Tutorial: Integrating products with OSLC

Tutorial Overview

This tutorial explains how to integrate tools with OSLC. The tutorial uses examples, starting with simple ones and building to more advanced topics such as implementing an OSLC Provider. It is organized into three parts:

- OSLC consumer topics
- OSLC provider implementation topics
- Advanced topics

It's intended for software developers with knowledge of the basics of web architecture, HTTP, RDF, and associated topics.

Organization

We start with a quick tour, showing OSLC in a nutshell, then present the story of how Nina improves her company's ALM process by integrating tools via OSLC.

- In **Part 1**, we start with the basic topics on the consumer side, explaining how Nina integrates her existing home-grown CRM web application with an OSLC Change Management (CM) provider. We'll explore UI Preview, HTTP operations on resources, and Delegated UI.
- In **Part 2**, we see how Nina enhanced her company's home-grown Defect Tracking system to support just enough of the OSLC CM specification to support the integrations developed in Part 1.

Audience

This tutorial is written for those who will be working directly with OSLC specifications, at the code level and implementing OSLC consumers and providers. If you are part of this audience, then you:

- Understand basics of software development
- Understand basics of web architecture and HTTP
- Understand basics of Linked Data and RDF
- Are abie to follow examples in XML, JSON, HTML, and JavaScript
- Are eager to learn more about those topics and OSLC

We try hard to keep this tutorial programming language agnostic; when we do have to show server-side logic we use Java and JSP to do so. Later versions of this document might expand to other languages and platforms. OSLC is a community effort and we'd love your help in adding examples in different programming languages to this tutorial, or

new tutorials, to help those on other platforms such as Perl, PHP, Python, Ruby, and the .Net languages.

Introduction

Welcome to the OSLC Tutorial. The goal of this document is to explain how to implement OSLC consumers and providers by examining realistic use cases and showing how to implement each in detail with lots of examples and working code.

We'll start simple, explaining how to interact and integrate with an OSLC-CM v2 provider. Then, we will progressively introduce more advanced features and build your knowledge to the point where you'll be able to build a complete OSLC provider implementation. As we progress, we will help you to follow along by looking at code for an OSLC consumer and OSLC provider and by exploring OSLC resources using the Firefox or Chrome Poster plugin.

To bring our examples to life, we'll tell the story of Nina, a developer and sysadmin who handles development infrastructure and uses OSLC specifications to integrate systems and put in place more efficient and effective work-flows for her team.

Following along

One of the best ways to learn a new technology is to experiment with it. In that spirit, you can follow along with the OSLC Tutorial by using the following software:

- **NinaCRM**: A fictional Customer Relationship Management (CRM) system that hosts OSLC UI Preview and OSLC Delegated UI examples, implemented as a Java EE web application. Get it from the OSLC Tools project on SourceForge. In the examples we assume that NinaCRM is installed at http://localhost:8181/ninacrm.
- **OSLC Bugzilla Adapter**: This is the adapter that Nina develops in Part 2, an OSLC Adapter for Bugzilla that implements the OSLC-CM specification. You can get it from the OSLC Tools project on SourceForge. In the examples, we assume this is installed at http://localhost:8282/bugz.
- **Firefox or Chrome Poster plugin**: This plugin makes it easy to issue HTTP requests supporting GET, PUT, POST and DELETE and setting headers. Examples show how to use Poster to manipulate OSLC resources.

The Appendix explains how to install and run the above software. Now let's quickly introduce OSLC so we can get started with the story.

Overview of OSLC

See a video overview of OSLC here: http://www.youtube.com/watch?feature=player_embedded&v=40mjwqGEKBU

Open Services for Lifecycle Collaboration (OSLC) is an initiative to define standards that enable easier and more effective integrations between the many tools that software and product developers use. OSLC is a different approach to tool integration that seeks to integrate the resources managed by those tools into the web of data. OSLC uses variety of web integration patterns, which are all either based on or complementary to what the World Wide Web Consortium (W3C) calls Linked Data.

To learn more about core OSLC concepts and architectural underpinnings, read the OSLC Primer. We'll be implementing many of these core resources and services in this tutorial.

For more information on the value of integrating with open protocols, read our whitepaper "The Case for Open Services".

Running the example applications

This section explains how to run the OSLC4J Bugzilla Adapter and the NinaCRM example application.

Installing prerequisites

Downloading and installing Eclipse

You'll be using Eclipse and some add-ons to simplify the setup.

You need Eclipse 3.6 or later. Download Eclipse here. Download the **Eclipse IDE for Java EE Developers**.

After you install Eclipse, start the Eclipse application.

In the Workspace Launcher window, create a new directory for your Eclipse workspace or accept the default Workspace location.

Installing EGit

All of the samples are in Git source control; EGit provides a GUI to work with Git repositories.

1. In Eclipse, click **Help → Install New Software**.
2. In the Install window, in the **Work with** field paste in the following: http://download.eclipse.org/egit/updates
3. Click **Add…**.
4. In the Add Repository window, in the **Name** field, type something memorable like `EGit`, then click **OK**.
5. Select Eclipse Git Team Provider and click Next.
6. On the Install Details page, click **Next**.
7. On the Review Licenses page, review the license, select **I accept the terms of the license agreement** and click **Finish**.
8. After EGit installs, on the Software Updates window click *Yes to restart Eclipse.

Installing M2Eclipse

Our samples use the Maven build automation tools; you'll use the M2Eclipse Maven plugin to manage Maven from Eclipse.

1. In Eclipse, click **Help → Install New Software**.

2. In the Install window, in the **Work with** field paste in the following: http://download.eclipse.org/technology/m2e/releases
3. Click **Add…**.
4. In the Add Repository window, in the **Name** field, type something memorable like `M2Eclipse`, then click **OK**.
5. Select Maven Integration for Eclipse and click Next.
6. On the Review Licenses page, review the license, select **I accept the terms of the license agreement** and click **Finish**.
7. After M2Eclipse installs, on the Software Updates window, click "Yes" to restart Eclipse.

Installing the OSLC4J toolkit

1. In Eclipse, open the Git Repositories view. (**Window → Show View → Other**, search for `Git repo` and click **OK**.)
2. Click Clone a Git Repository.
3. In the Clone Git Repository window, in the **URI** field paste the following: git://git.eclipse.org/gitroot/lyo/org.eclipse.lyo.core.git
   The **Host** and **Repository** fields will autofill. Leave the **Username** and **Password** fields empty.
4. Click **Next**.
5. On the Branch Selection page, select **master** and click **Next**.
6. For the **Destination**, select a folder for the files or accept the default of your Eclipse workspace.
7. Click **Finish**. `org.eclipse.lyo.core` will appear in the Git Repositories view.
8. In the Git Repositories view, right-click **org.eclipse.lyo.core** and click **Import Projects**.
9. In the Import Projects from Git Repository wizard, select **Import existing projects** and click **Next**.
10. Select all components of OSLC4J core (this is the default) and click **Finish**.

Next, build the OSLC4J project.

Build the OSLC4J projects

1. In Eclipse, open the Project Explorer view. (**Window → Show View → Project Explorer**)
2. In the Project Explorer view, expand **OSLC4JCoreRelEng**.
3. Right-click on **pom.xml**, then click **Run as → Maven Clean**.
4. Right-click on **pom.xml**, then click **Run as → Maven Install**. In the Console view, you'll see a lot fly by.

[Optional] Create an account at Bugzilla Landfill

Landfill is an always-running, open Bugzilla server that you can use if you don't want to use your own Bugzilla application or set up a new one.

Create an account here.

Downloading, building, and running the sample applications

Cloning the Lyo documentation and server repositories

The Eclipse Lyo documentation Git repository has the Bugzilla Adapter and NinaCRM sample applications.

1. In Eclipse, open the Git Repositories view. (**Window → Show View → Other**, search for `Git repo` and click **OK**.)
2. Click Clone a Git Repository.
3. In the Clone Git Repository window, in the **URI** field paste the following: git://git.eclipse.org/gitroot/lyo/org.eclipse.lyo.docs.git
   The **Host** and **Repository** fields will autofill. Leave the **Username** and **Password** fields empty.
4. Click **Next**.
5. On the Branch Selection page, select **master** and click **Next**.
6. For the **Destination**, select a folder for the files or accept the default of your Eclipse workspace.
7. Click **Finish**. `org.eclipse.lyo.docs` will appear in the Git Repositories view.
8. Repeat steps 2–7 for the server repository at git://git.eclipse.org/gitroot/lyo/org.eclipse.lyo.server.git.

Next, import the documentation projects:

1. In the Git Repositories view, right-click **org.eclipse.lyo.docs** and click **Import Projects**.
2. In the Import Projects from Git Repository wizard, select **Import existing projects** and click **Next**.
3. Select OSLC4JBugzilla and ninacrm and click Finish.

Finally, import the server projects:

1. In the Git Repositories view, right-click **org.eclipse.lyo.server** and click **Import Projects**.
2. In the Import Projects from Git Repository wizard, select **Import existing projects** and click **Next**.
3. Select the following projects:
4. org.eclipse.lyo.server.oauth.consumerstore
5. org.eclipse.lyo.server.oauth.core
6. org.eclipse.lyo.server.oauth.webapp
7. and click **Finish**.

Configuring the Bugzilla adapter

Configure the Bugzilla adapter to point to your Bugzilla application.

1. In Eclipse, open the Project Explorer view. (**Window → Show View → Project Explorer**)
2. In the Project Explorer view, find and edit the file `OSLC4JBugzilla/src/main/resources/bugz.properties`.
3. Edit the `bugzilla_uri` property to the URL of your Bugzilla server.
   If you're using Bugzilla Landfill, it will look similar to this:
   `bugzilla_uri=https://landfill.bugzilla.org/bugzilla-4.2-branch`
   There are multiple versions of Bugzilla running at landfill.bugzilla.org; be sure to select the version where you have a user ID.
4. For the `admin` property, provide your Bugzilla user ID.
   For Bugzilla Landfill, it will be the email address you used when you created your account:
   `admin=you@example.com`
   (This is the ID you will use to log in to the OAuth application).
5. Save bugz.properties.

Building the applications

First, update the project configurations for the projects.

1. In Eclipse, open the Package Explorer view. (**Window → Show View → Package Explorer**)
2. In the Package Explorer view, select the following packages:
3. ninacrm
4. org.eclipse.lyo.server.oauth.core
5. org.eclipse.lyo.server.oauth.consumerstore
6. org.eclipse.lyo.server.oauth.webapp
7. OSLC4JBugzilla
8. Right-click and select **Maven → Update Project**.
9. In the Update Maven Project window, verify that those 4 projects are selected and click **OK**.

Next, install the projects:

1. In the Package Explorer view, expand **org.eclipse.lyo.server.oauth.core**.
2. Find the file **pom.xml**
3. Right-click on **pom.xml** and select **Run as → Maven Install**. You should eventually see a success message in the Console view.
4. Repeat steps 1–3 for the following packages *in this order*:
   a. `org.eclipse.lyo.server.oauth.consumerstore`
   b. `org.eclipse.lyo.server.oauth.webapp`
   c. `OSLC4JBugzilla`

d. `ninacrm`

Running the sample applications

STARTING THE OSLC4J BUGZILLA ADAPTER:

1. In Eclipse, click **Run → Run Configurations**.
2. In the Run Configurations window, expand **Maven Build**.
3. Click OSLC4JBugzilla.
4. Click **Run**. This will start the application.

You will see a lot of messages in the Console view. The application will be running when you see this:

```
[INFO] Started Jetty Server
[INFO] Starting scanner at interval of 5 seconds.
```

In your web browser navigate to the OSLC Catalog at http://localhost:8080/OSLC4JBugzilla/services/catalog/singleton

Log in with your Bugzilla user ID and password.

STARTING NINACRM

- In Eclipse, click **Run → Run Configurations**.
- In the Run Configurations window, expand **Maven Build**.
- Click Launch NinaCRM.
- Click **Run**. This will start the application.

When the server starts, in your web browser navigate to http://localhost:8181/ninacrm to see the NinaCRM example.

Implementing an OSLC provider

In this section, we'll be creating an adapter to add OSLC Change Management support to Bugzilla, an open source Defect Tracking system. Although the Bugzilla adapter is available now as part of Eclipse Lyo, its features and architecture are broadly applicable to any adapter that adds OSLC support to an existing product.

This video explores some of the challenges and considerations in getting started with adding OSLC support to existing applications: http://www.youtube.com/watch?feature=player_embedded&v=-oXqudLmNMI

Planning out a partial implementation of OSLC-CM

Our integration use cases that we want to add to Bugzilla require only a partial implementation of the OSLC Change Management specification:

- **Service Provider and Catalogs**: These resources describe the services offered and make it possible for consumers of the OSLC CM service to find the ones they need. In Part 2, you will use these to help implement Automated Bug Creationso that the Testing team's build scripts can use Service Provider documents to locate a URL.
- **OSLC representations for bugs**: This means making each Bug available at a stable URI as an OSLC-CM Change Request resource, with RDF/XML and UI Preview representations via content negotiation. In Part 2, these RDF/XML representations will help automate customer notifications.
- **Delegated UI for Creation & Selection**: Enables users of other systems to create and select bugs in Bugzilla without leaving the web UI of those other systems. You'll use these dialogs in Part 2's to make it easy to link a customer incident to a Bugzilla bug.
- **Creation Factories for bugs**: Enables creation of new bugs by HTTP posting RDF/XML bug representations to the server. We also used this feature in Part 2 for Automated Bug Creation.

Although this leaves out some seemingly critical parts of OSLC (including UPDATE and DELETE via HTTP and OSLC Query), that's OK.

First, though, we need to decide how we'll add OSLC support to the existing applications.

Different approaches to implementing OSLC support

There are (broadly) three different approaches to implementing an OSLC-CM provider for Bugzilla (or any other software):

- The **Native Support** approach is to add OSLC-CM support directly into Bugzilla, modifying whatever code is necessary to implement the OSLC-CM specification.
- The **Plugin** approach is add OSLC-CM support to Bugzilla by developing code that plugs-in to Bugzilla and uses its add-on API.
- The **Adapter** approach is to create new web application that acts as an OSLC Adapter, runs along-side of Bugzilla, provides OSLC-CM support and "under the hood" makes calls to the Bugzilla web APIs to create, retrieve, update and delete resources.

Although any of these approaches are valid approaches for an OSLC implementation, here are some of the pros and cons of each:

Approach	Pros	Cons
Native approach	• Complete control over the quality implementation • Good approach for tool vendors shipping products with OSLC support	• You need control over the application code • You need to learn product's language and platform • Not a good approach for customers who want to add OSLC support to a vendor's products
Plugin Approach	• Uses established and supported mechanism to extend product and add OSLC support	• Limitations on plugins may limit quality of OSLC implementation • You need to learn product's language, platform, and plugin architecture
Adapter Approach	• Can be implemented without modifying the product • Can use your preferred platform and language	• Limitations of product's API may limit quality of OSLC implementation • May introduce redundant URL for product resources. For example, adapter-provided URLs must be used instead of native Bugzilla bug URLs

In short, the Native approach is the right approach for tool vendor who wants to add OSLC support to the products that they understand well. The Plugin and Adapter approaches are best for when you want to add OSLC support to a tool that you've bought from a tool vendor or obtained from an open source project. If the tool has a good Plugin API and you like the language/platform that it requires, then try the Plugin approach. If not, then an Adapter approach is probably best.

In our case, building an adapter makes the most sense.

Architecture for the adapter

Download the OSLC4J Bugzilla adapter. We'll be exploring the adapter instead of writing one from scratch.

The OSLC4J Bugzilla adapter is a RESTful web application built on Java EE with JAX-RS. It has the following additional dependencies:

- OSLC4J: part of Eclipse Lyo, OSLC4J is a Java toolkit that simplifies building OSLC applications
- J2Bugzilla: Java wrapper classes for Bugzilla's XML-RPC based web services interface

In addition, it uses the following helper classes (in the `utils` directory):

- **BugzillaHttpClient**: helper classes for doing HTTP GET requests against a Bugzilla server
- **HttpUtils**: helper classes for working with HTTP requests and responses
- **StringUtils**: helper classes for dealing with strings
- **XmlUtils**: helper classes for XML processing

Finally, the JAX-RS resource definitions are in `org.eclipse.lyo.oslc4j.bugzilla.services`.

NOTE: In older versions of this tutorial and Bugzilla adapter, we defined many individual servlets in the application's `web.xml` file; now, the OSLC4J Bugzilla adapter uses JAX-RS to handle URLs, requests, and resources.

Providing Service Providers and Catalogs

The next step in implementing the OSLC Change Management specification is to determine what high-level organizational concept in your product best maps to OSLC Service Providers – the central organizing concept of OSLC that represents a "container" of resources.

In Bugzilla, bugs are organized by Product. Before you can use Bugzilla, you have to tell the system which Products exist in order to report bugs against them.

Given that, in our adapter each Bugzilla Product will be represented by an OSLC Service Provider REST service. Each Service Provider will include URIs for a Delegated UI for bug selection, a Delegated UI for bug creation, a Query Capability so that bugs can be queried via HTTP GET, and a Creation Factory so that new bugs can be created via HTTP POST.

To enable client programs to find the Service Providers provided by Bugzilla (and because one Bugzilla instance can have multiple Products), we'll use an OSLC Service Provider Catalog. When a client wants to connect to Bugzilla, it first fetches the catalog, which provides a list of Service Providers. In the end, a client can start with the URI of the one Service Provider Catalog offered by Bugzilla and navigate to the Service Providers (one per Product in Bugzilla).

Here are the URLs that will be supported with our adapter (running at `/OSLC4JBugzilla/`) for our OSLC-CM implementation:

- http://HOST/OSLC4JBugzilla/services/catalog/singleton
  This URL will return the OSLC Service Provider Catalog
- `http://HOST/OSLC4JBugzilla/services/serviceProviders/{product_id}`
  Returns the OSLC Service Provider for the Product identified by {product_id} number
- `http://HOST/OSLC4JBugzilla/services/{product_id}/changeRequests`
  If using HTTP GET, returns a list of bugs in the product identified by {product_id}; if using HTTP POST, initiates the Creation Factory for creating a new bug and returns that new bug
- `http://HOST/OSLC4JBugzilla/services/{product_id}/changeRequests/{change_request_id}` Returns the Change Request identified by ID {change_request_id}, in a variety of content-types
- `http://HOST/OSLC4JBugzilla/services/{product_id}/changeRequests/selector` This URL is for the delegated UI selection dialog for the Product identified by ID {product_id}

- http://HOST/OSLC4JBugzilla/services/{product_id}/changeRequests/creator Returns Delegated UI creation dialog for the Product identified by ID {product_id}
- http://HOST/OSLC4JBugzilla/services/resourceShapes/changeRequest Returns the creation and queryResource Shape for Bugzilla bugs

Each of of the URLs above will be handled by a JAX-RS annotated method and our code will have to be able to form all of those types of URLs. That brings us to an important point about OSLC:

Clients don't need to form URLs

With OSLC, there's rarely any need to form URLs. Clients should not be constructing URLs, or making assumptions about how URLs are formed; **instead they should be able to navigate** to all of the other REST service URLs by following links from a Service Provider Catalog.

Basic application architecture

On the OSLC4J Bugzilla application, all REST services are handled by a Bugzilla Application JAX-RS servlet, which is mapped to the URL pattern `/services/*` in `OSLC4JBugzilla/src/main/webapp/WEB-INF/web.xml`:

```
<servlet>
  <servlet-name>JAX-RS Servlet</servlet-name>
  <servlet-
class>org.apache.wink.server.internal.servlet.RestServlet</
servlet-class>
  <init-param>
    <param-name>javax.ws.rs.Application</param-name>
    <param-
value>org.eclipse.lyo.oslc4j.bugzilla.services.BugzillaApplicati
on</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>JAX-RS Servlet</servlet-name>
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

Providing a Service Provider Catalog

In our adapter, we use a simple pattern to implement OSLC REST services: for each operation, a JAX-RS method will accept incoming requests, load the data necessary to render a response, perform the requested operation, and then render the resulting RDF or other representations.

The Service Provider Catalog is defined in **ServiceProviderCatalogService** (in the **org.eclipse.lyo.oslc4j.bugzilla.services**package). The catalog is available at the URL http://HOST/OSLC4JBugzilla/services/catalog/singleton and will list the OSLC Service Providers (one per Bugzilla product).

Defining a JAX-RS method for the Service Provider Catalog

In the file `ServiceProviderCatalogService.java` (in the **org.eclipse.lyo.oslc4j.bugzilla .servlet** package), view the JAX-RS annotation which defines the class that will run at http://HOST/OSLC4JBugzilla/services/catalog/:

```
@Path("catalog")
public class ServiceProviderCatalogService
{
    [ServiceProviderCatalog code]
}
```

That class has a variety of methods that will return a Service Provider Catalog in a variety of formats. More on those later on.

Retrieving Bugzilla product IDs

To build our catalog, we register the Bugzilla product IDs with the **ServiceProviderCatalogSingleton** class (in the**org.eclipse.lyo.oslc4j.bugzilla.servlet** package). The major activity happens in the **initServiceProvidersFromProducts()**method.

First, we create a connection to Bugzilla:

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

If there's a valid connection, we fetch a list of Bugzilla products:

```
GetAccessibleProducts getProductIds = new
GetAccessibleProducts();
bc.executeMethod(getProductIds);
Integer[] productIds = getProductIds.getIds();

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

Then for each Bugzilla product, we register an OSLC Service Provider:

```
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,product
Id);
    }
}
```

(Of particular note is the `parameterMap` HashMap, which will be used to add the Bugzilla productId to the URLs of our services in the **BugzillaChangeRequestService** class. More on that later.)

Displaying the Service Provider Catalog as HTML

Back in the file `ServiceProviderCatalogService.java` (in the **org.eclipse.lyo.oslc4j.bugzilla.services** package), view the `getHtmlServiceProvider()` method, which forwards the `catalog` object to a JSP template to produce the HTML:

```
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);
    }
}
```

Next, view the file `src/main/webapp/cm/serviceprovidercatalog_html.jsp`, which is the JSP template for displaying the catalog in HTML. It's a pretty typical HTML page and it reuses stylesheets from the Bugzilla application.

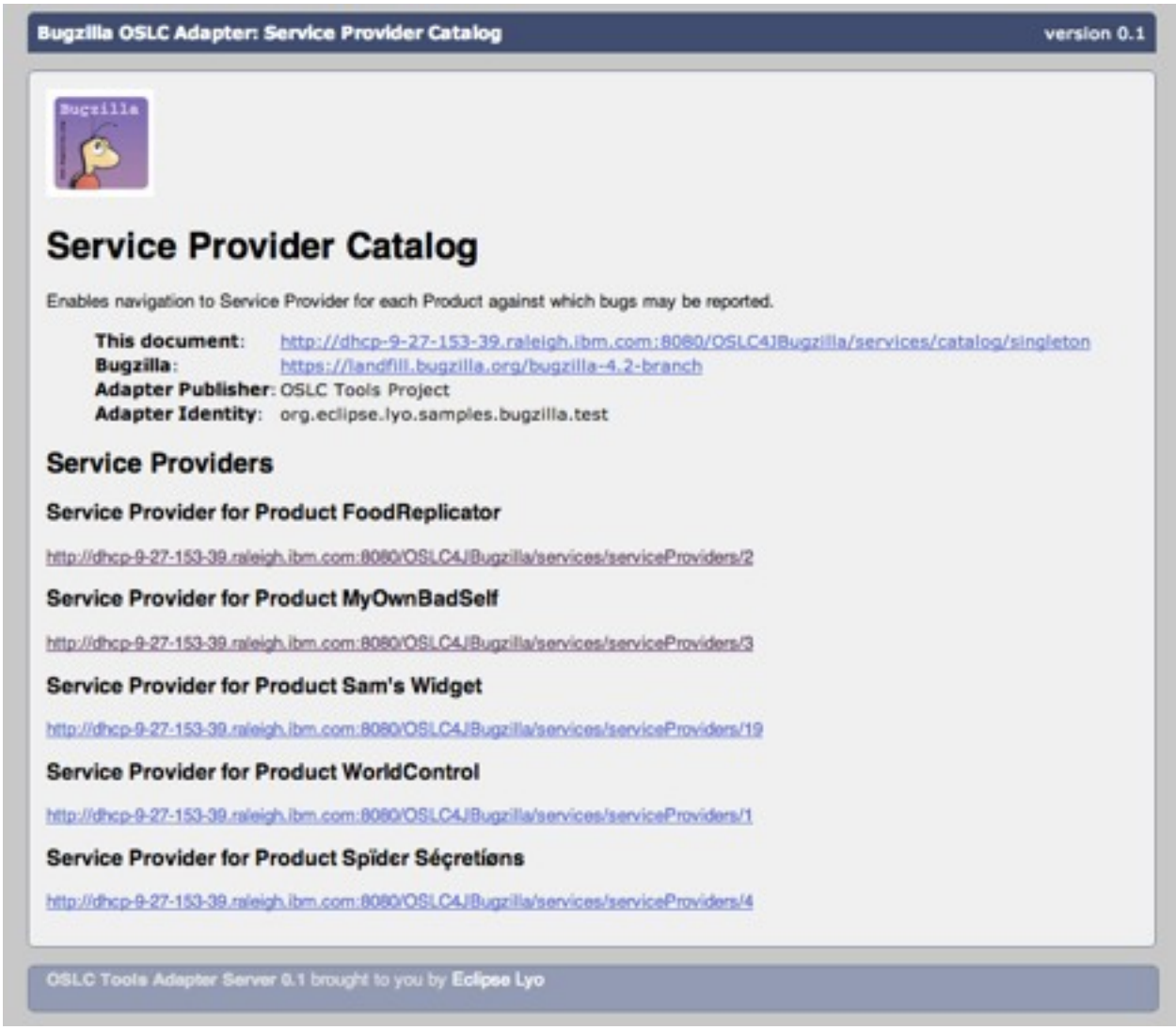
Of particular note are the dynamic segments. First, near the top of the file, the catalog variable is set from the passed catalog attribute:

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

And near the bottom of the file, we loop through the service providers in the catalog and output a heading with the name of the product (**getTitle()**) and a link (**getAbout()**):

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

If you're running the example applications, you can see this in action at http://localhost:8080/OSLC4JBugzilla/services/catalog/singleton.



Screen capture of the Service Provider Catalog in a web browser

Retrieving and displaying details about a Service Provider

Next, we'll display the details for each product as an OSLC Service Provider in an HTML page.

Similar to the Service Provider Catalog, the **ServiceProviderService** class in the **org.eclipse.lyo.oslc4j.bugzilla.services**package defines the URL structure:

```
@Path("serviceProviders")
public class ServiceProviderService
{
  [ServiceProviderClass code]
```

```
}
```

The class **ServiceProviderService** has several methods that fetch the appropriate data and present it in a variety of formats. For the moment, we'll explore the **getHtmlServiceProvider()** method:

```
@GET
@Path("{serviceProviderId}")
@Produces(MediaType.TEXT_HTML)
public void
getHtmlServiceProvider(@PathParam("serviceProviderId") final
String serviceProviderId)
{
    ServiceProvider serviceProvider =
ServiceProviderCatalogSingleton.getServiceProvider(httpServletRe
quest, serviceProviderId);

    Service [] services = serviceProvider.getServices();

    if (services !=null && services.length > 0)
    {
        //Bugzilla adapter should only have one Service per
ServiceProvider
        httpServletRequest.setAttribute("bugzillaUri",
BugzillaManager.getBugzillaUri());
        httpServletRequest.setAttribute("service", services[0]);
        httpServletRequest.setAttribute("serviceProvider",
serviceProvider);

        RequestDispatcher rd =
httpServletRequest.getRequestDispatcher("/cm/
serviceprovider_html.jsp");
        try {
            rd.forward(httpServletRequest, httpServletResponse);
        } catch (Exception e) {
            e.printStackTrace();
            throw new WebApplicationException(e);
        }
    }
}
```

Like the Service Provider Catalog class, we retrieve information from Bugzilla; however, here we only retrieve information about a single Product (from the URL / serviceProviders/{ProductId}). We then dispatch `serviceprovider_html.jsp` to display information about it in HTML.

Displaying the Service Provider as HTML

In `src/main/webapp/cm/serviceprovider_html.jsp`, view near the top of the file where we assemble all the URLs for OSLC services (which we'll cover later) such as query capability, delegated dialogs, and resource shapes:

```
<%
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();

%>
```

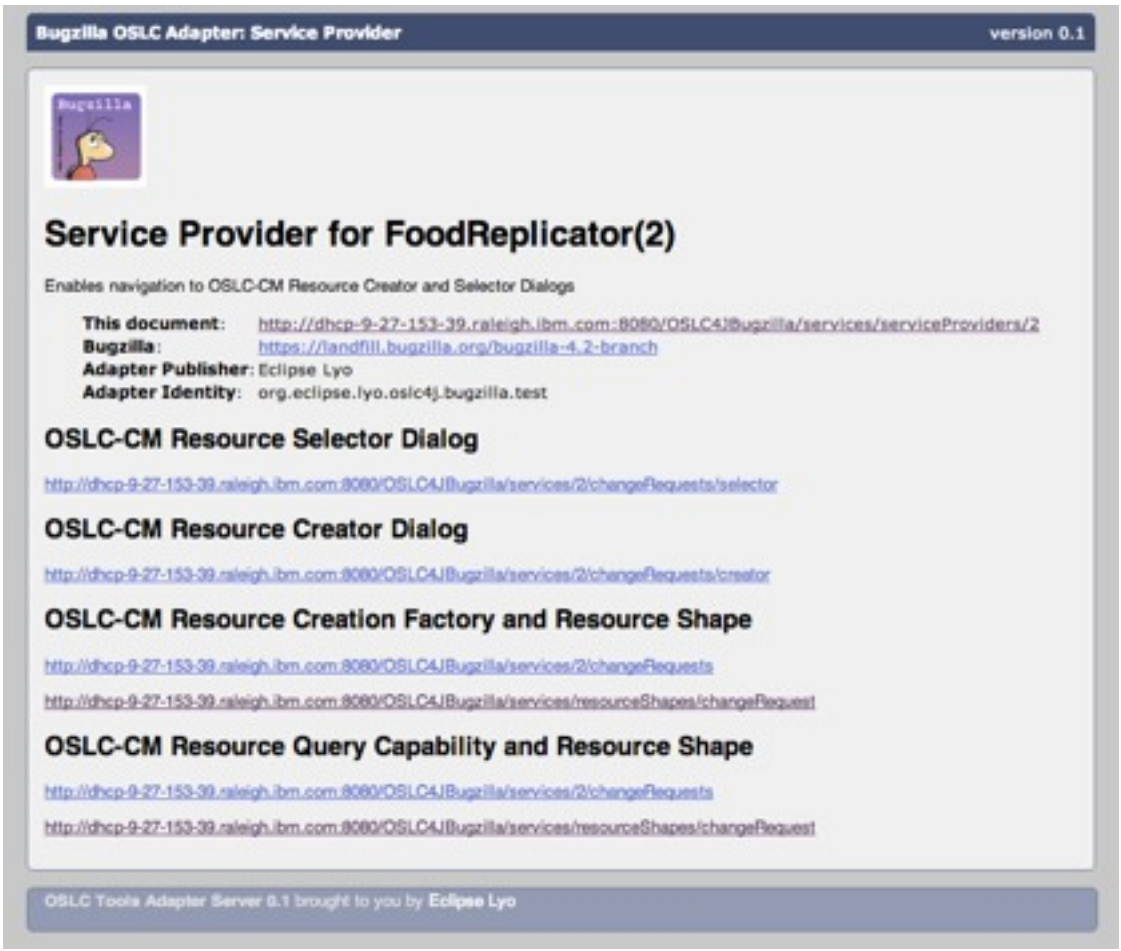
And towards the bottom, you'll find the HTML where we display those URLs:

```
<h2>OSLC-CM Resource Selector Dialog</h2>
<p><a href="<%= selectionDialog %>">
            <%= selectionDialog %></a></p>

<h2>OSLC-CM Resource Creator Dialog</h2>
<p><a href="<%= creationDialog %>">
            <%= creationDialog %></a></p>

<h2>OSLC-CM Resource Creation Factory and Resource Shape</h2>
<p><a href="<%= creationFactory %>">
            <%= creationFactory %></a></p>
<p><a href="<%= creationShape %>">
            <%= creationShape %></a></p>

<h2>OSLC-CM Resource Query Capability and Resource Shape</h2>
<p><a href="<%= queryCapability %>">
            <%= queryCapability %></a></p>
<p><a href="<%= queryShape %>">
            <%= queryShape %></a></p>
```

Browsing the Service Provider Catalog and Service Providers

If you're running the example applications, browse to http://localhost:8080/OSLC4JBugzilla/services/catalog/singleton.

Click on the link for any Service Provider (the number of Service Providers you'll see depends on the number of available Products on your Bugzilla server). You should see an HTML page with links to the available REST services, similar to this:



Providing RDF+XML or JSON representations of Service Providers and Service Provider Catalogs

Although the HTML representations we created above are useful as an educational and debugging tool, to connect to another tool (and comply with the specification!) we'll need to also create machine-readable formats, specifically RDF+XML and JSON.

Providing RDF+XML or JSON representations manually

One way you could create RDF+XML or JSON representations of these OSLC resources would be nearly the same as the HTML representation: build and gather the data for the resource and dispatch a JSP template to output it in the proper format.

Your JSP template for RDF+XML for the ServiceProvider might look similar to this:

```
<?xml version="1.0" encoding="UTF-8"?>
<%@ page contentType="application/rdf+xml" language="java"%>
<%@ page import="java.net.URI" %>
<%@ page import="jbugz.base.Product" %>
<%
// Load up the data sent in with the JSP template
String bugzillaUri = (String)
request.getAttribute("bugzillaUri");
Service service = (Service)request.getAttribute("service");
ServiceProvider serviceProvider =
(ServiceProvider)request.getAttribute("serviceProvider");

// Build the OSLC dialogs here
// ...

%>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:dcterms="http://purl.org/dc/terms/"
        xmlns:oslc="http://open-services.net/ns/core#">

<oslc:ServiceProvider
    rdf:about="<%= serviceProvider.getAbout() %>">
<dcterms:title>OSLC-CM Adapter/Bugzilla Service Provider:
Product <%= serviceProvider.getTitle() + "(" +
serviceProvider.getIdentifier() + ")" %></dcterms:title>
<dcterms:description>
Enables navigation to OSLC-CM Resource Creator and Selector
Dialogs
</dcterms:description>

<oslc:service>
    <oslc:Service>
        <oslc:domain rdf:resource="http://open-services.net/ns/
cm#" />
        <!-- URLs to your OSLC services (dialogs, etc.) will go
here -->
    </oslc:Service>
</oslc:service>

</oslc:ServiceProvider>
</rdf:RDF>
```

Although using another JSP template gives you complete control over the output, you have a higher risk of creating improperly formatted output. There are other options:

- **Use an RDF API**: build an RDF graph of triples and then serialize it into RDF+XML. This ensures proper formatting, but it's another API to work with.
- **Use an XML DOM API**: build up a DOM and then serialize it to XML.

Use whatever you're most comfortable with. However, with Eclipse Lyo there's an easier way.

Providing RDF+XML or JSON representations with OSLC4J

Open the `ServiceProviderService.java` class in the org.eclipse.lyo.oslc4j.bugzilla.services package and check out thegetServiceProvider method:

```
@GET
@Path("{serviceProviderId}")
@Produces({OslcMediaType.APPLICATION_RDF_XML,
OslcMediaType.APPLICATION_XML, OslcMediaType.APPLICATION_JSON})
public ServiceProvider
getServiceProvider(@PathParam("serviceProviderId") final String
serviceProviderId)
{
    httpServletResponse.addHeader("Oslc-Core-Version","2.0");
    return
ServiceProviderCatalogSingleton.getServiceProvider(httpServletRe
quest, serviceProviderId);
}
```

That's it for outputting Service Providers in a XML, RDF+XML, and JSON! You'll find similar code for the Service Provider Catalog in the `ServiceProviderCatalogService.java` file.

What's happening is the OSLC4J toolkit includes JAX-RS message body writers that serialize the Java representation of the Service Provider (or any OSLC resource) to RDF+XML, JSON, or XML. Likewise, it can convert OSLC resources in any of those formats back into Java objects. We'll explore this more in the next topic.

VIEWING THE MACHINE-READABLE FORMATS OF A SERVICE PROVIDER CATALOG

Let's try it out!

1. In Firefox or Chrome, open the Poster plugin. Poster is a browser plugin (for Firefox and Chrome) which can be used to send HTTP REST requests with full control over HTTP headers and their values.
2. For the **URL** field, type the URL for the Service Provider Catalog: http://localhost:8080/OSLC4JBugzilla/services/catalog/singleton
3. For the **User Auth** fields, type your Bugzilla username and password.

4. On the Headers tab, for the **Name** type `Accept` and for the **Value** type any of the following:
5. application/rdf+xml
6. application/json
7. application/xml
8. Then, click **Add/Change** to add the `Accept` header.
9. Click **Get** to execute the HTTP GET method. You should receive the complete Service Provider Catalog in the format you requested via `Accept` header. OSLC4J and JAX-RS produce the correct serialization based on the `Accept` header.

Next, try it with the URL for one of the Service Providers. (The exact URL will depend on the Product ID of the products on your Bugzilla server.)

Now, a client can start with a single URL (for the catalog) and navigate to all of the Service Providers. A client could use this to show a list of Products to a user and allow them to pick which ones to report bugs against, or query for existing bugs.

Providing OSLC representations of Bugzilla bugs

In the previous section we noted that we used OSLC4J to transform Plain Old Java Object (POJO) representations of OSLC resources into RDF, XML, and JSON formats. In this section, we'll look more closely at how OSLC4J defines OSLC resources. Then we'll make Bugzilla Bugs available as OSLC Change Management resources in a variety of formats.

What is OSLC4J?

OSLC4J, part of the Eclipse Lyo project, 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.

Defining OSLC resources with OSLC4J

OSLC4J comes with a sample Change Management application that includes the OSLC4J-annotated Java class representing a Change Request (as defined in the OSLC Change Management v2 specification).

The OSLC4J Bugzilla adapter includes that class (**ChangeRequest**) and extends it with Bugzilla-specific attributes (for example, "Product", "Platform", or other attributes that are not part of the OSLC CM specification); this extended change request is called a **BugzillaChangeRequest**.

Exploring the OSLC4J ChangeRequest class

Open the file `ChangeRequest.java` in the **org.eclipse.lyo.oslc4j.bugzilla.resources** package and explore the variables and methods. For reference, here is the definition of a Change Request in the OSLC Change Management specification.

First, observe the private variables at the top of the **ChangeRequest** class. These are the attributes of an OSLC CM V2.0 Change Request. Here are first several, which represent the relationships between Change Requests and other OSLC artifacts:

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

Further down are the primitive attributes of a Change Request:

```
private Boolean  approved;
private Boolean  closed;
private Date     closeDate;
private Date     created;
private String   description;
```

Further down, each attribute has an associated getter method. For example, here's the `getIdentifier()` method:

```
@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;
}
```

Note the OSLC-specific annotations before the method. These are used to not only automatically create OSLC resource shape documents, service provider documents, and service provider catalogs, but also assist with the serialization of Java objects to RDF or JSON:

- `@OslcOccurs` provides the cardinality of the attribute.
- `@OslcPropertyDefinition` providers the namespace qualified attribute name
- `@OslcReadOnly` indicates this attribute should appear in the resource shape as read only

Because the *default type in OSLC4J is a string*, there is no type annotation. Look for other attributes with the `@OslcValueType` annotation for examples of attributes that are not strings.

Extending ChangeRequest with Bugzilla attributes

Open the file `BugzillaChangeRequest.java` in the **org.eclipse.lyo.oslc4j.bugzilla.resources** package and explore the variables and methods for the Bugzilla-specific attributes.

As with the **ChangeRequest** class, the various getter methods (for example, `getVersion()`) have OSLC annotations.

MAPPING BUGZILLA ATTRIBUTES TO OSLC-CM PROPERTIES

To represent a Bugzilla bug as an RDF/XML document for an OSLC Change Management resource, we must map Bugzilla bug attributes to OSLC-CM ChangeRequest properties. The following attributes line up fairly clearly:

Bugzilla bug field	Maps to RDF Predicate	Prefixed name*
id	http://purl.org/dc/terms/identifier	dcterms:identifier
summary	http://purl.org/dc/terms/title	dcterms:title
status	http://open-services.net/ns/cm#status	oslc_cm:status
assigned_to	http://purl.org/dc/terms/contributor	dcterms:contributor
creation_time	http://purl.org/dc/terms/created	dcterms:created
last_change_time	http://purl.org/dc/terms/modified	dcterms:modified

(Prefix may be different depending on namespace prefix declaration in the XML)*

In the **BugzillaChangeRequest** class, the `fromBug()` method sets these properties. Near the top of the method, here is the code that sets the properties `dcterms:identifier`, `dcterms:title`, and `oslc_cm:status` properties from (respectively) the ID, Summary, and Status of the Bugzilla bug:

```
BugzillaChangeRequest changeRequest = new
BugzillaChangeRequest();
changeRequest.setIdentifier(bug.getID());
changeRequest.setTitle(bug.getSummary());
changeRequest.setStatus(bug.getStatus());
```

Bugzilla bugs also have attributes that do not map to any OSLC Change Management properties but that are required for Bugzilla. We should make these available in our RDF/XML representations by using the RDF predicates that Bugzilla defines for Bug lists, and we'll use the unique namespace `bugz` as a prefix (defined in `Constants.java` as shorthand forhttp://www.bugzilla.org/rdf#):

Bugzilla bug field	Maps to RDF Predicate	Prefixed name*
product	http://www.bugzilla.org/rdf#product	bugz:product
component	http://www.bugzilla.org/rdf#component	bugz:component
version	http://www.bugzilla.org/rdf#version	bugz:version
priority	http://www.bugzilla.org/rdf#priority	bugz:priority
platform	http://www.bugzilla.org/rdf#platform	bugz:platform
op_sys	http://www.bugzilla.org/rdf#op_sys	bugz:operatingSystem

(Prefix may be different depending on namespace prefix declaration in the XML)*

We set these properties in the `fromBug()` method in the **BugzillaChangeRequest** class. Here's the code that sets `bugz:product` and `bugz:component`:

```
changeRequest.setProduct(bug.getProduct());
changeRequest.setComponent(bug.getComponent());
```

You can explore the `fromBug()` method to see how the other properties are set.

Providing OSLC representations of Bugzilla bugs

Like with the **ServiceProviderService** and **ServiceProviderCatalogService** (discussed in in more detail in the previous section), the **BugzillaChangeRequestService** class has many JAX-RS methods to handle both

collections of BugzillaChangeRequests and individual BugzillaChangeRequests with a a variety of HTTP requests and output formats.

Open `BugzillaChangeRequestService.java` in the org.eclipse.lyo.oslc4j.bugzilla.services package.

Note the `@Path` annotation near the top of the class:

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

Recall that in `ServiceProviderCatalogSingleton.java` we registered a Service Provider for every Bugzilla product and used the product ID of the product in the URL for the Service Provider. So if the ID of a Bugzilla product is **2**, our base URL for the **BugzillaChangeRequestService** methods will be:

```
http://hostname:8080/OSLC4JBugzilla/services/2/changeRequests
```

Providing representations of Bugzilla Bugs as RDF+XML or JSON

As with Service Providers and the Service Provider Catalog, with OSLC4J we do not have to write manually code the RDF or JSON representation of a bug; the message body writers in OSLC4J automatically serialize the Java object into the appropriate machine-readable format.

The output is handled by the following methods in the **BugzillaChangeRequestService** class:

- `getChangeRequests()`: RDF/XML, XML and JSON representation of a change request collection
- `getChangeRequest()`: RDF/XML, XML, and JSON representation of a single change request

Without OSLC4J you could **dispatch a JSP template**, use an **RDF API**, or use an **XML DOM API** to generate the appropriate output format.

VIEWING THE RDF+XML OR JSON REPRESENTATION OF A COLLECTION OF BUGZILLA BUGS

The following assumes the Bugzilla adapter is running at `localhost:8080/OSLC4JBugzilla`

In Firefox or Chrome, open the Poster plugin.

For the **URL** field, type the URL for a list of all the bugs for a product:

```
http://localhost:8080/OSLC4JBugzilla/services/{productId}/changeRequests
```

For example, with a product ID of **1**:

```
http://localhost:8080/OSLC4JBugzilla/services/1/changeRequests
```

For the **User Auth** fields, type your Bugzilla username and password.

On the Headers tab, for the **Name** type `Accept` and for the **Value** type any of the following:

- `application/rdf+xml`
- `application/json`
- `application/xml`

Then, click **Add/Change** to add the `Accept` header.

Click **Get** to execute the HTTP GET method and you should receive the collection of bugs in the format you requested via`Accept` header. OSLC4J and JAX-RS produce the correct serialization based on the `Accept` header.

VIEWING THE RDF+XML OR JSON REPRESENTATION OF A BUGZILLA BUG

Follow the procedure above for a collection of bugs, but for the **URL** field, type the URL for a single bug:

```
http://localhost:8080/OSLC4JBugzilla/services/{productId}/
changeRequests/{bugId}
```

For example, with a Product ID of **1** and a Bug ID of **10**:

```
http://localhost:8080/OSLC4JBugzilla/services/1/changeRequests/
10
```

Click **Get** and you should receive the bug in the format you requested via `Accept` header.

Displaying a collection of Bugzilla bugs as HTML

OSLC4J can simplify providing collections of bugs in machine-readable formats, but we should also provide a more human-friendly HTML listing of Bugzilla bugs.

In `BugzillaChangeRequestService.java` in the **org.eclipse.lyo.oslc4j.bugzilla.services** package find the `getHtmlCollection()` method.

```
@GET
@Produces({ MediaType.TEXT_HTML })
public Response getHtmlCollection( @PathParam("productId") final
String productId, […] ) throws ServletException, IOException
{
    /* [code for returning a list of bugs with HTML] */
}
```

This method's basic activity is to retrieve a list of bugs for a Bugzilla product…

```
final List<BugzillaChangeRequest> results =
  BugzillaManager.getBugsByProduct(
    httpServletRequest,
    productId,
    page,
    limit,
    where,
    prefixMap,
    propMap,
    orderBy,
    searchTerms);
```

… and dispatch that list to a template (`/cm/changerequest_collection_html.jsp`):

```
httpServletRequest.setAttribute("results", results);

/**
 * …
 */

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

There are multiple parameters for this function that allow you to filter the collection with queries, paginate the results, and change the sort order. The Bugzilla Adapter does not use all of these parameters; however they are necessary for full support of OSLC Queries.

Open the file `/src/webapp/cm/changerequest_collection_html.jsp` in **OSLC4JBugzilla**. The HTML layout is nearly identical to that of the Service Providers and Catalog.

Towards the top, you'll see that we receive the data:

```
<%
  List<BugzillaChangeRequest> changeRequests =
(List<BugzillaChangeRequest>) request.getAttribute("results");
  ServiceProvider serviceProvider = (ServiceProvider)
request.getAttribute("serviceProvider");
  String bugzillaUri = (String)
request.getAttribute("bugzillaUri");
  String queryUri = (String)request.getAttribute("queryUri");
  String nextPageUri =
(String)request.getAttribute("nextPageUri");
%>
```

And towards the bottom of the file, we loop through the list of bugs and output the title/ summary and a link as HTML:

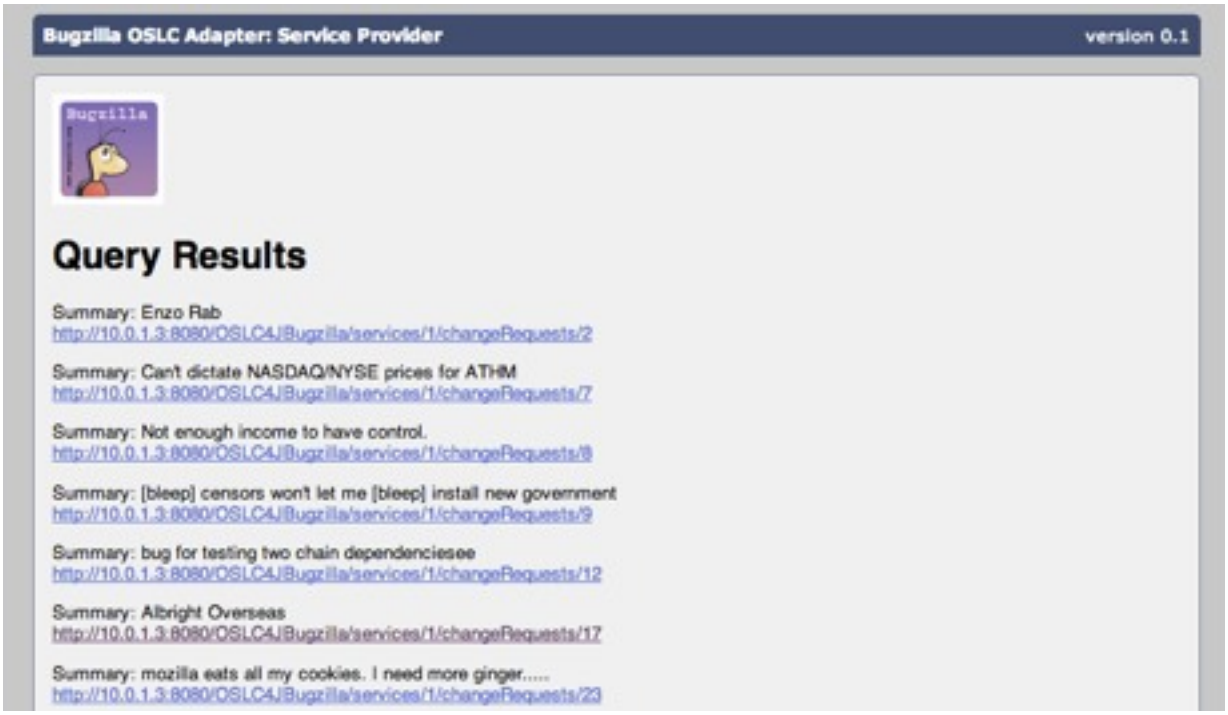
```
<h1>Query Results</h1>

<% for (BugzillaChangeRequest changeRequest : changeRequests)
{ %>
<p>Summary: <%= changeRequest.getTitle() %><br /><a href="<%=
changeRequest.getAbout() %>">
  <%= changeRequest.getAbout() %></a></p>
<% } %>
```

BROWSING ALL THE BUGS FOR A BUGZILLA PRODUCT

Let's try it out! From the Service Provider Catalog, you can navigate to a list of all bugs for a product.

1. If you're running the example applications, browse tohttp://localhost:8080/OSLC4JBugzilla/services/catalog/singleton.
2. Click on the link for any Service Provider for a product (for example, if the product ID is "1":http://localhost:8080/OSLC4JBugzilla/services/serviceProviders/1.
3. Then click on the first link under the **OSLC-CM Resource Query Capability and Resource Shape** heading. For example, if the product ID is "1": http://localhost:8080/OSLC4JBugzilla/services/1/changeRequests

You should see a page with links to the bugs, similar to this:



An HTML page in a browser showing a list of Bugzilla bugs

Forwarding HTML requests for single Bugzilla bugs

The Bugzilla application itself can create an HTML page with all the details about a bug – that's one of its primary features – so why recreate the wheel?

In the **BugzillaChangeRequestService** class, note the `getHtmlChangeRequest()` method:

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

Simple enough: given the ID number (`{changeRequestId}`) for a particular bug, the OSLC Bugzilla adapter will forward you directly to the web page for the bug in Bugzilla (`show_bug.cgi?id={changeRequestId}`). For example, a request from the adapter for bug 531 in Product 1 at the following URL…

```
http://localhost:8080/OSLC4JBugzilla/services/1/changeRequests/
531
```

… will forward you to the Bugzilla page for the bug at this URL:

```
https://bugzilla-host.example.com/show_bug.cgi?id=531
```

That's useful on its own, but OSLC also specifies a method called UI Preview for showing preview information about a resource in another tool. We'll tackle these rich preview formats in the next section.

Providing UI Previews

Once you establish relationships between resources using links, 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 that are in another application. 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 or focus on it, they can see a brief preview of that resource in a tool-tip or a pop-up window.

For all the details, see the OSLC UI Preview specification. (Don't worry; this one is pretty short.)

In this section, we'll explore how to make our Bugzilla Adapter into a provider of UI Previews. 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. Later in this tutorial, you'll be able to see your UI Preview in the NinaCRM application, because NinaCRM will support UI Preview as a Consumer.

Add UI Preview handling to the BugzillaChangeRequestService class

In the previous section we explored how the **BugzillaChangeRequestService** class handles requests for collections of**BugzillaChangeRequests** or individual **BugzillaChangeRequests**.

To add UI Preview support, we will add two methods to the service:

1. Provide an OSLC Compact Representation of a **BugzillaChangeRequest**.
2. Provide what is known as a small HTML preview of a **BugzillaChangeRequest**.

Provide the compact XML representation of a Bugzilla bug

Open the file `BugzillaChangeRequestService.java` in the org.eclipse.lyo.oslc4j.bugzilla.servcies package and find the `getCompact()` method.

Note the `@Produces` annotation:

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

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

The method first fetches the bug and converts it to a BugzillaChangeRequest:

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

Then, it copies the “About” and “Title” attributes:

```
compact.setAbout(getAboutURI(productId + "/changeRequests/" +
changeRequest.getIdentifier()));
compact.setTitle(changeRequest.getTitle());
```

Then to help identify the source of the bug, we add the Bugzilla icon (from the server) to our compact representation:

```
String iconUri = BugzillaManager.getBugzillaUri() + "/images/
favicon.ico";
compact.setIcon(new URI(iconUri));
```

Now we’ll build two Preview objects, `smallPreview` and `largePreview`, and pointers to the `smallPreview()` and `largePreview()` services in **BugzillaChangeRequestService**:

```
//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(new URI(compact.getAbout().toString() +
"/largePreview"));
compact.setLargePreview(largePreview);
```

And finally return the compact XML:

```
return compact;
```

As with many other methods, we needed to create the Compact and Preview objects and then let OSLC4J take care of serializing them to RDF.

VIEWING THE COMPACT XML REPRESENTATION OF A BUG.

The following assumes the Bugzilla adapter is running at `localhost:8080/OSLC4JBugzilla`

In Firefox or Chrome, open the Poster plugin.

For the **URL** field, type the URL for a single bug:

```
http://localhost:8080/OSLC4JBugzilla/services/{productId}/
changeRequests/{bugId}
```

For example, with a product ID of **1** and a bug ID of **10**:

```
http://localhost:8080/OSLC4JBugzilla/services/1/changeRequests/
10
```

For the **User Auth** fields, type your Bugzilla username and password.

On the Headers tab, for the **Name** type `Accept` and for the **Value** type the following:

```
application/x-oslc-compact+xml
```

Then, click **Add/Change** to add the `Accept` header.

Click **Get** to execute the HTTP GET method and you should receive OSLC compact XML representation of the bug. Examine the output to see how the `oslc:smallPreview` and `oslc:largePreview` resources are defined in the `oslc:Compact` resource.

If a consumer application wants to display a small or large preview of a **BugzillaChangeRequest**, the application can find the URLs to them using this `x-oslc-compact+xml` representation.

Now, let's set up the HTML for these previews.

Creating a method and JSP template for UI Previews

In the last section, the `getCompact()` method in the **BugzillaChangeRequestService** created preview resources pointing to `changeRequests/{id}/smallPreview` and `changeRequests/{id}/largePreview`.

Open the file `BugzillaChangeRequestService.java` in the **org.eclipse.lyo.oslc4j.bugzilla.servcies** package and find the `smallPreview()` method:

```
@GET
@Path("{changeRequestId}/smallPreview")
@Produces({ MediaType.TEXT_HTML })
public void smallPreview(@PathParam("productId")       final
String productId,
                         @PathParam("changeRequestId") final
String changeRequestId) throws ServletException, IOException,
URISyntaxException
{
    // Method code here
}
```

The `smallPreview()` method first fetches the bug and converts it to a **BugzillaChangeRequest**:

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

Then, it sets some attributes and dispatches a JSP:

```
BugzillaChangeRequest changeRequest =
BugzillaChangeRequest.fromBug(bug);

changeRequest.setServiceProvider(
  ServiceProviderCatalogSingleton.getServiceProvider(
    httpServletRequest,
    productId).getAbout());
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);
```

Now, let's look at that JSP template. Open the file `/src/webapp/cm/changerequest_preview_small.jsp` in **OSLC4JBugzilla** and browse the contents. The code near the top extracts the fields we want from the Change Request:

```
<%
BugzillaChangeRequest changeRequest =
(BugzillaChangeRequest)request.getAttribute("changeRequest");
String bugzillaUri = (String)
request.getAttribute("bugzillaUri");

Date createdDate = (Date) changeRequest.getCreated();
SimpleDateFormat formatter = new SimpleDateFormat();
String created = formatter.format(createdDate);
Date modifiedDate = (Date) changeRequest.getModified();
String modified = formatter.format(modifiedDate);

Person assigneePerson = (Person)
changeRequest.getContributors().get(0);
String assignee = "Unknown";
if (assigneePerson != null)
    assignee = assigneePerson.getMbox();
%>
```

Then those fields are output in a small table:

```
<table class="edit_form">
    <tr>
        <th>Status:</th>
        <td><%= changeRequest.getStatus() %></td>
        <th>Product:</th>
        <td><%= changeRequest.getProduct() %></td>
    </tr>

    <tr>
        <th>Assignee:</th>
        <td><%= assignee %></td>
        <th>Component:</th>
        <td><%= changeRequest.getComponent() %></td>
    </tr>

    <tr>
        <th>Priority:</th>
        <td><%= changeRequest.getPriority() %></td>
        <th>Version:</th>
        <td><%= changeRequest.getVersion() %></td>
    </tr>

    <tr>
        <th>Reported:</th>
        <td><%= created %></td>
        <th>Modified:</th>
        <td><%= modified %></td>
    </tr>
</table>
```

The method `largePreview()` works in a similar manner and uses the JSP template `changerequest_preview_large.jsp`.

VIEWING THE UI PREVIEW

With the adapter running, in a web browser navigate to the following URL:

```
http://localhost:8080/OSLC4JBugzilla/services/{productId}/
changeRequests/{bugId}/smallPreview
```

For example, with a product ID of **1** and a bug ID of **10**:

```
http://localhost:8080/OSLC4JBugzilla/services/1/changeRequests/
10/smallPreview
```

You should see a small table with details about the bug, similar to this:

A small table of information about a bug in the browser

Now, given a URI link to a resource, we can provide some human-readable and usable presentations for that link, including a quick peak into the Bug using UI Preview. Later in this tutorial we'll explore how other applications can discover and display these previews.

Providing a delegated UI for selection and search

Now that we have the basic services and resources defined for BugzillaChangeRequests (our Bugzilla-specific extension of OSLC ChangeRequests), our next step is to implement delegated user interface (UI) dialogs to allow the following actions from an OSLC Consumer:

- Search for and select Bugzilla bugs
- Create a new bug in Bugzilla

For example, a user of the NinaCRM product will be able to search for and add links to related bugs in Bugzilla without leaving the NinaCRM interface.

For more information on OSLC delegated UIs, see the section about them in the OSLC core specification.

Because delegated UI dialogs must accept user input and interact with Bugzilla to select or create bugs, they are more complex than collecting and describing bugs. Here's how we'll approach the process:

1. See how OSLC4J helps provide delegated UI locations to the Service Provider document
2. Understand how to define a basic dialog with the J2Bugzilla API and generate the appropriate responses
3. Add methods to **BugzillaChangeRequestService** for the selection and creation of change requests
4. Add forms and JavaScript code to handle interacting with the consumer of the dialogs
5. Test the dialogs to ensure the appropriate response is given

Adding the location of delegated UI dialogs to Service Providers

Because we're using OSLC4J, it's relatively trivial to add links to delegated UIs to our Service Provider documents.

Open `BugzillaChangeRequestService.java` in the **org.eclipse.lyo.oslc4j.bugzilla** package and search for `@OslcDialogs` (note the plural).

These use the imported annotations from OSLC4J:

- org.eclipse.lyo.oslc4j.core.annotation.OslcDialog,
- org.eclipse.lyo.oslc4j.core.annotation.OslcDialogs;, and
- org.eclipse.lyo.oslc4j.core.annotation.OslcQueryCapability;

Here are the annotations for the selection dialog and the capability to query for bugs:

```
@OslcDialogs(
{
    @OslcDialog
    (
         title = "Change Request Selection Dialog",
         label = "Change Request Selection Dialog",
         uri = "/{productId}/changeRequests/selector",
         hintWidth = "525px",
         hintHeight = "325px",
         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}
)
```

As with BugzillaChangeRequests, with the appropriate annotations OSLC4J handles the conversion of this information to XML or JSON for you – no additional templating required.

You can explore `/src/main/webapp/cm/serviceprovider_html.jsp` to see how we add the links to these to the HTML representation of a Service Provider (under the **Resource Selector Dialog** heading).

With this enabled, we've defined that the dialog for selecting or querying for bugs will be at the following URL (assuming your adapter is running at `localhost` and port `8080`):

```
http://localhost:8080/OSLC4JBugzilla/services/{productID}/
changeRequests/selector
```

Next, we must create the dialog.

Adding the dialog to search for and select bugs

As with our methods for returning BugzillaChangeRequests, we add another method to our**BugzillaChangeRequestService** class to handle requests for a delegated UI to select bugs.

In our OSLC4J Bugzilla Adapter, open `BugzillaChangeRequestService.java` and search for the `changeRequestSelector()` method:

```
@GET
@Path("selector")
@Consumes({ MediaType.TEXT_HTML, MediaType.WILDCARD })
public void changeRequestSelector(
    @QueryParam("terms")     final String terms,
    @PathParam("productId")  final String productId
    ) throws ServletException, IOException
{
    int productIdNum = Integer.parseInt(productId);
    httpServletRequest.setAttribute("productId", productIdNum);
    httpServletRequest.setAttribute("bugzillaUri",
BugzillaManager.getBugzillaUri());

httpServletRequest.setAttribute("selectionUri",uriInfo.getAbsolu
tePath().toString());

    if (terms != null ) {
        httpServletRequest.setAttribute("terms", terms);
        sendFilteredBugsReponse(httpServletRequest, productId,
terms);

    } else {
        try {
            RequestDispatcher rd =
httpServletRequest.getRequestDispatcher("/cm/
changerequest_selector.jsp");
            rd.forward(httpServletRequest, httpServletResponse);

        } catch (Exception e) {
            throw new ServletException(e);
        }
    }

}
```

In short, this method defines two things that could happen if you make a request to a URL with `/changeRequests/selector`:

- If the request URL *does not* have a `terms` parameter, we dispatch a JSP template (`changerequest_selector.jsp`) to display a form for the user to fill in the search terms and select a bug.
- If the request *does* have a `terms` parameter, the request is an AJAX request coming from the delegated UI that is running in a user's web browser (code that we'll discuss below). We call the `sendFilteredBugsReponse()` method that performs a Bugzilla search (using the J2Bugzilla API) and returns the results in a compact JSON format.

Let's explore the delegated UI to fill in search terms and select bugs in the JSP template.

Creating the delegated UI for selection

Open `/src/main/webapp/cm/changerequest_selector.jsp` and explore the contents.

The JSP page `changerequest_selector_dialog.jsp` provides the HTML and JavaScript for a Change Request Selector Dialog. There's a little bit of HTML, because the dialog is pretty simple.

Because a fair amount of JavaScript code is required to allow a user to enter search terms, display search results, allow the user to make a selection and then notify the delegated UI consumer that a selection has been made, we have stashed those methods in a `bugzilla.js` file that we load in the `<head>` of the returned HTML:

```
<script type="text/javascript" src="../../../bugzilla.js"></
script>
```

We'll explore the various JavaScript methods below.

Allowing users to search for bugs

In `changerequest_selector.jsp`, here's the form for the user to enter search terms:

```
<input type="search"
    style="width: 335px"
    id="searchTerms"
    placeholder="Enter search terms"
    autofocus>
<button
    type="button"
    onclick="search( '<%= selectionUri %>' )">Search</button>
```

Note that when you click the `<button>`, we call the JavaScript function `search()`, which we define in the file `src/main/webapp/bugzilla.js`.

Open `bugzilla.js`. Near the top, explore the `search()` function.

```
function search(baseUrl){
    var ie = window.navigator.userAgent.indexOf("MSIE");
    list = document.getElementById("results");
    list.options.length = 0;
    var searchMessage =
document.getElementById('searchMessage');
    var loadingMessage =
document.getElementById('loadingMessage');
    xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState==4 && xmlhttp.status==200) {
            // populate results
            txt = xmlhttp.responseText;
            resp = eval('(' + txt + ')');
            for( var x=0; x<resp.results.length; x++ ) {
                var item=document.createElement('option');
                item.text = resp.results[x].title;
                item.value = resp.results[x].resource;
                if (ie > 0) {
                     list.add(item);
                } else {
                     list.add(item, null);
                }
            }

            searchMessage.style.display = 'block';
            loadingMessage.style.display = 'none';
        }
    };
    terms = document.getElementById("searchTerms").value;
    xmlhttp.open("GET", baseUrl + "?terms=" +
encodeURIComponent(terms), true);
    searchMessage.style.display = 'none';
    loadingMessage.style.display = 'block';
    xmlhttp.send();
}
```

In short, this function does the following:

1. Removes any previous search results from the `<select id="results">` element
2. Creates an AJAX request
3. Get the search query from the value of the `<input id="searchTerms">` element.

4. Sends an AJAX request to `/changeRequests/selector` with a `terms=` parameter. Recall that this type of request to our adapter will run a search in Bugzilla and return the results in a JSON format.
5. A callback evaluates the search results with the `eval()` method, loops through the results, and adds each result as an `<option>` element to the `#results` element.
6. Reveals a loading message while making the request; hides the message when finished.

Now in our delegated UI, users can search for bugs in Bugzilla, see a list of results, and select amongst them from a `<select>` element. Next, we will allow them to do something with their selection.

Sending the selected bugs back to the OSLC consumer application

Go back to the file `changerequest_selector.jsp`. Just below the `<select id="results">` element that will hold search results, you'll see the form `<button>`s to submit the selected bug or bugs:

```
<button style="float: right;" type="button"
  onclick="javascript: cancel()">Cancel</button>
<button style="float: right;" type="button"
  onclick="javascript: select();">OK</button>
```

When you click the **OK** button, we call the JavaScript function `select()`, which is also defined in the `bugzilla.js` file.

Open `bugzilla.js` and search for the `select()` function:

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

This method finds the bug that the user has selected (`list.options[list.selectedIndex]`) and sends the title (`option.text`) and URL (`option.value`) of the bug in a response to the original window (the `sendResponse()` method).

Because we're sending data from a delegated UI in one browser window back to another application with a different host and in a different window, we must use either the `window.postMessage` method (if supported) or via `window.name` variables otherwise.

Explore the `sendResponse()` method to see how we build the JSON response and determine which cross-window method to use to send it to the requesting application:

```
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);
  }
}
```

We determine whether or not we want to use Window Name or postMessage by looking at the `location.hash` value for the page: either `#oslc-core-windowName-1.0` or `#oslc-core-postMessage-1.0`. We do this because the *requesting client* is the application that must indicate which cross-domain method *it* supports; the client does so by requesting our delegated UI with the appropriate hash. We'll explore this later when we access this delegated UI in the NinaCRM application.

You can further explore the `respondWithPostMessage()` and `respondWithWindowName()` methods in `bugzilla.js` to see how we send the data to the requesting window – it's taken almost entirely from the examples in the OSLC Core specification.

Note that the selection dialog above will show all Change Requests available for one Bugzilla Product. Some additional work could be done to make this more useful by doing some filtering up-front. For example, it might be useful to show only Change Requests that are assigned to the current user, or to prioritize recently created Change Requests.

We'll put this delegated UI to use later when we implement it in the NinaCRM sample application.

Next, we'll create a delegated UI that will allow users to create new bugs in Bugzilla.

Providing a delegated UI for creating bugs

Providing a delegated user interface (UI) dialog for creating new resources is similar to the process for providing one for selecting existing resources: we must create an HTML Form, the fields within that form, and then set up the server-side handling of the form submission.

Here's the plan:

1. Add the location of our delegated UI to our Service Provider representations
2. Create a utility method that accepts a **BugzillaChangeRequest** and creates a bug in Bugzilla
3. Create a service to handle requests to display a delegated UI to create bugs.
4. Create a service to accept a **BugzillaChangeRequest** via HTTP POST (sent from the delegated UI form) and create a new bug.

Adding the location of the delegated UI for creation to Service Providers

As with our delegated UI for selection, it's relatively easy to add the location of our delegated UI for creation to the various representations of service providers.

Open `BugzillaChangeRequestService.java` in the **org.eclipse.lyo.oslc4j.bugzilla** package and search for `@OslcDialog` (*not* plural). You'll see two occurrences: one near the top for the Selection Dialog and Query Capability and one farther down the Creation Dialog:

```
@OslcDialog
(
title = "Change Request Creation Dialog",
label = "Change Request Creation Dialog",
uri = "/{productId}/changeRequests/creator",
hintWidth = "600px",
hintHeight = "375px",
resourceTypes = {Constants.TYPE_CHANGE_REQUEST},
usages = {OslcConstants.OSLC_USAGE_DEFAULT}
)
```

With these annotations, OSLC4J handles the conversion of this information to XML or JSON for you – no additional templating required.

You can explore `/src/main/webapp/cm/serviceprovider_html.jsp` to see how to add the links to the HTML representation of a Service Provider (under the **Resource Creator Dialog** heading).

With this enabled, we've defined that the dialog for creating new bugs will be at the following URL (assuming your adapter is running at `localhost` and port `8080`):

```
http://localhost:8080/OSLC4JBugzilla/services/{productID}/
changeRequests/creator
```

Creating a bug from an OSLC BugzillaChangeRequest

Before we create the delegated dialog to create new bugs, we will need a server-side helper utility that can accept a **BugzillaChangeRequest** and use its information to create a new bug in Bugzilla.

Locate `BugzillaManager.java` in the `org.eclipse.lyo.oslc4j.bugzilla` package and explore the contents.

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.

Locate the `createBug()` method. We first retrieve bug properties from the passed **BugzillaChangeRequest**:

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

Next, if there are missing fields we set some defaults:

```
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);
}
```

Finally, we call j2bugzilla's methods to create a bug:

```
Bug bug = factory.createBug();
ReportBug reportBug = new ReportBug(bug);
bc.executeMethod(reportBug);
newBugId = Integer.toString(reportBug.getID());
```

And return the ID of the new bug:

```
return newBugId;
```

With that utility in place, we can now set up services for our application to serve up a delegated UI for a user to create a new bug.

Displaying the delegated UI to create new bugs

Retrieving valid field values from Bugzilla and dispatching a template

We will add another method to our **BugzillaChangeRequestService** class to create and display a delegated UI to create a new bug.

In our OSLC4J Bugzilla Adapter, open `BugzillaChangeRequestService.java` and search for the `changeRequestCreator()` method.

As with many of our other methods, we first establish which Bugzilla product we're working with from the URI:

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

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

Next, we use the j2bugzilla **GetLegalValues** API to retrieve the allowed values for the various bug fields. Here's the code for retrieving the legal values for the **Component** field:

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

We have similar code for the **Operating System**, **Platform**, and **Version** fields.

Finally, we set a few more attributes and dispatch them all to a .jsp template (`/cm/changerequest_creator.jsp`):

```
httpServletRequest.setAttribute("creatorUri",
uriInfo.getAbsolutePath().toString());
httpServletRequest.setAttribute("bugzillaUri",
BugzillaManager.getBugzillaUri());

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

Building the delegated UI to create new bugs

Now, open `/src/main/webapp/cm/changerequest_creator.jsp` and explore the contents.

As with the delegated UI for selection, note the addition of `bugzilla.js` in the `<head>` that has a variety of script methods that we'll explore soon.

```
<script type="text/javascript" src="../../../bugzilla.js"></
script>
```

Next, explore the HTML table and form. We populate the various `<select>` elements with the legal values that were passed in from the Java service. For example, here's the input for the **Component** field:

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

You'll see simliar code for the other fields. We also provide a free-form text `<input>` for the **Summary**…

```
<input name="summary" class="required text_input"
  type="text" style="width: 400px" id="summary" required
autofocus>
```

… and a `<textarea>` for the Description:

```
<textarea style="width: 400px; height: 150px;"
  id="description" name="description"></textarea>
```

Finally, when you click the **Submit Bug** button we call the JavaScript function `create()` (from the file `src/main/webapp/bugzilla.js`):

```
<input type="button"
  value="Submit Bug"
  onclick="javascript: create( '<%= creatorUri %>' )">
```

We'll explore the `create()` JavaScript method in more detail below.

Send the values for the new bug back to our adapter

Open `bugzilla.js` and explore the `create()` function:

```
function create(baseUrl){
  var form = document.getElementById("Create");
  xmlhttp = new XMLHttpRequest();
  xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState==4 && (xmlhttp.status==201)) {
```

```
      txt = xmlhttp.responseText;
      resp = eval('(' + txt + ')');
      // Send response to listener
      sendResponse(resp.title, resp.resource);
    }
  };
   var postData="";
   if (form.component) {
     postData +=
"&component="+encodeURIComponent(form.component.value);
   }
   if (form.summary) {
     postData +=
"&summary="+encodeURIComponent(form.summary.value);
   }
   if (form.version) {
     postData +=
"&version="+encodeURIComponent(form.version.value);
   }
   if (form.op_sys) {
     postData +=
"&op_sys="+encodeURIComponent(form.op_sys.value);
  }
  if (form.platform) {
    postData +=
"&platform="+encodeURIComponent(form.platform.value);
  }
  if (form.description) {
    postData +=
"&description="+encodeURIComponent(form.description.value);
  }
  xmlhttp.open("POST", baseUrl, true);
  xmlhttp.setRequestHeader("Content-type","application/x-www-
form-urlencoded");
  xmlhttp.setRequestHeader("Content-length",postData.length);
  xmlhttp.send(postData);
}
```

This method takes the values from our form, converts them into URL query string parameters, and sends data via POST to the `createHtmlChangeRequest()` method in our **BugzillaChangeRequestService** class via `XMLHttpRequest()`.

To see how our adapter handles that POST request, open `BugzillaChangeRequestService.java` and search for the `createHtmlChangeRequest()` method.

Note that this service expects an encoded URL via `POST` at the same URL path as our delegated UI (`"creator"`):

```
@POST
@Path("creator")
@Consumes({ MediaType.APPLICATION_FORM_URLENCODED})
```

Our `createHtmlChangeRequest()` first builds a **BugzillaChangeRequest** from the URL parameters:

```
BugzillaChangeRequest changeRequest = new
BugzillaChangeRequest();
changeRequest.setComponent(component);
changeRequest.setVersion(version);
changeRequest.setTitle(summary);
changeRequest.setOperatingSystem(op_sys);
changeRequest.setPlatform(platform);
changeRequest.setDescription(description);
```

Then we use the `createBug()` method from **BugzillaManager** (discussed above) to create a new bug in Bugzilla:

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

With the bug created, we gather some information about our new bug…

```
final Bug newBug =
BugzillaManager.getBugById(httpServletRequest, newBugId);
final BugzillaChangeRequest newChangeRequest =
BugzillaChangeRequest.fromBug(newBug);
URI about = getAboutURI(productId + "/changeRequests/" +
newBugId);
newChangeRequest.setAbout(about);

httpServletRequest.setAttribute("changeRequest",
newChangeRequest);
httpServletRequest.setAttribute("changeRequestUri",
newChangeRequest.getAbout().toString());
```

… and build a small JSON response to return.

```
httpServletResponse.setContentType("application/json");
httpServletResponse.setStatus(Status.CREATED.getStatusCode());
httpServletResponse.addHeader("Location",
newChangeRequest.getAbout().toString());
```

```
PrintWriter out = httpServletResponse.getWriter();
out.print("{\"title\": \"" +
getChangeRequestLinkLabel(newBug.getID(), summary) + "\"," +
        "\"resource\" : \"" + about + "\"}");
out.close();
```

Back in `bugzilla.js` the `onreadystatechange` callback evaluates that JSON response and uses the `sendResponse()` method (discussed in more detail in the previous section) to send some information about the new bug back to the consumer application (re-copied from above):

```
xmlhttp.onreadystatechange = function() {
  if (xmlhttp.readyState==4 && (xmlhttp.status==201)) {
    txt = xmlhttp.responseText;
    resp = eval('(' + txt + ')');
    // Send response to listener
    sendResponse(resp.title, resp.resource);
  }
};
```

We'll put this dialog to use in the NinaCRM application later on.

We now have the ability to use user interface delegation as a way to provide a simple way for consumer applications to both create and select bugs. We've also exposed this capability from our service provider resource definition.

Next, we'll explore how to make it easier for other applications to create new bugs programmatically.

Providing a creation factory

With OSLC you can allow people to create bugs via Delegated UI; however, like all UI approaches, an actual human user must be involved. What if you want to support automated bug creation; for example, enabling a build server to automatically create a bug whenever there is a test or a build failure?

To allow clients to create new bugs automatically, you need to support an OSLC Creation Factory as described in theOSLC Core specification.

Adding a method to the adapter to create BugzillaChangeRequests via HTTP POST

Recall that when we created a delegated UI for creating new bugs, we wrote code in the **BugzillaManager** class to use the **j2bugzilla** API for creation of bugs; we'll re-use the `createBug()` method for automated bug creation via POST.

Adding the Creation Factory to Service Provider documents

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

First search for the `@OslcCreationFactory` annotation:

```
@OslcCreationFactory
(
  title = "Change Request Creation Factory",
  label = "Change Request Creation",
  resourceShapes = {OslcConstants.PATH_RESOURCE_SHAPES + "/" +
Constants.PATH_CHANGE_REQUEST},
  resourceTypes = {Constants.TYPE_CHANGE_REQUEST},
  usages = {OslcConstants.OSLC_USAGE_DEFAULT}
)
```

Notice that we've specified a `resourceShapes` location; we'll cover that in more detail below.

As with our other services, OSLC4J uses this annotation and automatically adds the URI for the creation factory to our XML and JSON Service Provider documents. (You will have to manually add it to the `serviceprovider_html.jsp` template to add it to the HTML representation.)

Because we haven't set any different path, the creation factory will be available at the root path of our **BugzillaChangeRequestService** class which is `{productId}/`

`changeRequests`. For example, if your adapter is running at `localhost:8080` and the product ID is `1`, the creation factory will be at the following URL:

```
http://localhost:8080/OSLC4JBugzilla/services/1/changeRequests
```

Our adapter recognizes that you are invoking the Creation Factory (instead of requesting a listing of bugs) if you request that URL using HTTP POST (instead of GET).

Handling BugzillaChangeRequests received via POST

In `BugzillaChangeRequestService.java`, search for the the `addChangeRequest()` method. Note the JAX-RS annotations:

```
@POST
@Consumes({OslcMediaType.APPLICATION_RDF_XML,
OslcMediaType.APPLICATION_XML, OslcMediaType.APPLICATION_JSON})
@Produces({OslcMediaType.APPLICATION_RDF_XML,
OslcMediaType.APPLICATION_XML, OslcMediaType.APPLICATION_JSON})
```

The `@Consumes` annotation indicates that the method accepts a **BugzillaChangeRequest** in RDF/XML, XML, or JSON format; the `@Produces` annotation indicates that it will return the same.

The following code in the `addChangeRequest()` method creates a new Bugzilla bug from an OSLC **BugzillaChangeRequest** object (using the previously discussed `createBug()` method)::

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

Next, we convert the bug into a **BugzillaChangeRequest**:

```
final Bug newBug =
BugzillaManager.getBugById(httpServletRequest,
                    newBugId);

BugzillaChangeRequest newChangeRequest;

try {
  newChangeRequest = BugzillaChangeRequest.fromBug(newBug);
} catch (Exception e) {
  throw new WebApplicationException(e);
}
```

Then we return the new **BugzillaChangeRequest** as the body of a POST response:

```
URI about = getAboutURI(productId + "/changeRequests/" +
newChangeRequest.getIdentifier());
newChangeRequest.setServiceProvider(

ServiceProviderCatalogSingleton.getServiceProvider(httpServletRe
quest, productId).getAbout());
newChangeRequest.setAbout(about);
setETagHeader(getETagFromChangeRequest(newChangeRequest),
httpServletResponse);

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

Note that we set the `Location` header (via `Response.created()`) to the `about` URI for the new **BugzillaChangeRequest**; this is a SHOULD requirement of the Core specification.

Try it out!

If you can create new bugs on your Bugzilla application, you should be able to create a bug via HTTP to our adapter.

1. In Firefox or Chrome, open the **Poster** plugin.
2. In the **URL** field, type the URL for the Creation Factory (replace {ProductID} with a valid ID for a Bugzilla product):
   `http://oslc:8080/OSLC4JBugzilla/services/1/changeRequests`
3. In the **User Auth** fields, type your Bugzilla username and password.
4. On the **Headers** tab, for the **Name** type Content-Type and for the **Value** type application/rdf+xml
5. In the **Body** field enter the following example RDF/XML content. Change you@example.com to reflect your Bugzilla login/email; you might have to change some values depending on how your Bugzilla product has been configured, specifically `bugz:operatingSystem` and `bugz:component`.

```
 <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=you%40example.com">
          <foaf:mbox>you@example.com</foaf:mbox>
        </foaf:Person>
      </dcterms:contributor>
      <bugz:component>Server</bugz:component>
      <oslc_cm:severity>Unclassified</oslc_cm:severity>
    </oslc_cm:ChangeRequest>
 </rdf:RDF>
```

6. Click **Post** to execute the HTTP POST method. You should receive a **201 Created** status header and the response body should contain the RDF/XML **BugzillaChangeRequest** representation of the new bug.

Providing a ResourceShape document

To make it possible for client programs to automatically determine which BugzillaChangeRequest fields are required and the allowed values for those fields, we should provide a Resource Shape for every creation factory.

Resource Shapes are descriptive documents that define the set of OSLC Properties expected in a resource for specific operations (i.e. creation, update or query) along with the value types, allowed values, cardinality and optionality of each OSLC property. A client can use this information when creating new resources.

Fortunately, OSLC4J automates the creation of Resource Shape documents from our existing description of a **BugzillaChangeRequest**. All we must do is declare the location in our `@OslcCreationFactory` annotation:

```
resourceShapes = {OslcConstants.PATH_RESOURCE_SHAPES + "/" +
Constants.PATH_CHANGE_REQUEST}
```

Which indicates the resource shape will be located at the URI `resourceShapes/changeRequest`. For example if your Bugzilla adapter is running at `localhost:8080`, the Resource Shape will be available at this URL:

```
http://localhost:8080/OSLC4JBugzilla/services/resourceShapes/
changeRequest
```

You can open http://localhost:8080/OSLC4JBugzilla/services/resourceShapes/changeRequest in a browser to view the Resource Shape as an RDF/XML document.

(You can also use the Poster plugin with an `Accept` header of `application/json` to retrieve it in JSON format.)

Note that the document includes not only OSLC CM properties such as `relatedChangeRequest` or `inprogress`, but also Bugzilla-specific properties like `priority` and `version`; this indicates it was assembled from our **BugzillaChangeRequest** class.

Because these documents are not really meant to be human-readable, you don't have to build a HTML representation in a JSP template as we have for other resources.

Wrapping up

Our Bugzilla adapter now allows Bugzilla to be a reasonably complete OSLC CM provider application. In the next section, we'll take a different application (NinaCRM) and extend it to be an OSLC consumer that can take advantage of all the work we've done here.

Integrating with an OSLC provider

In this section, we'll integrate a sample homegrown Customer Relationship Management (CRM) with Bugzilla. Because Bugzilla (via our adapter that we built in the last section) is now an OSLC Change Management Provider, we can integrate NinaCRM with Bugzilla by making NinaCRM an OSLC-CM *Consumer*.

First up, we'll discuss the specific use cases that we want to support.

ASIDE: Why is the sample application called "NinaCRM"?

In older versions of this tutorial, we had a fictional protagonist named Nina who "designed" these applications. I've removed her story from this tutorial, but her legacy lives on in our sample application.

Sample use cases for an OSLC-CM Consumer

Our NinaCRM sample application presents a number of use cases that could benefit from OSLC.

NinaCRM is a Customer Relationship Management system that allows employees to store and track interactions with customers. First, let's explore how a typical interaction between a customer and a support representative:

1. Customer calls with a problem
2. Support rep brings up the record for the Customer, or creates one if necessary
3. Support rep finds the last incident involving the customer, or creates a new one if necessary.
4. Support rep searches for the customer's problem in the company's defect system (Bugzilla). If found, add the defect ID number to the Incident record.
5. If Defect includes work-around or fix, give it to the Customer
6. If the customer is satisfied with the solution, close the Incident

A better integration between Bugzilla and the NinaCRM system will make the process work more smoothly and efficiently. Here are the top items we want to target with our integration work:

1. **Linking Incidents to bugs is too difficult**: It takes support reps too much time to leave the CRM web UI and search for solutions to customer problems manually using Bugzilla. First, we should use OSLC UI previews to allow reps to see more details about linked bugs without leaving the NinaCRM application. Then we should automate the process of entering bugs by modifying NinaCRM system to use OSLC Delegated UI and let support reps search, create, and link to bugss without leaving the CRM web UI.
2. **Customer notifications are a manual process**: Customers can request notification whenever a specific bug is updated. Customer reps have to set aside time each week to review customer requests and check on bug status. We can surely automate this entire process, including writing and sending an email notification to each customer.

A Plan of Action

Here's our plan of action to add OSLC-CM support to NinaCRM.

Milestone 1: Use links and OSLC UI Preview

Modify NinaCRM to enable OSLC UI Preview for links to Bugzilla bugs

Milestone 2: Use OSLC Delegated UI for creating and selecting Bugzilla bugs to link to

Add to CRM's Incident page ability to link via Delegated UI

Milestone 3: Use OSLC protocol to automate customer notifications

- Create a program that can run as a scheduled job, e.g. via build system
- This program will query NinaCRM for list of notification requests and check the associated bugs
- If the bug has updated since last run, send an email to customer with summary

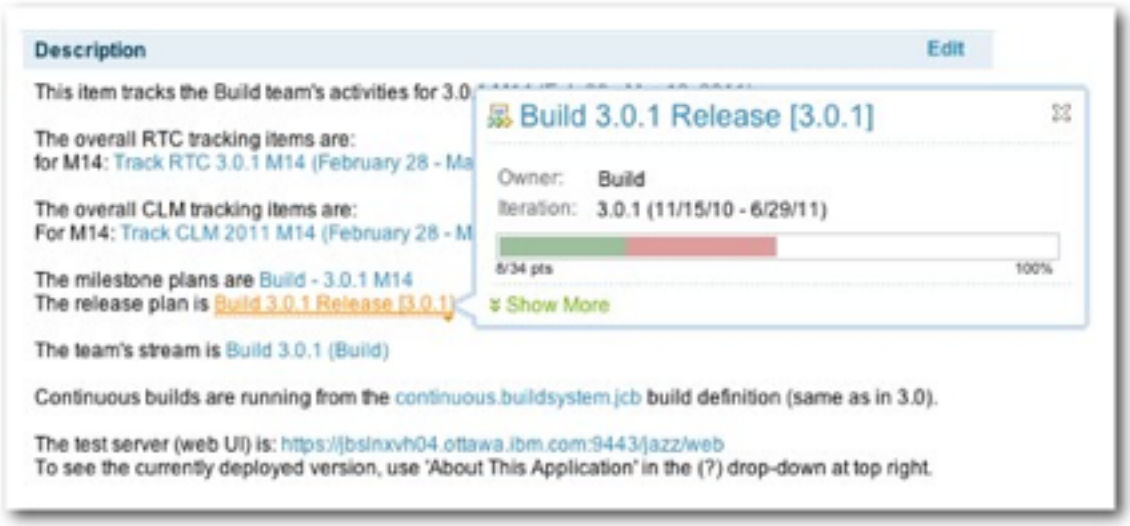
First up, we'll implement OSLC links and previews in NinaCRM.

Implementing links and UI previews

In this section, we'll add the ability to quickly preview linked resources to the NinaCRM sample application.

Introducing OSLC UI Preview

OSLC UI Previews makes it easy to show an in-context preview of a resource when a user "hovers" over the link to that resource, so the user can see what is at the other end and decide whether or not to click through to get more information. The illustration below shows UI Preview in action on IBM Rational's Jazz.net site. A user has put his mouse-pointer over a link to a Build and a preview of that build has appeared on the screen:



Sample UI preview

Here's how UI Preview works in an OSLC consumer:

1. Start with a link to a resource.
2. You send an HTTP GET request to that URL with an Accept header to indicate that you want the UI Preview representation.
3. The OSLC Provider will respond with the Compact representation, which includes links to HTML previews.
4. You send an HTTP GET for the small or large preview.
5. The OSLC provider returns HTML that you can show to the user.

We explored the OSLC Provider side of this in more detail earlier in this tutorial.

Example XML for a UI preview

Here's an example of the XML that an OSLC Provider will return when you request the UI Preview representation of a resource:

```
<?xml version="1.0" encoding="UTF-8"?>

<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:dcterms="http://purl.org/dc/terms/"
   xmlns:oslc="http://open-services.net/ns/core#">
   <oslc:Compact
     rdf:about="http://localhost:8080/OSLC4JBugzilla/services/1/
changeRequests/10">
     <dcterms:title>incidents common connexion</dcterms:title>
     <oslc:shortTitle>ChangeRequest 10</oslc:shortTitle>
     <oslc:icon rdf:resource="http://example.com/bugzilla/
images/favicon.ico" />
     <oslc:smallPreview>
       <oslc:Preview>
         <oslc:document
           rdf:resource="http://localhost:8080/OSLC4JBugzilla/
services/1/changeRequests/10/smallPreview" />
         <oslc:hintWidth>11em</oslc:hintWidth>
         <oslc:hintHeight>45em</oslc:hintHeight>
       </oslc:Preview>
   </oslc:smallPreview>
   <oslc:largePreview>
       <oslc:Preview>
         <oslc:document
           rdf:resource="http://localhost:8080/OSLC4JBugzilla/
services/1/changeRequests/10/largePreview" />
         <oslc:hintWidth>20em</oslc:hintWidth>
         <oslc:hintHeight>45em</oslc:hintHeight>
       </oslc:Preview>
       </oslc:largePreview>
   </oslc:Compact>
</rdf:RDF>
```

The sample above includes both a small (`<oslc:smallPreview>`) and large (`<oslc:largePreview>`). Both have a document URI (`rdf:resource`) which is the location of the HTML version of the preview. Each preview also includes a width (`oslc:hintWidth`) and height (`oslc:hintHeight`) which tell you how much space you should give the preview in your web page.

Implementing OSLC UI Preview

Adding the ability to view UI Previews for links in an application can be done almost entirely with HTML and JavaScript code within web pages that display links.

First, we must provide a server-side proxy service. We'll explain that first, then show how the HTML and JavaScript implementation works.

Working Around the Same Origin Policy with a Proxy Service

Anytime you display a link in a web page and you want to offer UI Preview for that link, you need to run some JavaScript to get the UI Preview representation of the link, parse the UI preview, and display a nice "tool tip" style popup that shows the UI Preview.

However, there's a problem: for security reasons, JavaScript code running in a browser cannot call just any URL. Code can only call URLs that have the Same Origin, in other words URLs with the same hostname and port number as the page that hosts the code. Since we want to be able to preview resources at any URL, we need a way to get around this restriction.

One way to work around the Same Origin Policy is to set up a proxy service inside each application that needs it, and that's how we're going to approach it.

The NinaCRM application includes a simple proxy service; we won't look at the proxy service in much detail. You can explore the source code in the file `ProxyServlet.java` in the **org.eclipse.lyo.samples.ninacrm** package. The service is mapped to the `/proxy` path and expects a `uri` parameter.

With the simple proxy service, you can send the URL `http://example.com/anything/etc` to the proxy with the following URL:

```
http://localhost:8181/ninacrm/proxy?uri=http://example.com/anything/etc
```

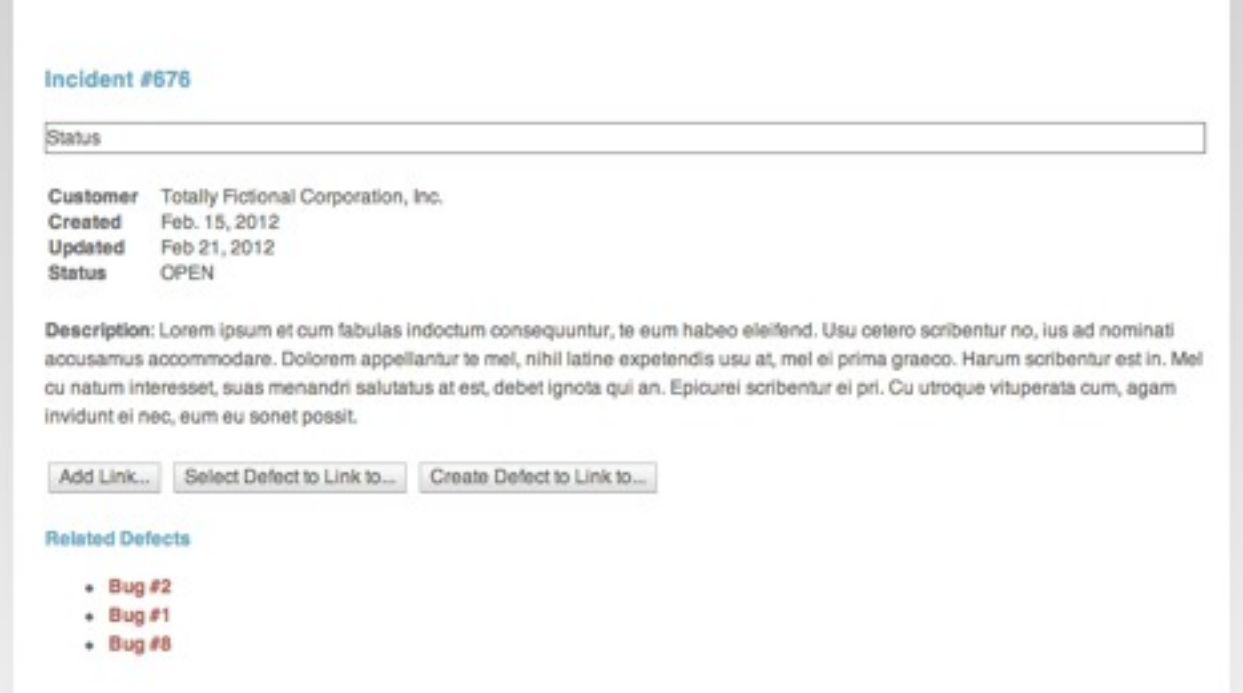
The proxy service will call the URL specified with the same method and headers as the original request and the return the results.

With a proxy service, we can now implement the rest of UI Previews in the browser.

Displaying Links to resources

Starting the NinaCRM sample application and the Bugzilla adapter.

Open http://localhost:8181/ninacrm/ in a web browser. You'll see a sample incident:



Sample incident #676 in the NinaCRM sample application

At the bottom, find the **Related Defects** heading. This is where we show links to related bugs in Bugzilla; the HTML is a simple unordered list:

```
<h3>Related Defects</h3>
<ul id="linkList">
    <li><a href="http://localhost:8080/OSLC4JBugzilla/services/
1/changeRequests/2">Bug #2</a></li>
    <li><a href="http://localhost:8080/OSLC4JBugzilla/services/
1/changeRequests/1">Bug #1</a></li>
    <li><a href="http://localhost:8080/OSLC4JBugzilla/services/
1/changeRequests/8">Bug #8</a></li>
</ul>
```

Note that the URLs are to our Bugzilla adapter.

Without any JavaScript, this would function perfectly well to navigate to the linked bugs. However, we can enhance these links to provide a UI preview when you hover over them.

Showing UI Previews via Dojo Tooltip Widgets

Throughout this, we'll be using the Dojo JavaScript toolkit to smooth out browser differences and build UI components like buttons and tooltips.

Open the file `index.jsp` in `/src/main/webapp/` and search for `dojo.addOnLoad(addPreviewMouseOverHandlers)`.

Here, when the page is done loading we use the `dojo.query()` method to get all the links on the page, and then for each link we add an event handler to run the `showPreview()` method when someone mouses over the link.

```
dojo.addOnLoad(addPreviewMouseOverHandlers);
var hostname ="localhost";

function addPreviewMouseOverHandlers() {
 dojo.query("a").forEach(function(elem) {
  elem.onmouseover = function() { showPreview(elem); };
 });
}
```

Here's the `showPreview()` method:

```
function showPreview(elem) { // (1)
  var previewURI = elem.getAttribute("href"); // (2)
  if (!previewURI) return;
  dojo.xhrGet({  // (3)
    url: "http://"+hostname+":8181/ninacrm/proxy?uri=" +
previewURI,
    handleAs:"xml",
    headers: {
      "Accept": "application/x-oslc-compact+xml", // (4)
    },
    load: function(data) {
      try {
        var previewData = parsePreview(data); // (5)
        var html = "<iframe src='" + previewData.uri + "' "; //
(6)
        var w = previewData.width ? previewData.width : "30em";
        var h = previewData.height ? previewData.height :
"10em";
        html += " style='border:0px; height:" + h + "; width:" +
w + "'";
        html += "></iframe>";
        var tip = new dijit.Tooltip({label: html, connectId:
elem}); // (7)
        tip.open(elem);
      } catch (e) { // (8)
        var tip = new dijit.Tooltip({label: "Error parsing",
connectId: elem});
        tip.open(elem);
```

```
      }
    },
    error: function (error) {
      var tip = new dijit.Tooltip({label: "Preview not found",
connectId: elem});
      tip.open(elem); // (9)
    }
  });
}
```

Here's how it works:

1. We pass the link as an argument
2. Retrieve the `href` from the link
3. We then use the `dojo.xhrGet()` method to request that URL using the proxy service.
4. We pass a header with name `Accept` and value `application/x-oslc-compact +xml` to indicate that we want the UI Preview representation of the linked resource.
5. When the data is returned, we parse it using the `parsePreview()` method (discussed below) to retrieve the URL for the preview, the height, and the width.
6. We start to build an HTML `<iframe>` that displays the preview URL
7. We use the Dojo tooltip to show the preview.

We also account for errors in parsing the XML or if the link fails to load.

Here's the `parsePreview()` method that we use to parse the XML:

```
function parsePreview(xml) { // (1)
  var ret = {};
  var compact = firstChild(firstChild(xml));
  var preview = firstChild(
    firstChildNamed(compact,'oslc:smallPreview')); // (2)
  if (preview) {
    var document = firstChildNamed(preview, 'oslc:document');
    if (document) ret.uri =
document.getAttribute('rdf:resource');
    var height = firstChildNamed(preview, 'oslc:hintHeight');
    ret.height = height.textContent;
    var width = firstChildNamed(preview, 'oslc:hintWidth');
    ret.width = width.textContent;
  }
  return ret;
}

function firstChild(e) { // (3)
  for (x=0; x<e.childNodes.length; x++) {
```

```
    if (e.childNodes[x].nodeType == Node.ELEMENT_NODE) {
      return e.childNodes[x];
    }
  }
}

function firstChildNamed(e, nodeName) { // (4)
  for (x=0; x<e.childNodes.length; x++) {
    if (e.childNodes[x].nodeType == Node.ELEMENT_NODE
      && e.childNodes[x].nodeName == nodeName) {
      return e.childNodes[x];
    }
  }
}
```

And here's how it works:

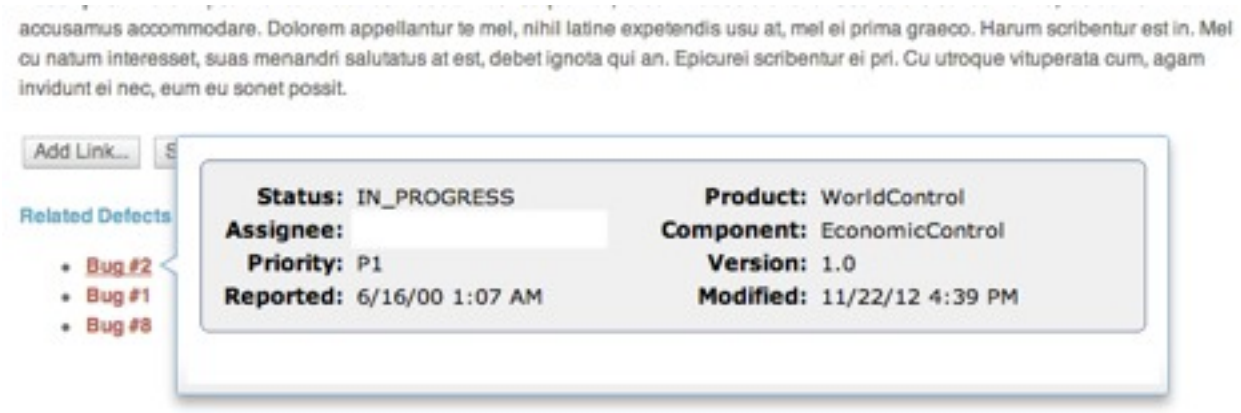
1. We pass the `parsePreview()` method the XML data from the OSLC provider.
2. We build and return an object (`ret`) with the following properties:
   - `uri`: the preview URL (note that this implementation only checks for the `oslc:smallPreview`; there could also be an `oslc:largePreview`)
   - `height`: the hinted height for the preview
   - `width`: the hinted width for the preview

The `firstChild()` and `firstChildNamed()` methods are simple tools to drill down into the XML and get to the nodes we care about.

Try it out!

If you're running the sample applications, open http://localhost:8181/ninacrm/ in a web browser.

Hover over any of the **Related Defects** links. (You will probably have to log in with your Bugzilla username and password.) You should see a tooltip appear with the small preview of the bug:



Small UI preview of a bug that is linked from the sample incident

Next up, we'll explore how to use OSLC Delegated UIs to allow our support reps to both select and create new bugs in Bugzilla without leaving the NinaCRM application.

Implementing OSLC Delegated UIs

In the previous section, we added the ability to display a small preview of OSLC4JBugzillailla bugs in the NinaCRM application. Now, we'll take it a step further and allow our support reps to create and select bugs in NinaCRM.

Introduction to OSLC Delegated UI

OSLC Delegated UI is a way for a web application to provide a UI for creating and selecting resources, one that can be used by other web applications.

To explain why it's called "Delegated" UI, consider our example: we want the NinaCRM system to be able to create and select bugs managed by the separate OSLC4JBugzillailla system; however, we do not want to create a new UI to collect the information or duplicate OSLC4JBugzillailla's methods of checking that the information is valid. Instead, we want to*delegate* the creation and selection of bugs to the OSLC4JBugzillailla system.

By using Delegated UI, Nina will enable support reps to add links from Incidents to bugs without leaving the comfort of the CRM system.

The mechanics of Delegated UI

Earlier in this tutorial, we walked through an implementation of Delegated UIs for OSLC4JBugzillailla, both for selecting bugs and creating new bugs. In addition to providing the UI and handling the results, the OSLC4JBugzillailla adapter (or any other OSLC provider application) announces in its Service Provider Documents the URL location and recommended size of the UI.

The application that wants to use the Delegated UI (the OSLC Consumer) creates an `<iframe>` for the Delegated UI so that the user can interact with it. The Consumer application must also listen to the `<iframe>` do something with the results of the user's actions.

You can learn more about Delegated UI in the OSLC Primer.

Parsing the Service Provider Documents

Note: The NinaCRM sample application does _not_ retrieve a Service Provider document and parse its contents to locate Delegated UIs; for simplicity, the URLs for the delegated UIs are hard-coded in the `index.jsp` file. A better implementation would be one that properly parses the Service Provider and thus could work with any OSLC Provider. We're not angels here; sorry.

As we noted when we implemented Service Providers and Catalogs, one of the cores of OSLC is that clients should not have to hard-code any URLs other than a Service Provider Catalog. Clients should be able to parse the Catalog and navigate from the Catalog to the Service Providers; the Service Providers will then expose the available OSLC services.

If you'd like to follow along with a real Service Provider Catalog or Service Provider, see the "Viewing the machine-readable formats of a Service Provider Catalog" section near the bottom of this section.

Below is a part of a sample RDF/XML representation of a Service Provider that exposes both a Delegated UI for selection (`<oslc:selectionDialog>`) and a Delegated UI for Creation (`<oslc:creationDialog>`):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:dcterms="http://purl.org/dc/terms/"
   xmlns:oslc="http://open-services.net/ns/core#">

   <oslc:ServiceProvider rdf:about="http://localhost:8080/
OSLC4JBugzilla/services">  <!-- (1)-->
      <dcterms:title>Service Provider Catalog</dcterms:title>
      <dcterms:description>
        Simple OSLC Reference Implementation (SORI) for Change
Management Service Document
      </dcterms:description>

      <!-- etc. etc. etc. omitted everything up to oslc:service
property -->

      <oslc:service>  <!-- (2)-->
        <oslc:Service>
           <oslc:domain rdf:resource="http://open-services.net/
ns/cm#"/>

        <!-- etc. etc. etc. omitted everything Delegated UI
information -->

           <oslc:selectionDialog>  <!-- (3)-->
              <oslc:Dialog>
                <dcterms:title>CM Resource Selector</
dcterms:title>
                <oslc:label>Picker</oslc:label> <!-- (4)-->
                <oslc:dialog  <!-- (5)-->
```

```
                  rdf:resource="http://10.0.1.3:8080/
OSLC4JBugzilla/services/1/changeRequests/selector"/>
               <oslc:hintHeight>325px</oslc:hintHeight> <!--
(6)-->
               <oslc:hintWidth>525px</oslc:hintWidth>
               <oslc:resourceType  <!-- (7)-->
                  rdf:resource="http://open-services.net/ns/
cm#ChangeRequest"/>
               <oslc:usage rdf:resource="http://open-
services.net/ns/core#default"/>
            </oslc:Dialog>
         </oslc:selectionDialog>

         <oslc:creationDialog> <!-- (8)-->
            <oslc:Dialog>
              <dcterms:title>CM Resource Creator</
dcterms:title>
               <oslc:label>Creator</oslc:label>
               <oslc:dialog rdf:resource=
                  "http://10.0.1.3:8080/OSLC4JBugzilla/
services/1/changeRequests/creator"/>
               <oslc:hintHeight>375px</oslc:hintHeight>
               <oslc:hintWidth>600px</oslc:hintWidth>
               <oslc:resourceType rdf:resource=
                  "http://open-services.net/ns/
cm#ChangeRequest"/>
               <oslc:usage rdf:resource="http://open-
services.net/ns/core#default"/>
            </oslc:Dialog>
         </oslc:creationDialog>
       </oslc:Service>
     </oslc:service>

  </oslc:ServiceProvider>
</rdf:RDF>
```

Note that both of the delegated UIs have a URL location (`rdf:resource`) and suggested sizes for the dialogs (`<oslc:hintHeight>` and `<oslc:hintWidth>`).

Since we now know this information, we can add delegated UIs to the NinaCRM application.

Adding Delegated UI dialogs to the NinaCRM

To follow along, start the NinaCRM application and the Bugzilla Adapter application. Open the sample NinaCRM incident page at http://localhost:8181/ninacrm/.

The HTML and JavaScript for the incident page are in the file `index.jsp in /src/main/webapp/`.

Buttons to launch the dialogs

First, we add two buttons to the HTML code of the incident page.

In the `index.jsp` file, search for the comment `Add link via OSLC Delegated Picker`. There you'll find the HTML for the buttons:

```
<button id="selectDefectButton" type="button"
  dojoType="dijit.form.Button" onclick="selectDefect()">
  Select Defect to Link to...
</button>

<button id="createDefectButton" type="button"
  dojoType="dijit.form.Button" onclick="createDefect()">
  Create Defect to Link to...
</button>
```

Note that each button uses the Dojo/Dijit button framework and launches a JavaScript method when clicked.

Next, we add the `selectDialog()` and `createDialog()` JavaScript methods.

The following methods use only the Post Message Protocol from the OSLC specification and will accordingly only work in newer browsers. To support older browsers, you should also implement the Window Name protocol. For more information, see the OSLC Core Specification and our our implementation of Delegated UIs.

Because the end-result of both actions is the same – we will be adding a link to either a new or existing bug in Bugzilla – both methods invoke the same `postMessageProtocol()` method with the appropriate URL:

```
function selectDefect() {
  postMessageProtocol(selectDialogURL);
}

function createDefect() {
  postMessageProtocol(createDialogURL);
}
```

Here's the `postMessageProtocol()` method that our buttons invoke:

```
var iframe;
function postMessageProtocol(dialogURL) {
  // Step 1
  dialogURL += '#oslc-core-postMessage-1.0';

  // Step 2
  var listener = dojo.hitch(this, function (e) {
    var HEADER = "oslc-response:";
    if (e.source == iframe.contentWindow &&
e.data.indexOf(HEADER) === 0) {
      // Step 4
      window.removeEventListener('message', listener, false);
      destroyFrame(iframe);
      handleMessage(e.data.substr(HEADER.length));
    }
  });
  window.addEventListener('message', listener, false);

  // Step 3
  iframe = document.createElement('iframe');
  iframe.width = 500;
  iframe.height = 375;
  iframe.src = dialogURL;

  // Step 4
  displayFrame(iframe);
  container.appendChild(iframe);
};
```

Here's how it works:

1. First, we add the hash `#oslc-core-postMessage-1.0` to the dialog URLs; this will tell the OSLC Provider application that we're using the Post Message protocol to communicate with the iframe.
2. We add an event listener that listens for a `postMessage` response with an `oslc-response` header and then invokes the `handleMessage()` method (see below)
3. We create an `iframe` element with the content of the dialog URL.
4. We then display the `iframe` element with the `displayFrame()` method (see below).

Here's the `displayFrame()` method, which is basically a wrapper for the Dojo/Dijit dialog method:

```
var dialog;
```

```
function displayFrame(frame) {
  if (!dialog) dialog = new dijit.Dialog();
  dialog.setContent(frame);
  dialog.show();
}
```

And here is the `handleMessage()` method, which evaluates the returned JSON representation of a bug for the URL and title:

```
function handleMessage(message) {
  var json = message.substring(message.indexOf("{"),
message.length);
  var results = eval("(" + json + ")");
  var linkname = results["oslc:results"][0]["oslc:label"];
  var linkurl = results["oslc:results"][0]["rdf:resource"];
  addLink(linkname, linkurl);
}
```

Finally, we add a link to the page with the `addLink()` method:

```
function addLink(linkname, linkurl) {
  dojo.xhrPost( {
    url: "http://"+hostname+":8181/ninacrm/data",
    headers: { "Content-Type": "application/x-www-form-
urlencoded" },
    postData: "linkname=" + linkname + "&linkurl=" + linkurl,
    load: function(data) {
      status("Added link!");
      var li = document.createElement("li");
      li.innerHTML = "<a href='" + linkurl + "'
onclick='showPreview()'>" + linkname + "</a>";
      var ul = dojo.byId("linkList").appendChild(li);
      addPreviewMouseOverHandlers();
    },
    error: function (error) {
      status("Error adding link!");
    }
  });
}
```

The `addLink()` method sends the title and URL of the bug to the `/ninacrm/data` service, which adds the information to the database. If successful, we add the link to the `<ul id="linkList">` element under the **Related Links** header.

Throughout the process, we update a status message with the `status()` method:

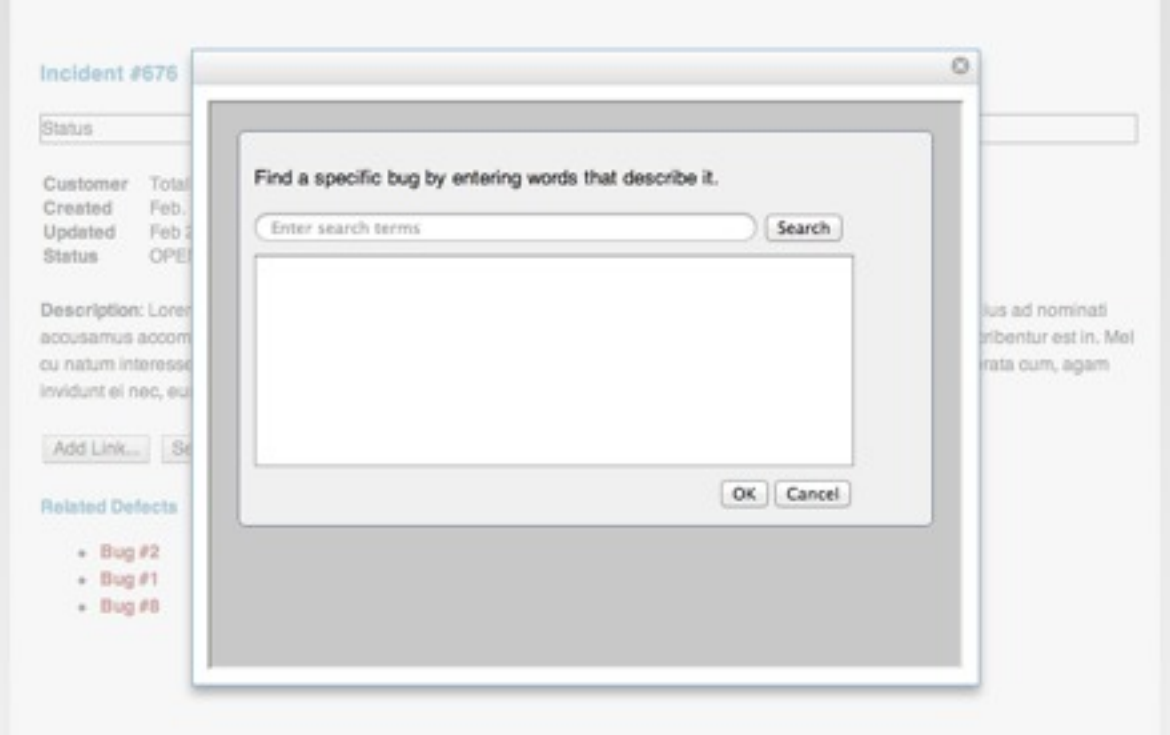
```
function status(msg) {
```

```
  document.getElementById("status").innerHTML = msg;
}
```

Results

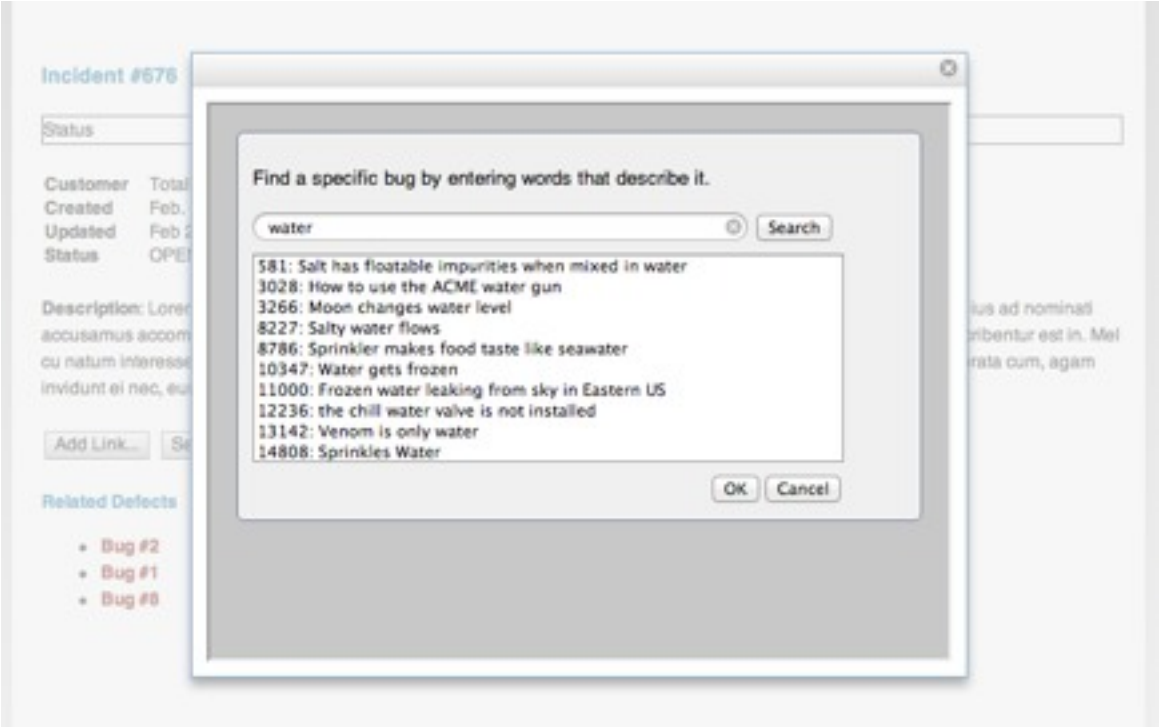
With all these in place, you should now be able to add links to our incident pages with delegated OSLC dialogs.

Here's the dialog for selection:



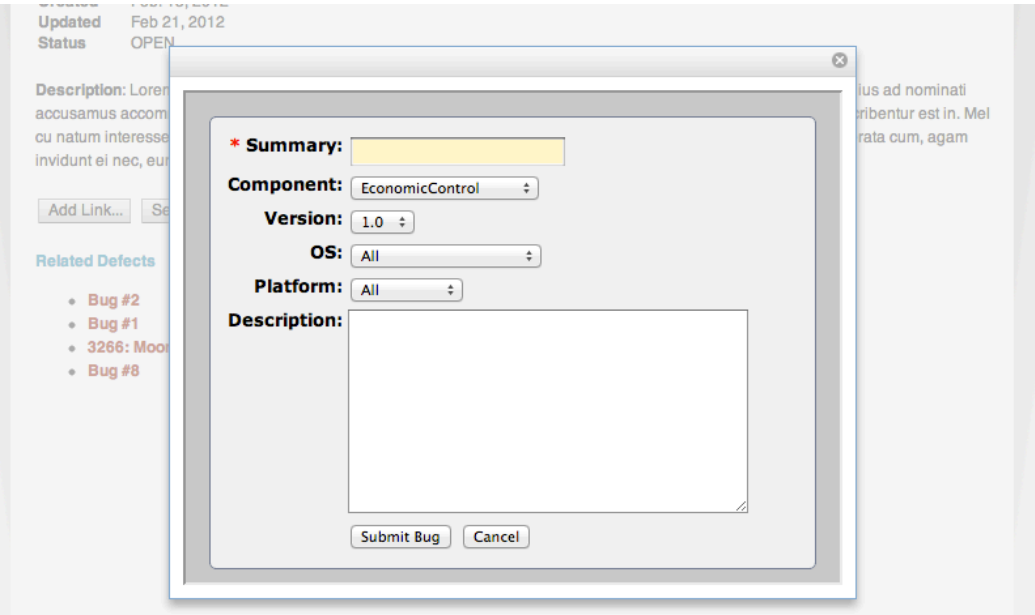
Sample delegated dialog for selecting bugs on the incident page

You can search for bugs directly:

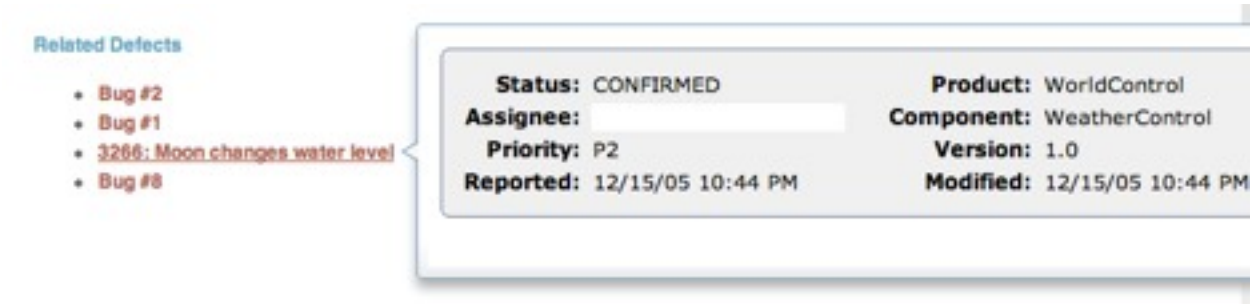


Using the delegated dialog for selection to search for water

You can also create new bugs right from the page:



Either way, the new or selected bug appears on the page (complete with the ability to see a UI preview of the bug):



Bug #3266 added to the incident page

At this point, we've completed our first milestone goals: the CRM system now uses links, OSLC UI Preview, and OSLC delegated dialogs to make it faster for support reps to find and create bugs.

Implementing a “Customers to notify” page

In this section, we’ll explore how the ability to parse OSLC resources can help us add the ability to automatically notify customers of bugs that have changed.

Our plan for automating customer notifications is pretty straight forward.

When our support reps create Incidents, customers are allowed to request notification for critical bugs, and this is recorded in the CRM system. To automatically send notifications to our customers when there are changes to bugs, we must do the following:

1. Query the CRM system to get all Notification Requests; each specifies the URL of a bug, date of last update and the customer’s notification email address.
2. For each Notification Request, check the associated bug to see if it has been updated, using HTTP Conditional GET to avoid retrieving and parsing bugs that have not been updated.
3. If a bug has been updated, then format a nice notification email and include a summary of the bug.

We can write a program that can run as a scheduled task on a Build Automation system or just plain old UNIX cron.

In this tutorial we won’t try to explain the whole program. We’ll focus on the OSLC-specific parts, which are retrieving an OSLC resource via HTTP GET and how to parse an OSLC resource to get property values like title, status, modification date and others.

Fetching an OSLC resource with HTTP GET

First, as a prerequisite, we have to set up our Notification program to run on a schedule and provide it with whatever network addresses, credentials and other information necessary to connect to the CRM system, Bugzilla, and the Email system. We won’t cover these details here.

Next, we need to write the code necessary to query the CRM system and get all Notification Requests and code that loops through the list. Then, For each bug, we wants to check and see if the bug has been updated since the last time the program ran. If the bug has been updated, then we want to fetch the bug in RDF/XML form and parse out the information need to form a notification email to the interested customer.

Exploring the RDF/XML form of a Bugzilla Bug

For this section, start the sample Bugzilla Adapter application. We assume it's running at `localhost:8080`.

With the adapter running, navigate to the following URL in a browser:http://localhost:8080/OSLC4JBugzilla/services/1/changeRequests/17

(You might have to substitute different ID numbers for the product and bug)

In a browser, you'll be forwarded to the HTML page in Bugzilla for the bug. That's nice for us, but not so useful for a program that must parse the data. How can we request an RDF/XML representation of a bug?

OSLC providers are required to provide an RDF/XML representation of resources; however, the normal rules of HTTP and Content Negotiation apply. **If you want RDF/XML then you should ask for it.** Specifically, use `Accept` headers.

If you send the same request with an `Accept` header with the content `application/rdf+xml`, you should receive RDF/XML back from the adapter.

You can explore this further with the Poster plugin and the Bugzilla adapter where we implemented OSLC representations of Bugzilla bugs.

Here's a sample Bugzilla bug represented as an RDF/XML BugzillaChangeRequest resource:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:oslc_data="http://open-services.net/ns/
servicemanagement/1.0/"
    xmlns:oslc_rm="http://open-services.net/ns/rm#"
    xmlns:oslc="http://open-services.net/ns/core#"
    xmlns:bugz="http://www.bugzilla.org/rdf#"
    xmlns:oslc_scm="http://open-services.net/ns/scm#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:oslc_qm="http://open-services.net/ns/qm#"
    xmlns:dcterms="http://purl.org/dc/terms/"
    xmlns:oslc_cm="http://open-services.net/ns/cm#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" >
  <rdf:Description rdf:about="http://localhost:8080/
OSLC4JBugzilla/services/1/changeRequests/17">
    <dcterms:contributor rdf:resource="http://localhost:8080/
OSLC4JBugzilla/person?mbox=tara%40bluemartini.com"/>
    <bugz:operatingSystem>Windows NT</bugz:operatingSystem>
```

```
    <rdf:type rdf:resource="http://open-services.net/ns/
cm#ChangeRequest"/>
    <oslc_cm:status>RESOLVED</oslc_cm:status>
    <oslc:serviceProvider rdf:resource="http://localhost:8080/
OSLC4JBugzilla/services/serviceProviders/1"/>
    <bugz:platform>PC</bugz:platform>
    <bugz:version>1.0</bugz:version>
    <dcterms:created>2000-06-29T22:07:00.000-04:00</
dcterms:created>
    <dcterms:title rdf:datatype="http://www.w3.org/1999/02/22-
rdf-syntax-ns#XMLLiteral">Albright Overseas</dcterms:title>
    <bugz:component>PoliticalBackStabbing</bugz:component>
    <oslc_cm:severity>Unclassified</oslc_cm:severity>
    <dcterms:modified>2009-11-14T14:36:54.000-05:00</
dcterms:modified>
    <bugz:priority>P4</bugz:priority>
    <dcterms:identifier>17</dcterms:identifier>
  </rdf:Description>
  <rdf:Description rdf:about="http://localhost:8080/
OSLC4JBugzilla/person?mbox=tara%40bluemartini.com">
    <rdf:type rdf:resource="http://xmlns.com/foaf/0.1/Person"/>
  </rdf:Description>
</rdf:RDF>
```

Note the variety of namespace definitions near the top of the document that define short prefix names for properties (e.g.,`dcterms`).

Inside the `<rdf:RDF>` root element, there is an `<rdf:Description>` element with an attribute of `rdf:about` that is the URI of the Change Request. The `<rdf:type>` element indicates that this is an OSLC CM request. Each XML element represents a property value of the Change Request.

OSLC resources use Dublin Core defined properties, like `title`, `description`, and `id`. They also use OSLC defined properties like `status`, `closed`, and `inprogress`. You can find a listing of the different types of properties allowed and required in the OSLC-CM specification. There are also Bugzilla specific properties like `component` and `priority`.

You can learn more about how our OSLC Bugzilla Adapter generates these representations here.

Parsing an OSLC resource

Note: The following discusses using an RDF/XML parser. If you are writing Java, you could also use – in fact, we recommend using – OSLC4J to convert RDF/XML

representations into Java objects, which will most likely be easier to work with. Consider the following to be guidance if you choose to approach this another way.

If you've parsed XML before, then the XML above probably does not look too challenging; however, RDF/XML is very flexible and XML parsing tools are not always the best way to process it. Fortunately, there are plenty of commercial and open source RDF parsing tools.

Why you need an RDF parser

You can see the flexibility of RDF/XML in action if you compare the RDF/XML for the Change Request above to theRDF/XML sample Change Request in the OSLC-CM specification.

RDF/XML allows RDF property values to be serialized in a variety of ways. For example, property values about a Change Request could be nested inside an `<oslc_cm:ChangeRequest>` element, as you see in the OSLC-CM samples, where the element itself indicates the Resource Type. Or they could be nested inside an `<rdf:Description>` element, as you see above, and the type indicated by an `<rdf:type>` value. Both are valid forms of RDF/XML and allowed by OSLC, so you will have to accept both forms in any parser code that you write. That's only one example.

Another reason to use an RDF parser is that RDF/XML is only one way to serialize RDF. Right now RDF/XML is the popular format and the one required by OSLC, but there are other formats including Turtle, N3 and soon an official RDF serialization for JSON. By using a full-featured RDF parser like Jena, which we discuss below, you can read and write any format with the same code.

Finding an RDF/XML parser

Instead of trying to write your own RDF/XML parser using XML tools, a better approach is to use an existing RDF/XML parser. There is one for every programming language and most are free and/or open source software. Below is a list of the more popular open source RDF tool-kits, the platform and licensing used by each.

- http://jena.apache.org - Jena (Java) License: ASL2
- http://www.openrdf.org/ - OpenRDF / Sesame (Java) License: ASL2 and BSD-like
- http://rdf.rubyforge.org/ - RDF.rb (Ruby) License: Public Domain
- http://librdf.org/bindings/ - Redland (C, Perl, PHP, Python and Ruby) License: ASL2
- http://razor.occams.info/code/semweb/ - SemWeb.NET (C# / .Net) License: GPL2

All of the above libraries support RDF parsing and serialization, some form of triple-store RDF storage, and SPARQL query… more than you'll need for a typical OSLC implementation.

For our implementation, we'll be using Jena.

How to use Jena to parse RDF/XML resource

Jena is an open source Java library that offers a wide variety of RDF tools including a parser that can handle RDF/XML and other RDF serializations. Using Jena is straight-forward and should be easy for a Java developer.

Before you start coding, you must get the Jena JAR and all of its dependency JARs into your development classpath. You can do this by downloading the Jena ZIP file which contains all of the necessary JARs, and add them to your IDE project. Or, if you are using Maven, then a dependency on groupId `com.hp.hpl.jena` and artifactId `jena` to your Maven POM file.

Let's attempt to GET an OSLC Change Request via HTTP, but this time we will do it in Java. When we get the results, we will parse them with Jena and pull out the properties that Nina needs: the OSLC-CM `fixed` value and the Dublin Core Terms `modified` date value.

Note that the following is not a complete Java class:

```
import java.net.HttpURLConnection; // (1)
import java.net.URL;
import com.hp.hpl.jena.rdf.model.Model;
import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.rdf.model.Property;
import com.hp.hpl.jena.rdf.model.Resource;
import com.hp.hpl.jena.rdf.model.Statement;

// class declaration and other things omitted

String resourceURI = "http://localhost:8080/OSLC4JBugzilla/
services/1/ChangeRequests/1";
URL url = new URL(resourceURI);  // (2)
HttpURLConnection conn =
(HttpURLConnection)url.openConnection();
conn.setRequestProperty("Accept", "application/rdf+xml");  //
(3)

Model model = ModelFactory.createDefaultModel();
model.read(conn.getInputStream(), resourceURI); // (4)

Resource resource = model.getResource(resourceURI); // (5)

// (6)
Property fixedProp = model.getProperty("http://open-
services.net/ns/cm#fixed");
```

```
Statement fixed = model.getProperty(resource, fixedProp); // (7)
System.err.println("Fixed = " + fixed.getString()); // (8)

// (9)
Property modifiedProp = model.getProperty("http://purl.org/dc/
terms/modified");
Statement modified = model.getProperty(resource, modifiedProp);
System.err.println("Modified = " + modified.getString());
```

Here's what it does:

1. Import the Java classes required for the example. Again, this is an incomplete Java class, so we've left out the class declaration, method declaration, and other bits;
2. Build a URL to a specific change request (hard-coded in the example);
3. Open a connection to that URL with an `Accept` header of `application/rdf+xml`;
4. Create a new Jena model and have it read the response. We pass in the `resourceURI` so Jena will know how to resolve relative links;
5. We use the Jena model to get the `Resource` for the the Change Request URI
6. We use the Jena model to get the Property object for the OSLC-CM `fixed` property by passing in the URL for the `fixed` property;
7. We get the value for the `fixed` property;
8. And output that value to `stdout`;
9. We repeat the same process to get the Dublin Core `modified` property.

With the ability to parse OSLC Change Request resources in RDF/XML form, you can fairly easily figure out the remainder of the application that will automatically notify customers if there have been any updates to critical bugs.

The power of OSLC representations

The real power of OSLC on display here is that although we've written this code with our OSLC-CM Adapter for Bugzilla in mind, it will work *equally well for any other application that provides data according to the OSLC-CM specification*. Because OSLC Providers should all expose the same types of data in the same standard formats, you can build integrations *for the OSLC standards and specifications* that should work with any compatible software. It's a different way of thinking about integrations that should help you make powerful, flexible, and future-proof ways to connect software. Cool stuff.

Next, now that we have a simple understanding of Jena, we'll use it to help us automatically create Bugzilla bugs – no human involvement required.

Implementing automated bug creation

In this section we'll be building the foundations for a Java service that can automatically create a new bug in Bugzilla whenever a build or a test (from another program) fails. Here's roughly how the entire system would work:

1. The build scripts (or testing programs) will be configured to report bugs against a product in Bugzilla; with the ID for the product, these applications can retrieve the OSLC Service Provider that represents that product.
2. The build scripts will retrieve the Service Provider for the product and then parse it to find the Creation Factory URL, or the URL to which you can POST to create new bugs
3. The scripts will create an RDF/XML representation of a new bug to be created; if there is an OSLC Resource Shape, they will use that to determine any required fields
4. The script will send the RDF/XML representation to the Creation Factory URL using HTTP POST; the adapter will then interact with Bugzilla to create a new bug.

As with the last section, we won't create an entire service; instead, we'll focus on the OSLC-specific parts, like parsing RDF resources and POSTing to an OSLC Creation Factory.

Using a Service Provider Catalog to find a Service Provider.

For our OSLC-CM Bugzilla Adapter (or any OSLC provider), the starting point for exploring OSLC capabilities is theService Provider Catalog document.

You can read more about implementing Service Provider Catalogs for our Bugzilla Adapter here. In short, we represent every Bugzilla Product as a Service Provider resource, and we collect all of those Service Providers in one Service Provider Catalog.

The general principle is that clients should only need to know the URL for the Catalog; from the Catalog, clients can navigate to the other OSLC services. In other words, *clients should not have to hard-code URLs to individual OSLC services*.

Here's a sample Service Provider Catalog document. You can see something similar if you're running the Bugzilla Adapter and run an HTTP GET request to `http://localhost:8080/OSLC4JBugzilla/services/catalog/singleton`with an `Accept` header of `application/rdf+xml`:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:oslc="http://open-services.net/ns/core#"
  xmlns:dcterms="http://purl.org/dc/terms/"
```

```
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <oslc:ServiceProviderCatalog rdf:about="http://localhost:8080/
OSLC4JBugzilla/catalog">
    <dcterms:title>OSLC-CM Adapter/Bugzilla Service Provider
Catalog</dcterms:title>
    <dcterms:description>
    Enables navigation to Service Provider for each Product
    against which bugs may be reported.
    </dcterms:description>
    <oslc:domain rdf:resource="http://open-services.net/ns/cm#"/
>

    <oslc:serviceProvider>
      <oslc:ServiceProvider rdf:about=
        "http://localhost:8080/OSLC4JBugzilla/services/
serviceProviders/2">
        <dcterms:title>FoodReplicator</dcterms:title>
      </oslc:ServiceProvider>
    </oslc:serviceProvider>

    <oslc:serviceProvider>
      <oslc:ServiceProvider rdf:about=
        "http://localhost:8080/OSLC4JBugzilla/services/
serviceProviders/19">
        <dcterms:title>Sam's Widget</dcterms:title>
      </oslc:ServiceProvider>
    </oslc:serviceProvider>

  </oslc:ServiceProviderCatalog>
</rdf:RDF>
```

The above example catalog has two `oslc:serviceProvider` values (i.e. http://open-services.net/ns/core#ServiceProvider). The URL of each provider is specified in the `rdf:about` attribute, and its title is specified as a `dcterms:title` property value. For example, the OSLC Service Provider resource for the FoodReplicator product is located at `http://localhost:8080/OSLC4JBugzilla/services/serviceProviders/2`.

As we discussed in the previous section, you should use an RDF parser like Jena to parse these documents programmatically.

Using a Service Provider to find a Creation Factory

Once we've navigated from a Catalog to a Service Provider resource, here's a sample of what you might see (as an RDF/XML document):

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dcterms="http://purl.org/dc/terms/"
  xmlns:oslc="http://open-services.net/ns/core#">
  <oslc:ServiceProvider rdf:about="http://localhost:8080/
OSLC4JBugzilla/services/serviceProviders/2">
    <dcterms:title>OSLC-CM Adapter/Bugzilla Service Provider:
       Product FakePortal(2)</dcterms:title>
    <dcterms:description>
       Enables navigation to OSLC-CM Resource Creator and
Selector Dialogs
    </dcterms:description>
    <oslc:service>  <!-- (1) -->
      <oslc:Service>
        <oslc:domain rdf:resource="http://open-services.net/ns/
cm#"/>

        <!-- selection and creation dialog information deleted
-->

        <oslc:creationFactory> <!-- (2) -->
          <oslc:CreationFactory>
            <dcterms:title>Change Request Creation Factory</
dcterms:title>
            <oslc:resourceType rdf:resource=  <!-- (3) -->
               "http://open-services.net/ns/cm#ChangeRequest"/>
            <oslc:label>CreationFactory</oslc:label>
            <oslc:creation rdf:resource=  <!-- (4) -->
               "http://localhost:8080/OSLC4JBugzilla/services/2/
changeRequests"/>
            <oslc:resourceShape rdf:resource=  <!-- (5) -->
               "http://localhost:8080/OSLC4JBugzilla/services/
resourceShapes/changeRequest"/>
            <oslc:usage rdf:resource=  <!-- (6) -->
               "http://open-services.net/ns/core#default"/>
          </oslc:CreationFactory>
        </oslc:creationFactory>
      </oslc:Service>
    </oslc:service>
  </oslc:ServiceProvider>
</rdf:RDF>
```

You can read more about implementing Service Providers and implementing creation factories for our Bugzilla adapter.

Of most interest to our team developing a way to automatically create bugs are the contents of the `<oslc:service>`element (**(1)**). The Service has an `oslc:creationFactory` property (**(2)**) with a value of `oslc:CreationFactory`. The creation factory has values that indicate it is for creating Change Requests (**(3)**), the URI for posting new Change Requests (**(4)**), and the URI of the Resource Shape (**(5)**) that lists the required fields for bug creation. The usage value (**(5)**) indicates that this is the default Creation Factory to use.

With this information, the build and testing scripts can parse the Service Provider document and discover the Creation Factory URL, which is the URL for posting new bugs.

First, though, let's explore the Resource Shape document.

Using a Resource Shape to determine required properties

It's not enough to just know the URL to POST bugs to; the testing scripts must also create a properly formatted OSLC-CM Change Request representation with the required property values and property values that are valid. Each Product defined in Bugzilla might have different required fields, custom fields and different allowed values.

It's entirely possible to confer with the Bugzilla system's administration and figure out the required and allowed values, *or* you could use the OSLC OSLC Creation Factory's Resource Shape document, which provides the same information.

An example Resource Shape document in RDF/XML form is below (you can see the Resource Shape from our Bugzilla Adapter at http://localhost:8080/OSLC4JBugzilla/services/resourceShapes/changeRequest):

```
<?xml version="1.0" encoding="UTF-8"?>
<oslc:Shape xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-
ns#"
   xmlns:dc="http://purl.org/dc/terms/"
   xmlns:oslc="http://open-services.net/xmlns/oslc-core#"
   xmlns:oslc_cm="http://open-services.net/xmlns/cm/1.0/"
   rdf:about="http://localhost:8080/OSLC4JBugzilla/shape?
productId=2">

   <dc:title>This is the resource shape for a new Bugzilla Bug</
dc:title>

   <oslc:property>
    <oslc:Property>
     <oslc:name>title</oslc:name>
```

```
      <oslc:propertyDefinition rdf:resource=
       "http://purl.org/dc/terms/title" />
      <oslc:valueType rdf:resource=
       "http://www.w3.org/1999/02/22-rdf-syntax-ns#XMLLiteral" />
      <oslc:occurs rdf:resource=
       "http://open-service.net/ns/core#Exactly-one" />
     </oslc:Property>
    </oslc:property>

    <oslc:property>
     <oslc:Property>
      <oslc:name>component</oslc:name>
      <oslc:propertyDefinition rdf:resource=
       "http://www.bugzilla.org/rdf#component" />
      <oslc:valueType rdf:resource=
       "http://www.w3.org/2001/XMLSchema#string" />
      <oslc:occurs rdf:resource=
       "http://open-service.net/ns/core#Exactly-one" />
      <oslc:allowedValue>Installer</oslc:allowedValue>
      <oslc:allowedValue>User Interface</oslc:allowedValue>
      </oslc:Property>
   </oslc:property>

   <!-- other properties omitted -->

</oslc:Shape>
```

Inside the root `oslc:Shape` element is the `dc:title` of the shape, which tells us the shape's purpose (1). Next there is a series of property values for the `oslc:property` property. Each one describes the requirements for the property at creation time. In the listing, we omit all but two of the property values: `dc:title` and `bugz:component`.

Each property has a name (`oslc:name`), a link to the property definition (`oslc:propertyDefinition`), the acceptable type of value (`oslc:valueType`) and the cardinality (`oslc:occurs`), and allowed values (`oslc:allowedValue`). In our example above, the `component` property accepts `String` values; the `occurs` value of `Exactly-one` indicates that it is required; and the allowed values are either `Installer` or `User Interface`.

With this information from the Resource Shape document, we can now create and post new bugs.

Forming an RDF/XML representation of a Bugzilla bug

Note: The following discusses using an RDF/XML parser. If you are writing Java, you could also use – in fact, we recommend using – OSLC4J to handle conversions between RDF/XML and plain old Java objects. Consider the following to be guidance if you choose to approach this another way.

Although you could generate RDF/XML through a variety of techniques, we recommend using a dedicated RDF toolkit like Jena.

We'll create a very simple method below: it accepts strings of various property values and returns an RDF/XML representation of a new bug.

To follow along, open the file `NewDefect.java` in the org.eclipse.lyo.samples.ninacrm.examples package and search for the `formNewBug()` method.

First, OSLC-CM requires Change Requests to have an RDF Type and a title:

```
Property bugType =
  new PropertyImpl("http://open-services.net/ns/
cm#ChangeRequest");

Property titleProp =
  new PropertyImpl("http://purl.org/dc/terms/title");
```

Next, we know from the Resource Shape document that new bugs must have property values for the Bugzilla bug properties product, version, component, platform and operating system. So we set up Jena property objects for each of those:

```
Property versionProp =
  new PropertyImpl("http://www.bugzilla.org/rdf#version");

Property componentProp =
  new PropertyImpl("http://www.bugzilla.org/rdf#component");

Property platformProp =
  new PropertyImpl("http://www.bugzilla.org/rdf#platform");

Property opsysProp =
  new PropertyImpl("http://www.bugzilla.org/rdf#opsys");
```

We are hard-coding required Bugzilla properties here. It would be better – more flexible and future-proof – to programmatically locate and parse a Resource Shape document to determine the required properties; for simplicity's sake, we do not do so here. We leave that as an exercise for the reader.

Note that **we did not set the Bugzilla Product**: the product can be determined by the choice of service that you use to post the new bug. (In other words, every Bugzilla product will have its own OSLC Creation Factory.)

Next, we set up a Jena Model object and adds namespace prefixes. These are not strictly necessary, but they will make the RDF/XML a little more readable and make it look more like the examples in the OSLC specifications, which is useful.

```
Model model = ModelFactory.createDefaultModel();
model.setNsPrefix("bugz",    "http://www.bugzilla.org/rdf#");
model.setNsPrefix("dcterms", "http://purl.org/dc/terms/");
model.setNsPrefix("oslc_cm", "http://open-services.net/ns/cm#");
```

Once the Model is set up, we create a Resource object using a base URI that is the empty string. We won't know the URI of the new bug until the OSLC-CM provider has created it and tells us the new URI via the HTTP Location header.

```
com.hp.hpl.jena.rdf.model.Resource resource =
model.createResource("");
```

Once we have a Resource, we are ready to add property values for each of the required properties:

```
resource.addProperty(RDF.type,     bugType);
resource.addLiteral(titleProp,     title);
resource.addLiteral(versionProp,   version);
resource.addLiteral(componentProp, component);
resource.addLiteral(platformProp,  platform);
resource.addLiteral(opsysProp,     opsys);
```

Finally, we write out the RDF model in RDF/XML format and return it in string form:

```
StringWriter sw = new StringWriter();
RDFWriter writer = model.getWriter();
writer.write(model, sw, "/");
sw.flush();
return sw.toString();
```

Using HTTP to POST a new bug

With the ability to build RDF/XML representations of a bug in place, we can write a simple example that posts an RDF/XML representation of a new bug to an OSLC-CM Provider:

```
public static void postNewBug(
                   String creationURL,
```

```
                    String title,
                    String version,
                    String component,
                    String platform,
                    String opsys) {

  String bug = formNewBug(title, version, component plaform,
opsys); // (1)

  try {
     URL createURL = new URL(creationURL);  // (2)

     HttpURLConnection conn =
(HttpURLConnection)createURL.openConnection();
     conn.setRequestMethod("POST"); // (3)
     conn.setDoOutput(true);
     conn.setRequestProperty("Content-Type", "application/rdf
+xml");  // (4)

     BufferedOutputStream out = new
BufferedOutputStream(conn.getOutputStream());
     out.write(bug.getBytes("UTF-8")); // (5)
     out.close();

     BufferedReader in = new BufferedReader( // (6)
      new InputStreamReader(conn.getInputStream()));
     String s;
     while ((s = in.readLine()) != null) {
       System.out.println(s);
     }
     in.close();

     int rc = conn.getResponseCode(); // (7)
     System.out.println("Return status: " + rc);
     System.out.println("Location: " +
conn.getHeaderField("Location")); // (8)
   }
   catch (IOException e) {
     e.printStackTrace();
   }
}
```

The above method accepts as arguments the URL for the creation factory and the required attributes.

First, we use the `formNewBug()` method (discussed above) to build the RDF/XML representation of the new bug from the passed property values (**(1)**).

Next, we create a URL object with the URL of the target Change Request Creation Factory (**(2)**) and use it to open an HTTP connection. We configure that connection for HTTP POST (**(3)**) and set the HTTP Content-Type to inform the server that we are sending RDF/XML data (**(4)**).

Finally, we write out the bug to the server (**(5)**). To confirm that the POST worked, we write out the results (**(6)**) and the response code (**(7)**) and the Location header (**(8)**). If all went well, the response code should be `201`, which means `Created`, and the Location will be the URI of the newly created bug.

Try it out!: If you'd like to more details or want to try to post a bug using RDF/XML, see our walkthrough of implementing a Creation Factory for our Bugzilla adapter.