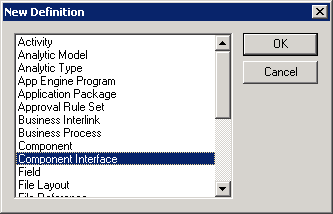
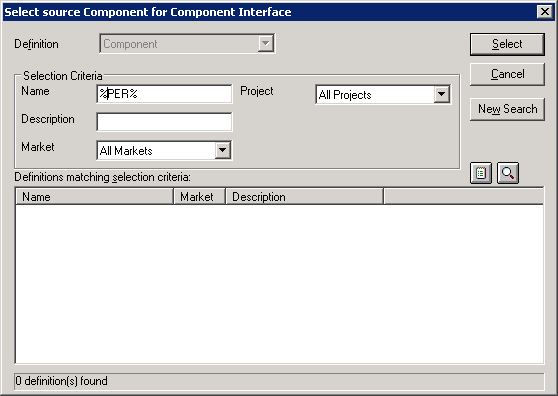
**Create CI**

In App Designer, click on File->New

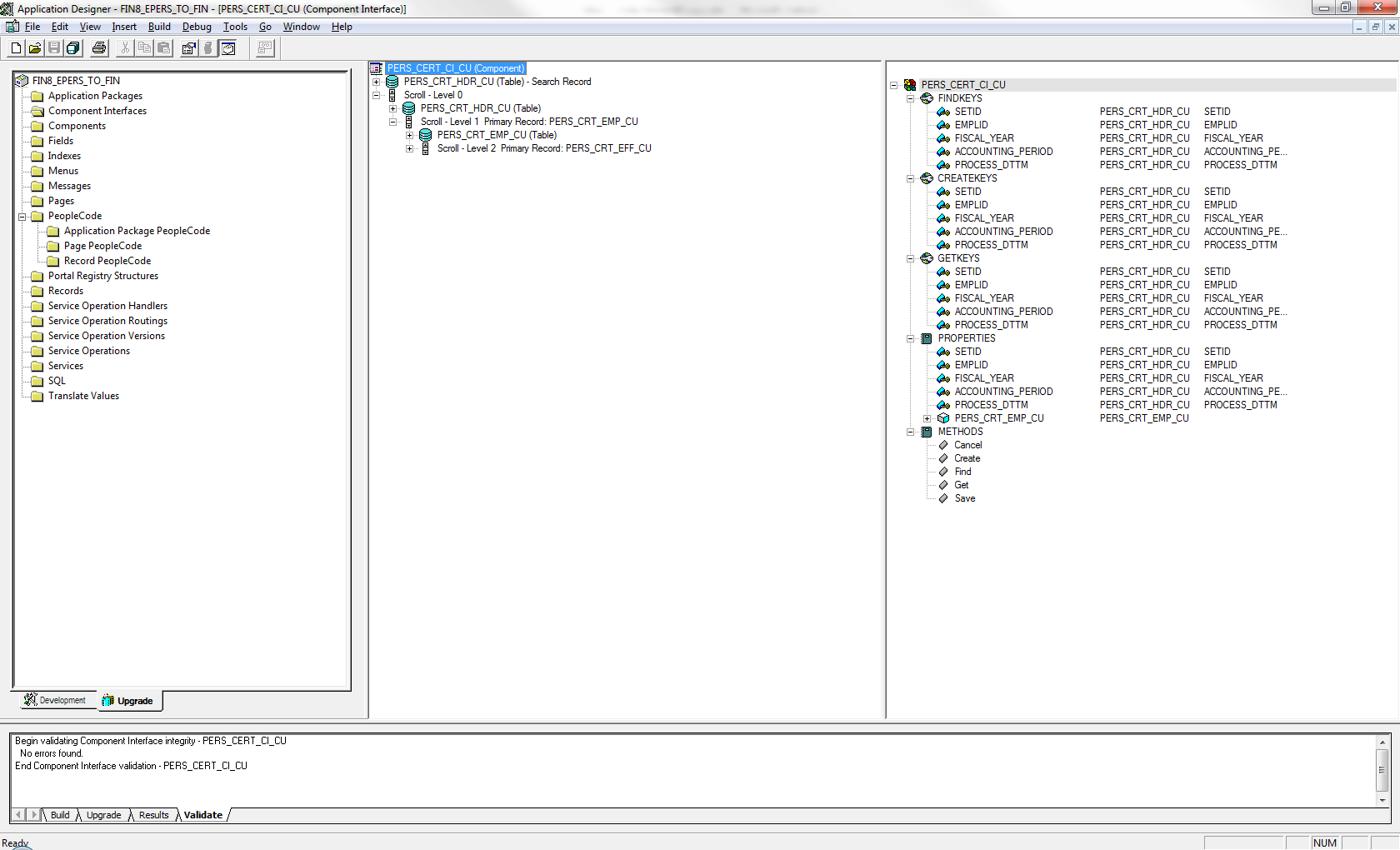
Then select “Component Interface” from the New Definition List.



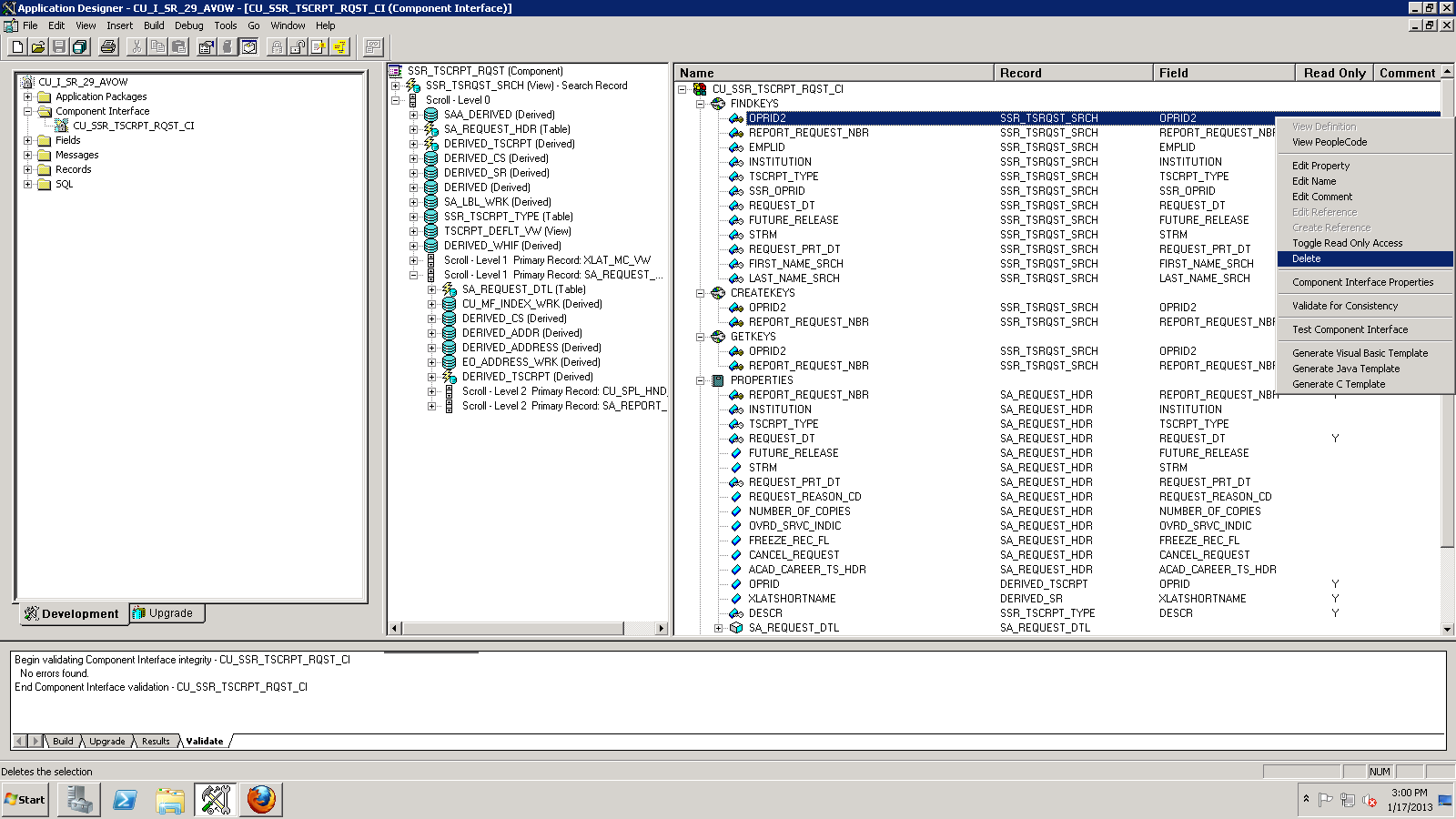
Then enter the name of the component from which you would like to create the component interface. Then click on select.



Select the component and the component interface will be created. You can use the component interface.



You can right click on any of the field to edit property, or toggle Read only or delete the field as needed.

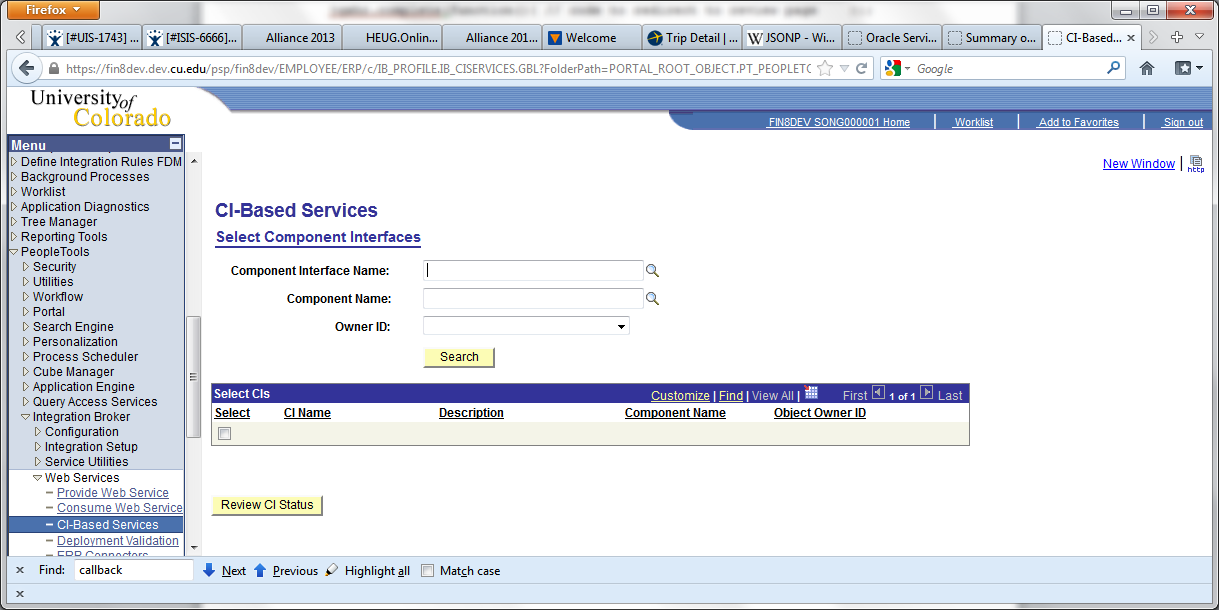


Save the component interface.

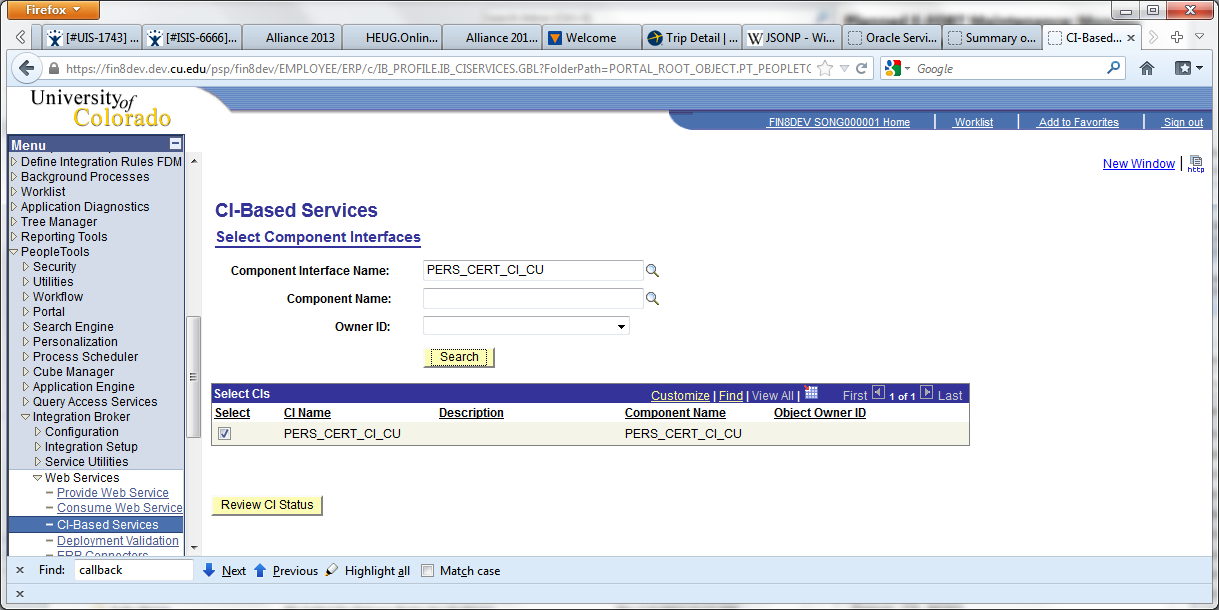
**Expose CI as web services**

Screen shots for exposing CI as Web Services.

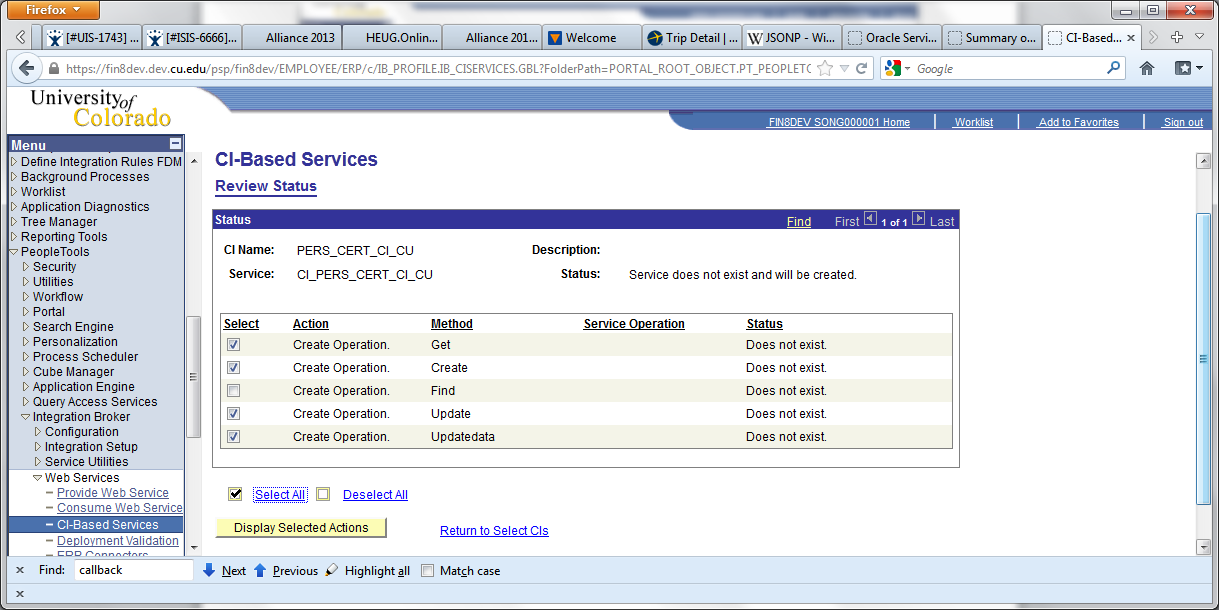
Step 1. Navigate to PeopleTools->Integration Broker -> CI-Based Services



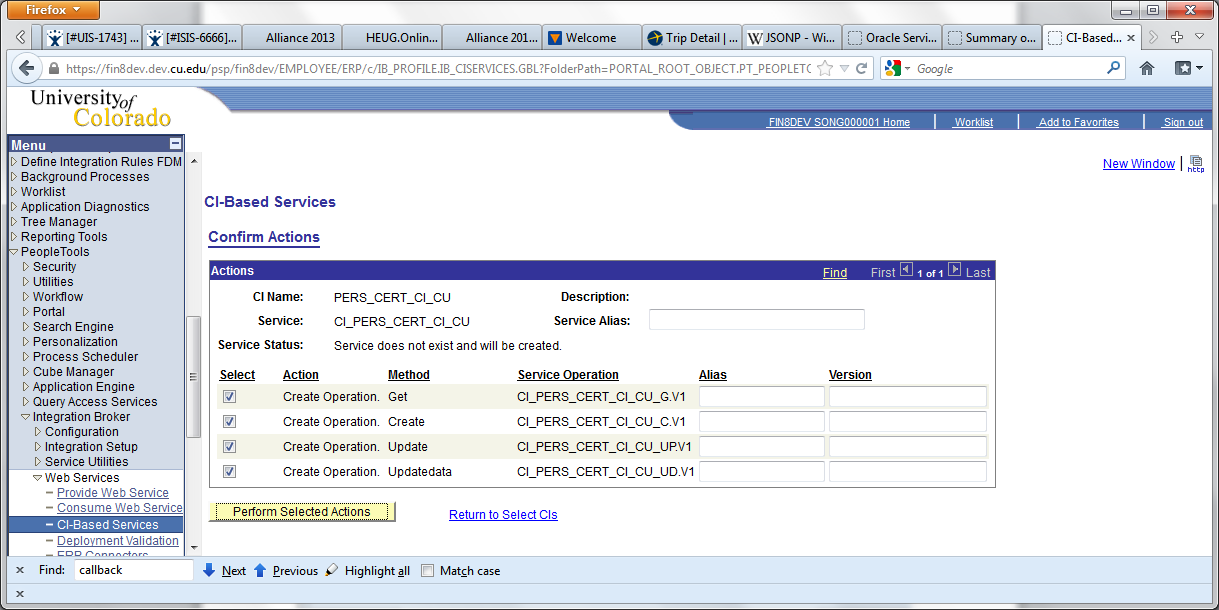
Step 2. Enter the Component Interface name and search, check the Component Interface then click on “Review CI Status” button.



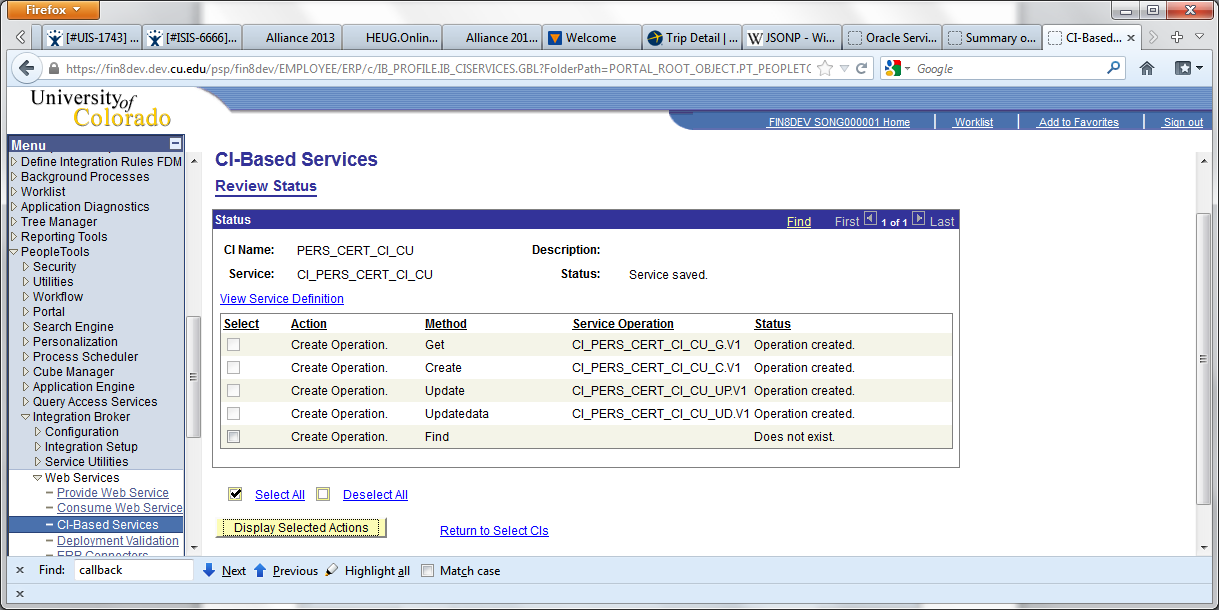
Step 3. Click on the operations need to be exposed (In this screen shot, only Create, Get, Update and UpdateData operations are checked. Find operation is not checked) then click on “Display Selected Actions” button.



Step 4. Enter Service Alias or Service Operation alias and version if you wish to. Then click on “Perform Selected Actions” button.



Step 5: Review the Status of the exposed CI-Based Web Services after the service and service operations are created.

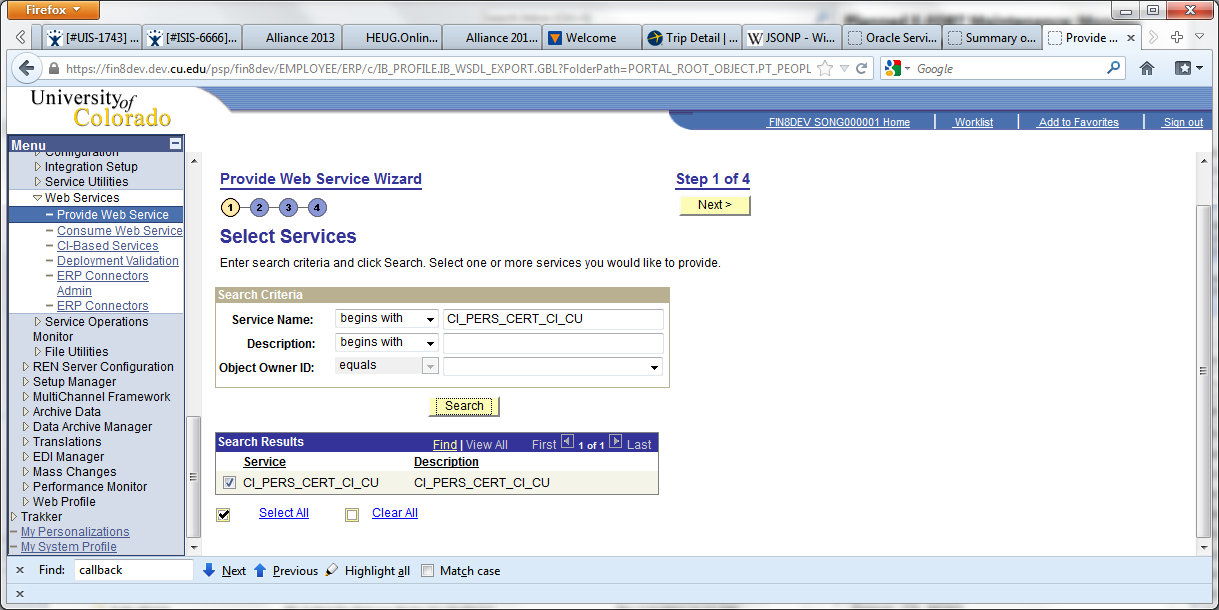


**Provide Web Service**

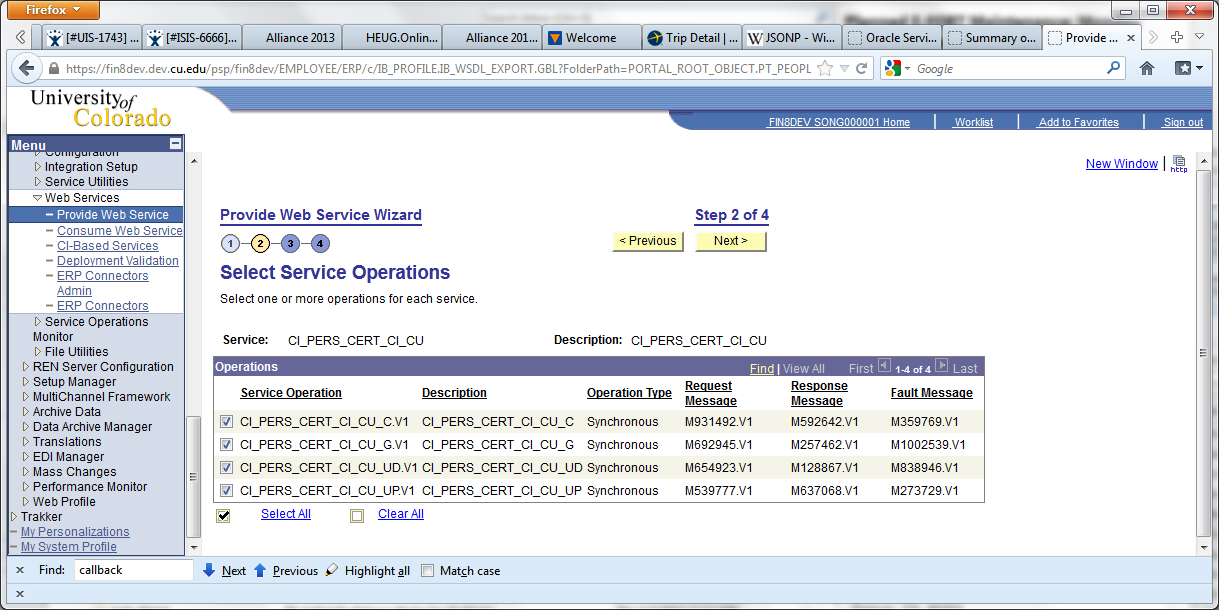
**After we create service and service operations for the Component Interface, we need to publish the web service so that external client will be able to invoke the web services.**

Step 1. Navigate to PeopleTools->Integration Broker -> Web Services -> Provide Web Service

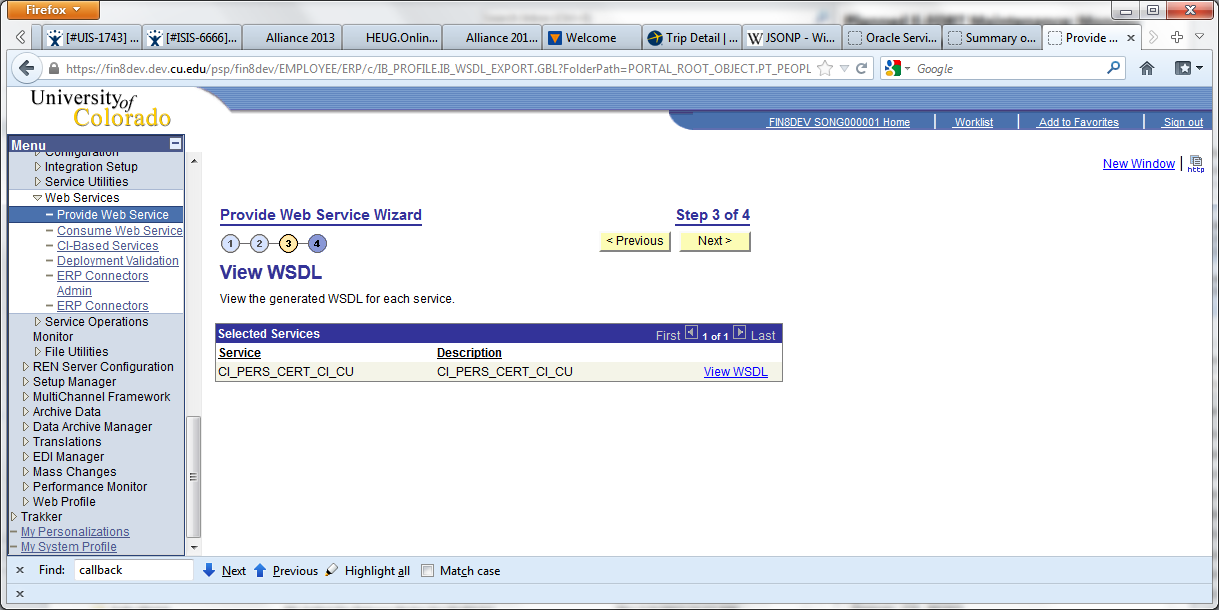
Enter the service name and click on search.



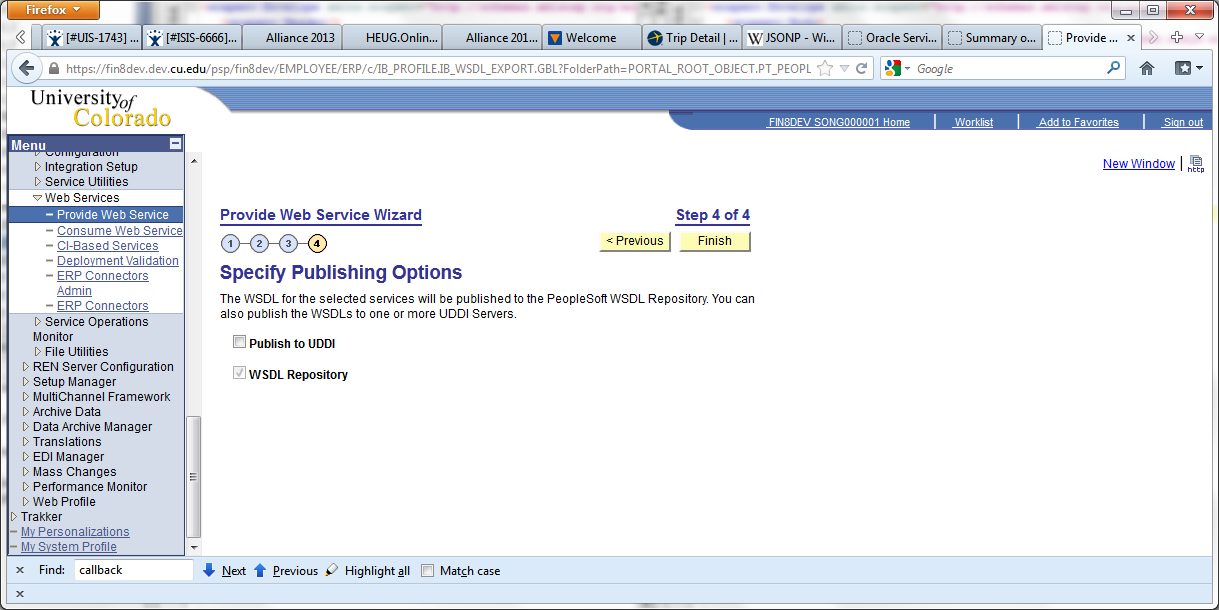
Step 2. Select service operations you would like to expose via web services. Then click on “Next” button.



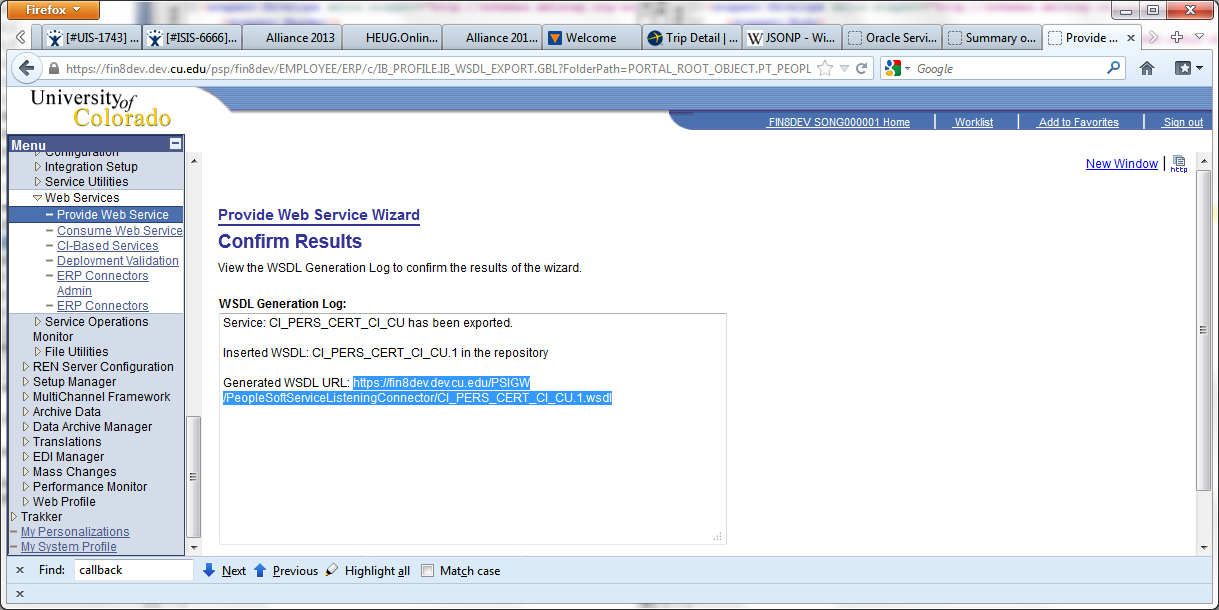
Step 3. Click on “Next” button.



Step 4. Click on “Finished” button.



Step 5. Open the WSDL URL using a browser to view the WSDL file.



**Need for Wrapper Web Services**

Ideally we can expose Component Interface as web services and then we can use the expose web services directly for our application. As we know that when we expose CI as web services, we would end up with 5 service operations typically. These service operations are CREATE, GET, FIND, UPDATE and UPDATEDATA. CREATE operation allows to create a new record. GET operation allows to pull out a record by providing keys as input parameters. FIND operation allows to find a collection of records then meet the given criteria. UPDATE service operation allows you to insert new row in children level record. UPDATEDATA service operation allows to update the data in at scroll level 0 and/or update the children level record.

In our application, we had issues with UPDATE and UPDATEDATA operations. The issue is that in order to call a correct UPDATE or UPDATEDATA service operation, we have to call GET service operation first and compare with the input from the frontend. And then find out if we need to call UPDATE or UPDATEDATA service operation.

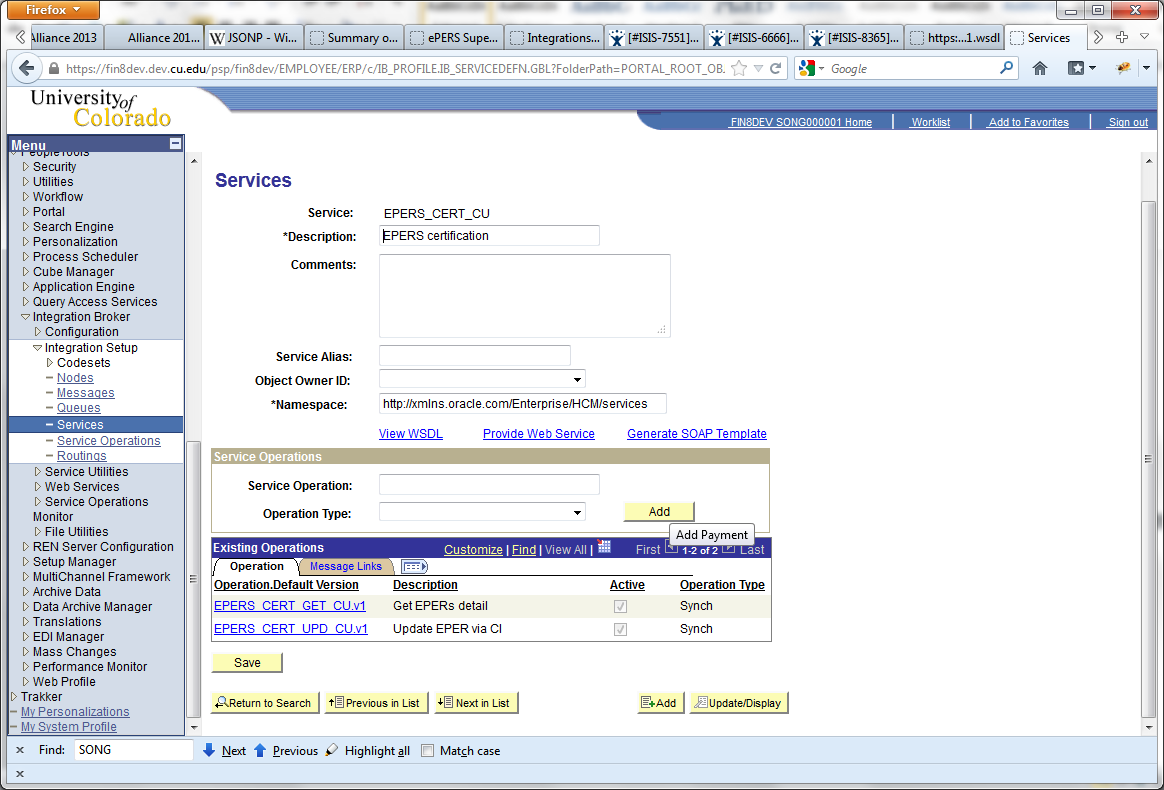
We didn’t want to have this business logic in Oracle Service Bus. As OSB only supposed to transform the data and handle operational branching. OSB should not include complex business logic. Meanwhile, we would not want to have this business logic in front end User Interface. Instead business logic should be in located in PeopleSoft.

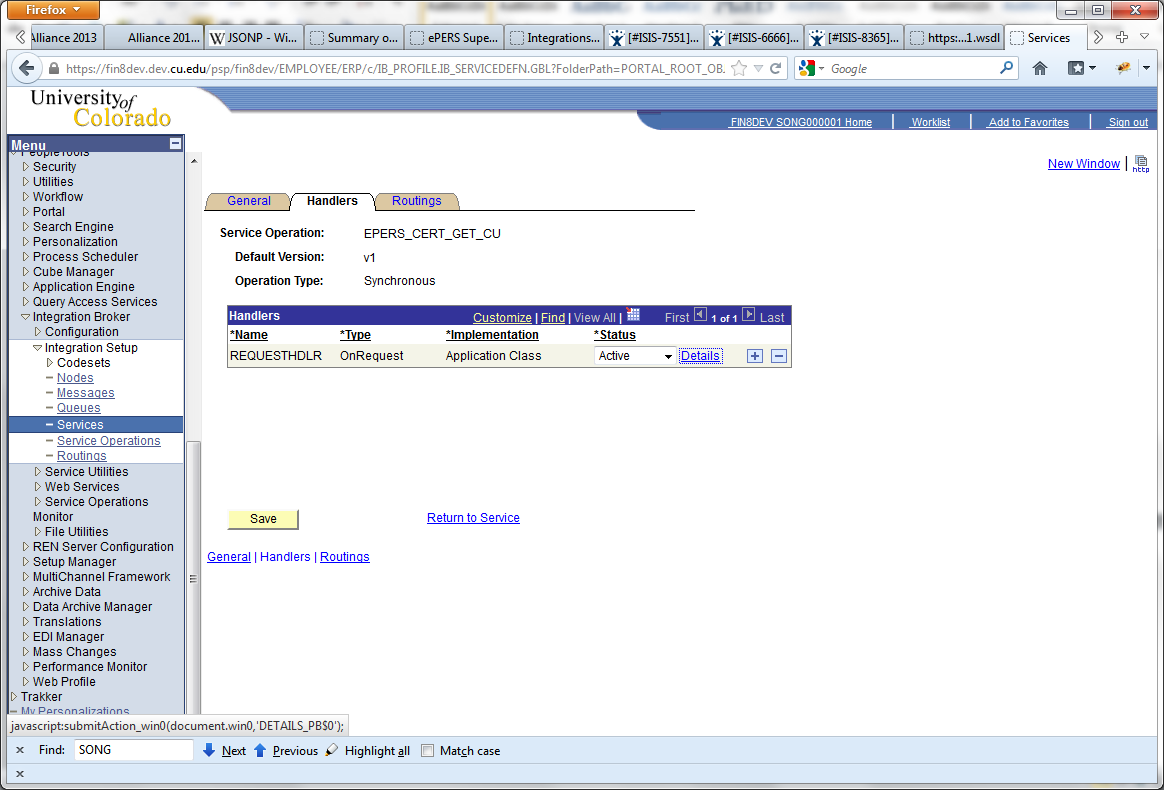
Therefore, instead of use the web service from exposing CI directly, we wrote a wrapper service in FIN using PeopleCode to handle when to update the data and when to insert a new item to the record. In this way, we could also simplify the request and response schema.

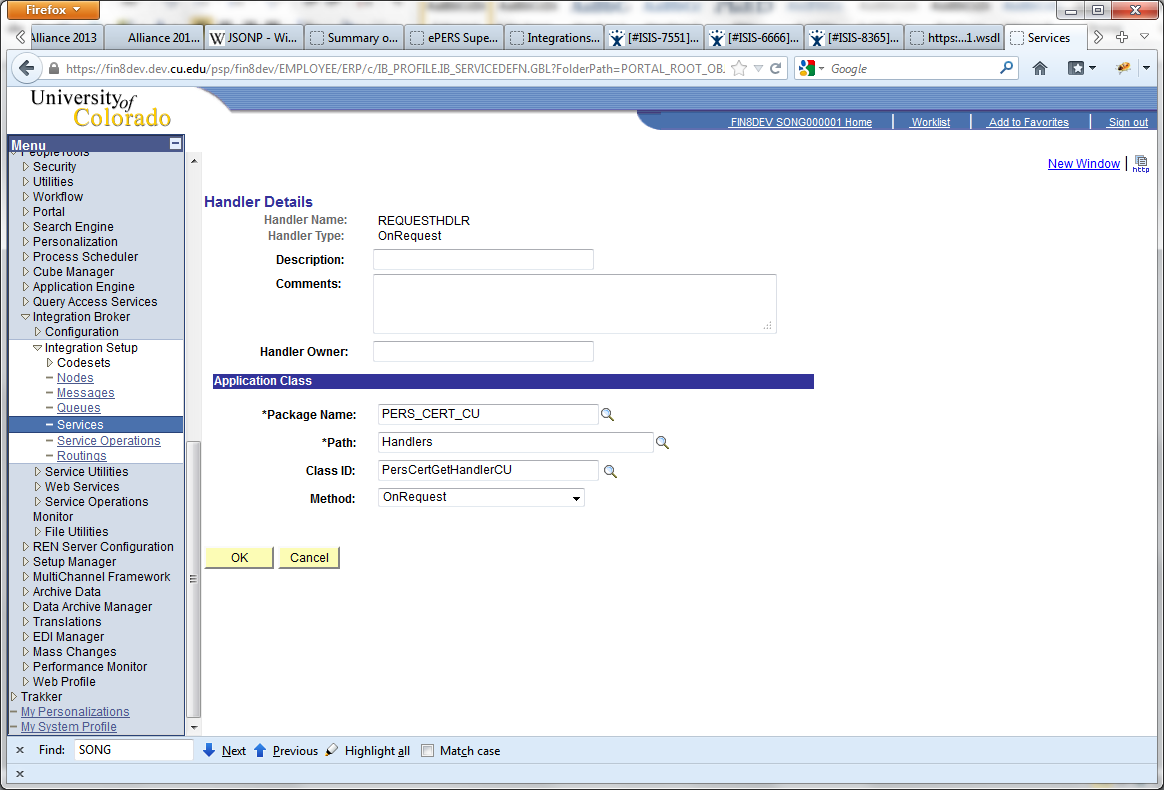
The steps of creating a wrapper web service.

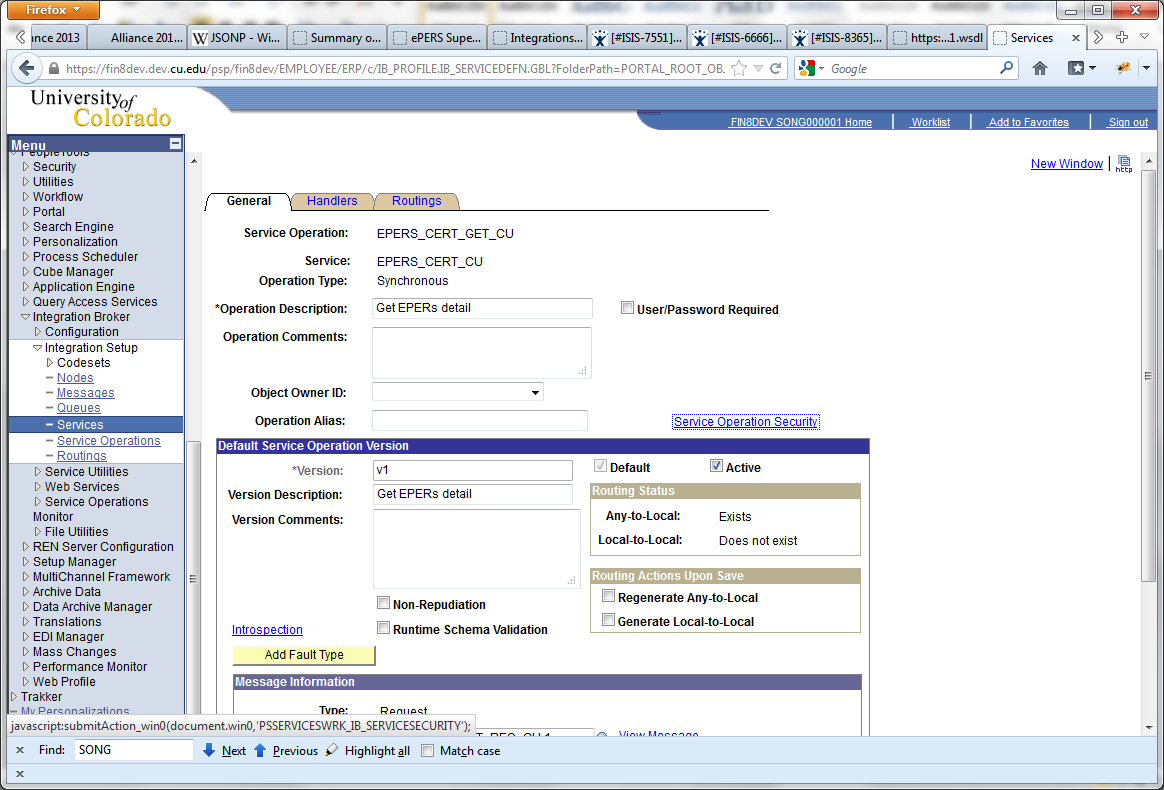
1. Create request and response message schema (PeopleTools->Integration Broker->Integration Tools -> Messages)  
2. Create request and response message (PeopleTools->Integration Broker->Integration Tools -> Messages)  
3. Develop application packages/classes   
4. Create Handler classes that implement PS\_PT:Integration:IRequestHandler

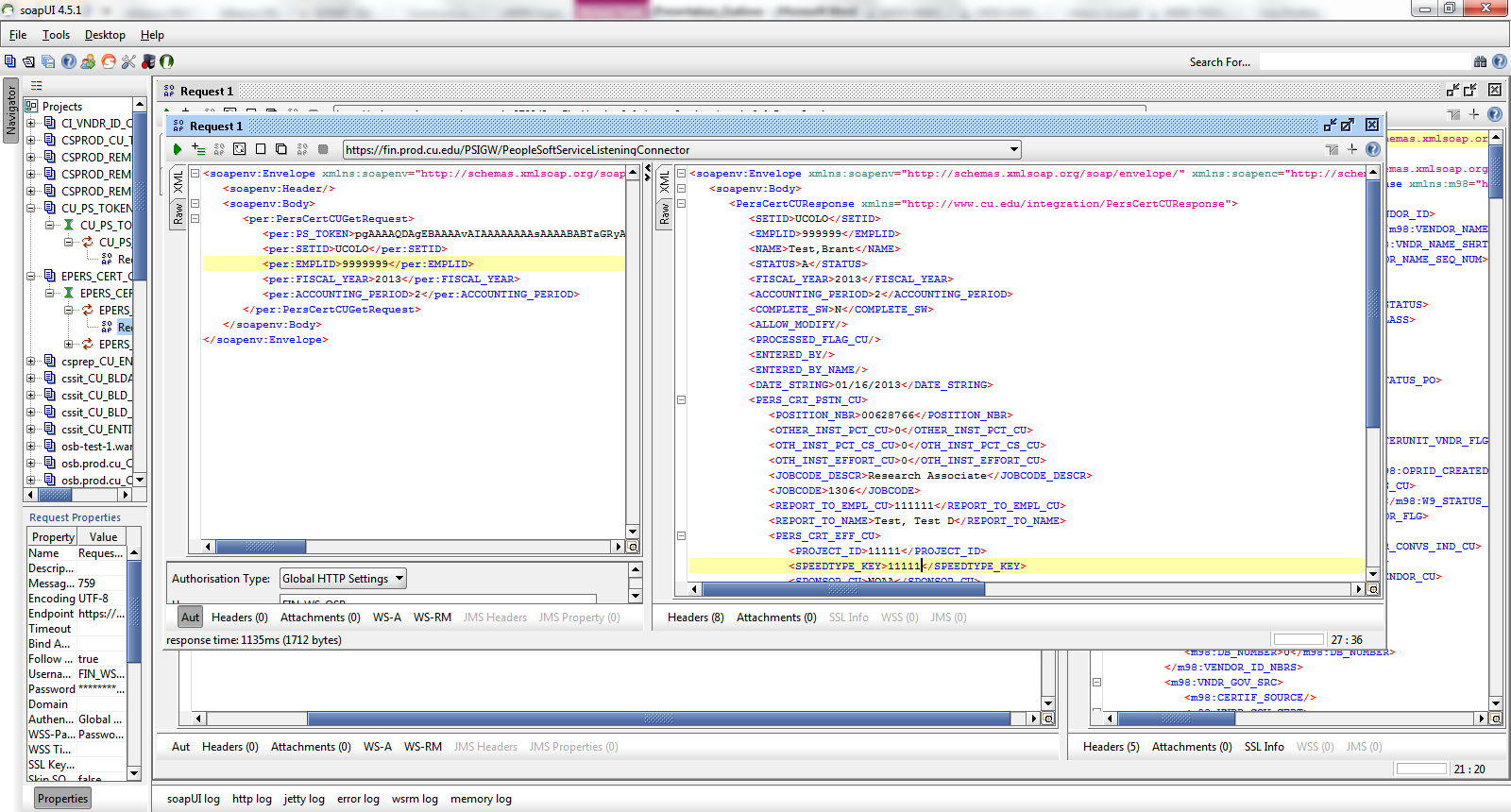
5. Add new service (PeopleTools->Integration Broker->Integration Set up->services)  
6. Add service operations (add service operation in the page where defines new service)  
7. Grant permission security.   
8. Provide web service (get the endpoint at the end of the process)  
9. Test with SOAPUI.

To Add a new service, navigate to Integration Broker->Integration Setup-> Services. Click on “Add new service” Tab. Then enter the new service name. Click on “Add” button.

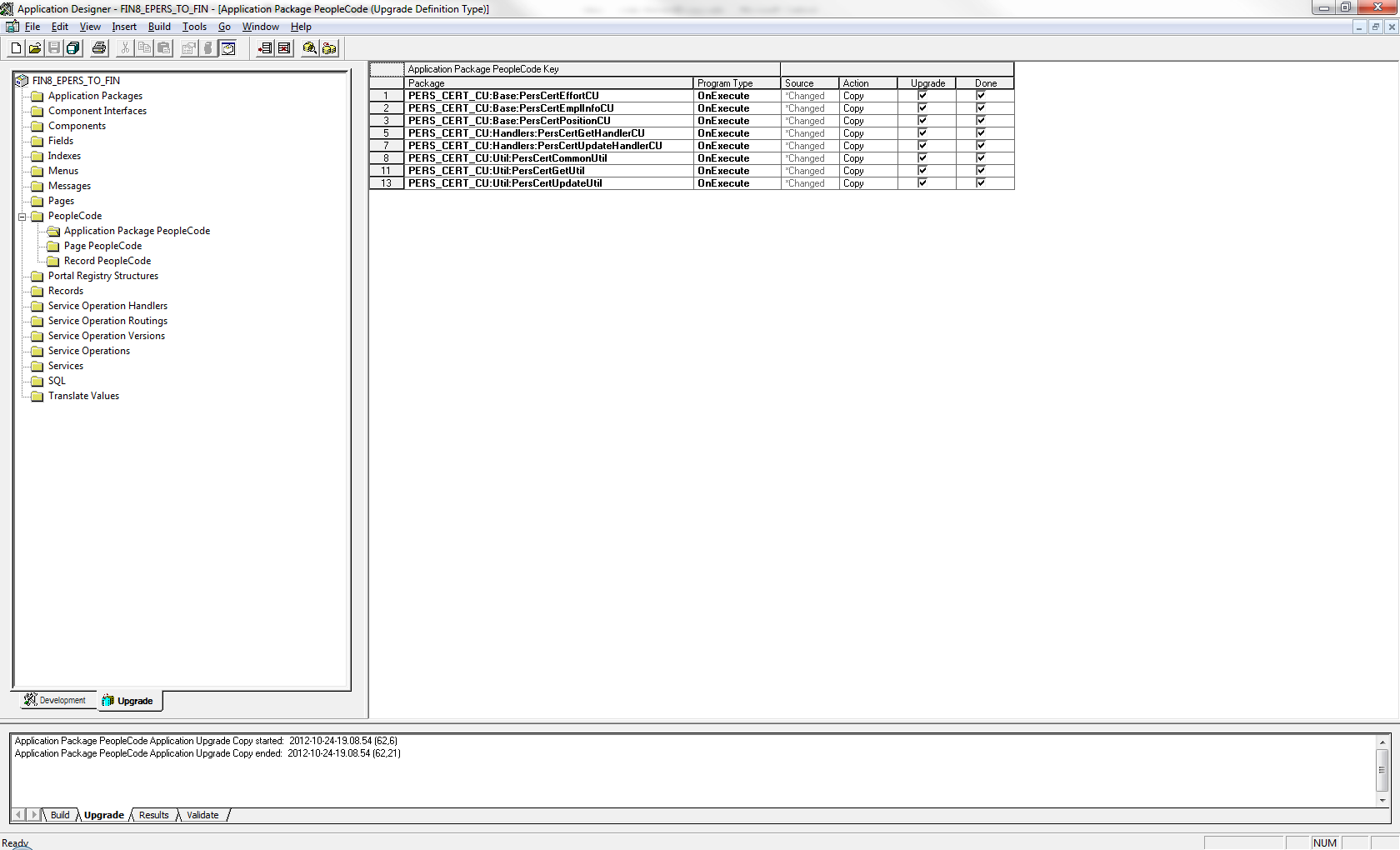








Screenshot - the Application Package PeopleCode for the wrapper web services.



**Oracle Service Bus**

The major tasks Oracle Service Bus performed in this project includes

1. JSON to XML transformation (transform JSON input to XML request for SOAP web service)
2. Conditional Branching based on the HTTP method
3. XML to JSON transformation (transform SOAP web service response XML to JSON)

**OSB Transform**

We used Java Callout for JSON to XML transformation. It seems to be more robust than using XSLT and XQuery for JSON to XML transformation.

Here is the method we have to JSON to XML transformation.

public static XmlObject Json2XmlObject(String jsonString)

{

JSONObject json = JSONObject.fromObject(jsonString);

XMLSerializer xmlSerializer = new XMLSerializer();

xmlSerializer.setTypeHintsEnabled(false);

xmlSerializer.setForceTopLevelObject(false);

String xml = xmlSerializer.write( json );

XmlObject xmlObject = null;

try

{

xmlObject = org.apache.xmlbeans.XmlObject.Factory.parse(xml);

}

catch (XmlException ex)

{

xmlObject = null;

Log logger = LogFactory.getLog("edu.cu.integration.JSONToXMLConverter");

logger.error("Couldn't create xml from Json String. " + ex.getMessage());

}

return xmlObject;

}

Here is the method for Xml to Json transformation.

public static String xmlToJson2(XmlObject xml)

{

XMLSerializer xmlSerializer = new XMLSerializer();

xmlSerializer.setSkipNamespaces( true );

xmlSerializer.setTrimSpaces( true );

xmlSerializer.setRemoveNamespacePrefixFromElements(true);

JSON json = xmlSerializer.read( xml.toString() );

return json.toString();

}

The java class requires a few library files. These library files are not native to the weblogic server so we have to install these library files the $DOMAIN\_DIR/lib.

The jars located in this directory will be picked up and added dynamically to the end of the server classpath at server startup. Therefore, you need to restart the weblogic server after adding new jar files to this directory.

Here are the list of library files.

* commons-beanutils-1.7.0.jar
* commons-collections-3.2.jar
* commons-lang-2.4.jar
* commons-logging-1.1.1.jar
* ezmorph-1.0.2.jar
* json-lib-2.2.3-jdk15.jar
* xmlbeans-2.3.0.jar
* xom-1.1.jar

XML to JSON transformation

We could also use XSLT to transform XML to JSON. We searched and found a working XSLT on the web. Here is the URL. http://code.google.com/p/xml2json-xslt/source/browse/trunk/xml2json.xslt?r=30

So far we do not have a generic XSLT to transform JSON to XML.

In GET HTTP method pipeline, we did use two XQuery files to translate JSON to XML as the parameters were passed in as query strings. The first XQuery is generic. It will transform query strings into xml format. The second XQuery is project specific. It will translate the generic xml to SOAP web service request for that particular project.

Example: Generic xml from query string.

<query-params xmlns="http://www.cusys.edu/restQueryStringPara

meter">

<param name="pstoken" value="PSTOKEN\_VALUE”> PSTOKEN\_VALUE</param>

<param name="callback" value="jQuery16406684299709741026\_1357613568428">jQuer

y16406684299709741026\_1357613568428</param>

<param name="emplid" value="999999">999999</param>

<param name="fiscalYear" value="2013">2013</param>

<param name="setid" value="UCOLO">UCOLO</param>

<param name="accountingPeriod" value="6">6</param>

</query-params>

Example: Project specific input

"<per:PersCertCUGetRequest xmlns="http://www.cusys.edu/restQ

ueryStringParameter" xmlns:per="http://www.cu.edu/integration/PersCertCUGetReque

st">

<per:PS\_TOKEN>PSTOKEN\_VALUE</per:PS\_TOKEN>

<per:EMPLID>999999</per:EMPLID>

<per:FISCAL\_YEAR>2013</per:FISCAL\_YEAR>

<per:ACCOUNTING\_PERIOD>6</per:ACCOUNTING\_PERIOD>

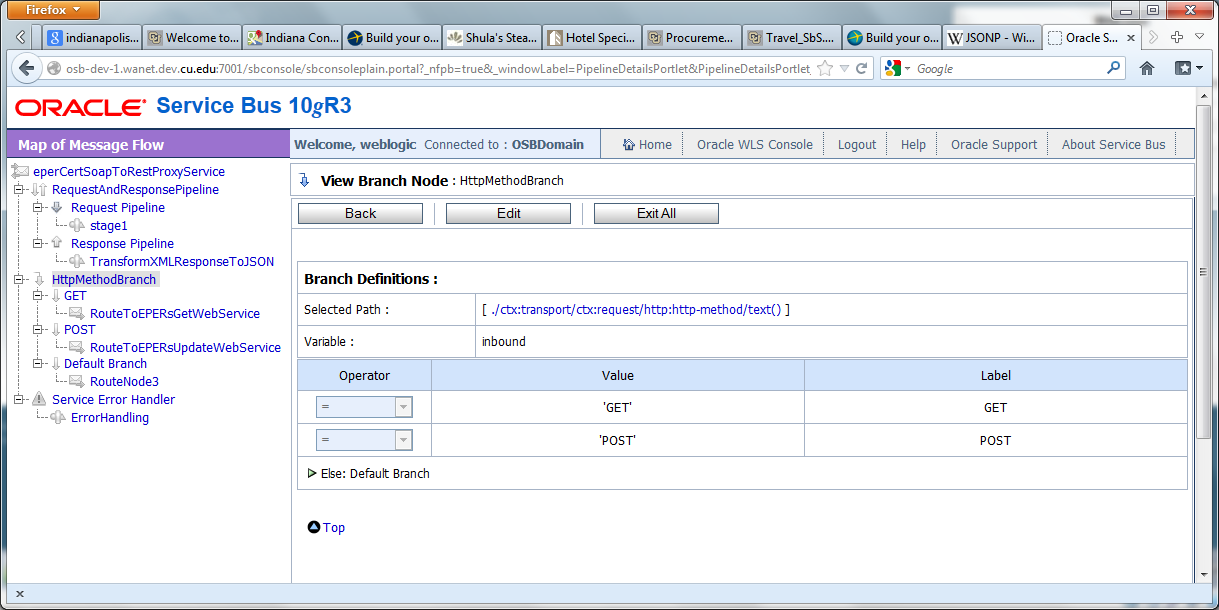
<per:CALLBACK>jQuery16406684299709741026\_1357613568428</per:CALLBACK>

</per:PersCertCUGetRequest>

In POST pipeline of one of our projects (EPER), the input JSON object was much complicated. It has parents, children and grandchildren levels of record data, we don’t have a generic XQuery or XSLT to translate JSON object to XML, so we had to use JAVA callout to transform JSON object to XML object.

We had previous problems with other java callout. Sometimes when we try to activate a change in sbconsole, OSB Servers couldn’t find the jar files to for the Java callout. To resolve this issue, we had to restart the managed servers. Because of this issue, we tried to use XSLT and XQuery to do the transformation as much as possible. In another project (FISCAL CERT), we used XSL and XQuery only for JSON to XML and XML to JSON transformation.

**Conditional Branching based on the HTTP method**



JSON and JSONP

JSONP stand for “JSON with padding”. It provides a method to request data from a server in a different domain.

We appended a callback parameter to the proxy service URL. The browser will generate the value of that callback parameter. It will look like something like this. jQuery16406684299709741026\_1357613568428

Here is the sample code of the JQuery $.getJSON function.

var jqxhr = $.getJSON(

"https://www.cu.edu/proxyServiceURL?callback=?", inputData,

function(data, textStatus, jqXHR){

// process response data here……..

}

)

.success(function() {})

.error(function(jqXHR, textStatus, errorThrown) {

alert("error: " + textStatus);

})

.complete(function() {});

});

On OSB side, we will see something like this. https://www.cu.edu/proxyServiceURL?callback=. jQuery16406684299709741026\_1357613568428&emplid=999999&fiscalYear=2013

We will capture this in callback value in OSB and then pad it to the pure JSON object in the response pipe line using XQuery function.

Example: fn:concat(fn:string($callback),"(", fn:string($xmlToJSONResult), ")")

So the response will be look like this.

jQuery16406684299709741026\_1357613568428 ({"emplid": "999999", "fiscalYear": 2013, "data": “here is the response data”});

**JSON POST and Internet Explore Cross Domain issue.**

JSONP only works for GET JSON file. It does not work for POST. For HTTP POST, HTML/JQuery page has to be on the same domain as service serving the data.

Here is the sample code of the $.post function

var jqxhr = $.post("https://ourProxyServiceURL?", inputDataString, function(data, textStatus, jqXHR) {

})

.success(function() { })

.error(function() { })

.complete(function() { });

// perform other work here ...

// Set another completion function for the request above

jqxhr.complete(function(){ // code to redirect to review page });

};

return false;

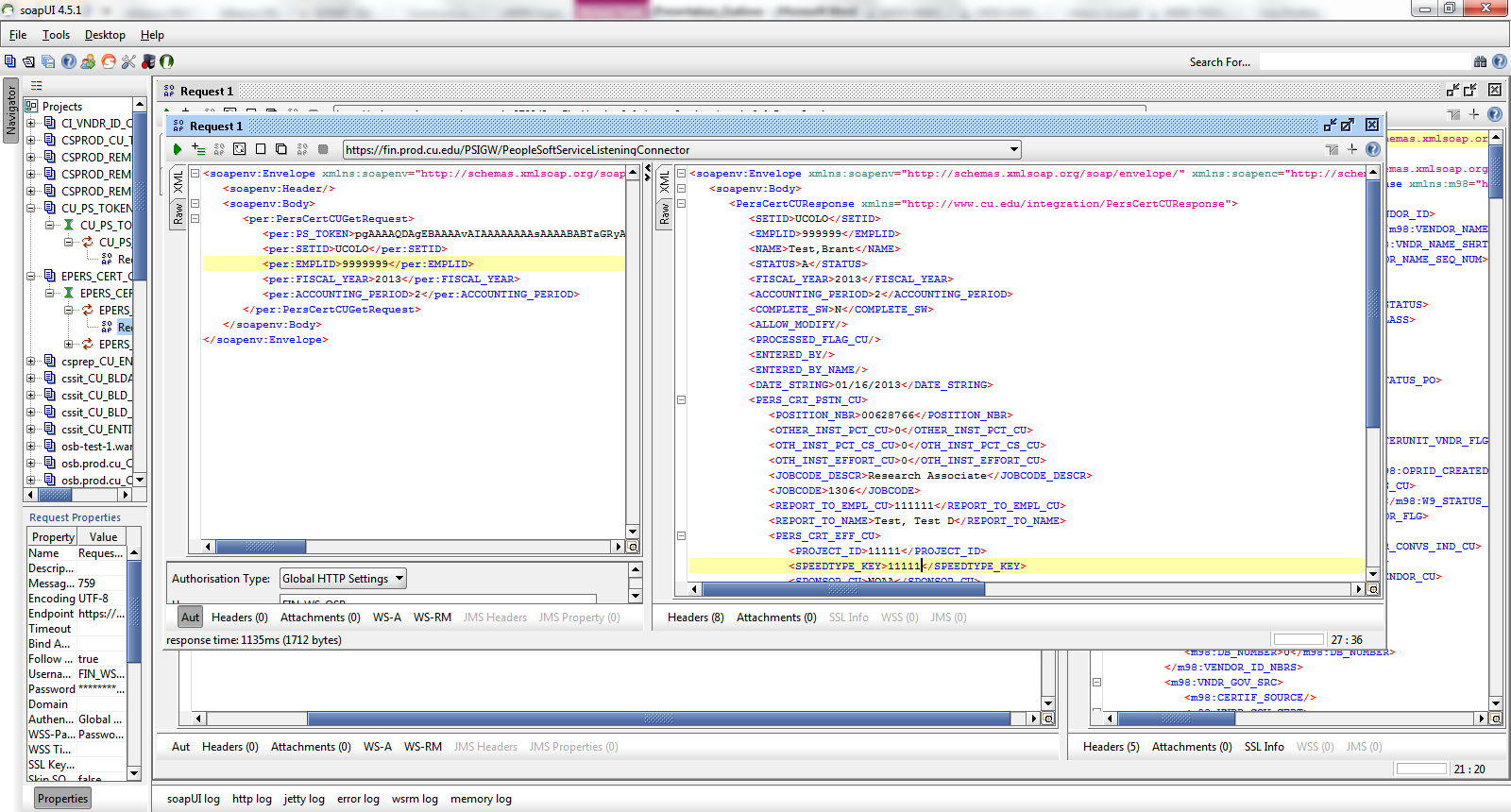
});

We had to either put the front end html on the same domain as Oracle Service Business proxy service, or make it works as if these two components are on the same domain.

Testing

**Use SOAPUI to test our PeopleSoft web services in our Finance System.**

Screenshots



**Use OSB test console to test Oracle Service Bus Proxy Service.**

We also used OSB test console to test Oracle Service Bus Business and Proxy Service.

Screenshots for testing HTTP GET

