Build rich Internet applications
Using OpenLaszlo, Eclipse Laszlo IDE, and Web Tools

Skill Level: Intermediate

Christopher Judd (cjudd@juddsolutions.com)
Freelance writer and consultant
Judd Solutions, LLC

B.J. Allmon (ballmon@pillartech.com)
Freelance Writer and Consultant
Pillar Technology Group

13 Dec 2005

Many users are dissatisfied with the capabilities and performance of today's HTML-based Web applications. Users want desktop application functionality with the ease of installation and accessibility that Web applications offer. This tutorial demonstrates how to develop, package, and deploy a rich Internet application using the open source OpenLaszlo framework, Eclipse-based Laszlo IDE, and Web Tools to build business applications that delight users.

Section 1. Before you start

This tutorial demonstrates how to develop, package, and deploy a rich Internet client written using the OpenLaszlo platform and the Web service with which it interacts. The rich client takes advantage of the back-end services using the common service-oriented architecture (SOA) approach of using Simple Object Access Protocol (SOAP) Web services. You'll also learn how to effectively use the Eclipse Laszlo IDE and Eclipse Web Tools to make development easy and productive.

About this tutorial

Because most Web applications are built on HTML and HTTP, the Web makes a great platform for applications used infrequently by users, such as auction or book store sites. However, this example fails for business applications needing rich user
interfaces due to the lack of state, limited set of components, and browser inconsistencies.

If HTML and HTTP are not the answer, what is? Rich Internet Clients (RICs), also referred to as Rich Internet Applications (RIAs), are the next generation of Web applications. RICs provide the usability, responsiveness, and reuse of client/server applications with the deployment, manageability, and availability of a traditional Web application.

This tutorial explores creating a rich Internet application, FluidMotion, using the OpenLaszlo platform.

Prerequisites

You should have some basic Java™, Web development, and XML skills. A basic understanding of Web services, XPath, and persistence is helpful but not necessary.

System requirements

You need to have the following software installed:

- Java™ Development Kit (JDK) V1.4 or higher
- Apache Tomcat V5.0.28

You also need to download the following software (this tutorial explains how to install and configure it):

- OpenLaszlo V3.0.2
- Eclipse Web Tools all-in-one 0.7.1
- IDE for Laszlo V2.1

Section 2. OpenLaszlo

This section provides a detailed look at OpenLaszlo, including how it fits into the Java 2 Platform, Enterprise Edition (J2EE) space and the available development tools.

Overview

OpenLaszlo is an open source rich-client platform for developing dynamic
data-driven applications. It combines XML, JavaScript, and XPath for object-oriented, event-driven user interfaces (UIs) that run in Macromedia Flash. It includes a rich set of components and services referred to as the run-time framework and an optional Java servlet for integration.

OpenLaszlo is made available under the open Common Public License. This is a popular license because, besides the common provisions of granting access to use and modify the source code, it also has provisions for commercial distributions. This means commercial products can be derived from the OpenLaszlo platform.

The XML format used by OpenLaszlo to layout the user interface is called Laszlo XML (LZX). OpenLaszlo includes a compiler that compiles LZX and JavaScript into a binary Shockwave Flash (SWF) file that the Flash player renders.

Running in the Flash player is ideal, because it's a ubiquitous platform. Macromedia claims it is on 96 percent of the computers connected to the Internet. No single browser can claim this. Increasingly it's also becoming available on many other devices, such as handheld computers. The Flash player also makes installation easy, because it automatically gets the latest version of the application and updates itself automatically. Moreover, Flash can be used to produce eye-catching applications, because it's the same platform used for developing Web-based games and movies. So why not just use Flash tools directly? Flash uses a movie metaphor with timelines and frames that makes it difficult to build business applications. OpenLaszlo, however, uses a common application framework metaphor with tables, trees, and components that business application developers are accustomed to.

**How OpenLaszlo fits with J2EE**

Open Laszlo fits beautifully into the J2EE stack as another presentation alternative in the client tier, as shown in Figure 1.

**Figure 1. OpenLaszlo as part of the J2EE stack**
Asynchronous Java + XML (Ajax) is a new term for an old technique of combining standard Web technologies like asynchronous JavaScript, XML, XHTML, and CSS. With the new acronym, many competing frameworks/tool kits have appeared. It remains to be seen which frameworks/tool kits will become the standard. But this technology is definitely gaining popularity, being partially fueled by some amazing things Google has done.

Macromedia Flex is a commercial product for developing business applications that run in the Flash player, the same technology that many Web games and movies play in. Flex applications are written in Maximum Experience Markup Language (MXML), an XML-based language for defining the UI, and also use ActionScript for events and flow control.

OpenLaszlo, which is the focus of this tutorial, is an open source alternative to Flex. Like Flex, it runs in the Flash player, uses XML to define the UI, and uses a scripting language for events and flow control as well as a server-side piece for data integration.

OpenLaszlo can communicate with back-end services over HTTP using one of three protocols: JavaRPC, eXtensible Markup Language -- Remote Procedure Call (XML-RPC), or SOAP. JavaRPC allows server-side Java methods to be invoked. XML-RPC and SOAP are standards-based Web services protocols that enable language-independent ways of invoking remote procedures over HTTP.

If you don't need all the power of a J2EE application server, OpenLaszlo applications can be run on the included Laszlo Presentation Server (LPS), which is a customized version of Apache Tomcat, or on a basic Web server like Apache or Microsoft Internet Information Server (IIS), which is referred to as the Standalone OpenLaszlo Output (SOLO).
Development tools

To start building Laszlo applications, you need the OpenLaszlo Development Kit (OLDK) (see Resources). The OLDK contains everything you need to build OpenLaszlo applications, including the LPS, run-time components and compiler, and tons of great documentation, demos, and sample code.

In addition to the OLDK, we recommend you use the IDE for Laszlo. It's an Eclipse plug-in that includes many great features, like Laszlo project and file wizards, a Laszlo Perspective, and a visual designer and debugger. Currently, the IDE for Laszlo is an IBM alphaWorks project. However, IBM and the Eclipse Foundation recently announced that IBM would donate the code base to the Eclipse Technology Projects.

The IDE for Laszlo is built on top of the Eclipse Web Tools Platform (WTP). This is ideal since the WTP is intended to build standards (HTML, XML, CSS) and J2EE-based Web applications. Because of the WTP foundation, the IDE for Laszlo has all the tools necessary for developing a multitier application. WTP also has support for developing Web services, as you will see later.

FluidMotion architecture

The FluidMotion application presented in this tutorial is a simple Work Order Management (WOM) application that allows an employee to submit a new work order to the database. WOM applications enable organizations to collect and manage facility and equipment maintenance requests, providing value to organizations by ensuring that requests are not forgotten and are completed in a timely fashion. The FluidMotion WOM in this tutorial allows the maintenance worker to review the work orders in a list and to update and close them when they have been completed.

The FluidMotion application architecture shown in Figure 2 consists of both client and server components. The client is written using the OpenLaszlo rich Internet client framework that is rendered by the Macromedia Flash Player, which is hosted by a Web browser. The client speaks SOAP to a Web service exposed through Apache Axis, an open source Web services framework. The Web service then uses Hibernate, an open source object relational mapping framework, to persist work orders to Derby, a lightweight Java database. The server component is hosted by the OpenLaszlo server, which is an extension of Apache Tomcat, an open source Web container (see Resources for details on all of these technologies).

Figure 2. FluidMotion application architecture
Section 3. Set up development tools

As you learned in the previous section, there are several pieces of software needed to set up a successful OpenLaszlo development environment. This section explains how to install and configure them.

OpenLaszlo Development Kit

The OpenLaszlo Development Kit comes with platform-specific installers, or the Dev Kit if your operating system is not explicitly supported. To install the development kit on Microsoft® Windows®:

1. If you have not done so already, download openlaszlo-3.0.2-windows-dev-install.exe.
2. Double-click openlaszlo-3.0.2-windows-dev-install.exe.
3. Read and accept the license agreement by clicking I Agree.
4. Select a directory to install the OpenLaszlo server, and click Install.
5. Click Finish.

Upon completing the installation, the LPS immediately starts up, and your default browser also starts if it's not already open. Your browser is directed to the local version of the OpenLaszlo documentation, demos, and sample code. After you've completed this tutorial, take some time to peruse this documentation. It's well worth it.

Eclipse Web Tools
Eclipse Web Tools has a dependency on Eclipse V3.1 as well as several other Eclipse projects: EMF, GEF, and Java EMF Model. Because of all the dependencies, there are three alternatives to installing WTP. If you already have Eclipse V3.1 installed, your first option is to download all the other Eclipse projects independently and install them separately. Another alternative is to use the preconfigured Eclipse.org update site in the Update Manager.

The third alternative is the all-in-one bundle, which includes Eclipse V3.1, WTP V0.7.1, and all the other dependencies. This is the preferred option if you don't already have Eclipse V3.1 installed.

Tip: Uncompressing Eclipse in an eclipse-3.1 directory is helpful if you need multiple versions of Eclipse installed on the same computer.

To install the all-in-one bundle, complete the following:

1. If you have not done so already, download wtp-all-in-one-0.7-win32.zip.
2. Uncompress wtp-all-in-one-0.7-win32.zip into the desired directory.
3. Start Eclipse by double-clicking `eclipse.exe` in the directory in which you installed the WTP all-in-one bundle.
4. When prompted for a workspace, keep the default, then click OK.

Tip: Only version 0.7 of WTP is required for the Laszlo IDE. However, version 0.7.1 has some improvements in the Web services tools, and you'll want to take advantage of those enhancements in future sections of this tutorial.

IDE for Laszlo

After you've installed WTP, install the Laszlo IDE plug-ins. They comes packaged as a .zip bundle that can be installed using the Eclipse Update Manager.

Warning: To complete these steps, you must be connected to the Internet.

To install the Laszlo IDE, complete the following steps:

1. If you have not done so already, download laszloIDE.zip.
2. If Eclipse is not currently running, start it by double-clicking `eclipse.exe` in the directory in which you installed the WTP all-in-one bundle.
3. Select Help > Software Updates > Find and Install.
4. On the Feature Updates page, select Search for new features to install, then click Next.
5. On the Update sites to visit page, shown in Figure 3, click **New Archived Site**.

Figure 3. Update sites to visit page in the Update Manager wizard

6. Browse and find the laszloIDE.zip file, then click **OK**.

7. On the Edit Local Site dialog, confirm the .zip location by clicking **OK**.

8. Back on the Update sites to visit page, with the laszloIDE.zip checked, click **Finish**.

9. On the Search Results page, check **laszloIDE.zip**, then click **Next**.

10. On the Feature License page, read and accept the license by selecting **I accept the terms in the license agreement**, then click **Next**.

11. On the Installation page, click **Finished**.

12. On the Feature Verification page, click **Install All**.

13. When prompted to restart, click **Yes**.

After Eclipse has restarted, the WTP and Laszlo IDE is installed, and you're ready to begin developing your Web services and OpenLaszlo client.
Section 4. Develop the Web services

In this section, you use Eclipse Web Tools to build a basic Web service for the FluidMotion application that the OpenLaszlo client will call. The Web service will use Hibernate to persist the data to an Apache Derby database. You begin by creating a dynamic Web project. Then you create the service as a Plain Old Java Object (POJO) and then run the Web services wizard to expose it as a SOAP service. You conclude by packaging it as a Web application.

Note: Before you begin developing the Web service, make sure any LPS servers are not running. Otherwise, Tomcat servers started by WTP error trying to bind to port 8080, which is the same port used by the LPS server. To shut down the LPS server, activate the command window, and press Ctrl+C.

Create a dynamic Web project

A Web service will ultimately be packaged as a Web application and deployed to a Web container. So you begin by creating a dynamic Web project that gives you a project with a structure necessary for a deployable Web application. In other words, it will contain a WEB-INF directory, WEB-INF/lib directory, and a web.xml file. You name this project $fms$ for FluidMotion Server.

To create the dynamic Web project, complete the following steps:

1. Select File > New > Project from the Eclipse main menu.
2. On the Select a wizard page, select Web > Dynamic Web Project, then click Next.
3. On the Dynamic Web Project page, enter the project name, then click Show Advanced.
4. Click New to add a new Tomcat server.
5. On the New Server Runtime page, select Apache > Apache Tomcat v5.0, as shown in Figure 4, and click Next.

Figure 4. Selecting the Apache Tomcat V5.0 run time
6. **Use Browse** to find the Tomcat installation directory.

7. Next, select a JDK from the JRE list, as shown in Figure 5, and click **Finish**. Note that Eclipse typically uses a Java Runtime Environment (JRE) by default, but Tomcat must be configured to use a JDK because it needs the compiling tools to convert JavaServer Pages™ (JSPs) to servlets. So make sure you have a JDK Runtime configured and selected here.

**Figure 5. Tomcat server configuration**
8. Back on the Dynamic Web Project page, click **Finish**.

9. If you're prompted for a license agreement, read the agreement, and accept it by clicking **I Agree**.

10. If prompted to open the J2EE perspective, click **Yes**.

Upon completing the Dynamic Web Project wizard, you will have a new dynamic Web project underneath the Dynamic Web Projects node, similar to the project shown in Figure 6.

**Figure 6. Contents of the new dynamic Web project**
Creating the service and data transfer objects

Creating the POJO service and data transfer objects (DTO) is identical to creating any other Java class with a couple of minor caveats. First, Web services containing collections are best exposed as arrays. This enables the Web Services Description Language (WSDL) that describes the service to be typed to handle the complex types appropriately. In addition, not all languages that support Web services have a concept of a dynamic untyped collection. Second, the DTOs must have no parameter constructor so the XML marshaling can create a new instance to hydrate.
Listing 1 is the **WorkOrderService** class exposed as a Web service in the next section.

### Listing 1. The **WorkOrderService** class

```java
package com.ibm.laszlo.services;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.ibm.laszlo.dto.Building;
import com.ibm.laszlo.dto.Floor;
import com.ibm.laszlo.dto.WorkOrder;
import com.ibm.laszlo.util.HibernateUtil;

/**
 * Service for work order management.
 */
public class WorkOrderService {

    /**
     * Finds a specific work order by id.
     * @param id unique id of work order.
     * @return work order.
     */
    public WorkOrder findWorkOrderById(int id) {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();
        WorkOrder workOrder = (WorkOrder) session.get(WorkOrder.class, new Integer(id));
        tx.commit();
        HibernateUtil.closeSession();
        return workOrder;
    }

    /**
     * Returns all work orders.
     * @return all work orders.
     */
    public WorkOrder[] findAllWorkOrders() {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();
        List workOrders = session.createQuery("from WorkOrder").list();
        tx.commit();
        HibernateUtil.closeSession();
        return (WorkOrder[]) workOrders.toArray(new WorkOrder[workOrders.size()]);
    }

    /**
     * Create a new work order.
     * @param contact contact person's name.
     * @param phone contact person's phone number.
     * @param email contact person's email.
     * @param description description of problem.
     * @param building building problem is in.
     * @param floor floor in building with problem.
     * @param severity severity of the problem.
     * @return new work order's id.
     */
    public int createWorkOrder(String contact, String phone,
```
String email, String description,
String building, String floor, int severity) {
    Session session = HibernateUtil.currentSession();
    Transaction tx = session.beginTransaction();
    WorkOrder workOrder = new WorkOrder(contact, phone,
            email, building, floor, description, severity);
    Date currentDate = new Date();
    workOrder.setDateRequested(currentDate);
    workOrder.setLastModified(currentDate);
    workOrder.setStatus(WorkOrder.STATUS_REQUESTED);
    session.saveOrUpdate(workOrder);
    tx.commit();
    HibernateUtil.closeSession();
    return workOrder.getId();
}

/**
* Collection of buildings and their floors.
* NOTE: Hard coded for simplicity but could be read from a
* database.
* @return
*/
public Building[] getBuildings() {
    List buildings = new ArrayList();
    Building building = new Building(1, "HQ”);
    building.addFloor(new Floor(1, "Floor 1”));
    building.addFloor(new Floor(2, "Floor 2"));
    building.addFloor(new Floor(3, "Mezzanine"));
    buildings.add(building);
    building = new Building(2, "Trump Tower”);
    building.addFloor(new Floor(1, "Trump 1"));
    building.addFloor(new Floor(2, "Trump 2"));
    building.addFloor(new Floor(3, "Trump 3"));
    buildings.add(building);
    return (Building[])buildings.toArray(
                new Building[buildings.size()]);
}

Notice how the class has five methods: findWorkOrderById(),
findAllWorkOrders(), createWorkOrder(), updateWorkOrder(), and
getBuildings(). These five methods will be exposed as SOAP operations. Also
notice that the getBuildings() and findAllWorkOrders() methods return
typed arrays rather than java.util.List. The WorkOrder, Floor, and
Building classes referenced in this service are simple JavaBeans with no
parameter constructors.

Note that if your services or DTO classes use any type of third-party libraries, such
as the WorkOrderService class uses Hibernate, those libraries' JARs need to be
included in the WebContent/WEB-INF/lib directory.

Exposing the POJO service as a Web service

Now that you have a POJO service, expose it as a Web service. This is called the
bottom-up approach, which means you start with a Java class and use it to generate
the WSDL describing that method. The alternative is the top-down approach where
you start with a WSDL and generate Java code from it.
To expose `WorkOrderService` as a SOAP Web service, complete the following:

1. Select **File > New > Other** from the Eclipse main menu.
2. On the Select a wizard page, select **Web Services > Web Service**, then click **Next**.
3. On the Web Services page shown in Figure 7, check **Generate a proxy**, **Test the Web service**, and **Monitor the Web service**, then click **Next**.

![Figure 7. Options generating the Web service](image)

4. On the Object Selection Page, choose the bean you want to expose as a Web service, such as `WorkOrderService`, then click **Next**.
5. On the Service Deployment Configuration page, just click **Next**. These are configurations for which project to add the Apache Axis JARs to. In addition, it's creating a new project to enable you to test your Web service without having to have the Laszlo client.
6. As shown in Figure 8, all the defaults on the Web Service JavaBean Identity page should be kept, so click **Finish**. This page enables you to explicitly check which methods will be exposed via the WSDL as well as the style of SOAP to use.

**Figure 8. Configuration for exposing methods and defining style**

Upon completing the Web service wizard, several things have happened. First, a new dynamic Web project was created with the same name, and the client was appended to the end of it. This project contains a consumer of your Web service and a JSP-based testing tool, shown in Figure 9. This test client is handy, because you don't have to have your OpenLaszlo client completed to test that your Web service worked properly.

**Figure 9. Web service testing client**
Second, if the Tomcat server is not already started, the server is started and added to the servers view (see Figure 10) for managing the starting, stopping, and redeploying of the application.

**Figure 10. Tomcat server running**

Third, both dynamic Web projects are deployed to the Tomcat server, and you can begin testing the services via the Web-based client. An internal Eclipse browser opens to the testing page automatically.

Finally, a TCP/IP monitor, shown in Figure 11, is configured so you can watch the SOAP traffic between the test client and your Web service.

**Figure 11. TCP/IP monitor watching traffic between the client and Web service**
Packaging the Web service

After you've tested your Web service using the test client, you're ready to package your service as a standard Web application or .war file to deploy it on a Web container. This is especially true for developing the OpenLaszlo client. Both the Tomcat server automatically started by Eclipse in the previous section and the OpenLaszlo server you need to use for developing the client run on port 8080. You need to package the Web service so it can be deployed to the OpenLaszlo server.

To "WAR" up the Web service, complete these steps:

1. Select **File > Export** from the Eclipse main menu.
2. Select **WAR file**, and click **Next**.
3. On the WAR Export page, select the Web module containing the Web service and a resulting destination .war file, and click **Next**.

Warning: Stop the Tomcat server started by Eclipse before starting the OpenLaszlo server.

To deploy the Web service to the OpenLaszlo server, complete the following:

1. Copy the .war file, fms.war, to the `<OpenLaszlo Server>/Server/tomcat-5.0.24/webapps` directory.
2. Start the OpenLaszlo server.

Note: After deploying the Web service to the OpenLaszlo server, the URL to the WSDL is http://localhost:8080/fms/wsd/WorkOrderService.wsdl.
Section 5. Develop the rich client

This section demonstrates how to configure the Laszlo IDE workspace, create a new Laszlo project, and start up the Laszlo server. Regarding development, it covers leveraging Laszlo components, event scripting, data binding, debugging, running the application inside the IDE, and packaging.

Configure Laszlo IDE -- Workspace-specific

Configuring the Laszlo IDE is workspace-specific. See the Eclipse Preferences window shown in Figure 12. The configuration panel can be located by selecting Window > Preferences from the Eclipse main menu.

The Laszlo IDE must be configured to run Laszlo applications from the IDE. The main requirement is to specify the LPS Web Root, Context Root, and application server port (Apache Tomcat). The LPS Web Root is where the installation instance of the Laszlo server resides and needs to point to the Context Root.

The port defined in Figure 12 is the default port that the Apache Tomcat servlet container listens on. The default design view and preferred setting is Local Design View. It allows for easier editing of the component attributes. Optionally, the Live Design View can be launched as the default view for editing. A browser of choice can be used to launch the application from the IDE.

Figure 12. Laszlo IDE configuration panel
Create a Laszlo project

Before you can begin developing Laszlo artifacts, you need to create a project to hold them. To create a Laszlo project, complete the following steps:

1. Select **File > New > Project** from the Eclipse main menu. The new project wizard should appear as shown in Figure 13.
   
   **Figure 13. New Laszlo Project wizard**
2. Select the wizard type **Laszlo Project**, then click **Next**.

3. Enter a project name, and click **Next** as shown in Figure 14. **Figure 14. Naming the new Laszlo Project wizard**
4. Define custom LPS options as shown in Figure 15, and click Finish. 
**Figure 15. Laszlo Project LPS settings**
Upon completing the Laszlo Project wizard, you're prompted to switch to the Laszlo perspective. A new project like the one shown in Figure 16 is then created.

**Figure 16. New Laszlo Project**
Notice in Figure 16 that the new Laszlo project contains two component libraries.

With the Laszlo project created, you’re ready to begin adding Laszlo or LZX files. To create a new Laszlo file, complete the following:
1. If you’re in the Laszlo Perspective, select **File > New > Laszlo File** from the Eclipse main menu.

2. As shown in Figure 17, enter a container directory relative to your project, a file name, and a file type of either **Canvas** for visual pages or **Library** for a collection of Laszlo classes, components, and/or utilities.

![Figure 17. New Laszlo file wizard](image)

Upon completing the new Laszlo file wizard, your project will contain a new file similar to the following:

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

Notice this is an .xml file containing two canvas tags, which are containers for visual components. Later in this tutorial you use this new file to create the home page.

**Start the Laszlo server**

Your OpenLaszlo server must be running before you can deploy and test a Laszlo application using Run as Laszlo application functionality.

To start the OpenLaszlo server on Windows, select **Start > Programs > OpenLaszlo Server > Start OpenLaszlo Server** from the Windows taskbar.
A bundled instance of Apache Tomcat executing the Laszlo explorer tries to launch. The Explorer can also be launched by navigating to http://localhost:8080/lps-3.0.2/laszlo-explorer/index.jsp when the LPS server is running.

To shut the server down, either select **Start > Programs > OpenLaszlo Server > Stop OpenLaszlo Server** from the Windows taskbar, or press **Ctrl + C** with the server console window activated.

**Lay out the application home page and navigation**

Begin the Laszlo application by developing the home page (index) and navigation. The navigation is two links to view and create work. The navigation makes use of the `<tree>` component that is populated by an XML data set. The menu code is made available as an include by wrapping the code inside a `<library>` tag.

Listing 2 shows the content of the menu.lzx file.

**Listing 2. The content of the menu.xml file**

```xml
<library>
  <!-- Menu data -->
  <dataset name="fm">
    <nav nav="View Work" url="viewWork.lzx?lzt=html"/>
    <nav nav="Create Work Order" url="createWork.lzx?lzt=html"/>
  </dataset>

  <!-- START NAVIGATION -->
  <view width="200" height="200" x="20" y="120">
    <tree datapath="/fm:" autoscroll="true" showroot="false">
      <tree datapath="*" text="@path{"@nav"}" open="true" isleaf="true" selectiontype="single">
        <method event="onactivate">
          var url = this.datapath.xpathQuery('@url');
          if (url !="null") {
            LzBrowser.loadURL(url,'_self');
          }
        </method>
      </tree>
    </tree>
  </view>
  <!-- END NAVIGATION -->
</library>
```

The index page shown in Figure 18 is a simple page that includes only the menu and a logo.

**Figure 18. Index page**
The code for the index performs an include of the menu and also includes an empty \(<\text{splash/}\) tag to present the user with a progress bar when the application is loading. The \(<\text{splash/}\) component comes in two flavors to control the presentation when the application is loading. The generic way, the empty tag or \(<\text{splash/}\)\), triggers a progress bar by default. Alternately, the \(<\text{splash/}\) tag can contain view elements that override the default progress bar with any custom image, element, or loading animation that you desire.

Listing 3 shows the content of the index.lzx file.

Listing 3. The content of the index.lzx file

```xml
<canvas>
  <view resource="/images/fm.gif"/>
  <menuseparator width="600" x="-1" y="97" height="6"/>
  <include href="lz/tree.lzx"/>
  <splash/>
  <include href="menu.lzx"/>
</canvas>
```

Lay out the Work Order form

The hill isn't steep if you're without knowledge of Laszlo components. Nor does it impede your ability to create sophisticated user interfaces. The Work Order shown in Figure 19 contains a simple form.
The Work Order form demonstrates using a few different built-in Laszlo components. An XML dataset-driven combo box is used for the building and floor selectors. The location dataset containing an XML representation of the building and floor data is shown in Listing 4.

Listing 4. The location dataset

```
<dataset name="location">
    <locations>
        <building id="1" name="HQ">
            <floor id="1">Floor 1</floor>
            <floor id="2">Floor 2</floor>
            <floor id="3">Mezzanine</floor>
        </building>
        <building id="2" name="Trump Tower">
            <floor id="1">Trump 1</floor>
            <floor id="2">Trump 2</floor>
            <floor id="3">Trump 150</floor>
        </building>
    </locations>
</dataset>
```

Listing 5 contains the code for buildings and floors combo boxes.
Listing 5. Buildings and floors combo boxes

Notice the `datapath` attribute on the `textlistitem`. This is what binds the values in the drop-down list to the location dataset described earlier. The text before the colon represents the dataset name. The value after is the XPath the component is bound to. The text and value use XPath to select the attributes of the dataset to be used as the text and values of the items in the drop-down list, respectively.

After a building is selected, an event populates the floors combo box by calling the `getFloors()` method and passing it to the active building ID. Listing 6 shows the `getFloors()` method called by the `onselect` event of the building combo box.

Listing 6. The `getFloors()` method

The contact, e-mail, phone, and description (with multiline attribute set to `true`) are `<edittext>` components. For example:

The following severity `<slider>` component allows a requestor to assign a severity to the work order:

The Submit button, below, is a standard `<button>` component and allows the form to be submitted. The form is going to be stubbed out for now. Later this form will be submitting to the Web service.
Lay out the work view

The work view shown in Figure 20 uses the <grid> component with other embedded components for reviewing the current work requests.

Figure 20. FluidMotion view work page (Laszlo grid component)

![FluidMotion view work page](image)

This grid was built with the static dataset in Listing 7 to model the server-side call that it connects to later.

Listing 7. Static dataset

```xml
<dataset name="work">
  <work>
    <wo id="100301">
      <date_req>11/15/2005</date_req>
      <date_last_mod>11/18/2005</date_last_mod>
      <bl_name>HQ</bl_name>
      <fl_name>Floor 3</fl_name>
      <contact>Tim Dennison</contact>
      <email>tden@rockstar.rock</email>
      <phone>679.111.1123</phone>
      <description>Need emergency service on my equipment!</description>
      <comments>I'm on it!</comments>
      <severity>Low</severity>
      <status>Complete</status>
    </wo>
    <wo id="100302">
      <date_req>11/18/2005</date_req>
      <date_last_mod>11/19/2005</date_last_mod>
      <bl_name>Trump Tower</bl_name>
      <fl_name>Trump 2</fl_name>
      <contact>Donald Trump</contact>
      <email>dtrump@trump.usa</email>
      <phone>603.239.4326</phone>
      <description>Need recruiter machine repaired.</description>
      <comments>This is on hold for now.</comments>
      <severity>Urgent</severity>
      <status>On Hold</status>
    </wo>
  </work>
</dataset>
```
This Laszlo <grid> component can contain <gridcolumn> components as shown in Listing 8.

Listing 8. Laszlo <grid> component can contain <gridcolumn> components

```xml
<grid datapath="work:/work[1]" contentdatapath="wo"
      showitems="7" height="135" width="625" id="wo_grid"
      x="180" y="115" multiselect="false">
  <gridcolumn resizable="false">Requested
    <text datapath="date_req/text()" text="$path{'text()'}"/>
  </gridcolumn>
  <gridcolumn resizable="true" width="90">Contact
    <text datapath="contact/text()"/>
  </gridcolumn>
  <gridcolumn resizable="false">Severity
    <text datapath="severity/text()"/>
  </gridcolumn>
  <gridcolumn resizable="false" width="90">Status
    <combobox defaulttext="Requested" editable="false"
              id="status" datapath="status/text()">
      <textlistitem text="Assigned" id="assigned" value="assigned"/>
      <textlistitem text="Active" id="active" value="active"/>
      <textlistitem text="Complete" id="complete" value="complete"/>
      <textlistitem text="On Hold" id="hold" value="hold"/>
      <textlistitem text="Cancelled" id="cancel" value="cancel"/>
    </combobox>
  </gridcolumn>
  <gridcolumn resizable="false" width="240">Comments
    <inputtext datapath="comments/text()" id="comments"
               text="comments/text()" multiline="true" height="35"
               selectiontype="multi"/>
  </gridcolumn>
</grid>
```

Run the application and debug

The Laszlo IDE allows the developer to launch the current file being developed from the IDE. You can leverage the Run As command shown in Figure 21 by simply right-clicking the document and selecting Run As > Laszlo Application. The IDE then launches.

Figure 21. Laszlo IDE Run As feature
Debug statements can be constructed by leveraging the `debug.write()` method. The debug statements are shown below:

```javascript
debug.write("getting floors for " + bl);
download.write("setting floor datapath to " + fl);
```

To see the debug window and results, set the debug attribute on the canvas tag to `true` as shown in Figure 22.

```
<canvas width="100%" height="100%" debug="true">
```

Figure 22. Laszlo Debugger window

Save the Laszlo as a .war file (bundle and deploy)

The Laszlo IDE allows the developer to bundle a standard J2EE Web archive (WAR) file by completing the following steps:

1. Select **File > Export** from the Eclipse main menu.
2. Select Laszlo WAR File as shown in Figure 23, and click Next.

Figure 23. Laszlo .war file Export tool

3. As shown in Figure 24, select a Laszlo project from the Project combo box, and select an appropriate destination file with a .war extension from the appropriate directory of the OpenLaszlo server in the LPS root directory.

Figure 24. Laszlo WAR Export utility
The Laszlo WAR Export utility bundles all of the necessary dependencies needed for running the application independently of the LPS server. In other words, it enables the application to be deployed and run on virtually any Java Web container. The dependencies include, but are not limited too, all the .jar files, the built-in Laszlo components, and the Laszlo DTD. It also includes all the base Laszlo components, CSS files, Java class files, Flash, and HTML. This feature makes packaging, deploying, and integrating the rich-client applications into the J2EE architectures effortless.

Section 6. Hook Web service and client together

Now that you've completed the Web service and laying out the rich client, they have to be wired together. In this section you look at two examples of how to call Web services from an OpenLaszlo client.

Simple Web service call

In an earlier example, the contents of the building and floors combo boxes were built
from a static location dataset. While this is fine for laying the application out, it's not a good idea for long-term maintenance. You don't want to have to update the code every time a new building is added. This could be externalized to an .xml file that is loaded from the Web container, but that too seems like too much work to maintain. Instead, this list and other pick lists are likely to come from the Web service and a database. This is why the Web service has a `getBuildings()` operation.

Listing 9 is an example of calling the `getBuildings()` operation.

**Listing 9. The `getBuildings()` operation**

```xml
<dataset name="location" />

<soap name="WorkOrderService" wsdl="http://localhost:8080/fms/wSDL/WorkOrderService.wsdl" autoload="true">
    <remotecall name="getBuildings" funcname="getBuildings" dataobject="location">
        <method event="ondata" args="value">
            debug.write(location.serialize());
        </method>
    </remotecall>

    <method event="onload">
        WorkOrderService.getBuildings.invoke();
    </method>

    <method event="onerror" args="error">
        debug.write('error:', error);
    </method>
</soap>
```

The XML has been removed from the location dataset. This is now dynamically populated with the XML from the call to the `getBuildings()` operation.

Next, define the SOAP server using the `soap` element. Here you need to give it a name and a URL to the WSDL generated with the Web service. For this example, autoload the Web service, because you want the combo box to be populated as soon as the user views the page.

Within the SOAP definition, you also need to declare the remote operations on the Web service that you're going to call using the `remotecall` element. Here you define the `getBuildings` operation. The `name` attribute is the local name that the JavaScript references when executing the operation, and the `funcname` attribute must match the name in the WSDL, which is the same as the Web service Java class. The `dataobject` attribute indicates which dataset will be populated with the results. Specify the location dataset used earlier.

Inside the `remotecall`, you can specify an `ondata` event. This event is called after the Web service returns with the results. In this example, you're just writing the result to the debugger window. This could also have been used to populate fields or something else. Even though this implementation of writing to the debugger is a simple one, it's still worth noting, because it's a power tool. Sometimes the Web
service produces XML that doesn’t look like the XML with which you originally designed your user interface. It can also be difficult to determine how the XML is interpreted by the OpenLaszlo platform. This example shows how you can call the `serialize` method on the dataset to spit out the XML in a pretty XML format. Listing 10 is the example from the Web service call, which is different than the original XML you designed with.

Listing 10. Call the `serialize` method on the dataset to spit out the XML in a pretty XML format

```xml
<location>
  <getBuildingsReturn>
    <floors>
      <floors>
        <name>Floor 1</name>
        <id>1</id>
      </floors>
      <floors>
        <name>Floor 2</name>
        <id>2</id>
      </floors>
    </floors>
    <name>HQ</name>
    <id>1</id>
  </getBuildingsReturn>
  <getBuildingsReturn>
    <floors>
      <floors>
        <name>Trump 1</name>
        <id>1</id>
      </floors>
      <floors>
        <name>Trump 2</name>
        <id>2</id>
      </floors>
    </floors>
    <name>Trump Tower</name>
    <id>2</id>
  </getBuildingsReturn>
</location>
```

Pay attention to a couple of things. First, the Web service did not use any attributes. Every data item was made an element. Second, `location`, the name of the dataset, is the outermost XML element. Third, `buildings` was replaced by `getBuildingsReturn`, the name of the method called with the word `Return` appended to it.

There are two remaining events that apply to the SOAP definition and not the remote call. They are `onload` and `onerror`. The `onload` is just a convenient way to invoke the method so the check box is populated when the page is initially viewed. Of course, to have it happen immediately on viewing, you also have to set `autoload` to `true` on the `soap` element. In the `onload`, you explicitly invoke the `getBuildings()` operation on `WorkOrderService`. In the `onerror` event, just write the error message to the debugger window so you can see the error.

Parameterized Web service call

The second Web service submits a new work order. Here you call the
createWorkOrder operation. Listing 11 is the definition that is included in the WorkOrderService soap definition from the previous section.

Listing 11. The definition included in the WorkOrderService definition

```
<remotecall name="createWorkOrder" funcname="createWorkOrder">
  <param value="${contact.text}" />
  <param value="${phone.text}" />
  <param value="${email.text}" />
  <param value="${description.text}" />
  <param value="${bl.text}" />
  <param value="${fl.text}" />
  <param value="${severity.value}" />
  <method event="ondata" args="value">
    debug.write('New Work Order ID = ' + value);
  </method>
</remotecall>
```

In this example, you also declare a remote call with a name and a funcname equal to the operation name in the WSDL and the method in the Web service class. Because this operation takes parameters, you have to include param elements. The parameters are in the same order as the Web services class createWorkOrder() method signature. Here, the values are referencing the text from the edittext components and other components. This remotecall also has an ondata event that writes the return value of the new work order ID to the debug window.

Warning: OpenLaszlo requires the Web service to return at least one value. It cannot just return a void, or the onerror event is called.

To invoke the call to createWorkOrder, add an onclick event to the button, and call the invoke method on the createWorkOrder remotecall:

```
<button isdefault="true" text="Submit" x="350"
  onclick="WorkOrderService.createWorkOrder.invoke()"/>
```

Deploy to Apache Tomcat

After the Web service and OpenLaszlo clients are complete, it's time to deploy the applications.

Because both the output of the Web service (fms) and the OpenLaszlo client (fm) are .war files, you should be able deploy them on any compliant Web container or J2EE application server. Because deployment is container specific, please see your container documentation for instructions to deploy the application. Apache Tomcat is a popular Web container, so this section reviews how to deploy the applications to this type of container.

To deploy to an Apache Tomcat server on Windows, complete the following:

1. Copy the exported fm.war and fms.war to the <jakarta-tomcat>/webapps
2. Start Apache Tomcat by executing the `<jakarta-tomcat>/bin/startup.bat` script.

That's all there is to it. To test your application, use a Web browser to navigate to `http://localhost:8080/fm/index.lzx`.

---

Section 7. Summary

With the current push toward rich internet applications, OpenLaszlo is poised to be a strong alternative to Ajax or Macromedia Flex. In this tutorial, you saw how simple it is to create attractive and responsive OpenLaszlo applications using the rich component library and a little bit of XML. You saw how well OpenLaszlo fits into the J2EE application architecture by integrating it with the business tier exposed as a SOAP Web service. Ultimately you learned how to package and deploy it to an Apache Tomcat Web container. Throughout the tutorial, you learned how to use many of the Eclipse Laszlo IDE and Eclipse Web Tools to simplify the development process.
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
<th>Download method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part 2 source code</td>
<td>os-laszlo_article_smile.zip</td>
<td>8MB</td>
<td>HTTP</td>
</tr>
</tbody>
</table>

*Information about download methods*
Resources

Learn

• Read "Build a Web-based client with the Eclipse Web Tools Platform" (developerWorks, November 2004), a tutorial showing how to build a complete Web-based auction client application using the Eclipse Web Tools Platform.

• Read "Build a Web service using the Eclipse Web Tools Platform," a tutorial showing how to build a Web service using the Eclipse Web Tools Platform.

• Read "Building Web Applications with Eclipse, WTP, and Derby" (developerWorks, September 2005), an article explaining how to build Web applications using the Web Tools Platform, Derby, and Jakarta-Tomcat.

• "Building apps using Asynchronous JavaScript with XML (Ajax)" (developerWorks, November 2005), a tutorial for getting started writing Ajax applications.

• Visit Google Maps and Google Suggest for examples of Ajax.

• Learn more about Macromedia Flex.

• Check out the "Recommended Eclipse reading list."

• Browse all the Eclipse content on developerWorks.

• New to Eclipse? Read the developerWorks article "Get started with Eclipse Platform" to learn its origin and architecture, and how to extend Eclipse with plug-ins.

• Expand your Eclipse skills by checking out IBM developerWorks' Eclipse project resources.

• To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.

• For an introduction to the Eclipse platform, see "Getting started with the Eclipse Platform."

• Stay current with developerWorks' Technical events and webcasts.

• Watch and learn about IBM and open source technologies and product functions with the no-cost developerWorks On demand demos.

• Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.

• Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

• Get the latest downloads, news, articles, and announcements of the following technologies: OpenLaszlo, Eclipse, Web Tools, and the new IDE for Laszlo,
Jakarta Apache Tomcat, Hibernate, Apache Derby database, Apache Axis.

- Download the IDE for Laszlo V2.1.
- Check out the latest Eclipse technology downloads at IBM alphaWorks.
- Download Eclipse Platform and other projects from the Eclipse Foundation.
- Download IBM product evaluation versions, and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.

Discuss

- The Eclipse Platform newsgroups should be your first stop to discuss questions regarding Eclipse. (Selecting this will launch your default Usenet news reader application and open eclipse.platform.)
- The Eclipse newsgroups has many resources for people interested in using and extending Eclipse.
- Participate in developerWorks blogs and get involved in the developerWorks community.

About the authors

Christopher Judd
Christopher Judd is the president and primary consultant for Judd Solutions, LLC, international speaker, open source evangelist, Central Ohio Java Users Group coordinator, and co-author of Enterprise Java Development on a Budget and Pro Eclipse JST. He has spent eight years developing software in the insurance, retail, government, manufacturing, service, and transportation industries. His current focus is consulting, mentoring, and training with Java, J2EE, J2ME, Web services, and related technologies.

B.J. Allmon
B.J. Allmon is a software consultant with Pillar Technology Group. He has worked mostly in J2EE Web application environments and has experience developing J2ME mobile software that integrates with Web services. He has a passion for learning and leveraging Java open source and has spent most of the past six years developing Web applications.