Call Flow 15](#_Toc7960185)

[Figure 14 – JSP Code Snippet, HTML Form using Resource URL with JSP filename in action attribute 16](#_Toc7960186)

[Figure 15 – JSP Code Snippet, HTML HREF using Resource URL with JSP filename 16](#_Toc7960187)

[Figure 16 – HTTP Post Request Showing Form Data Posting 16](#_Toc7960188)

[Figure 17 – Portlet Class Diagram 17](#_Toc7960189)

[Figure 18 – SessionDataBean Class Diagram 18](#_Toc7960190)

[Figure 19 – ServiceCallHelper Class Diagram 18](#_Toc7960191)

[Figure 20 – INDEX.JSP Source Code 20](#_Toc7960192)

[Figure 21 – JSP Fragment for Variable Declaration 20](#_Toc7960193)

[Figure 22 – Line 25 from JSPDeclaration.jspf File 21](#_Toc7960194)

[Figure 23 – Portlet Deployment Descriptor Showing Resource Bundle for Selected Portlet 21](#_Toc7960195)

[Figure 24 – Portlet Resource Bundle Showing Defined Properties 22](#_Toc7960196)

[Figure 25 – JSP Fragment for loading static resource 22](#_Toc7960197)

[Figure 26 – Line 9 from JSPIncludeStaticFiles.jspf 23](#_Toc7960198)

[Figure 27 – Static files directory showing JavaScript files for Portlet Modules 23](#_Toc7960199)

[Figure 28 – Code Snippet of doView method 24](#_Toc7960200)

[Figure 29 – Code Snippet of processAction method 24](#_Toc7960201)

[Figure 30 – Code Snippet of serveResource method 25](#_Toc7960202)

[Figure 31 – Code Snippet of populateRequestParamsInSessionBean method 26](#_Toc7960203)

[Figure 32 – Code Snippet callRequiredBackEndServiceOperation of method 27](#_Toc7960204)

[Figure 33 – Code Snippet of screenHandling method 27](#_Toc7960205)

[Figure 34 – Code Snippet of navigate method 28](#_Toc7960206)

[Figure 35 – Code Snippet of doQueryFormSubmission 29](#_Toc7960207)

[Figure 36 – Code Snippet of doPostUrl 30](#_Toc7960208)

[Figure 37 – Generate Client for Web Services 32](#_Toc7960209)

[Figure 38 – Web Service Client Window Service Definition 33](#_Toc7960210)

[Figure 39 – Web Service Client Window Output Folder 34](#_Toc7960211)

[Figure 40 – Method getInstance for return of ServiceDelegateImpl instance. 35](#_Toc7960212)

[Figure 41 – Abstract methods defined in ServiceDelegateFactory Interface. 35](#_Toc7960213)

[Figure 42 – Implementation of Abstract methods in ServiceDelegateFactoryImpl class. 36](#_Toc7960214)

[Figure 43 – WSDL Account Query Operations 36](#_Toc7960215)

[Figure 44 – Delegate interface for Account Query Operations 36](#_Toc7960216)

[Figure 45 – Delegate Implementation for Account Query 37](#_Toc7960217)

[Figure 46 – SadadServicesEndPointUrls reading URLs from SadadServiceEndPointUrls properties file 38](#_Toc7960218)

[Figure 47 – SadadServicesEndPointUrls Properties File 38](#_Toc7960219)

# Portal Project Introduction

SADAD Portal Portlet project has two artifacts.

1. ***SadadPortlets*** (Portlet Project – Java WAR)
2. ***SadadServiceClient*** (Service Client Project – Java JAR)

## SADAD Portlet Project

SADAD Portlet project is developed for SADAD Internal users as well as for SADAD Partners (Banks and Biller). The project has mainly two categories of portlets, which are as follows.

1. EBPP Portlets
2. SADAD Admin Portlets

### EBPP Portlets

The EBPP Portlets gives SADAD Internal users and SADAD Partner users to query their uploaded data in SADAD system. EBPP Portlets have following five main segments, each of which contains one or two portlets for query purpose.

1. Account
   1. Query Account
   2. Rejected Account
2. Bill
   1. Query Bill
   2. Rejected Bill
3. Customer
   1. Query Customer
4. Payment
   1. Query Payment
   2. Rejected Payment
5. Refund
   1. Query Refund

### SADAD Admin Portlets

1. SADAD
   1. SADAD Admin
2. Banks
   1. Branch Code
   2. District Code
3. Billers
   1. Bill Category
   2. Payment Type
   3. Service Type
   4. Access Channel

## SADAD Service Client Project

SADAD Service Client project is Java JAR file project which contains mainly the auto-generated Java classes from the WSDL files using RAD, Web Services 🡪 Generate Client.

Besides auto-generated classes and packages the following two packages have custom code for actual backend service call delegate.

* com.sadad.ebpp.portal.delegate.factory
* com.sadad.ebpp.portal.delegate.factory.clients

Further explanation regarding the development pattern and classes within the above two packages will comes later in detail section.

# Project SVN Location

The project source code resides in SADAD SVN repository at the following locations.

* <http://teamforge.sadad.com/svn/repos/sadad2/Trunk/Portal/SadadPortlets>
* <http://teamforge.sadad.com/svn/repos/sadad2/Trunk/Portal/SadadServiceClient>

# Project Setup

## Assumption

The assumption for this document is that RAD (Rational Application Developer) and WebSphere Portal (version 8.5 or 9.0) are already installed on user machine.

## Setup Maven Portal Dependencies & Development Environment.

1. Copy WAS\_pom.xml into a directory from where you can easily run mvn install command.
2. Rename WAS\_pom.xml to pom.xml.
3. Open the Command Prompt in the same directory where you copy the pom file.
4. Type the following command.
   * mvn install –D*serverInstallationFolder={D:\IBM\Portal\_80\WAS}*

*Change the highlighted value as per your Portal installation location.*

Repeat the above steps for Portal\_pom.xml to install Portal dependencies in your maven repository.

Same can be achieved via RAD Run Maven Configuration instead of mvn command line.

### POM Files for Maven Portal Setup

Attached are the files required to setup Maven based Portal development.



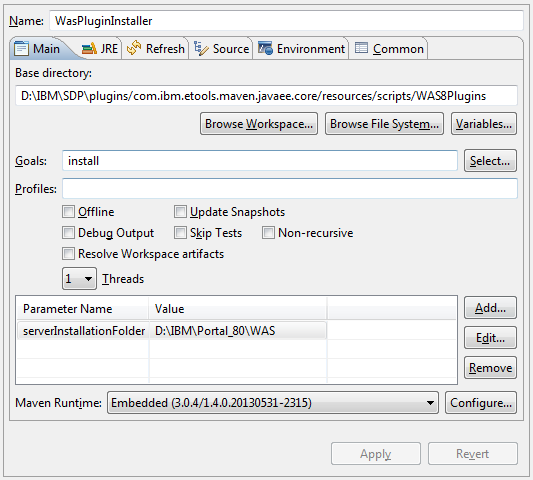


Figure 1 – RAD Maven Configuration for WAS Plugin

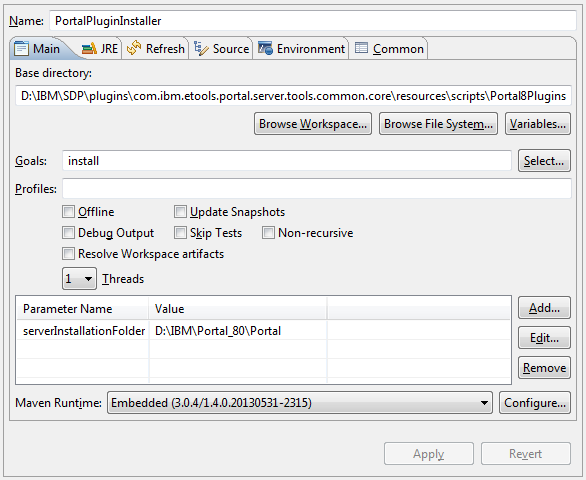


Figure 2 – RAD Maven Configuration for Portal Plugin

Once the WAS and Portal pom ran as Maven build successfully, edit the following file in your local maven repository. {LOCAL\_MAVEN\_REPO}\com\ibm\tools\target\was\8.0.0\was-8.0.0.pom.xml, and add the following as a dependency



## Project Setup in RAD

In Rational Application Developer (RAD), setup workspace by importing projects into workspace, RAD should have SVN Repository plugin installed. Open SVN Repository Exploring perspective and select the desired project (***SadadPortlet, SadadServiceClient***) to checkout, accept all default and click Finish.

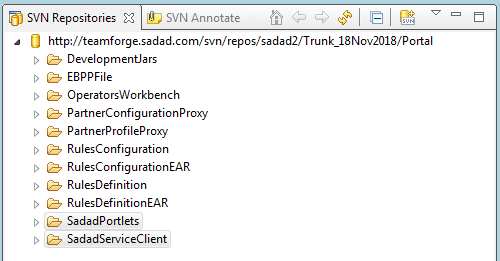


Figure 3 – RAD SVN Repository Perspective

Switch to JavaEE perspective, select both (***SadadPortlet, SadadServiceClient***) project right click, go-to Configure 🡪 Convert to Maven Project

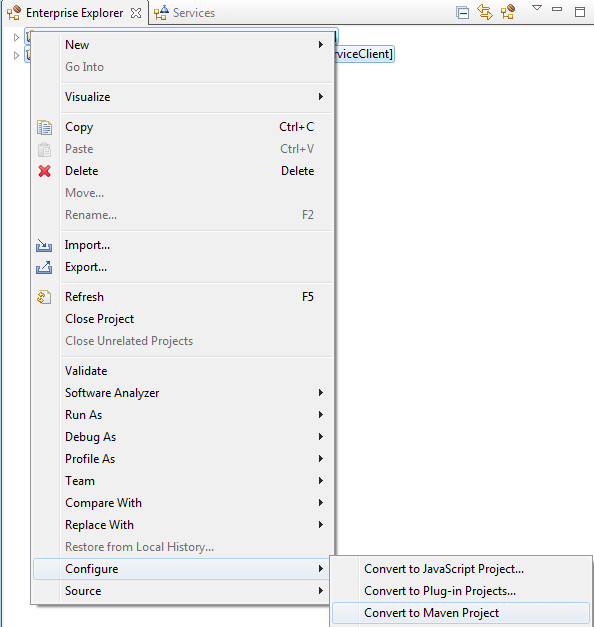


Figure 4 – RAD Java Configure Maven Project

After conversion to Maven, ***SadadPortlet*** project requires some tweaks in Project Facets. Right click to the ***SadadPortlet*** project, select Properties 🡪 Project Facets.

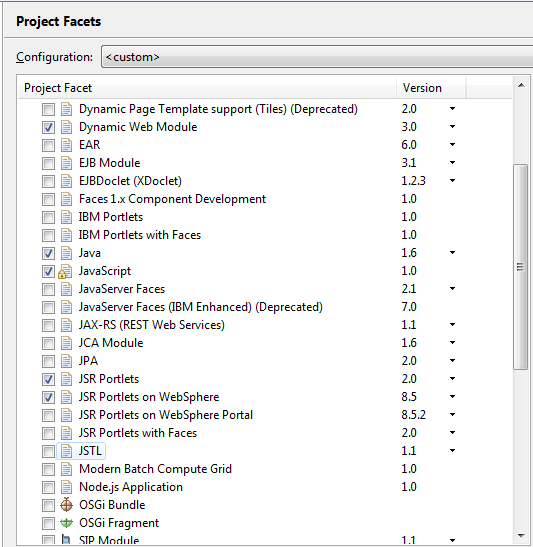
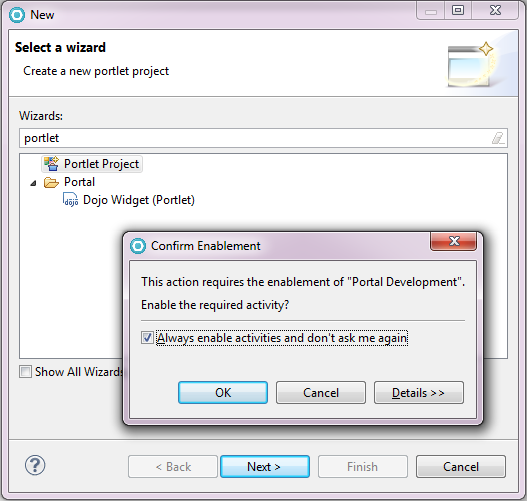


Figure 5 – RAD Project Facets for SADAD Portlet Project

***SadadServiceClient*** project setup is done and do not requires any change in Project Facets window.

In case if the current workspace is new and it does not have Portal / Portlet development functionality enabled, then open a New project wizard and select Portlet Project, if Portal Development functionality is not enabled then it will be enabled.

To check the Portal Development functionality in RAD open Portlet Deployment Descriptor file and it should opened in its own editor instead of any text or xml editor.

Figure 6 – RAD New Project Wizard

### Maven Build and Deployment using RAD

Right click on ***SadadServiceClient*** project, select Run As 🡪 Maven build, edit the configuration as below snapshot, for base directory click Browse Wrokspace and select the SadadService project. Type ***clean package install*** in Goals field, select ***Skip Test*** and click on Apply button to save this Maven configuration.

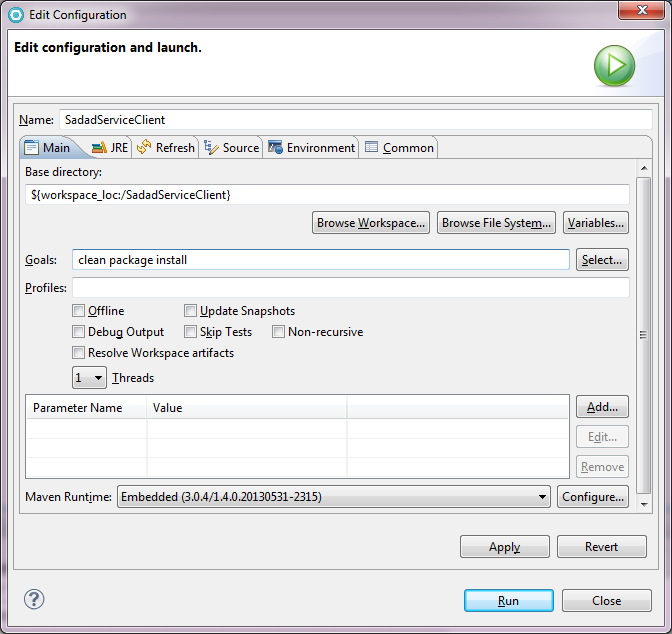


Figure 7 – RAD Maven Build Configuration Dialog

Repeat the same steps for ***SadadPortlet*** project, except in Goal field type ***clean package*** only as ***install*** is not required here, because this project is not defined as dependency for any other project.

# Project Development

Every portlet in this project has at least two JSP files, with one Portlet class having reference to one helper class and one session data bean. The helper class is used to call the backend services for retrieving and manipulating data, and Session Data Bean classes are POJO classes which are used to render data on JSP, and the data mapping between HTML forms fields and server classes.

## Used Technologies, Libraries and Framework

The project is developed using Java JSR-286 API without using any other Java framework, for the front-end development the HTML is from the S1-Portal screens, with some clean-up of HTML code mixed with custom JavaScript code using jQuery library, jQuery DataTable plugin component also used for rendering the data list.

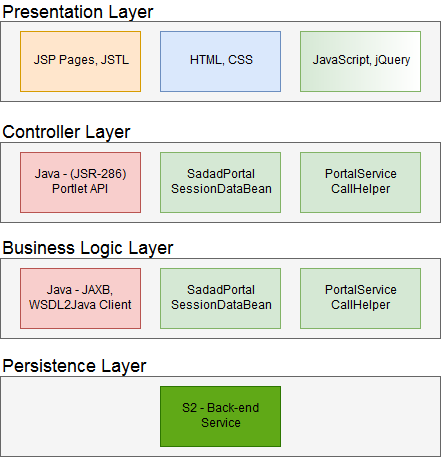


Figure 8 – Common Architecture of Project

## Common Architecture and Pattern

The common architecture and pattern used to develop this project are simply layered architecture, front-end is developed using HTML, CSS, and JavaScript with JSP, JSTL which used to capture the user input and pass it to the server for processing. The controller layer is Java Portlet 2.0 (JSR-286) API for receiving incoming user request and process it with the help of Data Session Bean (which is POJO class map the user requested data) and Service Call Helper (which is delegate class to call the backend services to fetch the data from database).

## Developing Portlet

To understand the project development style and pattern, take an example of a straightforward portlet and explore all the scenarios for portlet operations. Let's take Query Account Portlet as an example. The below snapshot shows the Query Account Portlet screen after the user inquired one account.

Using the form rendered in container 1, a user can select a biller and enter an account number to query an existing account in the system. The back-end system responds with the user account's details, the summary of the inquired account displayed in the second container. Usually summary or list has a link to the details page, which rendered in the container 1 with a back button.

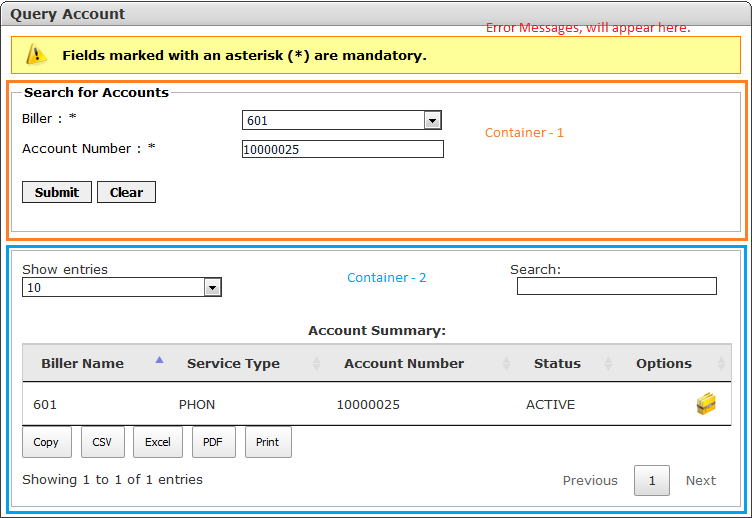


Figure 9 – Query Account Portlet Screen, show container 1, 2 and message container

## JSP Screen Formation

The portlet screen is formed using multiple JSP files, the main JSP file which used for rendering is ***<PortletName>***Index.jsp (referred as INDEX.JSP onwards). INDEX.JSP contains three placeholders. The very first placeholder is used to display error messages — the second container which is container 1 used to display HTML Forms for user input or Details about the inquired data, last container which is the container 2 used to display the Summary or the List of data user inquired. Figure 10 will clarify this paragraph, further explanation for INDEX.JSP is there in section [loading static resources dynamically](#_Loading_Static_Resource).

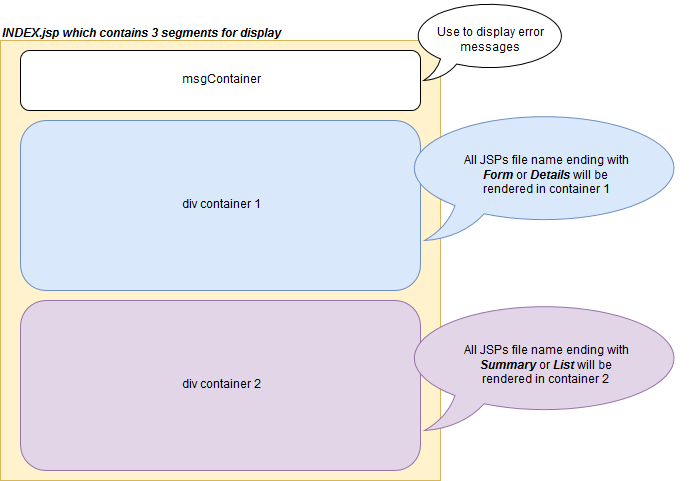


Figure 10 – JSP INDEX Page Formation

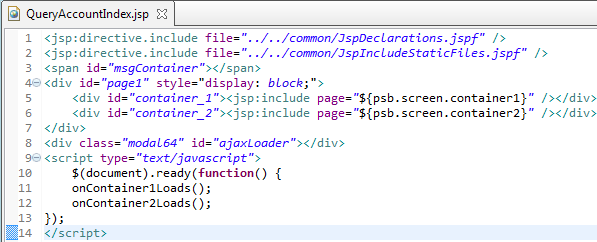


Figure 11 – INDEX.JSP Source Code

INDEX.JSP file source code is same across all the portlets, the reason not to use a single file instead of using portlet's owned INDEX.JSP is to give room for any changes in any module. INDEX.JSP is loading static resources like JavaScript and CSS files for details how it is loading static resources see the section [loading static resources dynamically](#_Loading_Static_Resource).

## HTTP Data Flow from Front-end to Back-end

All the HTTP Get calls for portlets are *RenderRequest* which are served by the *doView* method of the portlet class. Most of the HTTP Post calls from portlet screen are *ResourceRequest* which are served by the *serveResource* method of the portlet class, only a few calls like back button or finish button HTTP Post calls are *ActionRequest* and served by the *processAction* method of the portlet class.

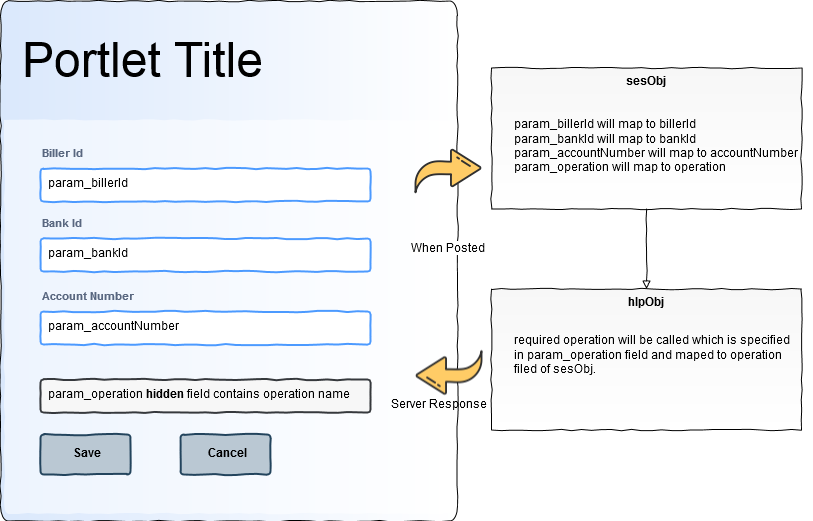


Figure 12 – High Level Request Data Flow

The *serveResource* method from the portlet class, calls the *populateRequestParamsInSessionBean* and *callRequiredBackEndServiceOperations* to pass the user input data to the backend service parameter and from backend service response to front-end data placeholders on a web page.

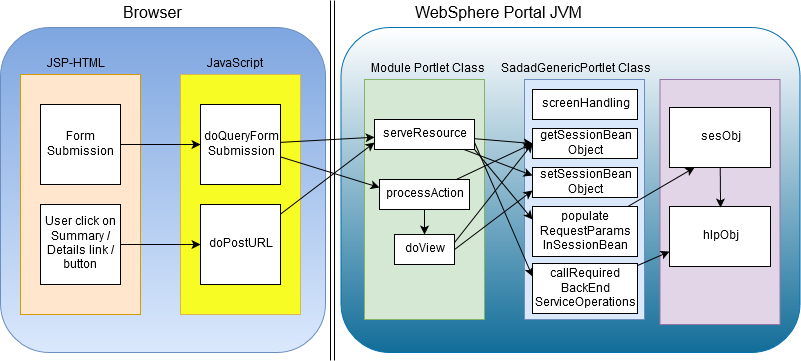


Figure 13 – Data / Call Flow

When calling HTTP Post call using *ResourceRequest* the resource id is always the name of JSP file which rendered on the web page. In Figure 14 JSP code snippet demonstrates the usage of portlet resource URL, and the ID attribute's value is the name of the JSP file which server sends in response. In Figure 15 JSP code snippet the portlet resource URL stored in a variable, and the ID attribute's values here is also the name of JSP file which server sends in response. Some additional parameters required to pass on to the server which is part of the resource URL, the naming convention of the parameter is important as this will automatically map to the data beans. In last the variable is used to call the resource URL and passed to the *doPostUrl* JavaScript method.



Figure 14 – JSP Code Snippet, HTML Form using Resource URL with JSP filename in action attribute

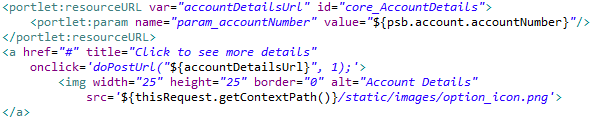


Figure 15 – JSP Code Snippet, HTML HREF using Resource URL with JSP filename

The naming convention used to send request parameters to the server should start with *param\_* followed by property name of session data bean class. For example, *billerId* property map to *param\_billerId*, *bankId* property map to *param\_bankId*, *paymentType* property map to *param\_paymentType* and so on, *param\_<propertyName>* contains the data required to call any backend service operation whereas *param\_operation* is the name of the backend service operation which is called upon the request if *param\_operation* is present.

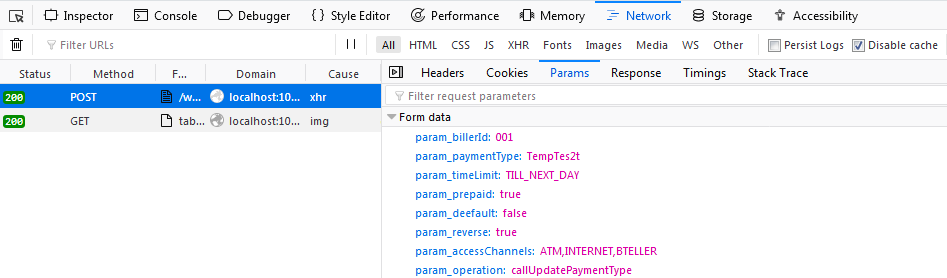


Figure 16 – HTTP Post Request Showing Form Data Posting

## Java Portlet Classes

Each portlet in the project has one portlet class which works as a controller. Each portlet class is an extension of *SadadGenericPortlet* class, and each portlet class contains an instance of one SessionDataBean and one Helper class object, SessionDataBean object helps to collect user input and displays the results on the view (JSP/HTML) page, and Helper object helps to call the backend web service operation. Every SessionDataBean is an extension of *SadadPortalSessionDataBean*, and every Helper is an extension of *PortalServiceCallHelper* abstract class.

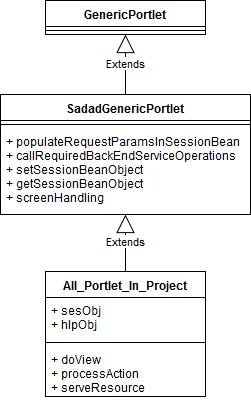


Figure 17 – Portlet Class Diagram

*SadadGenericPortlet* is an abstract class and contains some key methods which define the pattern of the whole development of this project. The method [*populateRequestParamsInSessionBean*](#_Populating_Request_Parameters) is the one which retrieves the incoming HTTP request parameter started with param\_ and populated its value in SessionDataBean object property which is followed by param\_, in case of property not found in SessionDataBean class and all its parents' classes then the generic error displayed to the user. Another critical method [*callRequiredBackEndServiceOperation*](#_Calling_Backend_Service) which is used to retrieve the name of the method from SessionDataBean field operation and invoke the method against Helper object, if operation name not provided then this will be ignored.

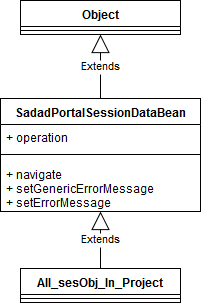


Figure 18 – SessionDataBean Class Diagram

*SadadPortalSessionDataBean* is an abstract class which contains an *operation*, *screen*, and *messageToDisplay* properties as protected and being inherited to all its subclasses; the critical methods names are *navigate*, *setErrorMessage* and *setGenericErrorMessage*, which helps in screen navigation and setting of error message on the frontend view.

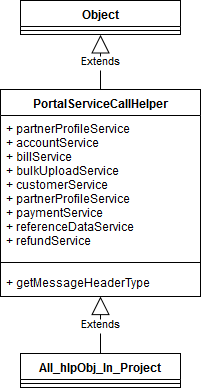


Figure 19 – ServiceCallHelper Class Diagram

*PortalServiceCallHelper* is an abstract class, parent of all Helper classes in the project which contains the instance of services delegates from the *SadadServiceClient* projects and three overloaded methods *getMessageHeaderType* for calling S2 backend services.

## Sequence Flow of Events and Portal Response

1. User access portlet, render request sent to the Portal server, the Portal server responds with a [*doView*](#_Case_of_Render) method.
2. User submits a form, *doQueryFormSubmission* method executes, or user click detail icon *doPostUrl* method executes, resource request sent to the Portal server, the Portal server responds with a [*serveResource*](#_Case_of_Resource_1) method.
3. User clicks on the back or finish button, action request sent to Portal server, [*processAction*](#_Case_of_Action) executed on Portal server, and Portal server responds with a [*doView*](#_Case_of_Render) method because after executing [*processAction*](#_Case_of_Action) method [*doView*](#_Case_of_Render) invoked as per Portlet lifecycle.

The above three scenarios are common for all the portlets, and each portlet has almost the same code for [*doView*](#_Case_of_Render), [*processAction*](#_Case_of_Action), and [*serveResource*](#_Case_of_Resource) method except the SessionDataBean and Helper class selection. For details about the Java code explanation see the [detailed section](#_Java_Portlet_and).

Service Endpoint Properties

SadadServicesEndPointUrls.properties

# Loading Static Resource Dynamically

INDEX.JSP file has the below common code. First two lines of code are the JSP directive to include two JSP fragments which load the resource bundle and load static resource for each module. Line 3 is the message container; line 4 is div block which further contains the two div container naming container\_1 and container\_2 which load the pages dynamically when the user refreshes the page, or the Ajax response will be rendered in container\_1 or container\_2 based on the naming pattern of the JSP files. Line 8 is the modal div which display the progress icon during Ajax call or post back call to the server. Line 9 – 14 is the JavaScript calls which executes once the page load finish, these functions could be empty or having custom logic based on the requirement of the portlet module.



Figure 20 – INDEX.JSP Source Code

## JSP Declaration Explanation

From the first line of INDEX.JSP another JSP fragment is loaded into the INDEX.JSP page.



Figure 21 – JSP Fragment for Variable Declaration

Line 1, 2 are the basic page import, content type and page encoding setting, from Line 4 – 8 required JSTL tag libraries are imported. Line 10 declare *defineObjects* from *portlet* tag library which will provide the *portletConfig*, *renderRequest*, *resourceRequest*, *actionRequest* objects in JSP. Line 11 – 24 based on the incoming request type, *thisRequest* variable is set with current request, Line 25 is using *thisRequest* variable to get the locale for the current request to load the resource bundle using *portletConfig* object for JSP label and caption rendering, and set it in variable name *bndlLang*. This *bndlLang* variable contains the dynamically loaded resource bundle which defines in the default resource bundle for the portlet module, *portletConfig.getResourceBundle* reads the resource file defined in the *portlet.xml* from that particular resource file reads the key *sadad.portlet.resource-bundle.base-name* which is another resource bundle file name, loaded and available on using *bndlLang*, which is used in every JSP to render any label or caption on web page from the resource bundle. Line 26 loads another resource bundle which is common and contains the generic error and info messages to display on JSP and set it in *bndlCommon* variable.

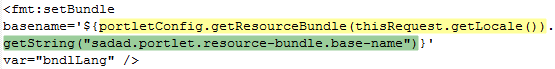


Figure 22 – Line 25 from JSPDeclaration.jspf File

Yellow highlighted line read *nl.ebpp.accounts.QueryAccountsPortlet* as defined in *portlet.xml* file, see the below Figure 15.

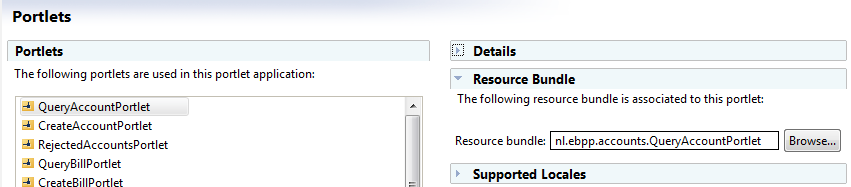


Figure 23 – Portlet Deployment Descriptor Showing Resource Bundle for Selected Portlet

Green highlighted line from Figure 14 reads the value of the defined key and returns resource bundle file name *nl.ebpp.core.EbppCorePortletResource*, which mean this file will be loaded in *bndlLang*, see the below Figure 16.

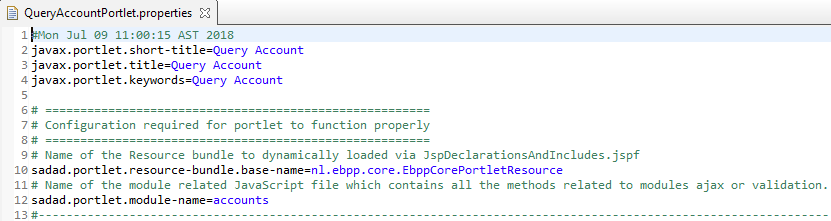


Figure 24 – Portlet Resource Bundle Showing Defined Properties

The reason behind using *sadad.portlet.resource-bundle.base-name* different than the portlet module default resource file is that in EBPP section all the labels in all screen (account, bill, payment, customer and refund) single resource bundle is used and the portlet module default resource file which is defined in *portlet.xml* is used to set the portlet name and titles only because in **WebSphere Portal Administration Console** the portlet titles are displayed using this default resource bundle.

## JSP Load Static Resources

The second line of INDEX.JSP loads another JSP fragment into the INDEX.JSP page.



Figure 25 – JSP Fragment for loading static resource

Line 1 is the check for loading the resources into the web page; all the static resources loaded only if the request type is *renderRequest*. From Line 2 – 12 stylesheets and JavaScript files are loaded using *thisRequest.getContextPath* variable defined in earlier loaded JSF fragment, here *thisRequest* can be replaced with *renderRequest* but leave it as is because *thisRequest* variable is used in other JSP files which are used to display HTML Forms or Details about the inquired data on the web page. Line 9 is loading one JavaScript file which is dynamically read from the resource bundle, this JavaScript file is containing specific code related to the module only, and in some cases these files are an empty files (to follow the same pattern just created an empty files), payment type portlets contain custom JavaScript file.



Figure 26 – Line 9 from JSPIncludeStaticFiles.jspf

Further explaining Line 9 from JSPIncludeStaticFiles.jspf, the blue highlighted part is loading default resource bundle which is defined in portlet deployment descriptor (*portlet.xml*), the brown highlighted part is reading the key *sadad.portlet.module-name* and concatenate it with Portlet.js to load the module related JavaScript file.

Line 14 – 19 keys *sadad-generic-info* and *sadad-generic-error* are loaded from common bundle and put in JSON *\_objKV* object, which is used in method to display error in case any Ajax request failed without getting any valid response or any business error from back-end server.

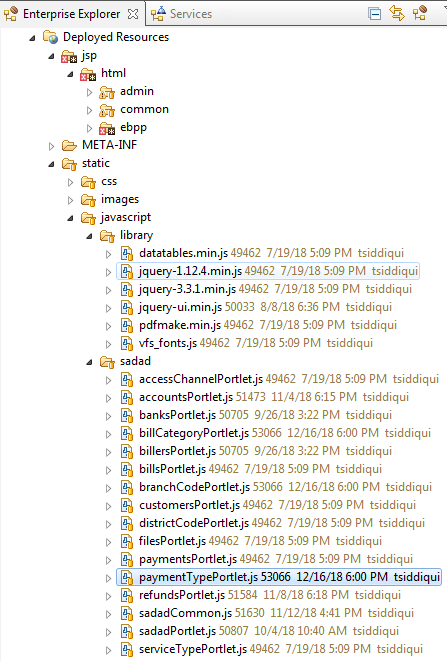
All the static files are placed under *static* folder, *static* folder contains *css*, *images*, *javascript/library* and *javascript/sadad* folder, *javascript/library* contain JavaScript files for jQuery and jQuery plugins like DataTables and PDF Make and *javascript/sadad* folder contain files for Sadad Portlet modules like accounts, bills, billers, banks and so on.

Figure 27 – Static files directory showing JavaScript files for Portlet Modules

# Java Portlet and Custom Class Code Explanation

## Case of Render Request

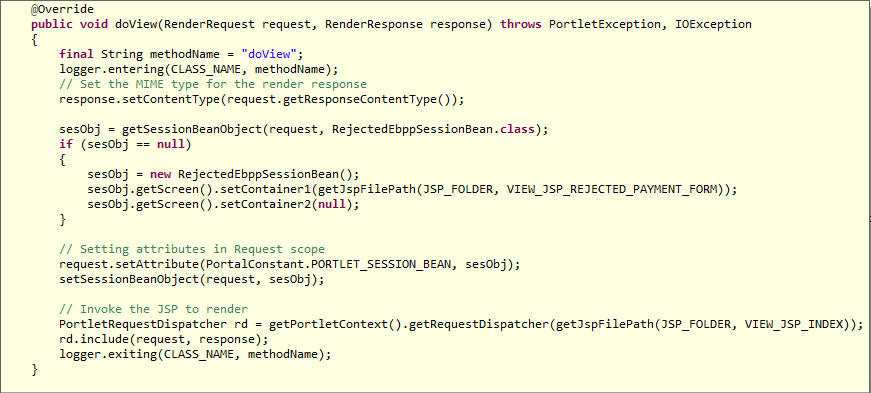


Figure 28 – Code Snippet of doView method

The *doView* method call the *getSessionBeanObject* of *SadadGenericPortlet* class, the parameter passed to the *getSessionBeanObject* will be different in each class depending on the SessionDataBean of the portlet, the getSessionBeanObject will return if *doView* is called first time by the user, otherwise existing SessionDataBean object will be returned and stored in *sesObj* instance variable. If *sesObj* is null then the required JSP (depending on each portlet view) is set in container 1 and container 2 is explicitly set to null (which actually sets the *NONE.jsp* in container 2). Place the same *sesObj* in request attribute and in portlet session scope. In last with the help of request dispatcher render the INDEX.JSP.

## Case of Action Request

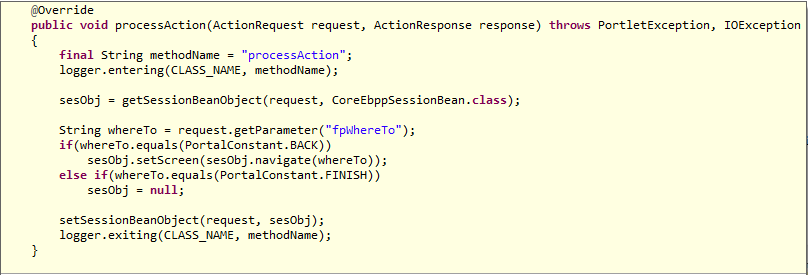


Figure 29 – Code Snippet of processAction method

The *processAction* method always called from the *back* and the *finish* button. When *processAction* method invoked it call the *getSessionBeanObject* to get the existing *sesObj* from the session, and manipulate its screen object, the request parameter *fpWhereTo* contains two possible values which are *back* or *finish*, in case of back *sesObj* navigate to previous screen and in case of finish *sesObj* become null and set the same *sesObj* in session scope. After the execution of *processAction*, the portlet container calls the *render* method and in our case to *render* method invoke *doView* method.

## Case of Resource Request

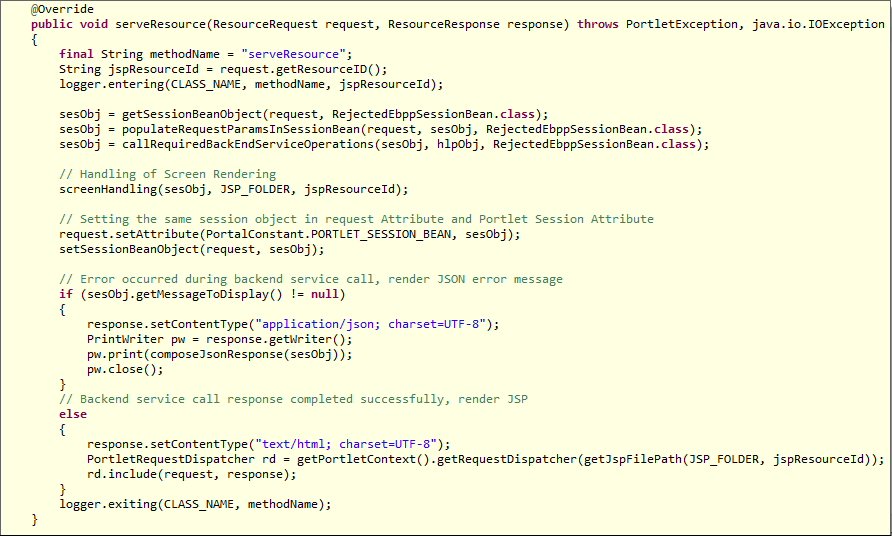


Figure 30 – Code Snippet of serveResource method

Most of the portlet calls to Portal server are resource request via HTTP POST, and it always invoked the *serveResource* method of the portlet class. The *serveResource* call the *getSessionBeanObject* method to get the *sesObj* from the session and call [*populateRequestParamsInSessionBean*](#_Populating_Request_Parameters_1) to populate incoming request parameter in *sesObj* and then call the [*callRequiredBackEndServiceOperation*](#_Calling_Backend_Service) method to invoke the required backend service operation to either retrieve the data from a database or to store the data into the database. The next method call is the [*screenHandling*](#_Frontend_Screen_Navigation) method from *SadadGenericPortlet* class which handles the display of JSP on container 1 or 2 based on the naming convention. Later the *sesObj* is saved in request scope and session scope for further retrieval and in last it checks any error message stored *sesObj* *messageToDisplay* property, if *messageToDisplay* is not null it means there is some exception occurred during backend service operation calls, and now the rendering of error message is only required instead of rendering of JSP. The response content type changed to *application/json* in case of error message otherwise *text/html* content type is the default for JSP rendering responses.

## Populating Request Parameters in Java Bean Dynamically

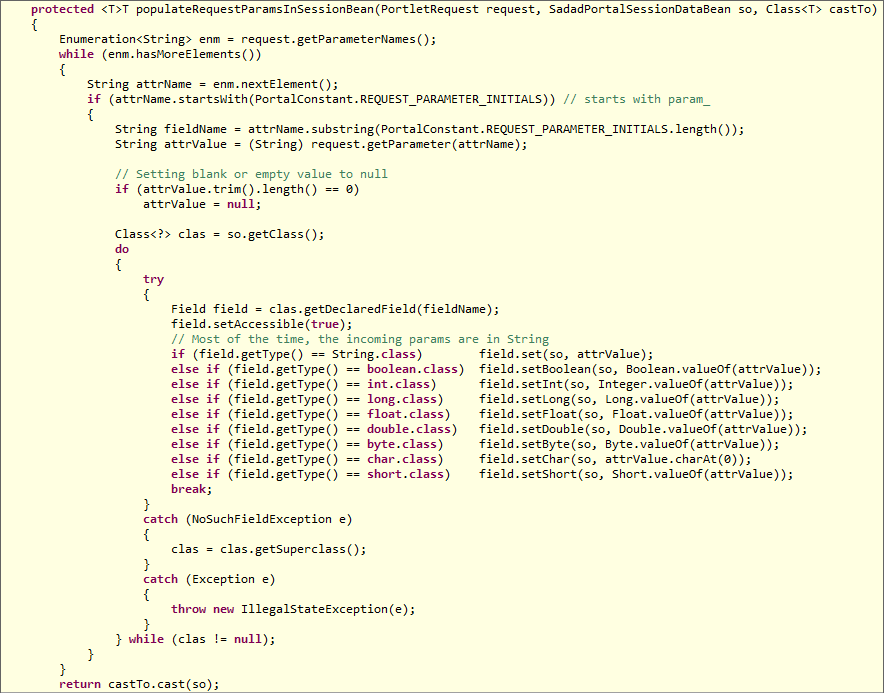


Figure 31 – Code Snippet of populateRequestParamsInSessionBean method

The *populateRequestParamsInSessionBean* method uses the Java Reflection API to dynamically set the values in SessionDataBean *sesObj* object. The method takes three parameters the first one is the portlet request parameter which is used to extract the incoming request parameter, and put it in the *sesObj* object which is the second parameter of the method. The third and last parameter is the *Class* parameter which is used to cast the *sesObj* to that specific class at the time of return. The method loops through all the incoming request parameter and checks the request parameter name starting with *param\_* for further processing, then strip the starting *param\_* part from the request parameter name to get the field name of the *sesObj* instance. If the length of the value of the incoming request parameter is zero, then the null value replaced with the blank value. Moreover, get the Class of *sesObj* and using Java Reflection API get its specific declared field using the field name which earlier extracted from request parameter, later by checking the field type of the parameter assign the value to the field by calling the appropriate *set* method according to the field type. In case of *NoSuchFieldException*, the parent class examined for the same field name and assigned the value to the field of the parent class, and return the *sesObj* after casting it into the third parameter Class name of the method.

## Calling Backend Service Operation Dynamically

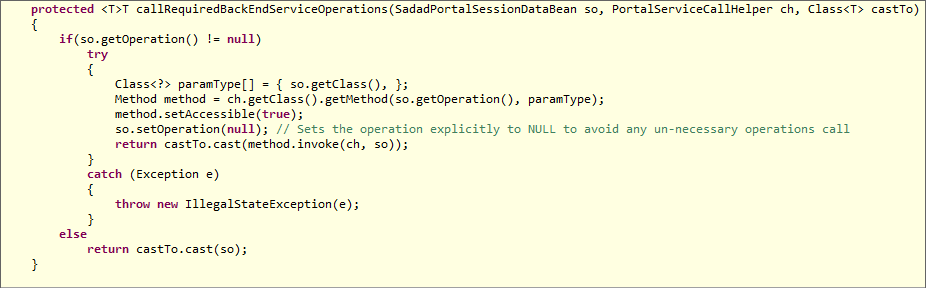


Figure 32 – Code Snippet callRequiredBackEndServiceOperation of method

The *callRequiredBackEndServiceOperation* method also uses Java Reflection API to invoke the required method dynamically. The method takes three parameters. The first parameter SessionDataBean *sesObj* used to retrieve the operation name which required to call and also passed as the parameter to the dynamic method when required operation invoked. The second parameter Helper *hlpObj* used to invoke the dynamic method, and the last parameter is the *Class* parameter used to cast the *sesObj* to the class at the time of return.

## Frontend Screen Navigation

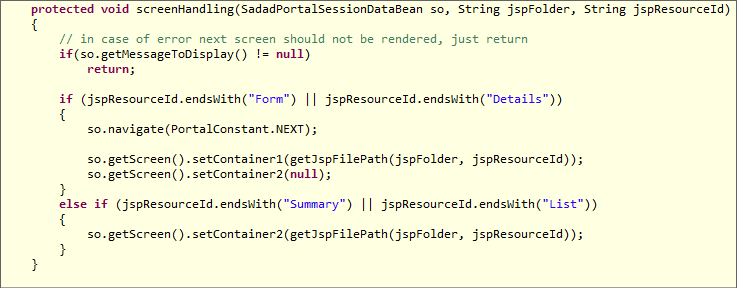


Figure 33 – Code Snippet of screenHandling method

The *screenHandling* method contains the logic of displaying JSP on different containers of the INDEX.JSP. The method takes three parameters, and the first parameter is *sesObj* which contains the *Screen* object and *Screen* object consist of *container1* and *container2* fields. The second parameter is the *jspFolder*, which is the name of the directory which contains the JSP file and the third parameter is the *jspResourceId* which is actually the name of the JSP file name without the file extension. The simple logic implementation is that if the file name ends with *Form* or *Details*, the file rendered in *container1* and if the file name ends with *Summary* or *List*, the file rendered in *container2*. In case of container1 [*navigate*](#_Pushing_and_Popping) method also called with argument value next which is pushing new *Screen* object into Java *ArrayDeque*.

## Pushing and Popping Screens into ArrayDeque

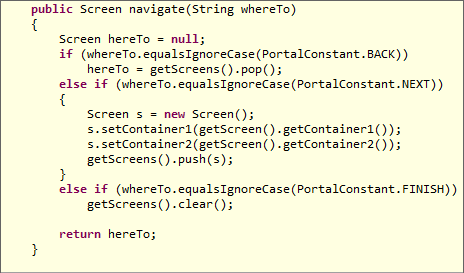


Figure 34 – Code Snippet of navigate method

The screen navigation handling requires the *Stack* type of data structure that is why we are using *ArrayDeque* which can be used as Stack, Linked List, Array or Double Ended Queue, the *screens* instance variable which is the type of *ArrayDeque<Screen>* declared in *SadadPortalSessionDataBean* class used to control the logic. The *navigate* method requires only one *String* parameter whose possible values are *NEXT*, *BACK* and *FINISH*. If the incoming *String* parameter value is *NEXT* a new *Screen* object is created with current container values and pushed into the *screens ArrayDeque* object. In case of *BACK*, the *pop* method is called for the *screens ArrayDeque* object which returns the topmost object in *ArrayDeque* and case of *FINISH* the *screens* object emptied, in both *NEXT* and *FINISH* cases the returning value is null.

# JavaScript Function Explanation

## HTML Form Submission via JS Function



Figure 35 – Code Snippet of doQueryFormSubmission

The project have some functions in JavaScript but the two are the key methods in written in JavaScript for posting data to the Portal Server, the above code snippet is the one of the method which use to send HTTP Post calls from Portal front end to backend Portal server. This method uses the jQuery Ajax call functionality to post HTML Form, the first parameter *formName* is only mandatory and others are optional, if *responseContainer* is given then the response will be rendered in the give response container otherwise container\_2 is the default response container as usually the Forms are rendered in container\_1.

## HTTP Post Call via JS Function

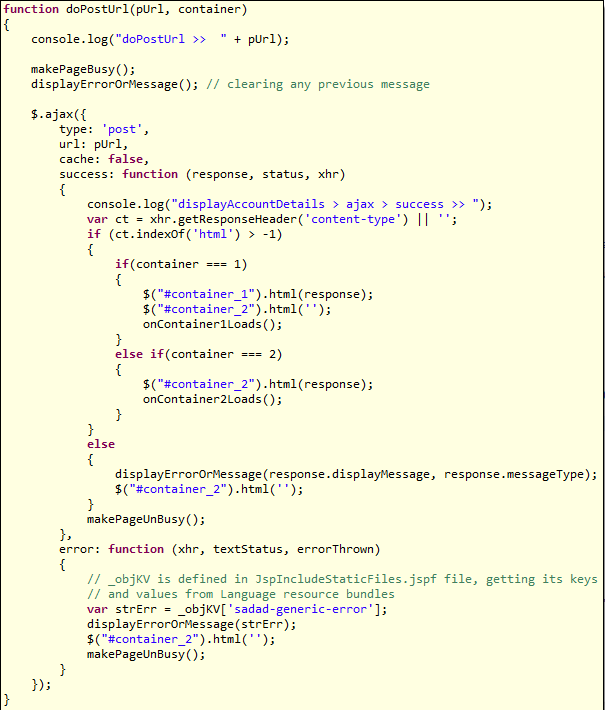


Figure 36 – Code Snippet of doPostUrl

The second important method in JavaScript code is doPostUrl, which is used to send HTTP Post calls to the Portal Server and it mainly invoke serveResource method of Portlet. The first parameter is the URL and the second is the container in which the rendering of response required. Both arguments are required to call the function.

# Application Level Cache

Application cached list of banks and billers from Partner Profile and list of district code, account type and access channel from Reference Data services on an application level in project’s servlet context level at the time of application startup and it is destroyed at application stop. The *com.sadad.portal.common.cache.ApplicationCacheInitializer* is the main class responsible for loading and unloading of cached data. The class *ApplicationCacheInitializer implements ServletContextListener* interface to implement the Servlet level caching. This cache will only be available to the Portlets which are part of this WAR project.

# Developing Web Service Client

The *SadadServiceClient* is Maven based JAR project which contains the classes auto generated using JAX-WS and some custom code to call the backend web services.

## Generate Client using JAX-WS

Copy the WSDL and XSD files into *SadadServiceClient\src\main\java\META-INF\wsdl* directory along with all dependent XSD files if any for the required service which need to be called from the Portal frontend.

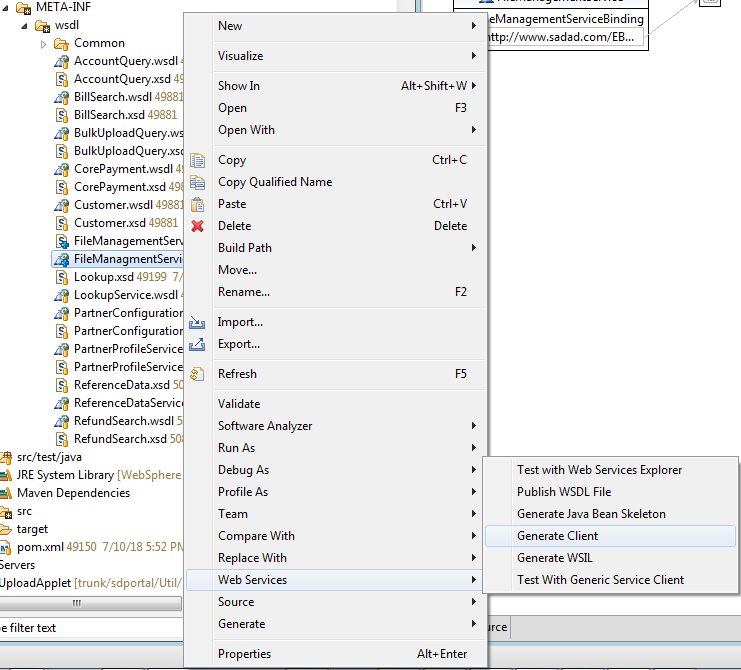


Figure 37 – Generate Client for Web Services



After copying the WSDL and XSD files right click on the WSDL file select Web Services 🡪 Generate Client option from the menu. The Web Service Client window will appear, here decrease the level of to the first level only which is Develop only.

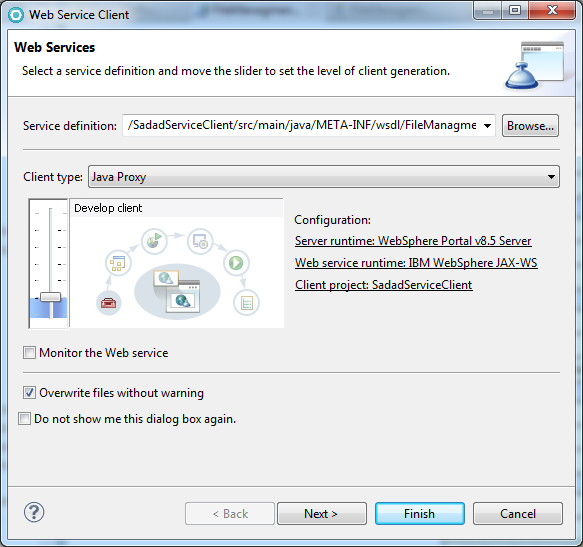


Figure 38 – Web Service Client Window Service Definition

Click Next button to further proceed with web Service Client generation. Overwrite files without warning can be unchecked if you do not want to overwrite the common files which are already generated with any previous WSDL client generation.

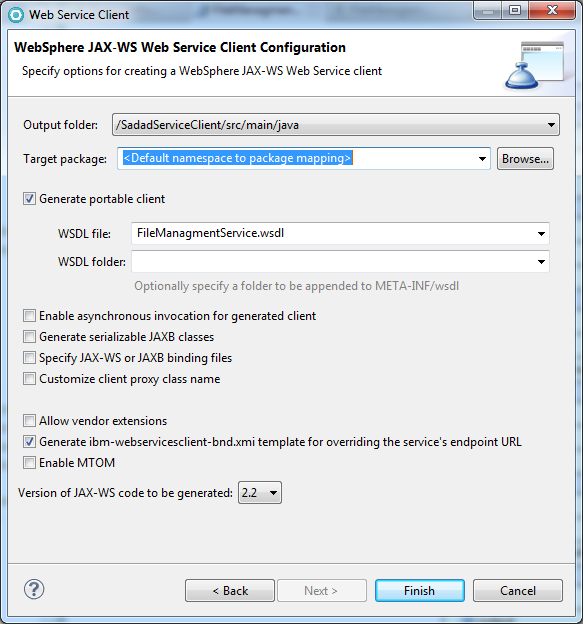
S

Figure 39 – Web Service Client Window Output Folder

Make sure that the output folder is *src/main/java* and click Finish button to start the generation of Java classes, wait for some time classes will be generated shortly depends on the size of the WSDL and XSD files.

## Custom Code for Web Service Client Project

As stated earlier in the introduction that two packages contains the custom code which actually the trigger point of calling backend service. The custom packages are

1. com.sadad.ebpp.portal.delegate.factory
2. com.sadad.ebpp.portal.delegate.factory.clients

The package name *com.sadad.ebpp.portal.delegate.factory* contains only the two Java files, the *ServiceDelegateFactory* interface and *ServiceDelegateFactoryImpl* which is the implementation class of *ServiceDelegateFactory* interface. The *ServiceDelegateFactory* interface defines the *getInstance* method for the singleton implementation and the abstract methods to get the delegate for the web service clients required in portal project.

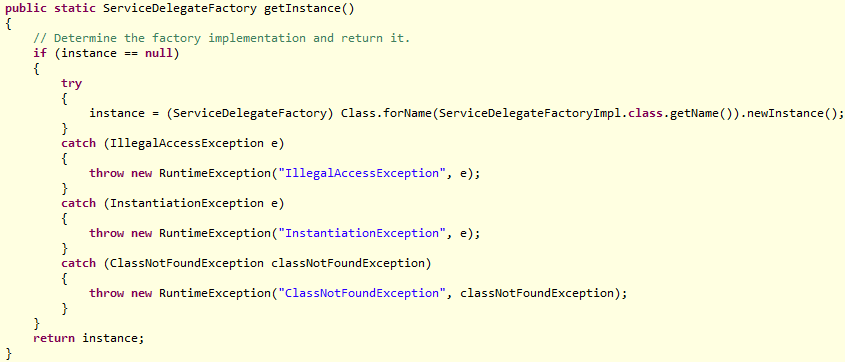


Figure 40 – Method getInstance for return of ServiceDelegateImpl instance.

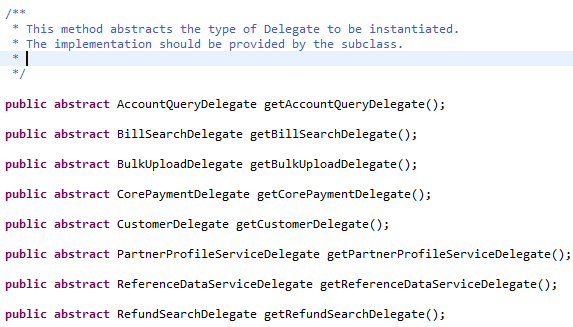


Figure 41 – Abstract methods defined in ServiceDelegateFactory Interface.

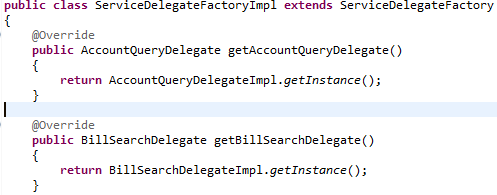


Figure 42 – Implementation of Abstract methods in ServiceDelegateFactoryImpl class.

The *ServiceDelegateFactoryImpl* class is an implementation of *ServiceDelageFactory* and it just implements the abstract methods of its interface, which is only the returning of an instance of actual service delegate. Each WSDL which is imported into the project and used to generate the classes and to call the backend services should have service delegate class.

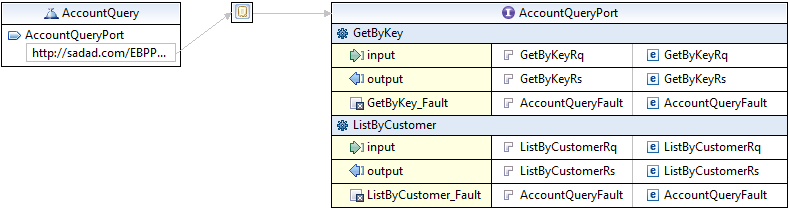


Figure 43 – WSDL Account Query Operations

Taking an example of *AccountQuery.WSDL* file which contains only two operations, we need to create *AccountQueryDelegate* interface and *AccountQueryDelegateImpl* implementation class. Both the interface and implementation classes are very simple as there is not much custom code inside both files. The delegate implementation class implements the singleton pattern, only to create single instance of delegate when calling service from Portal. If there are more methods in WSDL then required for Portal calls, in this case only the required method should be declared in delegate interface and implemented in the implementation class.

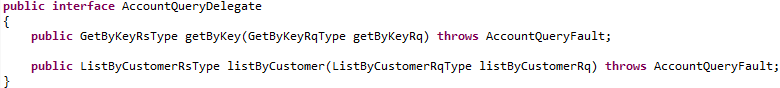


Figure 44 – Delegate interface for Account Query Operations

The above figure show the *AccountQueryDelegate* interface in which it shows the declaration of operations required to call on Portal interface for account query.

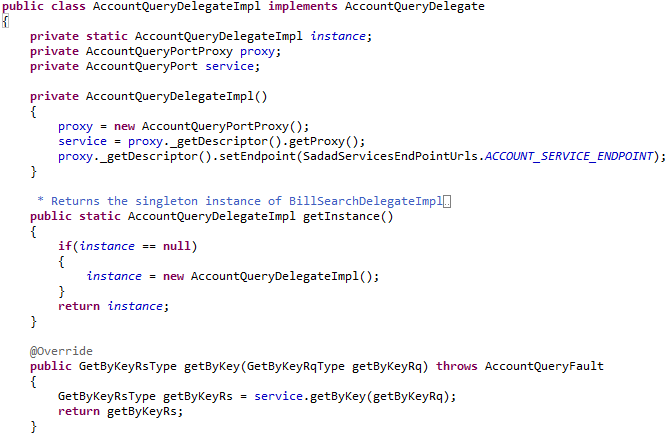


Figure 45 – Delegate Implementation for Account Query

The above snapshot showing the implementation code for *AccountQueryDelegate* interface in which the private constructor is defined along with the public static method name getInstance which returns the self-class type object instance, to make sure that the singleton pattern implemented. The other function calls method have one line implementation which only delegate the call to the service class which is auto generated from the WSDL and XSD files earlier.

## Service Endpoint Definitions

The last line of private constructor is uses set the endpoint of the service URL, this URL is defined in the properties file name *SadadServiceEndPointUrls.properties* which is saved under the *profile\_root/PortalServer/config,* this config directory is known as the portal properties directory and it is available in the server CLASSPATH, files are available to read just by file name without full directory path. *SadadServiceEndPointUrls.properties* is read by an interface which defined the constants for all the URL defined in the properties file.

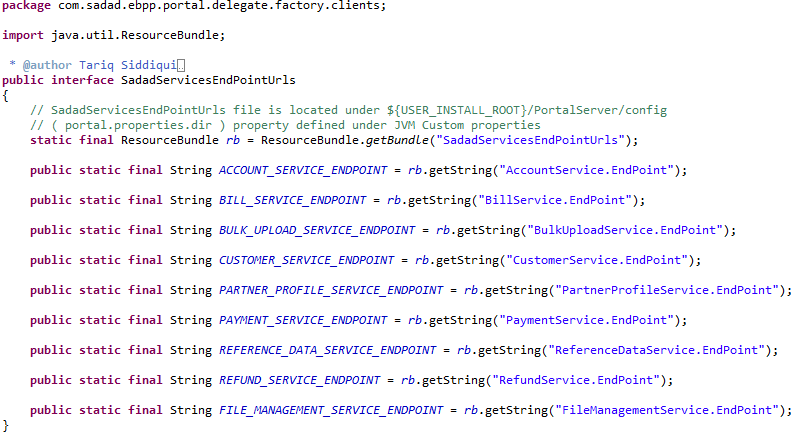


Figure 46 – SadadServicesEndPointUrls reading URLs from SadadServiceEndPointUrls properties file

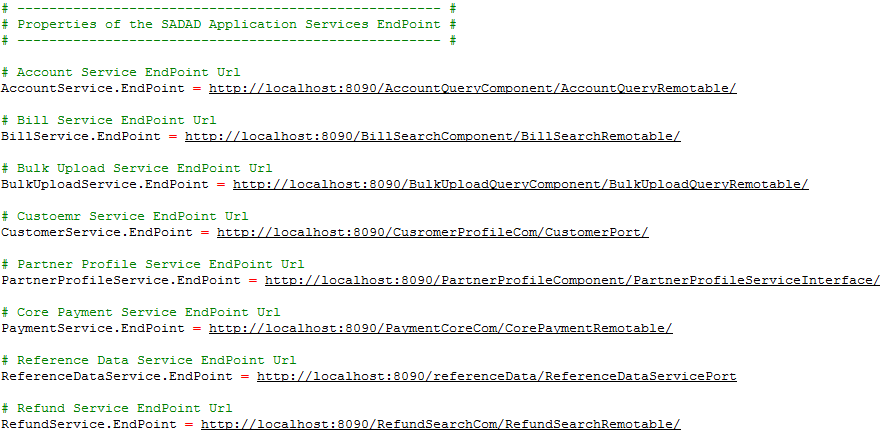


Figure 47 – SadadServicesEndPointUrls Properties File

# Reference

* Ibm.com. (n.d.). *Part 1. WAR deployment*. [online] Available at: https://www.ibm.com/developerworks/rational/library/maven-integration-portlet-projects/index.html [Accessed 19 Nov. 2018].
* Ibm.com. (n.d.). *Part 2. EAR deployment*. [online] Available at: https://www.ibm.com/developerworks/rational/library/maven-integration-portlet-projects-2/index.html [Accessed 19 Nov. 2018].