OSLC Workshop

OSLC Enable Your Tool in a Day

Lab Exercises

Contents

Overview

Objectives: In these labs, you'll go through steps to enable Bugzilla as an OSLC-CM service provider. This will make Bugzilla capable of receiving OSLC requests and providing OSLC responses. You will leverage some existing samples to validate the implementation. You will also work to improve the client samples to help test the work that you've done.

Pre-requisites:

- Some Java, JavaScript, Java Servlet, Java Server Pages (JSP) and HTML programming experience within an Eclipse IDE. Experience with JAX-RS Web Services and Resource Description Framework (RDF) a plus.

Length: 3-6 hours depending on which optional labs you include

Introduction

This workshop will focus on a number of details regarding OSLC implementations. It is intended to provide a foundation in the skills required for developing OSLC implementations. The example is based on a typical usage. Everyone's environment is unique, and the set of integration scenarios may need to be customized. Following these exercises will provide insight into how a common pattern can be used to provide an adapter for an existing tool, enabling it for OSLC integrations. This has the advantage of providing additional capability to an existing tool without requesting vendor changes or locating tool source code. You will look at alternatives along the way but focus on this key pattern.

The labs in this workshop make extensive use of OSLC4J from the Eclipse Lyo project. OSLC4J is a Java SDK for developing OSLC integrations. For more information, see:

http://wiki.eclipse.org/Lyo/LyoOSLC4J

Scenario

A developer named Nina must integrate a (totally fictional) web-based Customer Relationship Management (CRM) system named **NinaCRM** with **Bugzilla**. Nina has already done the work to OSLC-enable NinaCRM to act as an OSLC consumer, or client. You will focus on adapting Bugzilla so that it can act as an OSLC-CM provider. Since Nina is also looking at using Rational Team Concert within her company, she'll also try to connect related change requests in RTC with Bugzilla using the OSLC-enabled Bugzilla from this workshop.

The labs all include completed code samples which can be uncommented to save time and typing. You are encouraged to attempt the code change yourself.

Labs:

Completing all labs might require more than the time allotted for this workshop. The asterisked (*) labs are optional if time is an issue.

Lab 1: Enabling service discovery – this lab defines a simple set of service discovery resources and discusses the considerations that you must take into account when mapping a tool's data model to OSLC resources.

*Lab 2: Getting familiar with OSLC4J – this optional lab provides an introduction to defining OSLC resources and REST services using OSLC4J.

Lab 3: Creating OSLC delegated dialogs – create delegated web user interface dialogs that can be used to create and search for bugs.

Lab 4: Enabling link traversal and rich hover – Use OSLC techniques to link to Bugzilla bugs from within your tool and provide UI previews of bugs.

Lab 5: Creating a bug – now that you can provide a nice user interface dialog to view the bugs, you learn a programmatic way to create a bug

*Lab 6: Connecting to Rational Team Concert – in this optional lab, you'll highlight steps needed to connect your provider to an existing OSLC-enabled tool such as Rational Team Concert

Topics not covered:

There are a number of topics that will not be covered due to time constraints of this workshop. They are important concepts and are critical in supporting some scenarios.   Some of the concepts **excluded** are: query, authentication, dialog pre-fill, resource paging and concurrent modification. These topics and others are left to later workshops or online tutorials.

Icons

The following symbols appear in this document at places where additional guidance is available.

Icon	Purpose	Explanation
	Important!	This symbol calls attention to a particular step or command. For example, it might alert you to type a command carefully because it is case sensitive.
	Information	This symbol indicates information that might not be necessary to complete a step, but is helpful or good to know.
	Trouble-shooting	This symbol indicates that you can fix a specific problem by completing the associated troubleshooting information.

Lab 0   Getting set up

Objectives:

Perform the necessary steps to get the environment set up and ready for development and testing.

> In order to better accommodate partially completed labs, the code is structured as separate Eclipse projects named according for each lab. For example, "Lab1" project is the project you use for Lab 1.
>
> Once you complete a lab, you can either continue with current project or close it and pick up the next lab project as a clean starting point for the next lab.

For the latest setup instructions, refer to the online instructions under

https://oslc.github.io/developing-oslc-applications/integrating_products_with_oslc/running_the_examples.html

Lab 1   Enabling Service Discovery

| 1. Enable service discovery | → | 2. Get familiar with OSLC4J (optional) | → | 3. Create your first OSLC dialogs | → | 4. Enable link traversal and rich hover | → | 5. Support automated bug creation | → | 6. Connect to Rational Team Concert (optional) |
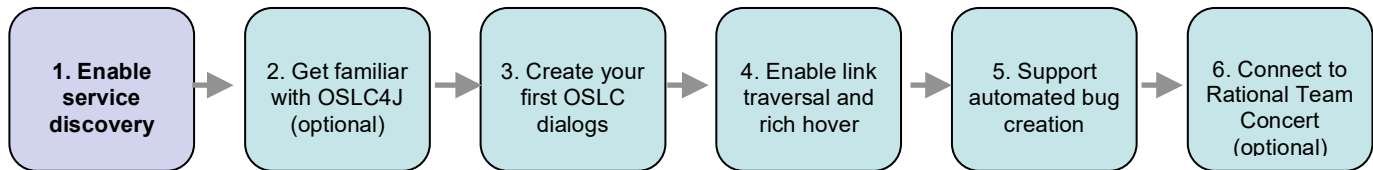
Objectives:

- Understand how the code is designed and arranged.

- Run and test your adapter.

- Add a service provider for each Bugzilla product to your service provider catalog.

- Provide an RDF/XML representation for your service provider catalog

Description

You'll start your work where an OSLC consumer would start, with service discovery. When a consumer wants to interact with an OSLC provider, it first does an HTTP GET to retrieve the provider's Service Provider Catalog. The catalog links to service providers, which provide the links needed to create and query Bugzilla bugs as OSLC Change Requests. Take a look at the OSLC Core specification for more information on Service Provider Catalog and Service Provider resources:

http://open-services.net/bin/view/Main/OslcCoreSpecification#Service_Provider_Resources

You'll implement adding service providers to the service provider catalog for your Bugzilla adapter and create a JSP to display the catalog and provider information.

Summary of tasks for this lab:

1.1 Understand the project structure

1.2 Run the adapter

1.3 Get a list of Bugzilla products and update catalog

1.4 Provide a service provider catalog RDF/XML representation

2.0   Understand the project structure
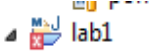
In order to make the needed changes, you will explore the structure and do some basic tests.

__1.   Open the first lab project named **Lab1**

Explore the contents of the project, namely the JAX-RS services, OSLC resource classes and JSPs. You can also explore these items further if you choose to do **Lab2.**   In particular, take a look at:

__a.   **ServiceProviderCatalogService.java** and **ServiceProviderService.java** in the **org.eclipse.lyo.oslc4j.bugzilla.services** package.   You can find the Java source in the **src/main/java** folder or press **Ctrl-R** to search by filename.

__b.   **SeviceProviderCatalogSingleton.java** in the **org.eclipse.lyo.oslc4j.bugzilla.servlet** package

__c.   **serviceprovidercatalog_html.jsp** and **serviceprovider_html.jsp** in the **src/main/webapp/cm** folder.

Verify there are no compiler or configuration errors

__d.   You should observe no red X decorator on the project such as

	Troubleshooting
	It is recommended to open the **Problems** view to see what problems exist with the project. Go to Window->Show Views->Other… and enter in Problems. See also Appendix A

2.0   Run the Adapter

__1.   Select the Run->Run Configurations menu item and then expand the Maven Build section and select **Launch Lab1** and click the **Run** button
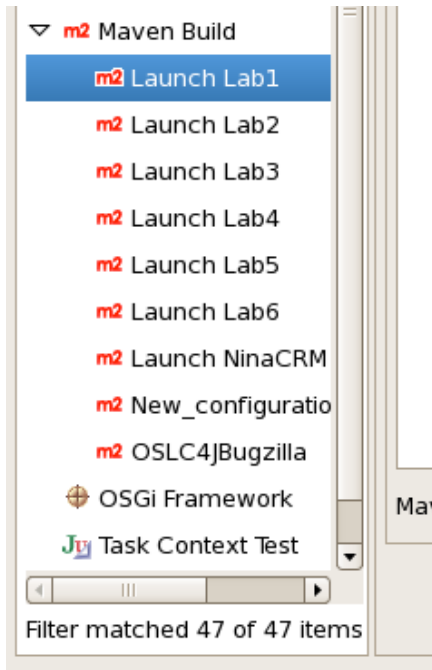
Observe that the server started successfully by switching to the **Console** view
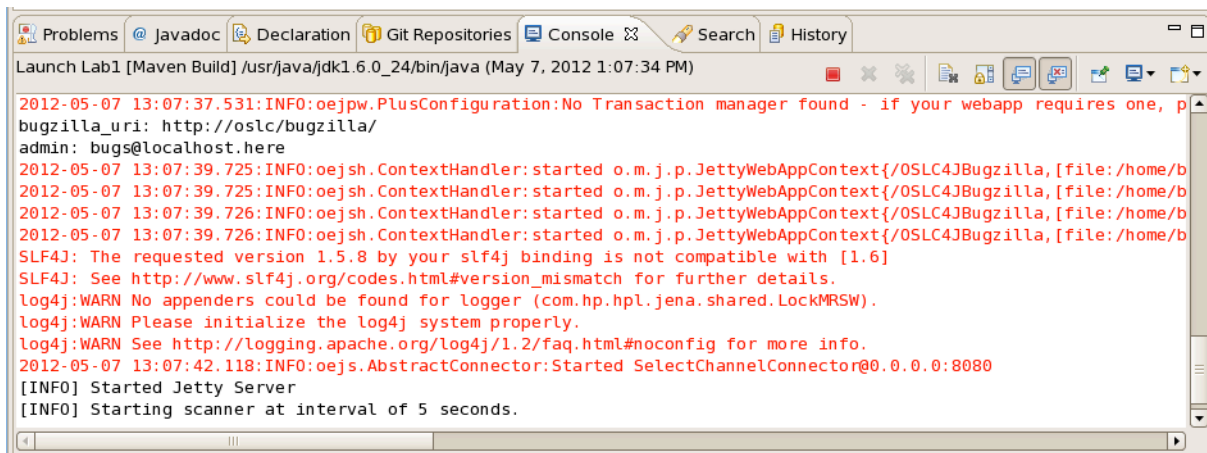
__2.   Validate the application is running by loading http://oslc:8080/OSLC4JBugzilla/services/catalog into the browser (Firefox) . You should see a page that provides some details of the implementation but it is incomplete. If you are prompted for a password, the user id is **bugs@localhost.here** and the password is **bugs4me**
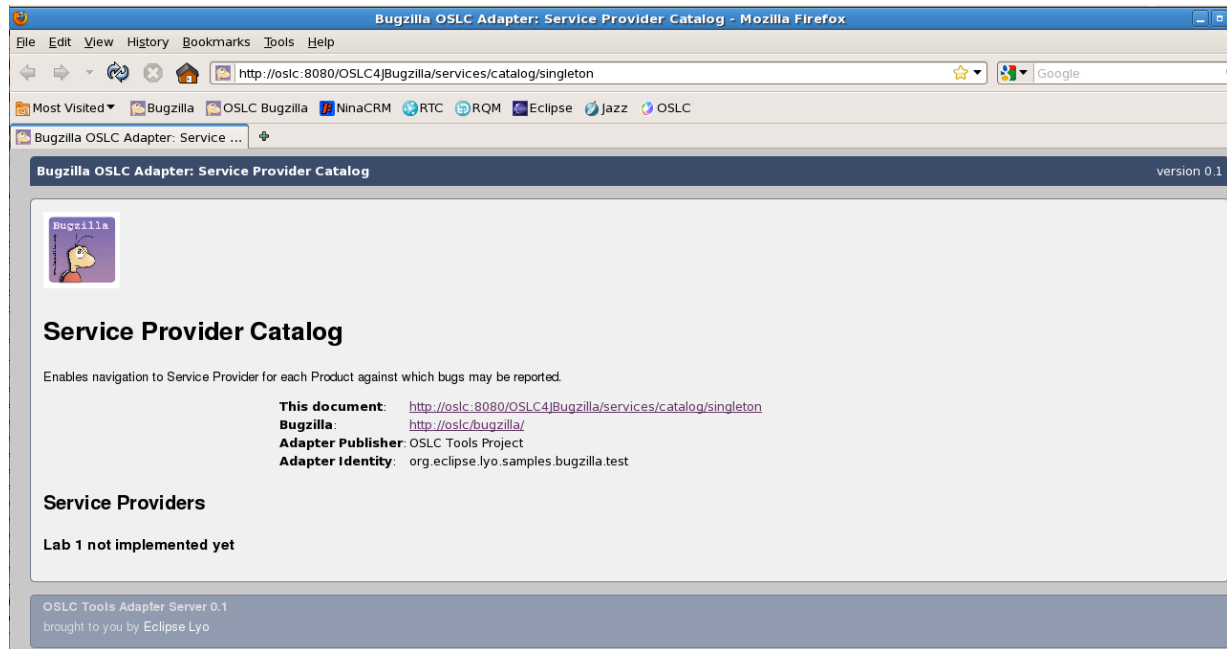
2.0   Get list of products and update catalog

In Bugzilla, bugs are organized by Product. For your adapter implementation, you've decided to represent each Bugzilla Product as an OSLC service provider. In the catalog you will list the service providers, one per Product. To do this, you'll update the catalog resource (**ServiceProviderCatalogSingleton** class) to get the list of Products from the J2Bugzilla API and create a service provider for each. You will then update the JSPs to give you an HTML representation of your catalog and service providers.

__3.   Open the **ServiceProviderCatalogSingleton.java** file in the **org.eclipse.lyo.oslc4j.bugzilla .servlet** package

__a. Update the **initServiceProvidersFromProducts()** method to fetch the list of products from Bugzilla

__b. Note, this should be located immediately following the lines where the BugzillaInitializer is used to get a connection.

__c. Get a list of product ids, such as the following example code which you will find commented out in the lab example file. You can enable the commented code or write the code yourself:

```
GetAccessibleProducts getProductIds = new GetAccessibleProducts();
bc.executeMethod(getProductIds);
```

```
        Integer[] productIds = getProductIds.getIds();


        String basePath = BugzillaManager.getBugzServiceBase();
```

__d. Next, register a new Service Provider for each Bugzilla product

```
        for (Integer p : productIds) {

            String productId = Integer.toString(p);

            if (! serviceProviders.containsKey(productId)) {

                GetProduct getProductMethod = new GetProduct(p);
                bc.executeMethod(getProductMethod);
                String product = getProductMethod.getProduct().getName();


                Map<String, Object> parameterMap =
                        new HashMap<String, Object>();
                parameterMap.put("productId",productId);
                final ServiceProvider bugzillaServiceProvider =
                BugzillaServiceProviderFactory.
                        createServiceProvider(basePath, product, parameterMap);

                registerServiceProvider(basePath,bugzillaServiceProvider,prod
                uctId);
                }

        }
```

The **parameterMap** serves a special purpose. You'll use that to add the Bugzilla productId to the URL of your services. When it is passed to the **createServiceProvider()** method, OSLC4J will use it as part of the service provider and resource URLs. If you want to peek ahead, see how productId is used in **BugzillaChangeRequestService.java.**

- Open the **ServiceProviderCatalogService.java** class in the **org.eclipse.lyo.oslc4j.bugzilla.services** package. No code changes are required right now, but notice how the method which provides an HTML version of the catalog forwards the Java catalog object as an HTTP request attribute to a JSP to produce the HTML:

```
@GET
@Path("{serviceProviderId}")
@Produces(MediaType.TEXT_HTML)
public void getHtmlServiceProvider(@PathParam("serviceProviderId")
                                        final String serviceProviderId)
{
  ServiceProviderCatalog catalog = ServiceProviderCatalogSingleton
                    .getServiceProviderCatalog(httpServletRequest);

  if (catalog !=null )
```

```
    {
        httpServletRequest
            .setAttribute("bugzillaUri", BugzillaManager.getBugzillaUri());
        httpServletRequest
            .setAttribute("catalog",catalog);

        RequestDispatcher rd = httpServletRequest
            .getRequestDispatcher("/cm/serviceprovidercatalog_html.jsp");

        try {
            rd.forward(httpServletRequest, httpServletResponse);
        } catch (Exception e) {
            e.printStackTrace();
            throw new WebApplicationException(e);
        }
    }
```

__4.   Open the **serviceprovidercatalog_html.jsp** file (located in **/src/main/webapp/cm**) to make use of the OSLC catalog information,

  __a. Identify where the catalog local variable is set at the beginning of the JSP:

```
        ServiceProviderCatalog catalog =
    (ServiceProviderCatalog)request.getAttribute("catalog");
```

__5.   Use the local catalog variable, to generate the HTML elements

  __a. Locate within the JSP where to make the changes, namely locate this statement:

```
<h3>Lab1 not implemented yet</h3>
```

  __b. Remove the reminder statement

  __c. Loop through all of the service providers in the catalog and print the Service Provider's title and a link to it. The **getAbout()** method returns the link for any OSLC resource

```
<% for (ServiceProvider s : catalog.getServiceProviders()) { %>
    <h3>Service Provider for Product <%= s.getTitle() %></h3>
    <p><a href="<%= s.getAbout() %>">
            <%= s.getAbout() %></a></p>
<% } %>
```

__6.   Ensure there are no compilation errors, if so…then you've got a little work to do. If not, onward!

__7.   You'll also create an HTML page for the details of a service provider. **ServiceProviderService.java** passes the details of a ServiceProvider to the JSP **serviceprovider_html.jsp** for display

  __a. Browse **ServiceProviderService.java** and see how the **getHtmlServiceProvider()** method passes a specific ServiceProvider object to **serviceprovider_html.jsp**

__2.   Edit **serviceprovider_html.jsp**

    __a. At the top of the JSP, enable the lines that retrieve all of the service Provider details, including URLs for services such as the query capability, creation factory and delegated dialogs.

```
String bugzillaUri = (String) request.getAttribute("bugzillaUri");
Service service = (Service)request.getAttribute("service");
ServiceProvider serviceProvider =
(ServiceProvider)request.getAttribute("serviceProvider");

//OSLC Dialogs
Dialog [] selectionDialogs = service.getSelectionDialogs();
String selectionDialog = selectionDialogs[0].getDialog().toString();
Dialog [] creationDialogs = service.getCreationDialogs();
String creationDialog = creationDialogs[0].getDialog().toString();

//OSLC CreationFactory and shape
CreationFactory [] creationFactories = service.getCreationFactories();
String creationFactory = creationFactories[0].getCreation().toString();
URI[] creationShapes = creationFactories[0].getResourceShapes();
String creationShape = creationShapes[0].toString();

//OSLC QueryCapability and shape
QueryCapability [] queryCapabilities= service.getQueryCapabilities();
String queryCapability = queryCapabilities[0].getQueryBase().toString();
String queryShape = queryCapabilities[0].getResourceShape().toString()
```

    __b. At the bottom of this file, enable the code which generates the HTML for the service provider details and remove the reminder.

__8.   Test your changes

    __a. From Firefox, reload http://oslc:8080/OSLC4JBugzilla/services/catalog and you should now see a list of two Service Providers.   Select the Service Provider URLs to get detailed information on each.
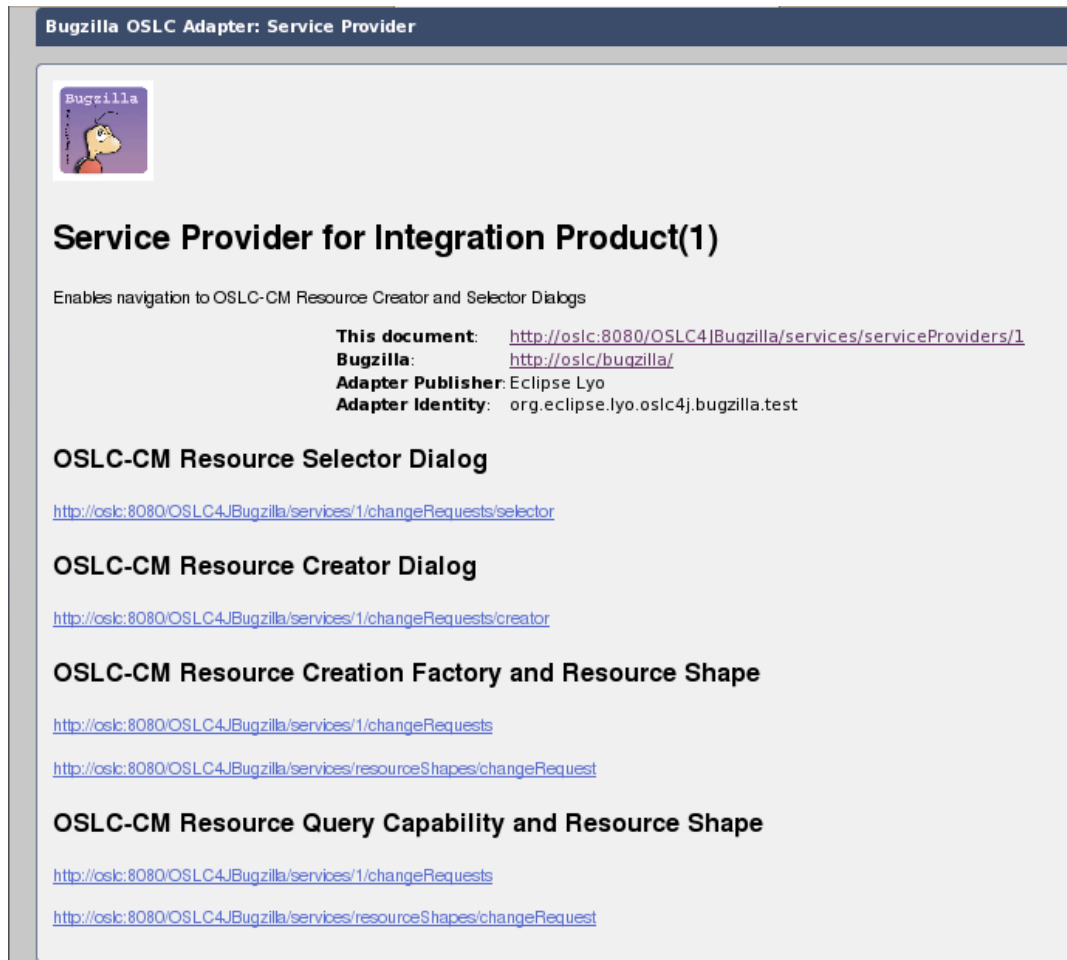
__b.

__c. Hopefully all is well and you get the result as above. If not, check the **Console** view for any errors. Also verify the previous steps. It is also useful to re-launch the server in debug mode and step through the adapter.

2.0 Provide an RDF/XML and JSON Service Provider Catalog representation

The previous section focused on completing the HTML representation of a service provider catalog and service provider resources. This was a useful educational and debugging tool, though in order to connect the adapter to another tool, you'll need to put it in a machine readable format such as RDF/XML or JSON.

__9. Locate and open the **ServiceProviderCatalogService.java** servlet

__10. Locate the **getServiceProviderCatalog()** method which produces RDF/XML, XML and JSON. Add or enable this code.

```
ServiceProviderCatalog catalog =  ServiceProviderCatalogSingleton
      .getServiceProviderCatalog(httpServletRequest);
```

```
    if (catalog != null) {

        httpServletResponse.addHeader(Constants.HDR_OSLC_VERSION,"2.0");
        return catalog;
    }
```

Notice you did not actually add any code to produce RDF or JSON.   What's going on here? OSLC4J includes JAX-RS message body writers capable of serializing the Java representation of the catalog (or any OSLC resource) to RDF/XML, XML and JSON. OSLC4J also handles converting RDF/XML, XML or JSON coming in over the wire to Java objects representing OSLC resources.   More on this **Lab 2**.

__11.   Test current changes

For this you'll use a plugin within Firefox called REST Client to give you control over the requests being sent to your adapter

__a.   Launch REST Client from Firefox by selecting Tools->REST Client

__b.   Enter the **Request URL** to be http://oslc:8080/OSLC4JBugzilla/services/catalog/singleton

__c.   Click **Add Request Header** and add a new header with the name **Accept** and the value **application/rdf+xml**

__d.   Execute the HTTP GET method by clicking the **GET** action

__e.   You will get a response with the Catalog in RDF/XML.
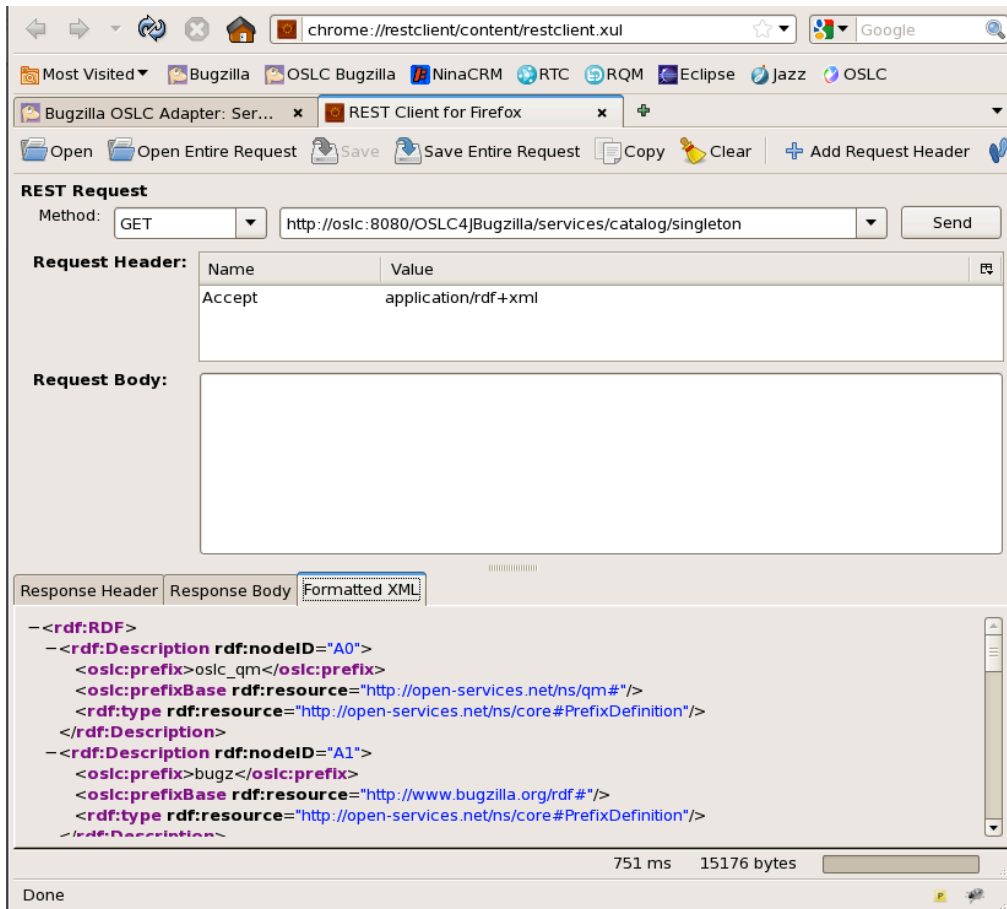
    f.   Try changing the Accept header to **application/xml** or **application/json** and see how the response differs. OSLC4J and JAX-RS produce the correct serialization based on the Accept header.

There are online RDF/XML validators that are useful for additional validation. W3C provides one at http://www.w3.org/RDF/Validator/

2.0 End of Lab1

It is recommended to leave this project behind and pick up with the next project by starting the **Lab2** project

1) Close all open editors (files)

2) Shutdown any running servers (click the red Stop button in the Console window).

3) Optionally, close **Lab1** project

2.0   Summary

You've made good progress on running and testing your adapter. You've accomplished something useful too - you were able to produce a functional OSLC catalog with service providers based on Bugzilla products and provided HTML, RDF/XML, XML and JSON representations for them.

Lab 2　　　　Getting familiar with OSLC4J – OPTIONAL

1. Enable service discovery	→	**2. Get familiar with OSLC4J (optional)**	→	3. Create your first OSLC dialogs	→	4. Enable link traversal and rich hover	→	5. Support automated bug creation	→	6. Connect to Rational Team Concert (optional)
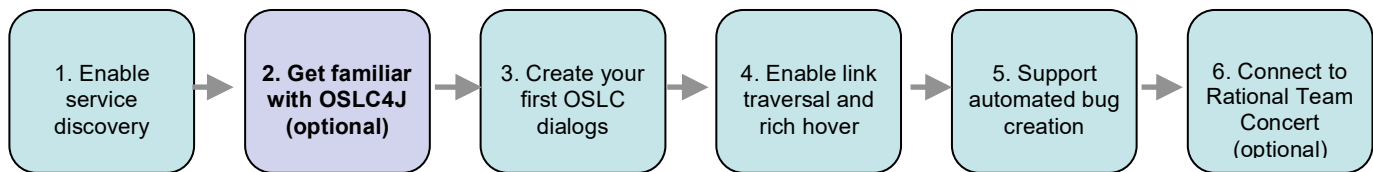
Objectives

- Explore defining OSLC resources using OSLC4J

- Explore using JAX-RS to provide OSLC REST services

- Provide HTML, RDF/XML, XML and JSON representations of OSLC Change Requests which are backed by Bugzilla bugs

Description:

OSLC4J is a Java SDK for developing OSLC provider or consumer implementations. OSLC resources can be modeled with plain old Java objects (POJOs) which are annotated to provide the information OSLC4J needs to create resource shapes, service provider documents, and to serialize/de-serialize OSLC resources from Java to representations such as RDF or JSON.

This lab is optional. If time is constrained, you can proceed to Lab 3.

2.0　Define OSLC Resources

The OSLC Change Management V2 specification defines a Change Request resource.   OSLC4J comes with a sample Change Management application which includes the OSLC4J-annotated Java class representing a Change Request.

For the Bugzilla adapter, you'll include the base **ChangeRequest** straight from OSLC4J and then extend it with some additional Bugzilla-specific attributes like product, platform, operating system, etc.   You'll call this extended change request a **BugzillaChangeRequest**.

Locate the **Lab2** project

__1.　For the first activity in this lab, find the **ChangeRequest.java** and **BugzillaChangeRequest.java** classes in the **org.eclipse.lyo.oslc4j.bugzilla.resources** package and browse them to get familiar with the contents.   The private variables at the top of the **ChangeRequest** class are the attributes of an OSLC CM V2.0 Change Request.   The first several represent the relationships between Change Requests and other OSLC artifacts

```java
private final Set<Link> affectsPlanItems    = new HashSet<Link>();
private final Set<Link> affectsRequirements = new HashSet<Link>();
private final Set<Link> affectsTestResults  = new HashSet<Link>();
```

Further down are some of the primitive attributes of a ChangeRequest

```java
private String   status;
```

```
    private String   title;
    private Boolean  verified;
```

__2.   Scroll down in the **ChangeRequest** class to the **getXYZ()** methods for these attributes. Notice that they are prefaced by some OSLC-specific annotations which provide additional information about the attributes. OSLC4J uses these notations for several purposes.   They are used to automatically create OSLC resource shape documents, service provider documents and service provider catalogs.   They are also used to assist with serialization of Java objects to RDF and JSON, You'll use **getIdentifier()** as an example:

```
    @OslcDescription("A unique identifier for a resource. Assigned by the
service provider when a resource is created. Not intended for end-user
display.")
    @OslcOccurs(Occurs.ExactlyOne)
    @OslcPropertyDefinition(OslcConstants.DCTERMS_NAMESPACE + "identifier")
    @OslcReadOnly
    @OslcTitle("Identifier")
    public String getIdentifier()
    {
        return identifier;

    }
```

> **@OslcOccurs** gives you the cardinality of the attribute, **@OslcPropertyDefinition** providers the namespace qualified attribute name **and @OslcReadOnly** indicates this attribute should appear in the resource shape as read only.   There is no type for this attribute – the default type for OSLC4J is a string. See other example getters which use **@OslcRange** for non-string example

__3.   Now you will look in **BugzillaChangeRequest.java.** Notice that it extends **ChangeRequest** and adds six additional attributes: product, platform, component, version, priority and operating system. Have a look at the getters for these attributes as well for the OSLC annotations.

2.0   Define OSLC services using JAX-RS

The HTTP methods (GET, POST, PUT, DELETE) used by OSLC to interact with resources are defined in 3 classes in the Bugzilla adapter: **ServiceProviderCatalogService, ServiceProviderService and BugzillaChangeRequestService.**   These classes define all of the methods to interact with the Catalog, the ServiceProviders and the BugzilllaChangeRequests using HTTP and the various media formats for each (RDF/XML, XML, JSON and HTML).   The services are implemented as JAX-RS annotated methods.

__4.   Open **BugzillaChangeRequestService.java** in the **org.eclipse.lyo.oslc4j.bugzilla.services** package. By the time you are done, this class will have a lot of methods in it – don't get overwhelmed.   It is structured to deal with collections of BugzillaChangeRequests and individual BugzillaChangeRequests using various HTTP methods (GET, POST, PUT, DELETE) and producing and consuming various formats (RDF/XML, XML, JSON, HTML).   Browse through some of the methods.

__a. Notice the following statement at the top of the class:

```
@Path("{productId}/changeRequests")
```

Recall that in **ServiceProviderCatalogSingleton.java,** you registered a Service Provider for each product. When the Service Provider was created, you put the product ID on the URL path for this Service Provider.   As an example for productId 2, the URL to the collection of changeRequests would be **http://hostname:8080/OSLC4JBugzilla/services/2/changeRequests**

__5.   Four basic services for reading (GET-ing) BugzillaChangeRequests need to be written or enabled: read a collection as RDF/XML, XML or JSON, read a collection as HTML, read a single BugzillaChangeRequest as RDF/XML, XML or JSON and finally, read a single BugzillaChangeRequest as HTML

__a. Add or enable code in two methods to get collections of change requests: **getChangeRequests()** and **getHtmlCollection()**.   The first returns collections of BugzillaChangeRequests as RDF/XML, XML and JSON.   The second forwards the request to a JSP to produce a paged HTML representation of the collection.

```
@GET
@Produces({OslcMediaType.APPLICATION_RDF_XML,
           OslcMediaType.APPLICATION_XML,
           OslcMediaType.APPLICATION_JSON})
public BugzillaChangeRequest[] getChangeRequests(
                          @PathParam("productId")final String productId,
                          @QueryParam("oslc.where") final String where,
                          @QueryParam("page") final String pageString)
                              throws IOException, ServletException
{

  int page=0;
  int limit=999;


     final List<Bug> bugList = BugzillaManager
        .getBugsByProduct(httpServletRequest, productId, page, limit);
     final List<BugzillaChangeRequest> results =
        changeRequestsFromBugList(httpServletRequest, bugList, productId);

     return results.toArray(new BugzillaChangeRequest[results.size()]);

  }
```

This method simply calls two utility methods – one to get all of Bugzilla bugs for this **productId** and another to convert the bugs to an array of OSLC **BugzillaChangeRequests** and returns the array.

OSLC4J's JAX-RS message body writers will take care of serializing the array to the correct representation. If you have time, browse the utility methods as well.

__6.   Once all four methods in Lab 2 have been enabled in the **BugzillaChangeRequest** class, you can go to the corresponding JSPs for the HTML representations.

__a. Edit **changerequest_collection_html.jsp**. This JSP is passed a Java **List** of **BugzillaChangeRequests**. You need to add a table and then loop through the List and output the title and link for each item. Add or enable the following code to the JSP:

```
<div id="header">
   <div id="banner"></div>
      <table border="0" cellspacing="0" cellpadding="0" id="titles">
         <tr>
            <td id="title">
               <p>
                   Bugzilla OSLC Adapter: Service Provider
               </p>
            </td>
            <td id="information">
               <p class="header_addl_info">
                  version 0.1
               </p>
            </td>
         </tr>
      </table>
</div>
<div id="bugzilla-body">
   <div id="page-index">
       <img src="../../images/resources/bugzilla.gif" alt="icon"
width="80" height="80" />
       <h1>Query Results</h1>
       <% for (BugzillaChangeRequest changeRequest : changeRequests) { %>
          <p>Summary: <%= changeRequest.getTitle() %><br /><a href="<%=
changeRequest.getAbout() %>">
             <%= changeRequest.getAbout() %></a></p>
             <% } %>
          <% if (nextPageUri != null) { %><a href="<%= nextPageUri
%>">Next Page</a>
       <% } %>

   </div>
</div>
```

__7.   Test the changes for collections to make sure they are working well

__a. Start the Adapter (Run -> Run Configurations and run **Launch Lab2**).

__b. Test the HTML representation of a collection of BugzillaChangeRequests for the service provider for productId = 1:

   __i. In Firefox, go to **http://oslc:8080/OSLC4JBugzilla/services/1/changeRequests**

  __c. Test a non-HTML representation

   __i. In Firefox, go to Tools->REST Client

   __ii. Click Add Request Header and add a header with the name **Accept** and a value of either **application/rdf+xml, application/xml** or **application/json**

   __iii. Enter **http://oslc:8080/OSLC4JBugzilla/services/1/changeRequests** as the URL

__8. Now for the single BugzillaChangeRequest services. Add or enable the code for **getChangeRequest() and getHtmlChangeRequest().** These methods provide the representations for single change requests. The @Path(`"{changeRequestId}"`) annotation indicates that JAX-RS will pass the last item on the URI path into the changeRequestId parameter.

  __a. For a single change request as RDF/XML, XML and JSON, you'll get the Bug using j2Bugzilla and then call the **BugzillaChangeRequest.fromBug()** utility method to convert it to a BugzillaChangeRequest.   You'll have set the resource URI ("about") and serviceProvider URI explicitly:

```
@GET
@Path("{changeRequestId}")
@Produces({OslcMediaType.APPLICATION_RDF_XML,
OslcMediaType.APPLICATION_XML, OslcMediaType.APPLICATION_JSON})
    public BugzillaChangeRequest getChangeRequest(@PathParam("productId")
                                            final String productId,
          @PathParam("changeRequestId") final String changeRequestId)
          throws IOException, ServletException, URISyntaxException
  {
      final Bug bug = BugzillaManager
                    .getBugById(httpServletRequest, changeRequestId);

      if (bug != null) {
         BugzillaChangeRequest changeRequest = null;

         changeRequest = BugzillaChangeRequest.fromBug(bug);

         changeRequest.setServiceProvider(ServiceProviderCatalogSingleton
               .getServiceProvider(httpServletRequest,
                                 productId).getAbout());
         changeRequest
           .setAbout(getAboutURI(productId + "/changeRequests/" +
                       changeRequest.getIdentifier()));
          return changeRequest;
      }
     throw new WebApplicationException(Status.NOT_FOUND);
   }
```

  __b. For a single BugzillaChangeRequest as HTML, the adapter will just redirect to Bugzilla to

---

display the Bug natively:

```
@GET
@Path("{changeRequestId}")
@Produces({ MediaType.TEXT_HTML })
public Response getHtmlChangeRequest(@PathParam("productId")
                                          final String productId,
        @PathParam("changeRequestId") final String changeRequestId)
        throws IOException, URISyntaxException
{
    String forwardUri = BugzillaManager.getBugzillaUri() +
                        "show_bug.cgi?id=" + changeRequestId;
    httpServletResponse.sendRedirect(forwardUri);
    return Response.seeOther(new URI(forwardUri)).build();
}
```

__9.   **Save**. You've now enabled the code which gives you HTML representations for single change requests.   Time for the final tests of collections and single change requests:

__a. Start the **Lab 2** server if not already started: Run->Run Configurations and run the **Launch Lab 2** configuration.

__b. Test the HTML representations for the service provider for productId = 1:

__i.   Collection: **http://oslc:8080/OSLC4JBugzilla/services/1/changeRequests**

__ii.   Single: Click on one of the change requests in the collection.
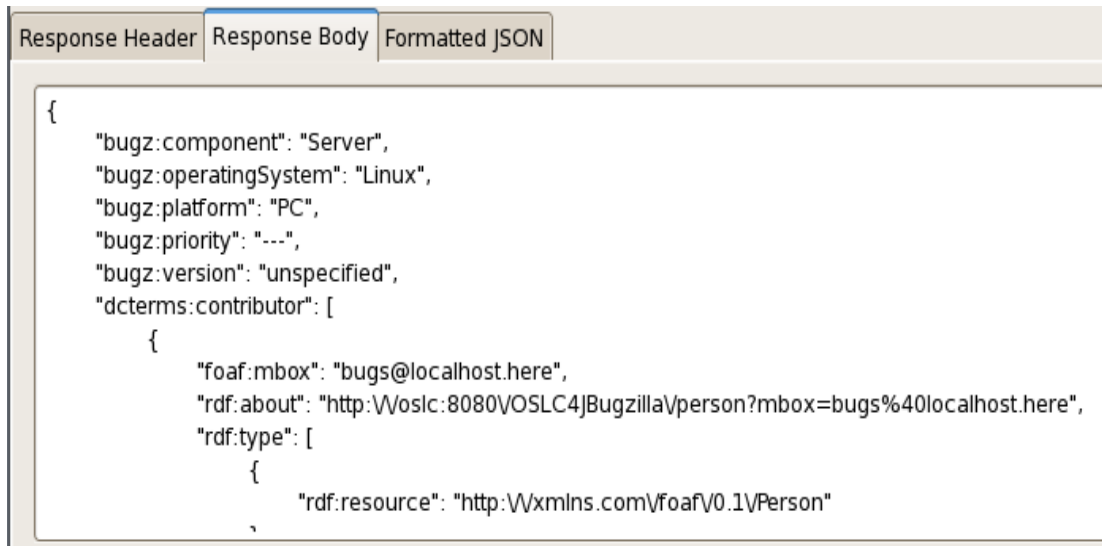
__c. Test a non-HTML representation

__i.   In Firefox, go to Tools->REST Client

__ii.   Click Add Request Header and add a header with the name **Accept** and a value of either **application/rdf+xml, application/xml** or **application/json**

__iii.   Use the url **http://oslc:8080/OSLC4JBugzilla/services/1/changeRequests/37** to see bug 37 in product 1.

__iv. View the response in the Response Body tab.   For JSON, it should look like this:

```json
{
    "bugz:component": "Server",
    "bugz:operatingSystem": "Linux",
    "bugz:platform": "PC",
    "bugz:priority": "---",
    "bugz:version": "unspecified",
    "dcterms:contributor": [
        {
            "foaf:mbox": "bugs@localhost.here",
            "rdf:about": "http:\/\/oslc:8080\/OSLC4JBugzilla\/person?mbox=bugs%40localhost.here",
            "rdf:type": [
                {
                    "rdf:resource": "http:\/\/xmlns.com\/foaf\/0.1\/Person"
                }
```

2.0   End of Lab2

It is recommended to leave this project behind and pick up with the next project by starting the **Lab3** project

1) Close all open editors (files)

2) Shutdown any running servers (click the red Stop button in the Console window).

3) Optionally, close **Lab1** project

2.0   Summary

This lab provided an introduction to defining OSLC resources and REST services using the Eclipse Lyo OSLC4J SDK (http://eclipse.org/lyo). You now have the basics for exposing Bugzilla bugs as OSLC BugzillaChangeRequests and you are able to retrieve them in a variety of representations which can be used by humans and other tools. In the next lab you'll see how OSLC can let you create or select these change requests from the NinaCRM tool.

Lab 3   Creating your first OSLC dialogs

| 1. Enable service discovery | 2. Get familiar with OSLC4J (optional) | **3. Create your first OSLC dialogs** | 4. Enable link traversal and rich hover | 5. Support automated bug creation | 6. Connect to Rational Team Concert (optional) |
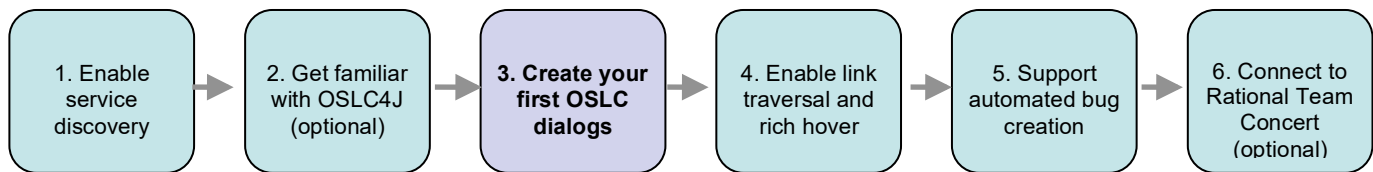
Objectives

- Explore additional aspects of implementing an OSLC provider

- Build delegated user interfaces as a way to provide a simple way for consumer applications to create and select bugs

- Expose this capability from your service provider resource definition

Description:

You have completed service description resources that enable consuming tools to discover your application and you have basic resource and services defined for OSLC BugzillaChangeRequests. Now you'll implement an OSLC Delegated UI to provide a dialog that can be used to create or select and link to bugs from any OSLC consumer. For example, a user of the NinaCRM product will be able to link CRM Incident records to Bugzilla bugs without leaving the NinaCRM web interface. For more information on Delegated UI, take a look at the OSLC Core specification section on the topic here:

http://open-services.net/bin/view/Main/OslcCoreSpecification#Delegated_User_Interface_Dialogs

In order to achieve this, you'll need to break down your work into the following steps:

1. Understand how to define a basic dialog

    a. Use the J2Bugzilla API to customize the dialog contents

    b. Generate appropriate responses

2. See how OSLC4J helps provide delegated UI locations to the Service Provider document

3. Add methods to **BugzillaChangeRequestService** for the selection and creation of change requests

4. Add HTML forms and JavaScript code to handle interacting with the consumer of the dialogs

5. Test the dialogs in a sample app to ensure the appropriate response is given

2.0 Creation dialog updates

Providing a Delegated UI involves creating an HTML Form, the fields within that form, and then the server-side handling of the form submission.

Locate the **Lab3** project

__1.   First, create a utility method that lets you create a Bugzilla bug from an OSLC BugzillaChangeRequest

    __a.   Locate **BugzillaManager.java** in the **org.eclipse.lyo.oslc4j.bugzilla** package and browse the code.   This class contains several static utility methods for interacting with Bugzilla using the j2bugzilla library. **BugzillaChangeRequestService.java** makes use extensive use of the methods in this class

    __b.   Find the commented out **createBug()** method, and add or enable code to create a Bugzilla bug from an OSLC BugzillaChangeRequest. First, retrieve the bug properties from the BuzillaChangeRequest and set some defaults for any missing fields.

```java
            final int productId = Integer.parseInt(productIdString);

            final BugzillaConnector bc = BugzillaManager
                        .getBugzillaConnector(httpServletRequest);

            GetProduct getProducts = new GetProduct(productId);
            bc.executeMethod(getProducts);

            final Product product = getProducts.getProduct();

            String summary = changeRequest.getTitle();
            String component = changeRequest.getComponent();
            String version = changeRequest.getVersion();
            String operatingSystem = changeRequest.getOperatingSystem();
            String platform = changeRequest.getPlatform();
            String description = changeRequest.getDescription();

            BugFactory factory =
                new BugFactory().newBug().setProduct(product.getName());

            if (summary != null) {
                factory.setSummary(summary);
            }
            if (version != null) {
                factory.setVersion(version);
            }
            if (component != null) {
                factory.setComponent(component);
            }
            if (platform != null) {
                factory.setPlatform(platform);
            } else
                factory.setPlatform("Other");
```

```
        if (operatingSystem != null) {
            factory.setOperatingSystem(operatingSystem);
        } else
            factory.setOperatingSystem("Other");

        if (description != null) {
            factory.setDescription(description);
        }
```

__c. Then call j2bugzilla's method to create a bug and report it

```
        Bug bug = factory.createBug();
        ReportBug reportBug = new ReportBug(bug);
        bc.executeMethod(reportBug);

        newBugId = Integer.toString(reportBug.getID());
```

__2. Locate **BugzillaChangeRequestService.java** in the **org.eclipse.lyo.oslc4j.bugzilla.services** package and explore its contents. Find the commented out **Lab 3** method **createHtmlChangeRequest()** . This method will handle the HTML request from the browser, query Bugzilla for valid field values for a new Bug and forward the request to a JSP to present the form to the user.

__a. Add or enable code in **changeRequestCreator()** to use the j2Bugzilla **GetLegalValues** API to retrieve the legal values for fields and pass them to the JSP

```
BugzillaConnector bc =
    BugzillaManager.getBugzillaConnector(httpServletRequest);

Product product = BugzillaManager.getProduct(httpServletRequest, productId);
httpServletRequest.setAttribute("product", product);

GetLegalValues getComponentValues =
    new GetLegalValues("component", product.getID());
bc.executeMethod(getComponentValues);
List<String> components = Arrays.asList(getComponentValues.getValues());
httpServletRequest.setAttribute("components", components);
```

This pattern repeats for the platform, operating system, version and status attributes

__b. Also add or enable the code in the **createHtmlChangeRequest()** method just below. This code will be called when the user submits the form with the info needed to create a new Bugzilla bug.

__3. Locate **changerequest_creator.jsp** and explore its contents. This JSP is used to generate the creation dialog

__a. Find the appropriate location in the file to add selection lists for: components, versions, operating systems and platforms

---

__b.   Now add an HTML table with selection entries for each field in the form. In <select> statements which populate selectors with the legal values passed in from the Java service.   Example for Component:

```
 <tr>
 <td>Component: </td>
   <td>
   <select name="component">
   <% for (String c : components) { %>
     <option value="<%= c %>"><%= c %></option>
     <% } %>
   </select>
  </td>
 </tr>
```

__c.   Repeat for the remaining attributes (or uncomment the table code in the JSP). Save your changes. Notice the Submit button will call a JavaScript method called **create()** when pressed.   This method is in **bugzilla.js.**

__4.   Locate **bugzilla.js** (in the webapp folder) and browse its contents.   The **create()** method is called by the JSP when the **Submit** button is pressed**. create()** sends the form data back to **BugzillaChangeRequestService's createHtmlChangeRequest()** method to create the new bug.

__a.   Add or enable code in the **create()** method which posts the form data back to **BugzillaChangeRequestService** .

Save your changes and test

__b.   Go to Run->Run Configurations and Run **Launch Lab3**

__c.   In Firefox, go to http://oslc:8080/OSLC4JBugzilla/services/serviceProviders/1.

__d.   Click the Resource Creator Dialog for this service provider.   You should see your creation form and be able to create a new Bug.

__e.   Notice that when selecting Create, a new bug is created within the Bugzilla server. Feel free to navigate to the native Bugzilla Web UI and locate this newly created bug.

  The creation form does not disappear when you press Submit right now. You still need a consumer that knows how to handle events when resources are created or selected and will tackle that later in this lab.

2.0   OSLC4J and delegated UIs

A quick detour – how did the Creation Dialog, Creation Factory, Selection Dialog and Query Capability dialogs get into the service provider document back in lab 1? If you want to refresh your memory, you can point Firefox at http://oslc:8080/OSLC4JBugzilla/services/serviceProviders/1 . OSLC4J provides annotations to help create those URLs. You still have to create the code to implement the delegated UIs, but OSLC4J can help your service provider advertise them.

__5.   Open **BugzillaChangeRequestService.java** and look for **@OslcDialogs**.   You'll see two occurrences – one near the top for the Selection Dialog and Query Capability and one farther down for Creation Factory and Creation Dialog.

Since you're going to create a selection dialog next, here are the OSLC4J annotations for it:

```
@OslcDialogs(
    {
        @OslcDialog
        (
            title = "Change Request Selection Dialog",
            label = "Change Request Selection Dialog",
            uri = "/{productId}/changeRequests/selector",
            hintWidth = "450px",
            hintHeight = "300px",
            resourceTypes = {Constants.TYPE_CHANGE_REQUEST},
            usages = {OslcConstants.OSLC_USAGE_DEFAULT}
        )

    })
    @OslcQueryCapability
    (
        title = "Change Request Query Capability",
        label = "Change Request Catalog Query",
        resourceShape = OslcConstants.PATH_RESOURCE_SHAPES + "/" +
Constants.PATH_CHANGE_REQUEST,
        resourceTypes = {Constants.TYPE_CHANGE_REQUEST},
        usages = {OslcConstants.OSLC_USAGE_DEFAULT}

    )
```

2.0   Update selection dialog to generate events

You have had a look at the creation dialog, so now you will look at the selection dialog. You'll want to update this dialog as you already have this dialog but it does not generate events as defined by OSLC.

__6.   Optionally, close any previously opened files in the editors

__7.   Locate **BugzillaChangeRequestService.java** and find the commented out method called **changeRequestSelector()**. Notice **the @Path("selector")** for this method. That means the URL will contain "**changeRequests/selector**" to load the UI. This method serves two purposes:

__a.   When called without a "**terms**" parameter on the URL, it forwards the request to the delegated UI JSP to display a form for the user to fill in the search terms

__b.   When called with a "**terms**" parameter on the URL from the delegated UI, it performs a Bugzilla search and returns the results in an **oslc:results** response.

__c.   Enable the code in **changeRequestSelector()** and save.

Locate **changerequest_selector.jsp** and explore the contents

__d. Enable the code that creates the search form. Note that it calls the **bugzilla.js select()** method when OK is selected:

```
<div style="width: 400px; margin-top: 5px;">
    <button style="float: right;" type="button"
        onclick="javascript: cancel()">Cancel</button>
    <button style="float: right;" type="button"
        onclick="javascript: select();">OK</button>

</div>
```

__8.   Now locate **bugzilla.js** again and browse the code

__a. You need to add a method to generate either a Window Name response (**#oslc-core-windowName-1.0)** response or a Post Message response (**#oslc-core-postMessage-1.0)** response depending on which protocol your client is using.

```
function sendResponse(label, url) {

  var oslcResponse = 'oslc-response:{ "oslc:results": [ ' +
    ' { "oslc:label" : "' + label + '", "rdf:resource" : "' +
    url + '"} ' + ' ] }';

   if (window.location.hash == '#oslc-core-windowName-1.0') {
  // Window Name protocol in use
    respondWithWindowName(oslcResponse);
   } else if (window.location.hash == '#oslc-core-postMessage-1.0') {
  // Post Message protocol in use
    respondWithPostMessage(oslcResponse);
   }


}
```

> **Delegated UI Protocol Selection**
>
> It is up to the consumer to indicate to you (the provider) which protocol to use. This is often dependent on which browser this application is running and what it supports. The postMessage method is the preferred method.

__9.   Now you will implement the **select()** method that will make use of this.

__a. Find the user's selection and call the **sendResponse()** method you just implemented.

```
function select(){

    list = document.getElementById("results");
    if( list.length>0 && list.selectedIndex >= 0 ) {
        option = list.options[list.selectedIndex];
        sendResponse(option.text, option.value);
    }
  }
```

__b.   Save your changes, check for errors

__10.   Test the selection dialog by placing the URL in the browser
http://oslc:8080/OSLC4JBugzilla/services/1/changeRequests/selector Try searching for bugs
with the string "3rd" in them.

__11.   Extra credit, use Firebug (a Firefox plugin) accessible from Tools->Firebug to walk through the
JavaScript code that you just developed.

__12.   Test selection dialog with NinaCRM
This is a sample OSLC consumer application that has some hard-coded linkages to your bugzilla
adapter

__a.   Run->Run Configurations and select **Launch NinaCRM**

__b.   Select the **NinaCRM** bookmark from the Firefox toolbar

__c.   Select the button "Select Defect to Link to…"

__d.   Fill in the search criteria, suggest to use the term "3rd"

__e.   Select a bug and then "ok"

__f.   You should see the selection you made added to the list

You can also test delegated bug creation from **NinaCRM**

2.0   End of Lab3

It is recommended to leave this project behind and pick up with the next project by starting the **Lab4**
project

1)   Close all open editors (files)

2)   Shutdown any running servers (click the red Stop button in the Console window).

3)   Optionally, close **Lab3** project

2.0   Summary

This lab explored many aspects of implementing an OSLC provider. You now have the ability to use
user interface delegation as a way to provide a simple way for consumer applications to create and
select bugs. You've also exposed this capability from your service provider resource definition. Now that
you have links, you'll explore more how you can access those links and preview resource information.

Lab 4   Enabling link traversal and rich hover

| 1. Enable service discovery | 2. Get familiar with OSLC4J (optional) | 3. Create your first OSLC dialogs | **4. Enable link traversal and rich hover** | 5. Support automated bug creation | 6. Connect to Rational Team Concert (optional) |
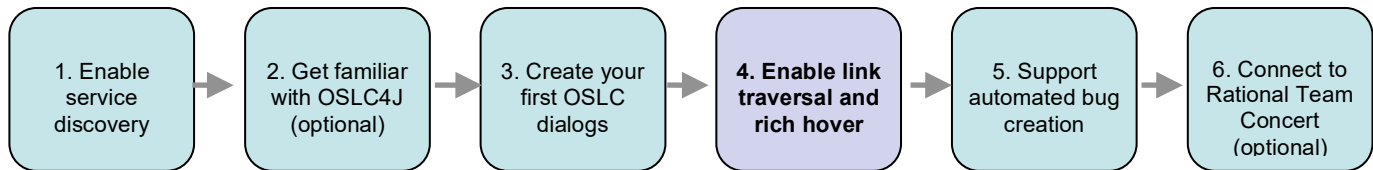
Objectives:

- Take a link represented as a URI and provide some human-readable and usable presentations for that link

- Understand how to present a quick peak into the Bug using UI Preview

Description:

Once you establish relationships between resources, you can enable a very useful form of integration known as UI Preview. When a user is viewing a resource in a web browser, they might see a list of links to related resources. UI Preview makes it easy for them to learn about those resources in context and without leaving the web page that they are looking at. When users "hover" over a link with their mouse, they can see a brief preview of that resource in a tool-tip or a pop-up window.

For all of the details of UI Preview, the protocol and resources involved, take a look at the OSLC UI Preview specification http://open-services.net/bin/view/Main/OslcCoreUiPreview. It's short and fairly easy to understand, even if you don't know much about OSLC.

In this section, you'll explore how to make the Adapter into a UI Preview Provider, so that when a user sees a link to Change Requests, they can see a UI Preview as long as the application that is displaying that link is a UI Preview Consumer. At the end of this lab, you'll be able to see your UI Preview in the NinaCRM example application, because it supports UI Preview as a Consumer.

Steps:

1. Add UI Preview handling to BugzillaChangeRequestService

   a. Add logic to detect the UI Preview Content-type (application/x-oslc-compact+xml)

   b. Dispatch to a JSP page to render the HTML preview

2. Create JSP to render HTML for a UI Preview

2.0   Understand Project Structure

Just as you did in the previous labs, you will explore the Lab 4 structure and do some basic tests.

__1.   Open the first lab project named **Lab4**

__2.   Explore the contents of the project, namely the JAX-RS services and the JSPs

__3.   Verify there are no compiler or configuration errors

2.0   Add UI Preview handling to BugzillaChangeRequestService

In this lab, you already have a **BugzillaChangeRequestService** class that can handle HTML requests for collections of **BugzillaChangeRequests** or individual **BugzillaChangeRequests** (Lab 2). It also has the capability to provide delegated selection and creation UIs (Lab 3)

To add UI Preview support, you will add two methods to the service. The first will provide an OSLC Compact Representation of a **BugzillaChangeRequest**. The second will provide what is known as a small HTML preview of a **BugzillaChangeRequest**.

__4.   Open the **BugzillaChangeRequestService.java** file in the
   org.eclipse.lyo.oslc4j.bugzilla.services package.

Find the **getCompact()** method

  __a. Note the **@Produces** annotation

```
@Produces({OslcMediaType.APPLICATION_X_OSLC_COMPACT_XML})
```

      As with other media types, OSLC4J will handle serialization to the correct media type.

  __b. Add code to build a Compact representation of a **BugzillaChangeRequest**. First fetch the bug and convert to a full **BugzillaChangeRequest**. Then, copy the "about" and "title" attributes. You'll also add a URL to the Bugzilla icon on the Bugzilla server to the compact representation. Finally you'll build a **Preview** object pointing to your **smallPreview()** service in **BugzillaChangeRequestService** and add the **Preview** to the **Compact** object and return the **Compact** object to OSLC4J

```
final Bug bug = BugzillaManager.getBugById(httpServletRequest,
                                           changeRequestId);
if (bug != null) {
    final Compact compact = new Compact();
    BugzillaChangeRequest changeRequest = null;

    changeRequest = BugzillaChangeRequest.fromBug(bug);

    compact.setAbout(getAboutURI(productId + "/changeRequests/" +
                changeRequest.getIdentifier()));
    compact.setTitle(changeRequest.getTitle());
    String iconUri = BugzillaManager.getBugzillaUri() +
                "/images/favicon.ico";

    compact.setIcon(new URI(iconUri));
      //Create and set attributes for OSLC preview resource
        final Preview smallPreview = new Preview();
```

```
        smallPreview.setHintHeight("11em");
        smallPreview.setHintWidth("45em");
        smallPreview.setDocument(new URI(compact.getAbout().toString() +
                                "/smallPreview"));
        compact.setSmallPreview(smallPreview);
        //Use the HTML representation of a change request as the large
preview as well
        final Preview largePreview = new Preview();
        largePreview.setHintHeight("20em");
        largePreview.setHintWidth("45em");
        largePreview.setDocument(changeRequest.getAbout());
        compact.setLargePreview(largePreview)
        return compact;

    }
```

__3.   Make sure your new code compiles, and restart the Adapter.   Run the adapter using Run->Run Configurations and running **Launch Lab4**

__4.   Now, you will verify that the **getCompact()** service is working. Try using the REST Client tool to access the UI Preview descriptor for a Bugzilla bug via the adapter, for example try the URL below.

http://oslc:8080/OSLC4JBugzilla/services/1/changeRequests/1

Set the Accept header to **application/x-oslc-compact+xml** and click the GET button. Assuming you have bug #1 in Bugzilla, you should see something like the below screenshot:
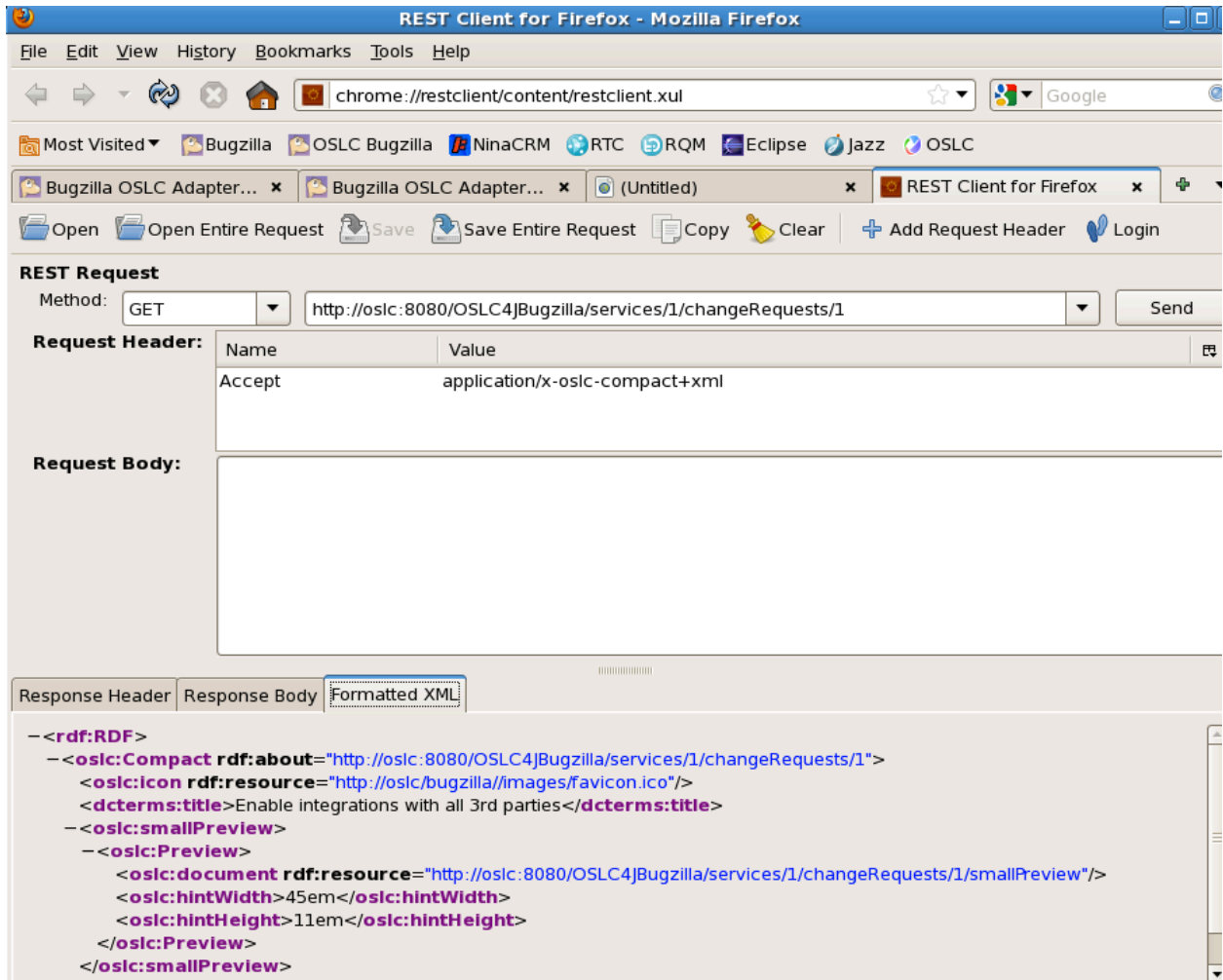
JAX-RS sent the GET request to the **getCompact()** method based on the Accept header. Examine the output to see how the **oslc:smallPreview** and **oslc:largePreview** resources are defined in the **oslc:Compact** resource.   In your Java code, you needed to create the **Compact** and **Preview** objects and then let OSLC4J take care of serializing them to RDF.

Consumers wanting to display a small or large preview of a **BugzillaChangeRequest** can find the URLs to them using **the x-oslc-compact+xml** representation.   Let's provide the HTML for the preview and test it out. Now you will enable the small and large previews.

2.0 Create a UI Preview JSP and dispatch to it

In the previous section, you added a new method to **BugzillaChangeRequestService.java** called **getCompact()** which created a **Preview** resource pointing to changeRequests/{id}/smallPreview. First you'll create a method to handle HTTP requests to that URL for HTML content.

__5.   Update the Java code to forward requests for previews to the JSPs

__a. Open **BugzillaChangeRequestService.java** and find the commented out **smallPreview()** method.   Add or enable code to retrieve the Bugzilla bug, convert it to a BugzillaChangeRequests and forward it to your JSP:

```java
final Bug bug = BugzillaManager
                 .getBugById(httpServletRequest, changeRequestId);

if (bug != null) {
   BugzillaChangeRequest changeRequest =
                 BugzillaChangeRequest.fromBug(bug);
   //Set the service provider from the catalog
   changeRequest
         .setServiceProvider(ServiceProviderCatalogSingleton
         .getServiceProvider(httpServletRequest, productId)
         .getAbout());
   //Set the "about" for this change request
   changeRequest.setAbout(getAboutURI(productId +
         "/changeRequests/" + changeRequest.getIdentifier()));

   final String bugzillaUri =
         BugzillaManager.getBugzillaUri().toString();
   httpServletRequest.setAttribute("changeRequest", changeRequest);
   httpServletRequest.setAttribute("bugzillaUri", bugzillaUri);

   RequestDispatcher rd = httpServletRequest
       .getRequestDispatcher("/cm/changerequest_preview_small.jsp");
   rd.forward(httpServletRequest,httpServletResponse);
   return;

}
```

__b. Repeat this pattern for **largePreview()**, but forwarding to **changerequest_preview_large.jsp**

__6.   Update your JSPs to present the small and large preview

__a. Locate and open **changerequest_preview_small.jsp** and browse the contents. The code at the top extracts the fields you want to display from the BugzillaChangeRequest

__b. Add or enable the code which creates a table and displays the fields and Save

__c. Repeat for **changerequest_preview_large.jsp**

__7.   Make sure your new code compiles and restart the Adapter.

__8.   Start the NinaCRM example, as described in Lab 2. Navigate to the main page of NinaCRM by clicking the NinaCRM shortcut on the toolbar and use the Delegated UI to create or select a bug in Bugzilla. Once you've done that and you see the link in the page hover over it and you should see your preview, something like what's shown below:
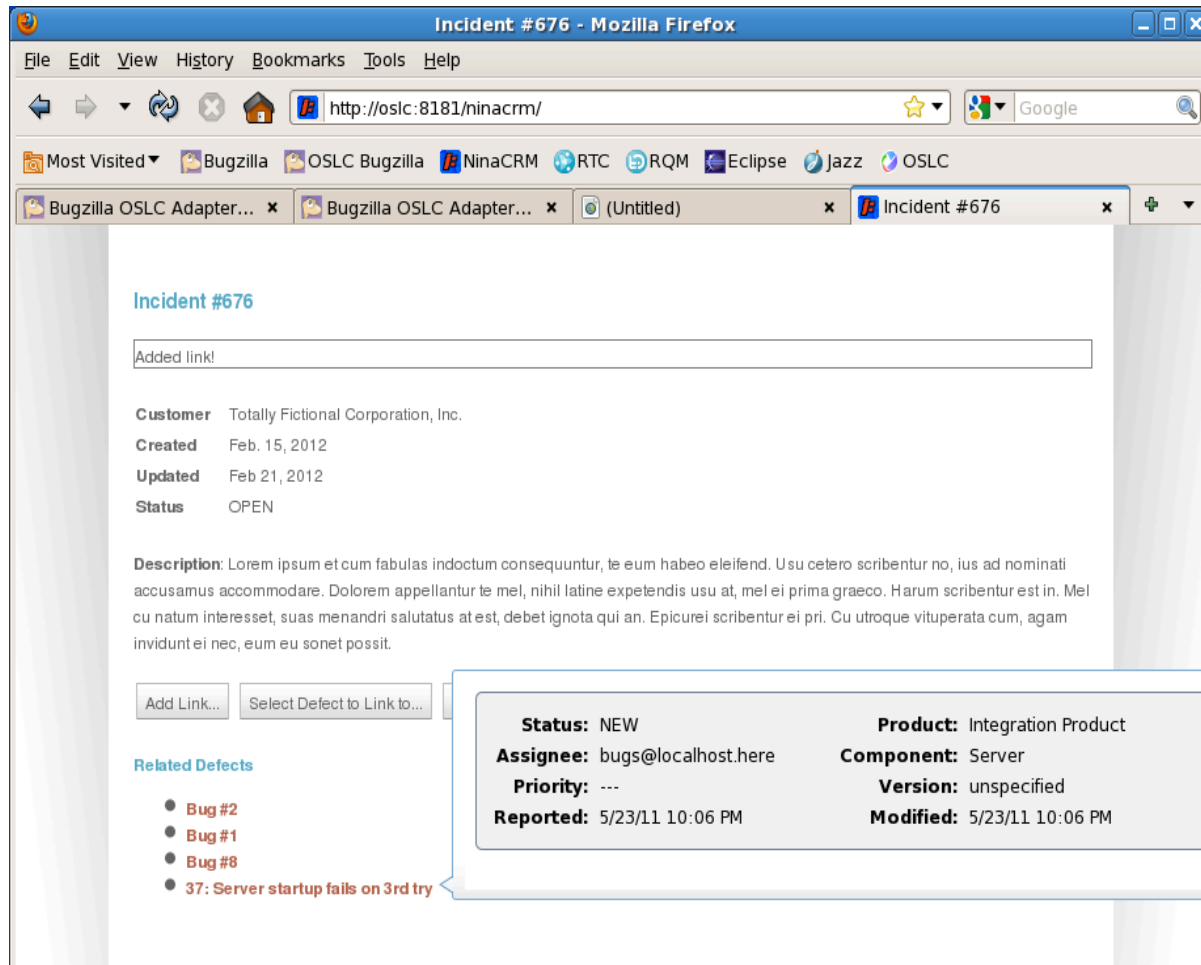
2.0   End of Lab4

It is recommended to leave this project behind and pick up with the next project by starting the **Lab5** project

1) Close all open editors (files)

2) Shutdown any running servers (click the red Stop button in the Console window).

3) Optionally, close **Lab4** project

2.0   Summary

In this lab you learned how to take a link represented as a URI and provide some human-readable and usable presentations for that link. You highlighted how to use the full HTML representation for the Bug and how to present a quick peak into the Bug using the UI Preview provided by the service provider when the compact representation (**application/x-oslc-compact+xml)** of a Bug is requested.

Lab 5   Supporting automated bug creation

1. Enable service discovery	2. Get familiar with OSLC4J (optional)	3. Create your first OSLC dialogs	4. Enable link traversal and rich hover	**5. Support automated bug creation**	6. Connect to Rational Team Concert (optional)
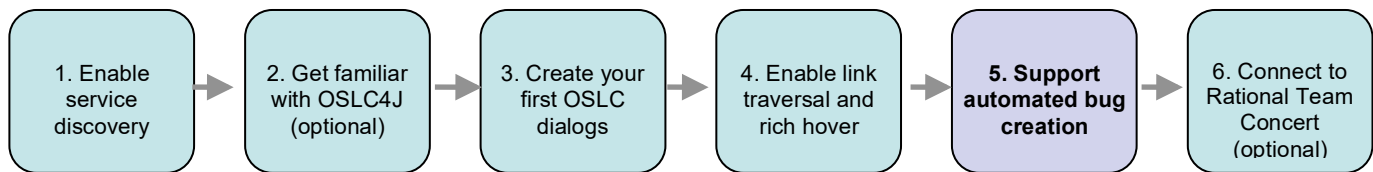
Objectives:

Enable tools to programmatically create bugs using the HTTP POST method using a simple RDF/XML representation of a bug

Description:

With OSLC, you can enable bug creation via Delegated UI, but like all UI approaches, a user must be involved. What if you want to support automated bug creation; for example, enabling a build server to file a bug whenever there is a test or a build failure? To support automated bug creation, you need to support an OSLC Creation Factory as described in the OSLC Core specification. That's what you'll do in this lab.

OSLC4J takes care of adding the Creation Factory to the service provider document based on the annotations. You'll implement the code to actually create the new bugs.

Steps:

- Explore OSLC4J's ability to indicate which JAX-RS method represents the Creation Factory for inclusion in the Service Provider document

- Support HTTP POST of BugzillaChangeRequests in RDF/XML, XML or JSON formats

Recall that in Lab 3, you enabled code in **BugzillaManager.java** to use the j2bugzilla API for creation of bugs using your delegated creation UI.   You'll re-use the **createBug()** method for automated bug creation via POST.

2.0   Understand Project Structure

As you've done in the previous labs, you will explore the Lab 5 structure and do some basic tests.

__1.	Open the first lab project named **Lab5**

Verify there are no compiler or configuration errors

2.0   Add a method to the adapter to create BugzillaChangeRequests via POST

To enable clients to find the Creation Factory URL to use for creating bugs within

---

__2.   Open the **BugzillaChangeRequestService.java** in the **org.eclipse.lyo.bugzilla.services** package and browse the code.   Find the **addChangeRequest()** method.

__a.   Note the OSLC4J annotations before the method declaration. **The @OslcCreationFactory** annotation is used by OSLC4J to advertise the URI of this method as your Creation Factory.

__b.   The **@Consumes** JAX-RS annotation for this method indicates it can accept a BugzillaChangeRequest that is RDF/XML, XML or JSON. Once again, OSLC4J's message body handlers will take care of de-serializing the RDF representation in the HTTP POST request into a Java BugzillaChangeRequest object to work with.

__c.   Add or enable the code to create a Bugzilla bug from an OSLC BugzillaChangeRequest object.

```
final String newBugId = BugzillaManager.createBug(httpServletRequest,
                         changeRequest, productId);
final Bug newBug = BugzillaManager.getBugById(httpServletRequest,
                         newBugId);
```

__d.   Convert the new bug into a BugzillaChangeRequest and return it as the result of this method. OSLC4J will correctly serialize it and send it back as the body of POST response. It will also set the Location header of the response to the resource ("about") URI for the new BugzillaChangeRequest. Add or enable the code to do this:

```
BugzillaChangeRequest newChangeRequest;
try {
     newChangeRequest = BugzillaChangeRequest.fromBug(newBug);
} catch (Exception e) {
     throw new WebApplicationException(e);
}
URI about = getAboutURI(productId + "/changeRequests/" +
                          changeRequest.getIdentifier());
 newChangeRequest
    .setServiceProvider(ServiceProviderCatalogSingleton
    .getServiceProvider(httpServletRequest, productId).getAbout());
 newChangeRequest.setAbout(about);

return Response.created(about).entity(changeRequest).build();
```

__a.   Make sure your new code compiles and start the Adapter (Run->Run Configurations and run **Launch Lab6**)

__3.   Now, you will use the REST Client tool to create a bug via HTTP POST.

__a.   Start the REST Client tool and set the URL below.

http://oslc:8080/OSLC4JBugzilla/services/1/changeRequests

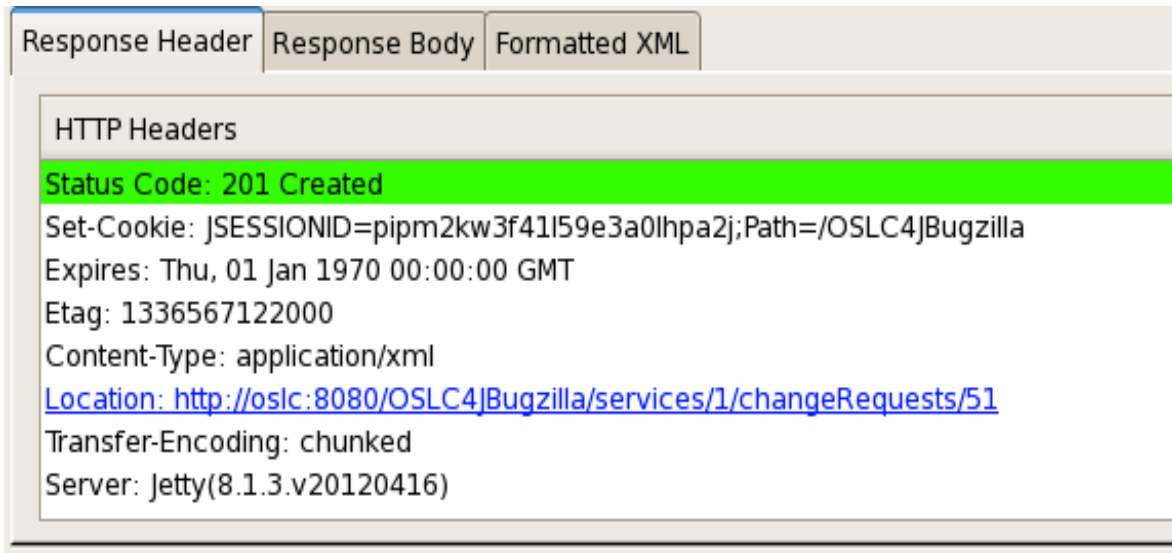__b.   Set the content-type header to **application/rdf+xml**

__c.   Set the REST Request Method to POST

__d. Enter the RDF/XML data below into the Request Body field (note that you may need to substitute the correct values for the things shown in bold below using the correct values for your Bugzilla installation, and the productId you picked above. If you do not wish to retype this XML, it can be found in the **Lab5** project in **src/text/resources/newBugzillaChangeRequest.xml**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:oslc="http://open-services.net/ns/core#"
    xmlns:bugz="http://www.bugzilla.org/rdf#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:dcterms="http://purl.org/dc/terms/"
    xmlns:oslc_cm="http://open-services.net/ns/cm#">
  <oslc_cm:ChangeRequest>
    <bugz:operatingSystem>Linux</bugz:operatingSystem>
    <rdf:type rdf:resource="http://open-services.net/ns/cm#BugzillaChangeRequest"/>
    <oslc_cm:status>NEW</oslc_cm:status>
    <bugz:priority>---</bugz:priority>
    <dcterms:title>New bug entered from OSLC Adapter</dcterms:title>
    <bugz:version>unspecified</bugz:version>
    <bugz:platform>PC</bugz:platform>
    <dcterms:contributor>
      <foaf:Person
rdf:about="http://oslc:8080/OSLC4JBugzilla/person?mbox=bugs%40localhost.here">
        <foaf:mbox>bugs@localhost.here</foaf:mbox>
      </foaf:Person>
    </dcterms:contributor>
    <bugz:component>Server</bugz:component>
    <oslc_cm:severity>Unclassified</oslc_cm:severity>
  </oslc_cm:ChangeRequest>
</rdf:RDF>
```

__e. POST the data and look at the results. You should see a response header like the screen shot below and the Response Body should contain the RDF/XML of the new BugzillaChangeRequest

```
Response Header  Response Body  Formatted XML

  HTTP Headers

Status Code: 201 Created
Set-Cookie: JSESSIONID=pipm2kw3f41l59e3a0lhpa2j;Path=/OSLC4JBugzilla
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Etag: 1336567122000
Content-Type: application/xml
Location: http://oslc:8080/OSLC4JBugzilla/services/1/changeRequests/51
Transfer-Encoding: chunked
Server: Jetty(8.1.3.v20120416)
```

2.0   End of Lab5

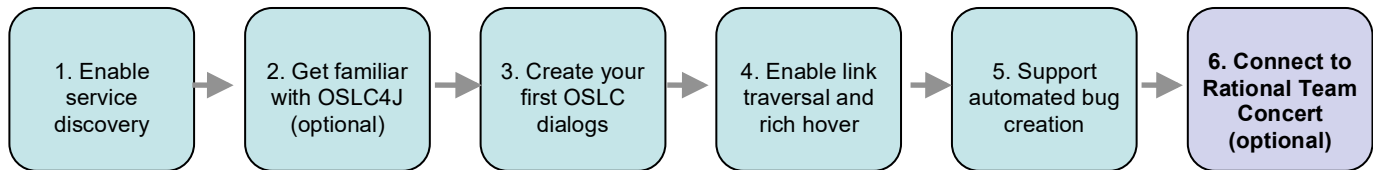It is recommended to leave this project behind and pick up with the next project by starting the **Lab6** project

1) Close all open editors (files)

2) Shutdown any running servers (click the red Stop button in the Console window).

3) Optionally, close **Lab5** project

2.0   Summary

In this lab you enabled tools to be able to programmatically create Bugs using the HTTP POST method using a simple RDF/XML representation of a Bug.

Lab 6   Connecting to Rational Team Concert - OPTIONAL

1. Enable service discovery	→	2. Get familiar with OSLC4J (optional)	→	3. Create your first OSLC dialogs	→	4. Enable link traversal and rich hover	→	5. Support automated bug creation	→	**6. Connect to Rational Team Concert (optional)**

Objectives:

- Take what you've built up from the previous labs and use it to connect to Rational Team Concert.

- This is an optional lab and can be skipped if you are constrained on time.

Description:

Now that you've mostly created an OSLC provider, you'd like to connect it to a tool that you have that is an OSLC consumer. You'd like to have RTC Work Items linked to Bugzilla bugs. It would also be handy to link your test artifacts to Bugzilla bugs directly. This lab will focus on RTC as the consumer of your Bugzilla services, most of the work to link RQM to Bugzilla is the same and steps could be repeated. Let's get RTC hooked up to Bugzilla.

This lab will explore the steps necessary to achieve these integrations that leverage the work that has been done so far. It will show some additional requirements that are needed such as a Jazz Root Services document and OAuth.

> ⚠️ **Important!**
> This lab does not provide comprehensive documentation on how to enable these integrations. It helps guide you through the steps to enable them. It does NOT replace product documentation.

Steps:

- Understand additional steps needed to connect to RTC and RQM

    o Jazz Root services

    o OAuth 1.0a

- Explore some additional requirements

    o XML format for service provider resources

2.0   Add Jazz Root Services support

The intent of the Jazz Root Services is for Jazz Team Servers and other Jazz-integrated applications to expose what services they offer. For your purposes you'll only need to expose a subset of the entire specification. To learn more about these specification, see:

https://jazz.net/wiki/bin/view/Main/RootServicesSpec

https://jazz.net/wiki/bin/view/Main/RootServicesSpecAddendum2

__1.   Locate **RootServicesService.java** in the **org.eclipse.lyo.oslc4j.bugzilla.services** package and explore its contents. Add or enable the code to pass your adapter's base URI and Service Provider Catalog URI to the JSP which will produce the rootservices document:

```
request.setAttribute("baseUri", BugzillaManager.getBugzServiceBase());
request.setAttribute("catalogUri",
                  ServiceProviderCatalogSingleton.getUri().toString());
request.setAttribute("oauthDomain", BugzillaManager.getServletBase());
final RequestDispatcher rd =
          request.getRequestDispatcher("/cm/rootservices_rdfxml.jsp");
 rd.forward(request, response);
 request.flushBuffer();
```

__2.   Locate the **rootservices_rdfxml.jsp** and explore its contents

__a.   Add the location to the service provider catalog URL to the <oslc_cm:cmServiceProviders> element like:
```
<oslc_cm:cmServiceProviders rdf:resource="<%= catalogUri %>" />
```

__3.   Add the OAuth parameters. These URLs match the URLs used by the Eclipse Lyo OAuth webapp (see the next section for details on that webapp).

__c.   Add the URLs for the OAuth realm and domain names
```
        <jfs:oauthRealmName>Bugzilla</jfs:oauthRealmName>
        <jfs:oauthDomain><%= oauthDomain %></jfs:oauthDomain>
```

__d.   Add the remaining OAuth URIs such as:
```
<jfs:oauthRequestConsumerKeyUrl rdf:resource="<%= baseUri +
                "/oauth/requestKey" %>" />
        <jfs:oauthApprovalModuleUrl rdf:resource="<%= baseUri +
                "/oauth/approveKey" %>" />
        <jfs:oauthRequestTokenUrl rdf:resource="<%= baseUri +
                "/oauth/requestToken" %>"/>
        <jfs:oauthUserAuthorizationUrl rdf:resource="<%= baseUri +
                "/oauth/authorize" %>" />

        <jfs:oauthAccessTokenUrl rdf:resource="<%= baseUri +
```

```
"/oauth/accessToken" %>"/>
```

2.0   Add OAuth support using the Eclipse Lyo OAuth Web App

The Eclipse Lyo project contains 3 projects which provide OAuth functionality. The net.oauth package from http://code.google.com/p/oauth/ is the OAuth library used by these projects

- org.eclipse.lyo.server.oauth.core – Core classes for token handling and interactions with the consumer store to manage consumer keys.

- org.eclipse.lyo.server.oauth.consumerstore – Persistence for consumer keys. Uses Jena and Derby to persist keys in an RDF store in Derby.

- org.eclipse.lyo.server.oauth.webapp – A JAX-RS and JSP web application to provide REST services for token requests and key handling. It also provides an administrative UI for provisional key approval and approved key display.

 It is beyond the scope of this workshop to go into the inner workings of an OAuth provider. You will go through the steps required to "hook up" the OAuth provider to the Bugzilla adapter for use in a Rational Team Concert integration.

__5.   First you'll look at how the Eclipse Lyo OAuth web application is included as a dependency of your Bugzilla Adapter.

__a. Find and open the file **pom.xml** in **Lab6.** Go to the Dependencies tab.   This is the Maven XML dependency editor. Notice oauth-core, oauth-consumer-store and oauth-webapp are all dependencies with a folder icon.   This means Maven is smart enough to resolve these projects from your workspace.

__b. Now find and open **CredentialsFilter.java** in the **org.eclipse.lyo.oslc4j.bugzilla.servlet** package.   This is an HttpServletFilter which is called any time your Bugzilla Adapter is called using HTTP.   In the previous labs it has taken care of checking for Basic Authorization Credentials for you and challenging the user for login if they are not there. Find the **doFilter()** method and add or enable code to perform validation of any OAuth information on the request:

```java
try {
    try {
        OAuthMessage message = OAuthServlet.getMessage(request, null);
        if (message.getToken() != null) {
            OAuthRequest oAuthRequest = new OAuthRequest(request);
            oAuthRequest.validate();
            BugzillaConnector connector = keyToConnectorCache.get(message
                                                    .getToken());
            if (connector == null) {
                throw new OAuthProblemException(OAuth.Problems.TOKEN_REJECTED);
            }

            request.getSession().setAttribute(CONNECTOR_ATTRIBUTE, connector);
        }
    } catch (OAuthProblemException e) {
```

```
        if (OAuth.Problems.TOKEN_REJECTED.equals(e.getProblem())))
            throwInvalidExpiredException(e);
        else
            throw e;
    }
} catch (OAuthException e) {
    OAuthServlet.handleException(response, e, OAUTH_REALM);
    return;
}
```

The logic is a little complicated due to the fact that Jazz servers will only re-start the OAuth process when a particular OAuth problem message is received and **throwInvalidExpiredException**() handles that.

___b.   You can test this by loading **http://oslc:8080/OSLC4JBugzilla/rootservices** into the browser after starting the adapter (Run->Run Configurations and run **Launch Lab6**)


2.0   Establish a friend

The adapter now has a catalog, service providers, rootservices and OAuth support is enabled. You can now establish a Jazz friend relationship. This is needed in order to enable RTC as consumers of another server's services.

___1.   Start the RTC server. From the Red Hat desktop, select **Applications->Jazz Team Server->Start Jazz Team Server**

___2.   From Firefox, load the RTC web UI. File->New Tab, then select the **RTC** toolbar shortcut.

___a.   Login as **clmuser** and password **clmuser**

___b.   Navigate to the Application Administration page
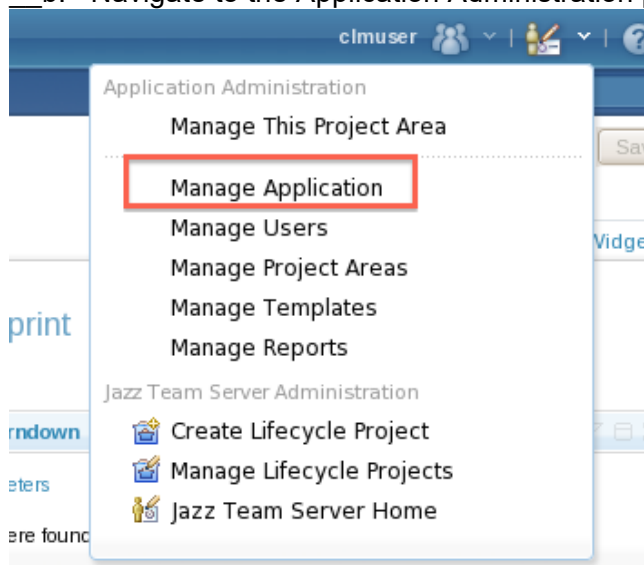
___c.   Navigate to Communications->Friends (Outbound)

___d.   Select Friend List **Add…**

__3.   Fill in the dialog with the root services URL **http://oslc:8080/OSLC4JBugzilla/rootservices** and other details (any secret will do, they just need to match).

Add Friend

1 Add Friend   2

Add Friend

Property	Value
Name	Bugzilla Enter a name to identify this entry in the friends list
Root Services URI	http://oslc:8080/OSLC4JBugzilla/rootservices The URI for root services on the server you want to add as a friend. Format: https://<hostname><:port-number>/<context>/rootservices
OAuth Secret	•••••• Enter a code phrase to be associated with the new OAuth consumer key from the friend server.
Re-type Secret	•••••• Re-type your code phrase to help prevent typos.
Trusted	☑ Trusted consumers will be able to share authorization with other trusted consumers and users will not be prompted for approval to access data. It is recommended that external web sites or products are considered as untrusted.

Create Friend

< Back   Next >   Finish   Cancel

__a.   Select **Create Friend** and then click the **Next** button.   Click the **Grant access for the provisional key** link to start the process of approving RTC's request to talk to the Bugzilla adapter using OAuth

__c.   When prompted, login with user **bugs@localhost.here** and password **bugs4me** and then click **Continue**

__d.   On the authorization UI (provided by the Lyo OAuth webapp), click **Allow** and the **Finish**
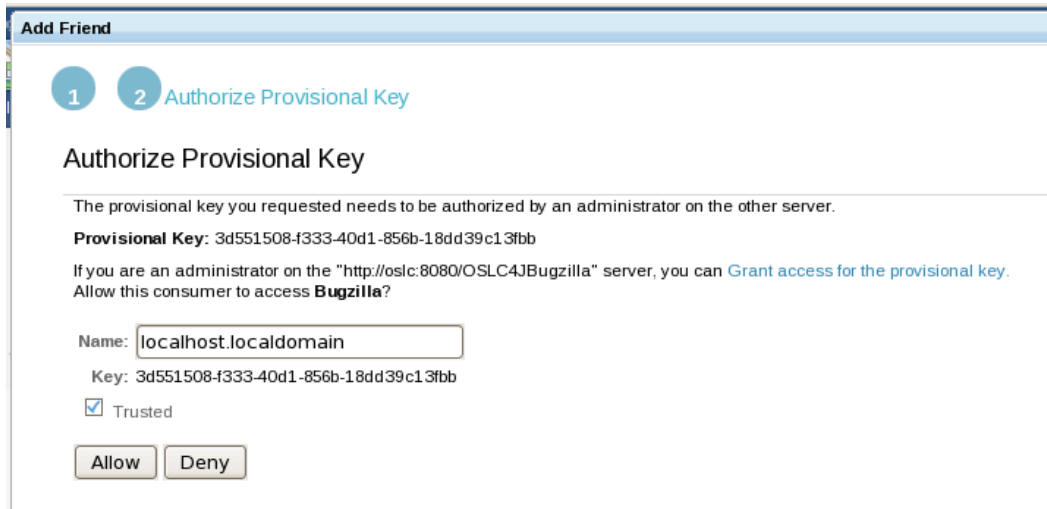
__e. The Bugzilla adapter is now "friends" with RTC

Now that you have the servers trusting each other, you will link the RTC project to a Bugzilla product.

2.0   Associate a Project Area to a Product

__1.   Open the Project Area editor, locate the **Project Areas** toolbar item just below **Application**
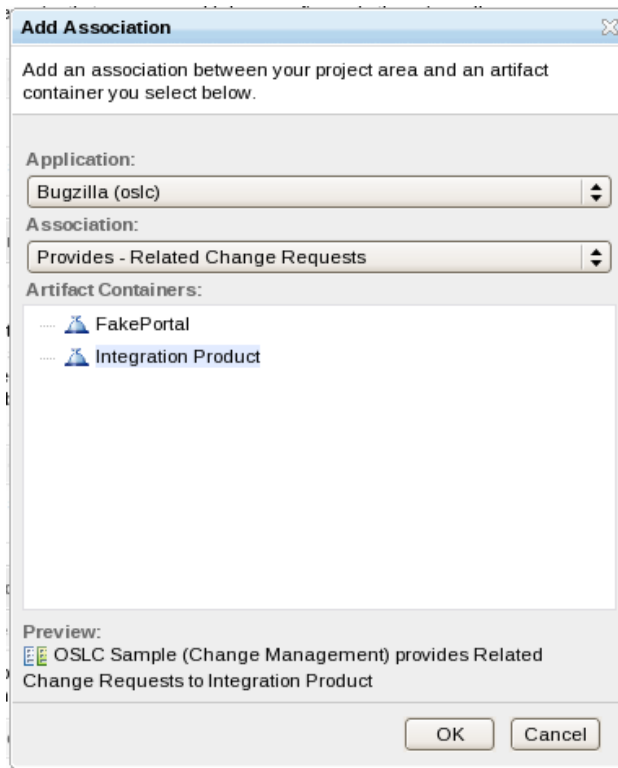
Administration header.

__a. Select the project area **OSLC Sample (Change Management)**

__b. Scroll down to the **Associations** section and select **Add…**

__c. You will be presented with a dialog of all friend servers who have artifact containers (service providers in OSLC terminology) that RTC can consume. Select your Bugzilla application from the Application dropdown list. If prompted, provide the Bugzilla credentials

__d. Next select Association and pick the Artifact Container **Integration Product** and then select **OK**

Add Association

Add an association between your project area and an artifact container you select below.

Application:
Bugzilla (oslc)

Association:
Provides - Related Change Requests

Artifact Containers:
- FakePortal
- Integration Product

Preview:
OSLC Sample (Change Management) provides Related Change Requests to Integration Product

OK   Cancel

__f.   In the top right, select **Save**

__g.   That's it. RTC is now hooked up to Bugzilla.

__h.   Now you will use this to link a Work Item to a Bug

2.0   Link a Work Item to a Bug

Now that you have the servers "friended" and the RTC Project area associated to a Bugzilla Product, now the general Work Items users (probably the general and most often usage of RTC) have the ability to link to Bugzilla.

In order to link to a bug, you need to open a Work Item in the Work Item editor

__1.   Select the "Home" toolbar dropdown, then select the **OSLC Sample** Project under "Change and Configuration Management" section such as:
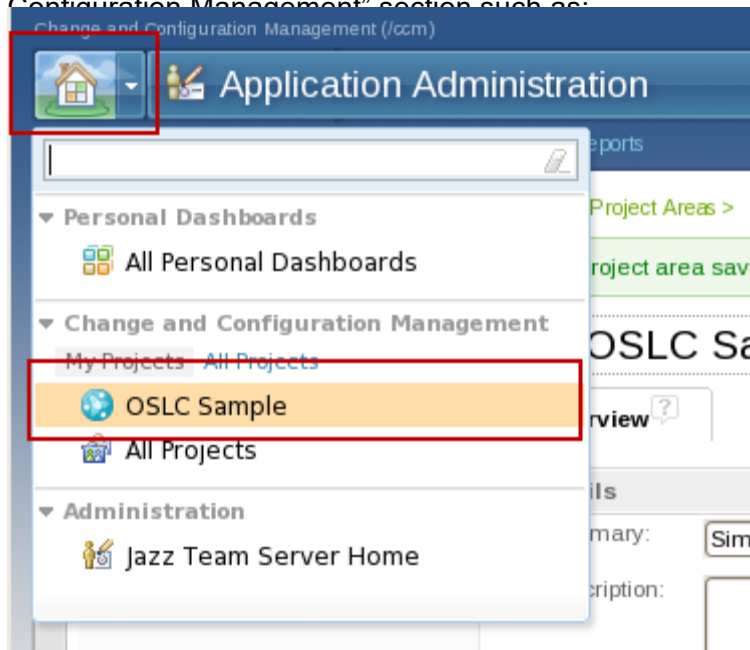
__2.   Next create a Work Item to work with by selecting Work Items -> Create Defect

  __a. Navigate to the **Links** tab

  __b. Within the **Links** section, select **Add -> Related Change Request**

  __c. You will be prompted to select link or create (note it may take a couple seconds for RTC to fetch the service provider resource from your Bugzilla provider)

  __d. Select "Link to existing : Bugs" and then select **OK**

  __e. You will now be presented with a Bugzilla search dialog (doesn't this look familiar from before?)

  __f. Enter a search term, try entering "3rd" and then select Search

  __g. Select a bug from the list, then select **OK**

__3.   Notice that an entry is added to the "Related Change Request" links section with the Bugzilla icon, bug id and the summary from the bug such as:
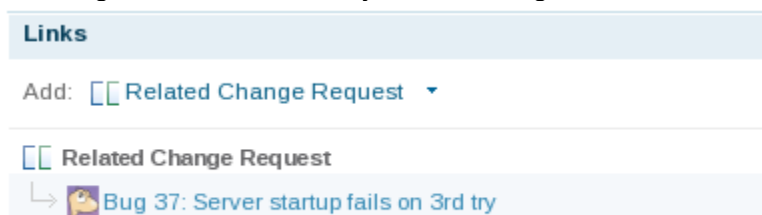
__4.   Next move the mouse pointer over the link (but don't click on it) and you'll observe the UI preview (again, remember this from before?)
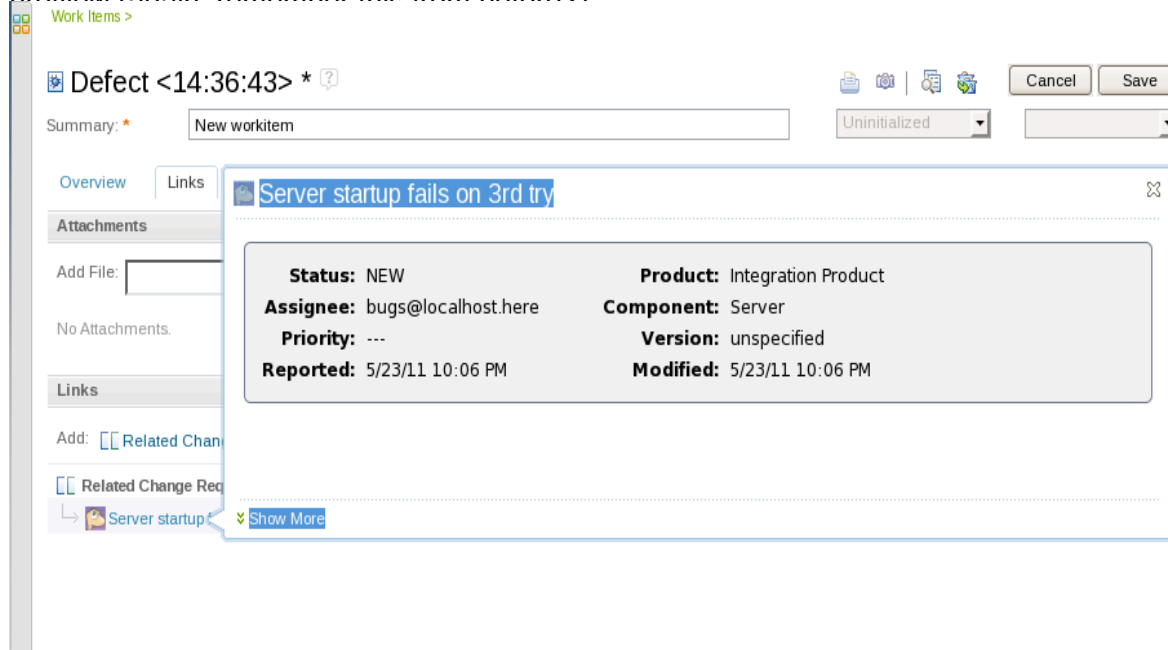
__5.   You can save your Work Item but you'll be prompted with some choices, why? The Work Item save operation is trying to update the Bugzilla Bug with a link back to the Work Item. Since you haven't implemented that yet, RTC reports an error. You can select the option to ignore the error and save anyway. OR you can implement the link update operation on the Bugzilla provider.

2.0   Summary

You learned in this lab how to take what you've built up from the previous labs and use it to connect to Rational Team Concert. You now have a very good start at an implementation that could be used more broadly. There are still some items that should be handled, like handling updates to Bugs. Even though this was a quick run through some many key concepts, it is intended to give you a better understanding of these concepts and how to apply them in your own integrations.

Appendix A.   Troubleshooting

There are always a number of expected things that you might encounter. If there is anything that you may not be able to overcome, please see a proctor or the workshop leader.

In Eclipse, check for compilation errors in the Eclipse project

In Eclipse, Make sure multiple servers aren't running (or right servers are running)

In Eclipse, check Console for Server errors.

Firefox browser has installed a add-in called Firebug that can be used to inspect the web page's DOM and JavaScript.

If RTC isn't responding, make sure the Jazz Team Server has been started from the Applications menu item from the desktop.

Ask a proctor for help.

Appendix B.   Sources

The sources for this workshop are based on materials that are available in an open source project around OSLC enablement material. Also there are additional materials available to help with consuming OSLC services and more advanced topics are available around authentication and query. See references below for sources:

Eclipse Lyo project http://eclipse.org/Lyo and http://wiki.eclipse.org/Lyo

> Everyone is encouraged to join and contribute to these efforts that can greatly help build the ecosystem of OSLC-based tool integration

OSLC Tutorial

> Check http://open-services.net for a the development of a 2 part tutorial where you can establish your own development environment and follow along with the examples.
>
> Part 1 Consuming OSLC Services
>
> Part 2 Providing OSLC Services

General Resources

> As new resources and enablement material are being made available be sure to check out the OSLC website for these developments at http://open-services.net

Appendix C.   Legal

2.0   Notices

The material in this guide is Copyright (c) IBM Corporation 2011,2012.

2.0   About this Content

June 12, 2012

License

The Eclipse Foundation makes available all content in this plug-in
("Content"). Unless otherwise indicated below, the Content
is provided to you under the terms and conditions of the Eclipse
Public License Version 1.0 ("EPL") and Eclipse Distribution
License Version 1.0 ("EDL"). A copy of the EPL is available
at *http://www.eclipse.org/legal/epl-v10.html*
and a copy of the EDL is available at
http://www.eclipse.org/org/documents/edl-v10.php.

For purposes of the EPL, "Program" will mean the Content.

If you did not receive this Content directly from the Eclipse
Foundation, the Content is being redistributed by another party
("Redistributor") and different terms and conditions may
apply to your use of any object code in the Content. Check the
Redistributor's license that was provided with the Content. If no such
license exists, contact the Redistributor. Unless otherwise indicated
below, the terms and conditions of the EPL and EDL still apply to any
source code in the Content and such source code may be obtained at
http://www.eclipse.org

NOTES

NOTES

NOTES