LZProject
An OpenLaszlo 4.0 Blueprint Application

May, 2007

Overview

OpenLaszlo is the premiere open source platform for creating Rich Internet Applications (RIAs). OpenLaszlo applications are written in LZX, an XML language that uses custom tags and industry-standard JavaScript to describe client behavior. LZX applications are compiled by the OpenLaszlo compiler, which is bundled as part of the Laszlo Presentation Server (LPS), and delivered either as SWF files to be executed by the Adobe Flash Player, or as DHTML ("Ajax") to be rendered directly by the native JavaScript engines of internet browsers such as Mozilla Firefox or Internet Explorer. LZX applications can be deployed either using the OpenLaszlo server as a proxy connection, or as Standalone OpenLaszlo Objects—which we call "SOLO" deployment. LZX applications typically communicate with "back ends"—which may be written in Java, PHP, SQL, etc—by sending XML over HTTP.

For more on OpenLaszlo, please see the OpenLaszlo white paper, and the OpenLaszlo Explorer on OpenLaszlo.org.

LZProject is an application, written in LZX, for tracking projects. It includes both the client application, written in LZX, and all necessary back end components, including Java servlets and a small database. It's been designed to be simple to deploy and easy to modify. We hope that you'll download and use the application as you read this paper, and, if you're a software developer, that you'll download the sources and play with them too.

This document walks you through the LZProject application, logical section by logical section. It illustrates the principles of building a complete OpenLaszlo web application, both client side and server side, and demostrates best practices for integrating OpenLaszlo 4.0 applications with a JSP/Servlet back end. OpenLaszlo does support SOAP and remote RPC protocols, but for the sake of simplicity, LZProject is built using Representational State Transfer (REST) web services.

This document is intended for developers who are familiar with XML and JavaScript, JSP/Servlet technology and SQL databases. Familiarity with OpenLaszlo and LZX will make it much more comprehensible, so if you're not an LZX developer you may want to, at a minimum, invest ten minutes to go through the self-paced tutorial Laszlo in Ten Minutes before proceeding. An understanding of the difference between proxied and SOLO deployment, as explained in the OpenLaszlo white paper and documentation, will be especially beneficial.

LZProject Variants

LZProject comes in two flavors: "proxied" and SOLO.

The "proxied" version of LZProject includes everything you need to build and delploy the complete web application for proxied deployment. It contains the OpenLaszlo 4.0 (LPS) server, an Apache Derby embedded SQL database ("Derby"), and a combination of JSP/Servlet technology for the backend part of the application.

The "SOLO" variant contains everything you need to build and deploy the client application, written in LZX. Once a SOLO application has been compiled and deployed, the OpenLaszlo Server is no longer used, so LPS is not included in the SOLO version.

Software Requirements for LZProject

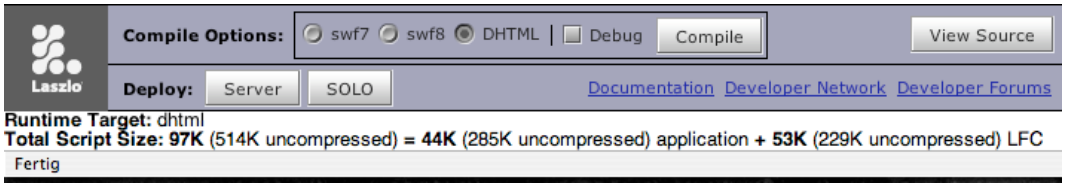
The proxied version of LZTRack can be deployed on any machine meeting the following requirements.

- Java SDK 1.4 or greater.
- A Java Servlet container supporting at least version 2.2 of the Java Servlet Specification.
- 512 MB physical RAM.
- 800 MHz or higher recommended.

The SOLO version requires a Java web application containing two SOLO deployed versions of LZProject (DHTML/Ajax and SWF7), the Java back end and the Derby database. This version will run on any system capable of running a Java servlet container such as Tomcat 5.0 or higher.

OpenLaszlo runtime options

As we mentioned above, OpenLaszlo applications can be compiled, from the same source, to run either as a DHTML/Ajax application or as a Flash application (SWF movie embedded into an HTML page). The Developer's Consol of the OpenLaszlo server allows you to select wether you're compiling to JavaScript code or Flash byte code with a single button push.



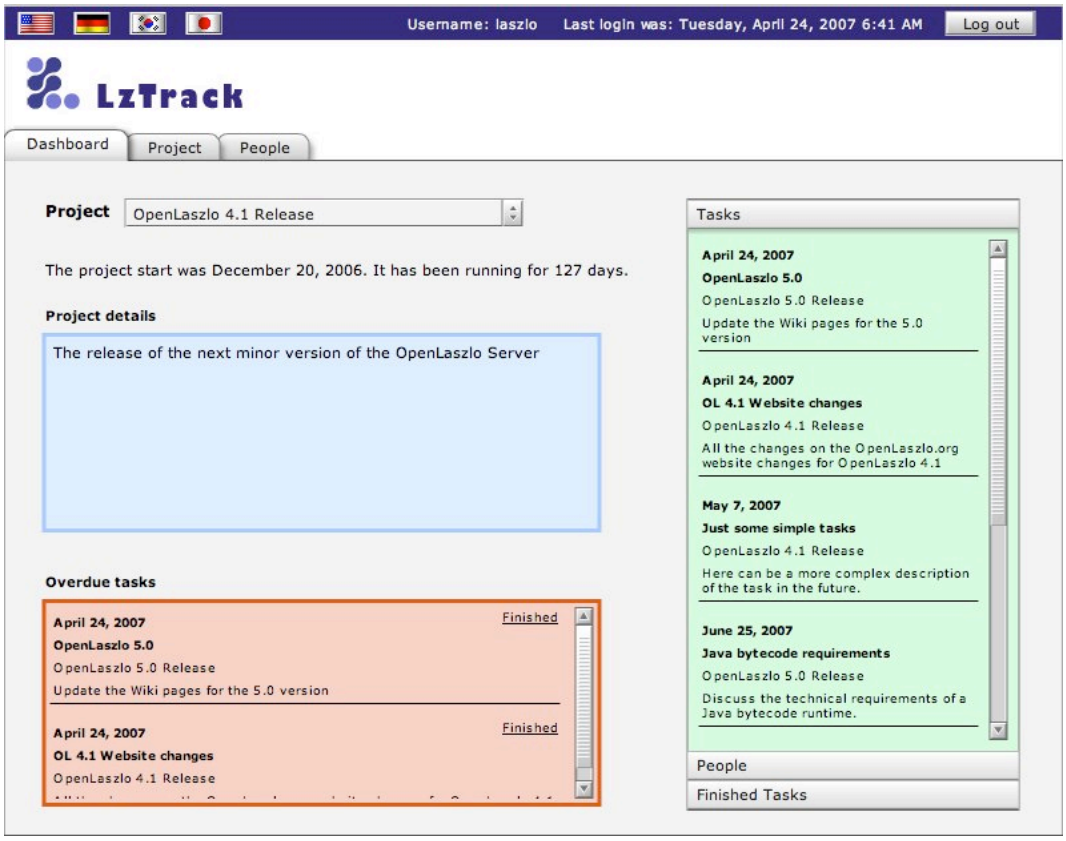
With OpenLaszlo Ajax or Flash is a matter of one click…

Compared to traditional page-based HTML web applications, which consisti of a number of generated HTML pages, Rich Internet Applications are more complex. This complexity brings enormous usability benefits to users and enormous economic benefits to deployers, but it does place a burden on developers.

OpenLaszlo manages this increased complexity by providing you a programming language, LZX, optimized for exactly that task.

Application review - LZProject

LZProject is a simple tool for tracking project tasks. Let's take a look at its welcome screen:



LZProject dashboard screen

The application contains the following functionality:

- Authentication and general user management
- Create, Read, Update, and Delete (CRUD) operations through web service calls
- Integration with Java backend through REST web services
- Internationalization, with the ability to switch between locales without an application reload
- Automatic resizing with scrollbars that appear and disappear according to need
- LZ X-Ray, a tool for testing web service calls at runtime

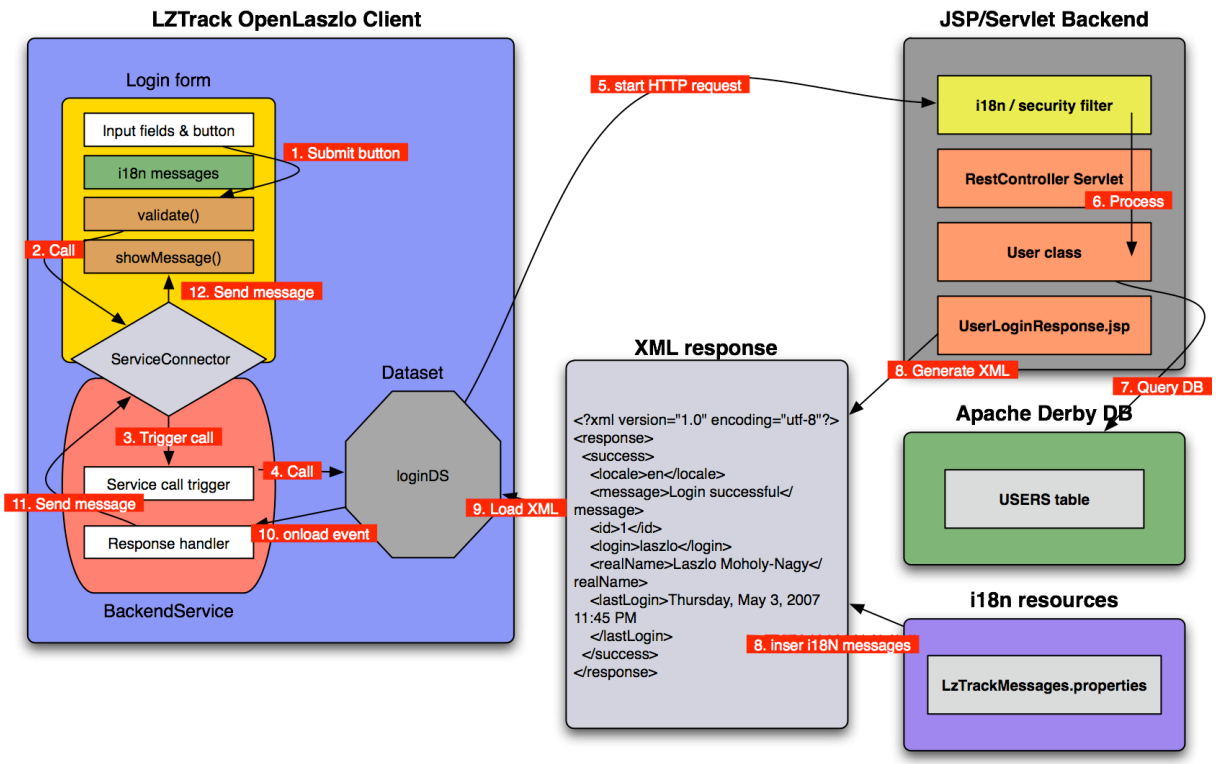
LZProject Client Architecture

The LZProject client comprises:

- View area – the visual components of the application, the interface, as it is displayed in the browser
- messages used for localization
- Service connectors – connecting the visual components to the client-side backend service classes
- Backend services – the classes connecting to the server-side REST services
- Datasets – the object representation of XML files passed from the server into the LZProject client
- Datapointers – datapointers are used to access the XML nodes in the dataset as well as to monitor the arrival of data in the applications
- Events and the corresponding methods which are used to handle the application flow

Data Flow

The following diagram shows the anatomy of the login part of LZProject application and the flow of a login service call.

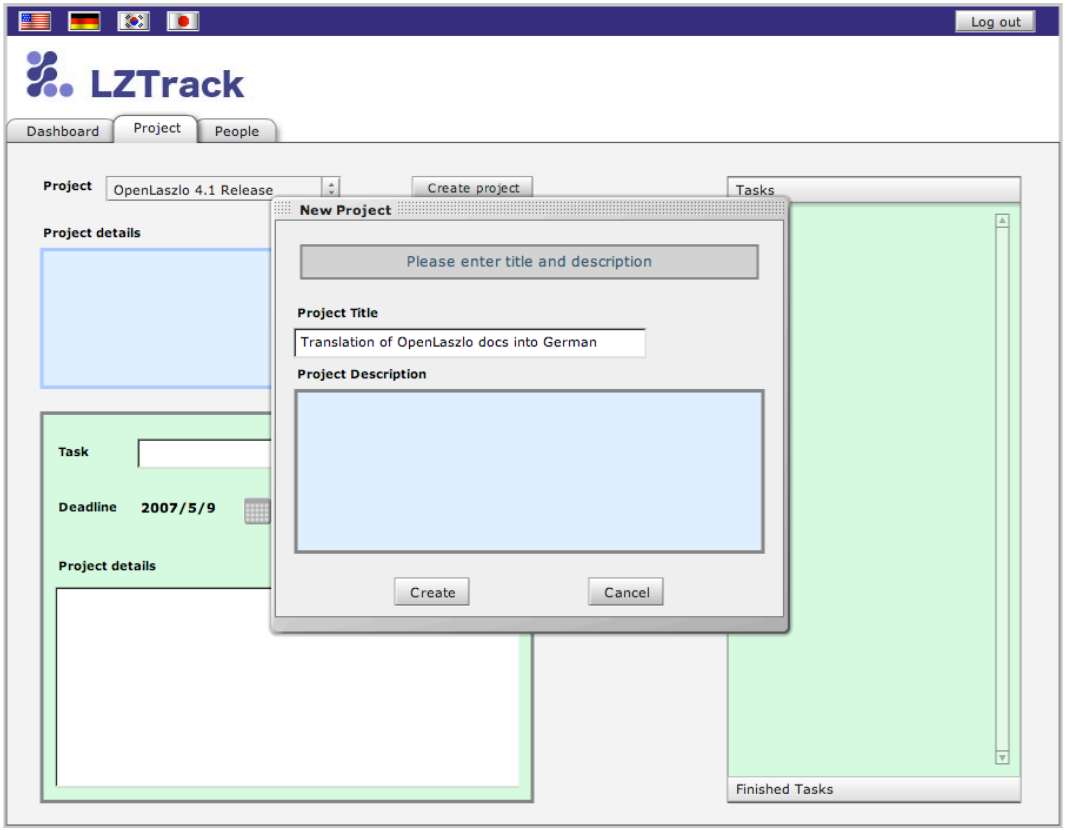


Anatomy of LZProject login

The diagram shows 12 steps that are executed in a successful login.

1. A login and password are entered and the submit button is clicked. The client side validation is started by calling the validate() method of the login form.
2. If there are valid entries the service connector, a class responsible for managing the communication between forms and the corresponding backend service classes starts a login service call.

3. The login action for the backend service is triggered.
4. The backend service collects the values from the form, adds them as GET parameters to the request and calls doRequest() on the dataset.
5. The dataset sends the request out to the login web service.
6. The server side processing of the request requires several steps. The request passes throught the "i18n" filter for localization, where the best matching locale is added to the HTTP session. The security filter checks if the request is trying to access a protected web service.
7. The REST controller instantiates a User object and calls the User.login() method. Within the login() method, an SQL query is executed on the table USERS. The data is correct, the user is authenticated and the user object is saved in the session.
8. The REST controller forwards to lzproject/webservice/UserLoginReponse.jsp to generate the XML response. Within the JSP the locale is retrieved from the session (which has been put there by the I18NFilter) and the corresponding resource bundle (for example, LzTrackMessages_de.properties, for German), is loaded. The localized messages are inserted into the XML result file.
9. The dataset parses the XML file and loads the data into the client.
10. The datapointer used for monitoring a successful result receives an ondata event. In turn the handleSuccess() method of the BackendService is called.
11. The handleSuccess() method passes the localized message to the ServiceConnector.
12. The ServiceConnector puts the received message into the text field.



Response message when creating a new project – missing project description

The combination of datasets, datapointers, events triggered by the arrival of a new XML file calls for a code structure which makes the interplay of the different code pieces clearly visible and readable. LZProject bundles the sequence of method calls, request and events in a declarative way to provide superior code readability. Strict naming conventions are used to ensure maintainability of the code base.

Naming conventions used in LZProject

A few naming conventions are used to simplify the interplay of the application pieces. Here's a list of them:

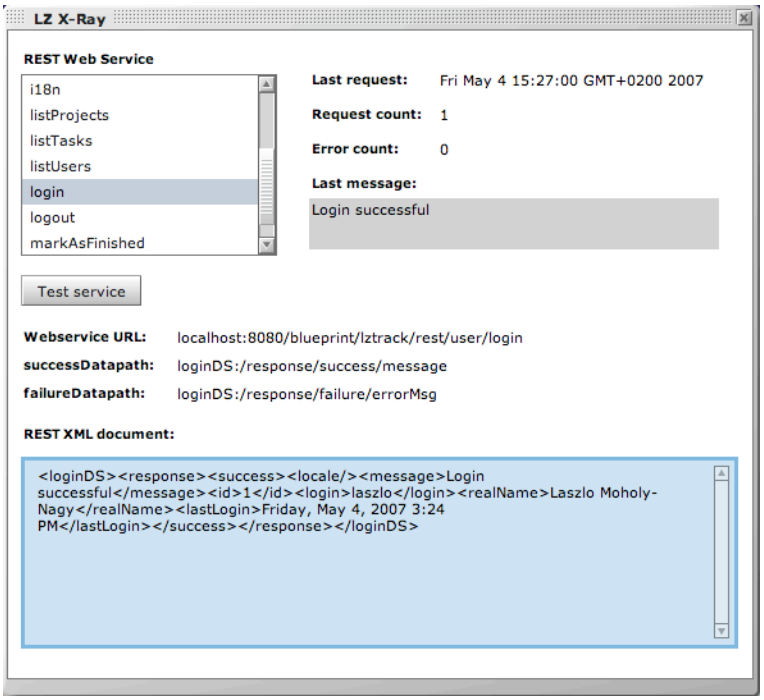
- BackendService class: Every instance of the class has a name attribute which is chosen based upon the following schema:
    - The verb describing the action which this service represents
    - An optional noun describing the object of the operation

  Examples: "login", "listProjects", "createUser"
- Datasets: The corresponding dataset for every service are named by attaching "DS" to the BackendService name.
  Examples: "loginDS", "listProjectsDS", "createUserDS"
- ServiceConnector: The service connector instances carry the name of the BackendService instance they refer to plus the string "Conn".
  Examples: "loginConn", "createUserConn"

With these conventions, when you look at a service connector contained in a form it is immediately clear to which backend service it connects. The same is true for the datasets. On top of that it is easily possible to come up with an in-application testing tool for web services. Just open the context menu anywhere in LZProject and you'll see a menu entry called LZ X-Ray. Select it an a window will open up showing a list of all backend service instances.

LZ X-Ray

The structure of LZProject makes it easy to use the information contained in the backend service instances to generate a list of web services and test them out of the application. LZ X-Ray provides you with a simple interface to select a service instance and see statistics on it.



LZ X-Ray lets you easily test your web services

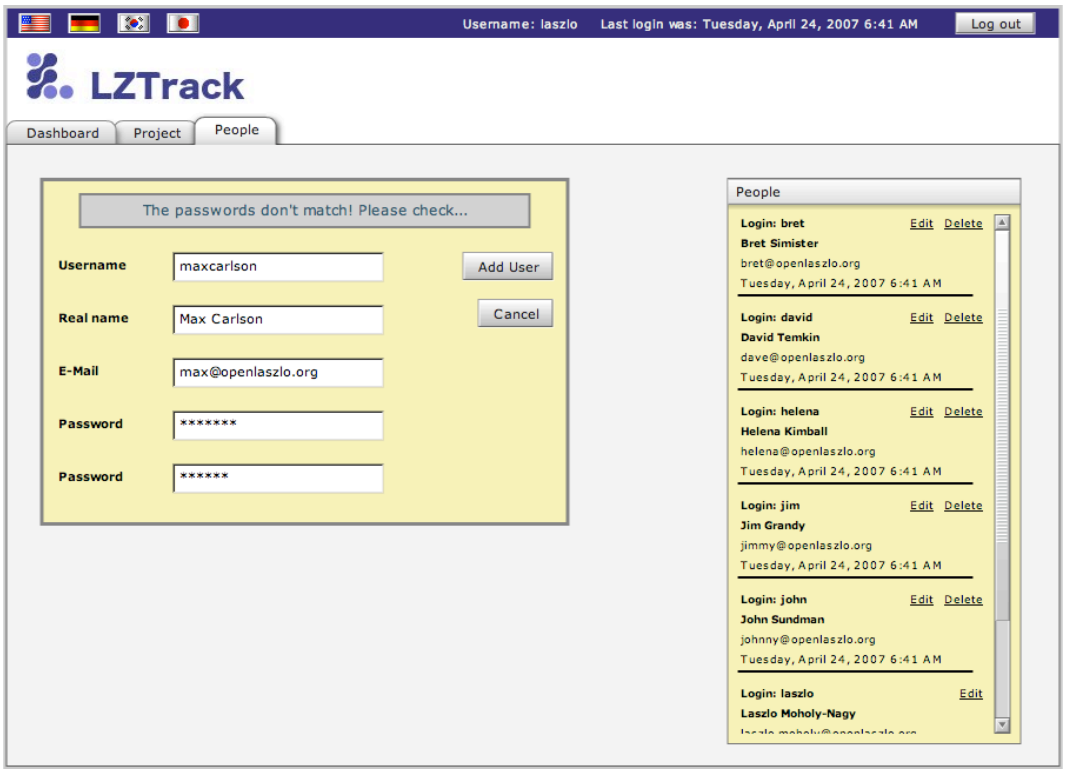
You can access the following information with LZ X-Ray:

- The timestamp of the last request made to the service
- The total number of request to the service
- An error count, showing the total number of errors for this request
- The URL the web service is using
- The datapath values for success and failure
- The XML document returned by the web service

You can actively start a request to the web service selected and even watch the changes in real time when request are started out of the normal LZProject interface. LZ X-Ray is just a simple example of the possibilities for building client-side application testing tools.

The View

Now that we have examined the general structure of LZProject, let's take a look at the implementation. The view area of the application contains all the visual elements of the interface.



Creating a user, validation failed

As an example we'll take the form for creating a new user inside LZProject. It's a pretty simple form containing a text field for user name, real name, e-mail and two password fields.

```
<class name="UserCreateEditForm" extends="BorderedBox" width="$once{this.content.width+26}"
     height="${this.content.height+26}" borderColor="#777777" bgcolor="#f8f3aa" inset="10">

         ...

   <!-- DISPPLAY AREA -->
  <view name="content" width="$once{classroot.innerWidth}">

     <simplelayout axis="y" spacing="5" />

     <BorderedBox name="messageBox" align="center" y="60" height="0"
         width="${parent.width-40}" borderColor="#777777"
              bgcolor="#cccccc" borderSize="2" visible="true" clip="true">
         <method name="setText" args="t">
             if (t.length > 0) {
              this.textField.setText(t);
              this.setAttribute("visible", true);
              this.animate('height', 30, 400);
             } else {
              this.animate('height', 0, 400);
             }
         </method>
         <text name="textField" align="center" fgcolor="#2c4c62" fontsize="12" />
     </BorderedBox>

     <view height="10" />

     <view name="buttonYRef" height="40">
         <text y="3" fontsize="10" fontstyle="bold"
             datapath="i18nDS:/app/login/user/text()" />
         <edittext name="login" x="100" width="180" />
     </view>
```

```
      <view height="40">
          <text y="3" fontsize="10" fontstyle="bold"
              datapath="i18nDS:/app/main/peopleTab/realName/text()" />
          <edittext name="realName" x="100" width="180" />
      </view>

      <view height="40">
          <text y="3" fontsize="10" fontstyle="bold"
              datapath="i18nDS:/app/main/peopleTab/email/text()" />
          <edittext name="email" x="100" width="180"/>
      </view>

      <view height="40">
          <text y="3" fontsize="10" fontstyle="bold"
              datapath="i18nDS:/app/login/password/text()" />
          <edittext name="password1" x="100" width="180" password="true"/>
      </view>

      <view height="40">
          <text y="3" fontsize="10" fontstyle="bold"
                datapath="i18nDS:/app/login/password/text()" />
          <edittext name="password2" x="100" width="180" password="true"/>
      </view>

      <state name="createState" pooling="true" apply="true">
          <button name="createButton" y="${parent.buttonYRef.y}"
              datapath="i18nDS:/app/main/peopleTab/newUser/text()"
              align="right" options="ignorelayout" onclick="classroot.createUser()" />
      </state>

      <state name="updateState" pooling="true">
          <button name="editButton" y="${parent.buttonYRef.y}"
              datapath="i18nDS:/app/main/peopleTab/editUser/text()"
          align="right" options="ignorelayout" onclick="classroot.updateUser()" />
      </state>

      <button name="cancelButton" y="${parent.buttonYRef.y+40}"
datapath="i18nDS:/app/main/peopleTab/cancel/text()"
          align="right" options="ignorelayout" onclick="classroot.clear()" />

   </view>
</class>
```

The form for entering new users or updating existent information

The form contains the edittext fields for the user input and three buttons: one for creating a new user, one for updating the information of an existing user and a button to cancel the operation and reset the form.

The *create* and the *update* buttons are each put inside a <state> tag, as we only need either one of them. In LZX, states provide a concise and powerful way to control procedural behavior with a simple declarative syntax. The message box for displaying any information to the user will only be visible on user interaction. The form contains methods for resetting the form, simple validation of data and for loading the data of an existing user into the form.

```
<!-- METHOD SECTION -->
   <method name="clear">
       searchSubnodes("name", "login").setText('');
       searchSubnodes("name", "realName").setText('');
       searchSubnodes("name", "email").setText('');
       searchSubnodes("name", "password1").setText('');
       searchSubnodes("name", "password2").setText('');
       parent.user = null;
       content.createState.apply();
       content.updateState.remove();
```

```
</method>

<method name="collectValues">
    var values = new Array();
    values['login'] = parent.searchSubnodes("name", "login").getText();
    values['password'] = parent.searchSubnodes("name", "password1").getText();
    values['email'] = parent.searchSubnodes("name", "email").getText();
    values['realname'] = parent.searchSubnodes("name", "realName").getText();
    if (this.user != null) {
        values['id'] = user.userId;
    }
    return values;
</method>

<method name="createUser">
    createUserConn.callService();
</method>

<method name="loadUser" args="user">
    this.user = user;
    parent.searchSubnodes("name", "login").setText(user.login);
    parent.searchSubnodes("name", "realName").setText(user.realName);
    parent.searchSubnodes("name", "email").setText(user.email);
    content.createState.remove();
    content.updateState.apply();
</method>

<method name="updateUser">
    updateUserConn.callService();
</method>

<method name="validate" args="conn">
    var errorMessage = null;
    var name = searchSubnodes("name", "login").getText();
    var pass1 = searchSubnodes("name", "password1").getText();
    var pass2 = searchSubnodes("name", "password2").getText();
    var email = searchSubnodes("name", "email").getText();
    var realName = searchSubnodes("name", "realName").getText();
    if (name.length == 0 || email.length == 0 ||
         realName.length == 0 || pass1.length == 0) {
        errorMessage = this.i18n.missingParameters;
    } else if (pass1 != pass2) {
        errorMessage = this.i18n.passwordMissmatch;
    }
    if (errorMessage != null) {
        conn.showValidatonError(errorMessage);
        return false;
    } else {
        return true;
    }
</method>
```

Methods inside the form containing the logic for validation and data collection

ServiceConnector

Service connectors are used to manage the application flow between the view area and the backend services. They act as a middleware between application layers, taking care of:

- Starting client side validation when the form contains a validate() method.
- Collecting the data entered into the form and passing it onto the backend service.
- Passing response messages to the forms.

Service connectors are placed inside the view element which acts as the form to the backend service. Once again, the name “createUserConn” marks that this connector connects to the “createUser” backend service.

```
<!-- Connector used for createUser action -->
<ServiceConnector name="createUserConn" form="$once{parent}">
  <method name="handleResult" args="message">
     form.searchSubnodes("name", "messageBox").setText(message);
  </method>
</ServiceConnector>

<!-- Connector used for updateUser action -->
<ServiceConnector name="updateUserConn" form="$once{parent}">
  <method name="handleResult" args="message">
     form.searchSubnodes("name", "messageBox").setText(message);
  </method>
</ServiceConnector>
```

The service connectors inside the UserCreateEditForm class

Backend service class

The BackendService class acts as a controller managing communication between the view area and the dataset used for the requests to the backend. It contains the datapointers used for tracking a web service call's success or failure, which in turn are used to call the handleError(), handleFailure() and handleSuccess() methods.

- **handleSuccess** – as the name suggests, this method is called when the web service returns an XML file signaling a success of the desired operation. The handleSuccess() method is called when the data of the node, the XPath expression in the successDatapath attribute points to, changes.
- **handleFailure** – failure here defines a server side logical failure of an operation, like trying to delete a non-existent user, or login in with a wrong password. In such a case a localized message will be returned as part of the XML file to provide more information. The handleFailure() method is called when the data of the node, the XPath expression in the failureDatapath attribute points to, changes.
- **handleError** – this is only called when the HTTP request itself fails. Requesting an invalid URL on the server would lead to a call of handleError().

Lets have a look at the BackendService instance used for the initial login:

```
<!-- Login -->
<dataset name="loginDS" src="rest/user/login" type="http" request="false" proxied="false" />
<BackendService name="login"
      successDatapath="loginDS:/response/success/message"
      failureDatapath="loginDS:/response/failure/errorMsg"
      clearDelay="2000">
  <handler name="ontrigger" args="obj">
   Debug.write('login triggered');
   this._connector.form.disable();
   this.prepareParams(obj.collectValues());
  </handler>
  <method name="handleFailure" args="errorMsg">
   Debug.write('Error message: ', errorMsg);
   // either client side validation or server side
   // authentication failed. Display error message
   this._connector.form.showError(errorMsg);
  </method>
  <method name="handleSuccess" args="msg, p">
   // Extract the login for this user
   var dataPointer = p.ownerDocument.getPointer();
   dataPointer.setXPath('/response/success/login');
```

```
    // store the login name as canvas attribute for use within
    // logout process
    canvas.login= dataPointer.getNodeText();
    this._connector.handleResult(msg);
        // Now load all data for the application
        canvas.services.listProjects.startRequest();
        canvas.services.listUsers.startRequest();
        canvas.services.listTasks.startRequest()
        canvas.setAttribute('loggedIn', true);
    </method>
    <method name="clearMessage">
    if (canvas.loggedIn) {
        // user authenticated, show main screen
        this._connector.form.setAttribute('visible', false);
        this._connector.form.reset();
        app.setAttribute('visible', true);
    } else {
        // clear the error message and show the default
        // login message
        loginScreen.showMessage('loginMessage');
        loginScreen.enable();
    }
    </method>
</BackendService>
```

BackendService.lzx (lzproject/modules/BackendService.lzx)

The clearMessage() method in the BackendService class is called two seconds after handleFailure() or handleSuccess() has been called. The existence of a clearMessage() method is optional; the method will only be called if it exists. The delay can be adjusted by the clearDelay attribute of the BackendService class. This method is used to clear any messages displayed in forms after the specified amount of time. To understand the necessity for this functionality let's analyze the login procedure at application start-up.

1. Login and password are entered
2. The form is validated on the client side
3. In case of validation (login and password contain values) a request is sent to the login web service
4. Login and password are correct, a positive result is sent back to LZProject
5. The handleSuccess() method is called, a message will be displayed signaling that the login attempt has been successful.
6. After two seconds the clearMessage() method is called. The login screen will be made invisible and the main application screen will be shown.

The login screen could be immediately closed after a successful login without giving any feedback on the state of the login process. Showing a message confirming the successful login makes the application look friendlier. While LZProject is was not made to show the capabilities of OpenLaszlo to create fancy animations and state transitions, LZX was created with exactly that in mind. Be sure to check out the many other OpenLaszlo demo applications to get a feel for the kind of dramatic user interfaces that LZX allows you to create.

Multi-step processes in OpenLaszlo applications including state-changes and multiple animations are used to create what we call a *Cinematic User Experience*. Laszlo Webtop, a newly released commercial product by Laszlo Systems, is designed to support the delivery of integrated suites of OpenLaszlo RIAs.
Java developers looking for an optimized high-level framework for creating rich Internet applications should evaluate Laszlo Webtop as a platform for their projects.

Localization

For the localization of LZProject a powerful combination of Java server side coding and LZX constraints are used. On the server side all localized messages are stored in resource bundles. A servlet filter (*org.openlaszlo.lzproject.I18NFilter*) is used to extract the preferred language passed by the browser to the server as part of the HTTP headers. The filter then checks for the best matching resource bundle available, based on the language code (such as "en" for English, or "ko" for Korean). That language code is stored in the HTTP session and later retrieved by the JSP generating the XML output or inside the business classes for projects, users and tasks.

Here's an extract of lzproject/webservice/ApplicationXML.jsp showing the locale being retrieved from the HTTP session and output of localized messages through the Apache I18N taglib.

```
<?xml version="1.0" encoding="utf-8"?>
<%@page pageEncoding="UTF-8" contentType="text/xml" %>
<%@page import="java.util.*" %>
<%@page import="org.openlaszlo.lzproject.model.*" %>
<%@ taglib uri="http://jakarta.apache.org/taglibs/i18n-1.0" prefix="i18n" %>
<%
Locale myLocale = (Locale) session.getAttribute("myLocale");
User user = (User) session.getAttribute("user");
%>
<i18n:bundle baseName="de.openlaszlo.lzproject.LzTrackMessages"id="bundle" locale="<%= myLocale
%>"/>
<app>
<%
if (user != null) {
%>
   <user>
      <login><%= user.getLogin()%></login>
      <realName><%= user.getRealName()%></realName>
      <lastLogin><i18n:formatDateTime value="<%=user.getLastLogin()%>" dateStyle="full"
timeStyle="short" locale="<%=myLocale%>"/></lastLogin>
   </user>
<%
}
%>
...
```

ApplicationXML.jsp used for localization

On the client side this file is used for updating all text inside the application, which is not saved in the SQL database. The login screen is a good example of localization. If the text is used as a label on a normal input field title, it is sufficient to use a datapath tag on the text field to have it auto-updated with every load of the XML file.

```
<class name="LoginScreen" extends="BorderedBox" bgcolor="$once{COLORS.LOGIN_BG}"
     width="500" height="310" borderColor="$once{COLORS.LOGIN_BORDER}">
```

```
    <!-- i18n message container -->
    <node name="i18n" datapath="i18nDS:/app/login">
        <attribute name="loginSuccess" type="string" value="$path{'success/text()'}" />
        <attribute name="loginMessage" type="string" value="$path{'loginMessage/text()'}" />
        <attribute name="missingData" type="string" value="$path{'missingData/text()'}" />
    </node>

    ...

    <!-- DISPPLAY AREA -->
    <BorderedBox name="messageBox" x="20" y="60" height="32" width="${parent.width-40}"
         borderColor="#6eaed8" bgcolor="#c6dff0">
        <method name="setText" args="t">
            this.textField.setText(t);
        </method>
        <text name="textField" align="center" fgcolor="#2c4c62" fontsize="14"
            datapath="i18nDS:/app/login/loginMessage/text()" />
    </BorderedBox>

    <text x="20" y="107" fontsize="14" datapath="i18nDS:/app/login/user/text()" />
    <edittext name="loginName" x="110" y="105" width="310" fontsize="14" height="30"
         doesenter="true" maxlength="8">laszlo
        <method name="doEnterDown">
            parent.commit.onclick.sendEvent()
        </method>
    </edittext>

    <text x="20" y="145" fontsize="14" datapath="i18nDS:/app/login/password/text()" />
    <edittext name="password" x="110" y="143" width="160" fontsize="14" height="30"
        password="true" doesenter="true">
        <method name="doEnterDown">
            parent.commit.onclick.sendEvent()
        </method>
    </edittext>

    <button name="commit" x="110" y="190" fontsize="14"
        onclick="classroot.loginConn.callService()"
        datapath="i18nDS:/app/login/loginButton/text()"/>

</class>
```

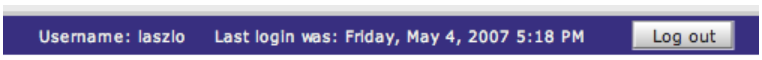
LoginScreen.lzx with localized text fields and error messages

For the client side validation a different approach has been chosen. To avoid constant requests to the server, all necessary messages are stored as attributes of a node called *i18n*. If the client side validation fails, the messages are locally available and just have to be displayed.

```
    <!-- i18n message container -->
    <node name="i18n" datapath="i18nDS:/app/login">
        <attribute name="loginSuccess" type="string" value="$path{'success/text()'}" />
        <attribute name="loginMessage" type="string" value="$path{'loginMessage/text()'}" />
        <attribute name="missingData" type="string" value="$path{'missingData/text()'}" />
    </node>
```

All messages, which might be needed for client side validation are stored here

In some cases, where it's necessary to perform modifications of the text passed into the application through the XML file, an event handler is added to the text element. The text field's datapath doesn't directly point to a text node within the dataset. The text which shall be retrieved from the dataset is instead stored in an attribute added dynamically to the text field instance. Within the handler the text changes are handled through JavaScript code. The information in the application header on the last login time and the login name is generated as follows:




```
<text fgcolor="#eeeeee" fontsize="10" datapath="user/login">
  <attribute name="i18nUser" type="string"
         value="$path{'../../login/user/text()'}" />
  <handler name="ondata" args="p">
      if (p != null) {
          this.setText(this.i18nUser + ": " + p.childNodes[0]);
      }
  </handler>
</text>
<text fgcolor="#eeeeee" fontsize="10" datapath="user/lastLogin" resize="true">
  <attribute name="i18nLastLogin" type="string"
      value="$path{'../../main/lastLogin/text()'}" />
  <handler name="ondata" args="p">
      if (p != null) {
          this.setText(this.i18nLastLogin + ": " + p.childNodes[0]);
      }
  </handler>
</text>
```

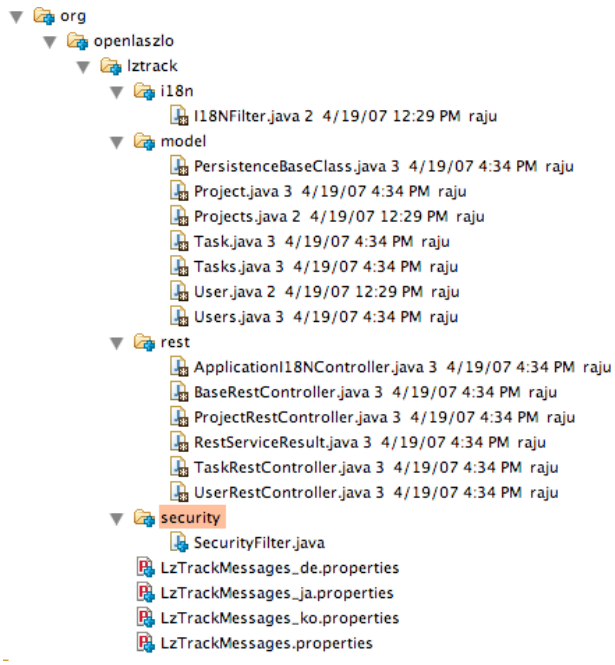
Login name and time of last login in the application header

The Java backend

While implementing the Java backend for LZProject, one of our goals was to not use any of the well-established Java web frameworks like Spring, Struts or any ORM technology, as that would take the focus away from integration of an OpenLaszlo application with Java. You'll find the main ingredients of a Java web application here, only simplified to highlight the technical concept of integrating OpenLaszlo with Java.

The backend for LZProject consists of a number of files:

- A Servlet Filter that acts as a security filter
- A Servlet Filter for i18n, automatically selecting the best locale based on the existing resource bundles
- Java Servlets acting as REST controllers for the web service request
- Java POJOs acting as our business model, making JDCB calls to the Apache Derby database
- A number of resource bundles containing the localization messages
- JSP files for generation of XML result pages



The files of the Java backend for LZProject

Let's look into the different parts of the backend to get a better understanding of the inner functioning.

The REST controller servlets

The REST controller servlets are the central contact points for the web service requests sent out of the LZProject application. The controller extracts the REST command from the URL and calls the corresponding methods of the model classes. The request parameters – if applicable – are retrieved and the setter methods on the model classes are called. Each controller maintains a list of JSP pages to which the request is forwarded, depending on the result of the operation.

```
// Check for all known commands
if (command.equals("login")) {
  // Remove user object from session
  session.removeAttribute("user");
  // Start the login process
  log.debug("User login: " + request.getParameter("login"));
  log.debug("User password: " + request.getParameter("password"));
  user.setLogin((String) request.getParameter("login"));
  if (request.getParameter("password") != null) {
     user.setPassword((String) request.getParameter("password"));
  } else {
     // To avoid null reference problems set a dummy password
     user.setPassword("");
  }
  result = user.login(session, result);
  dispatcher = getServletContext().getRequestDispatcher(USER_RESPONSE_JSP);
} else if (command.equals("logout")) {
  // Start the logout process
  if (session.getAttribute("user") != null) {
     user = (User) session.getAttribute("user");
  }
  result = user.logout(session, result);
```

```
   if (!result.isError()) {
      dispatcher = getServletContext().getRequestDispatcher(USER_RESPONSE_JSP);

   }
} else if (command.equals("create")) {...
```

The doPost() method of the UserRestController class – checking for known REST commands.

Every call to one of the methods of the model classes, like User.login(), returns a RestServiceResult object containing localized messages, error status and in case of a SQL error the SQL state and exception message. The *RestServiceResult* is added to the request and then passed to the JSP.

```
<?xml version="1.0" encoding="utf-8"?>
<%@page pageEncoding="UTF-8" contentType="text/xml" %>
<%@page import="java.util.*" %>
<%@page import="org.openlaszlo.lzproject.model.*" %>
<%@page import="org.openlaszlo.lzproject.rest.*" %>
<%@ taglib uri="http://jakarta.apache.org/taglibs/i18n-1.0" prefix="i18n" %>
<%
Locale myLocale = (Locale) session.getAttribute("myLocale");
%>
<i18n:bundle baseName="org.openlaszlo.lzproject.LzTrackMessages"id="bundle"
locale="<%=myLocale%>"/>
<response>
<%
User user = (User) session.getAttribute("user");
RestServiceResult result = (RestServiceResult) request.getAttribute("result");
if (! result.isError()) {
%>
<success>
<locale><%=myLocale%></locale>
<message><%= result.getMessage() %></message>
<id><%= user.getId()%></id>
<login><%= user.getLogin()%></login>
<realName><%= user.getRealName()%></realName>
<lastLogin><i18n:formatDateTime value="<%=user.getLastLogin()%>" dateStyle="full"
timeStyle="short" locale="<%=myLocale%>"/></lastLogin>
</success>
<% } else { %>
<failure>
<errorMsg><%= result.getMessage() %></errorMsg>
<%if (result.getSqlState() != null && result.getSqlState().length() > 0) { %>
<errorNo><%= result.getSqlState() %></errorNo>
<%    } %>
</failure>
<% } %>
</response>
```

Use of of Jakarta I18N tag lib in UserLoginResponse.jsp

While the *RestServiceResult* object already contains localized messages, there is more localization within the JSPs, such as formatting date or time strings. That localization could be done through Java code inside the JSP. The Jakartka I18N tag lib, which was developed for the Jakarta Struts framework, shows a more elegant, tag based approached to localization within a JSP.

The model

The model classes (User, Project, Task) are POJOs (plain old Java objects) with some code to connect to the Derby DB trough JDBC. All the model classes extend PersistenceBaseClass, which provides the code necessary to load the Apache Derby DB.

```
protected void dbConnect() {
  String dbPath = this.realPath + "/" + this.dbName;
  try {
     logger.debug("Loading Derby embedded driver");
     Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
     conn = DriverManager.getConnection("jdbc:derby:" + dbPath);
     conn.setAutoCommit(true);
     logger.info("Connection to Derby established");
  } catch (Exception e) {
     logger.error("Error loading Derby DB " + this.dbName
           + " from path " + dbPath + "!");
     logger.error(e.getLocalizedMessage());
  }
}
```

PersistenceBaseClass – opening a connection to Apache Derby DB

All other model classes contain methods for the REST web services offered to the LZProject application, like the *User.create()* method shown here. A request to lzproject/rest/user/create results in a call to the User.create() method by the *UserRestController*.

```
public RestServiceResult create(RestServiceResult result) {
  Statement stmt;
  log.debug("Executing command User.create()");
  if (login != null && realName != null && login.length() > 0
        && realName.length() > 0 && email != null && email.length() > 0) {
     this.dbConnect();
     try {
        stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
            ResultSet.CONCUR_READ_ONLY);
        stmt.executeUpdate("INSERT INTO USERS (LOGIN, REAL_NAME, PASS, EMAIL,
LAST_LOGIN) "
                           + "VALUES ('"+ this.login + "','"
                           + this.realName + "','"
                           + this.password + "', '"
                           + this.email + "', CURRENT_TIMESTAMP)");
        stmt.close();
        Object[] args = { this.realName, this.login };
        result.setMessage(MessageFormat.format(bundle
            .getString("user.create.success"), args));
     } catch (SQLException e) {
        // 23505: unique key violation
        if (result.getSqlState().equals("23505")) {
           Object[] args = { this.login };
           result.setMessage(MessageFormat.format(bundle
                 .getString("user.create.userAlreadyExists"), args));
        } else {
           // Generate a debug error message and add the error to the
           // result set.
           log.error("Error while creating user");
           log.error(e.getLocalizedMessage());
           log.error(e.getStackTrace());
        }
     }
```

```
        if (this.conn != null) {
            this.dbDisconnect();
        }
    } else {
        log.warn("User could not be created! Missing parameters...");
        result.setError(true);
        result.setMessage(bundle.getString("user.create.missingParameters"));
    }
    return result;
}
```

Method for creating a new user in class User

Apache Derby, an embedded database written in Java, can be used like any standard JDBC database. In case of an SQL exception the SQL state will be saved to the *RestServiceResult* object and passed to the response JSP file.

The database – Apache Derby

Apache Derby, an Apache DB subproject, is an open source relational database implemented entirely in Java and available under the Apache License, Version 2.0. Some key advantages of this choice include:

- Derby has a small footprint -- about 2 megabytes for the base engine and embedded JDBC driver.
- Derby is based on the Java, JDBC, and SQL standards.
- Derby provides an embedded JDBC driver that lets you embed Derby in any Java-based solution.
- Derby also supports the more familiar client/server mode with the Derby Network Client JDBC driver and Derby Network Server.
- Derby is easy to install, deploy, and use.

We used Derby for LZProject for exactly those reasons. Anyone interested in downloading and running LZProject on their own system does not have to install a MySQL database locally on their machine. Just running a Java servlet container like Tomcat or Jetty and dropping the SOLO version of LZProject into it is enough.

The Derby database files inside LZProject can be found inside *WEB-INF/lzprojectdb* folder. There are two additional JAR files in the *WEB-INF/lib* folder:

- derby.jar Java archive
- derbytools.jar Java archive

Note that although we used Apache Derby for LZProject, that doesn't mean it is the recommended database for building backend systems for OpenLaszlo applications. As an embedded database being loaded for every request made to the backend the performance will not be nearly as good as the performance of standard SQL database servers like MySQL.

Localization of the application

The localization of LZProject is based on the interaction of a number of different components.

- The I18NFilter class handles the selection of the best matching locale.
- Resource bundle files contain the localized messages, situated in the WEB-INF/classes/org/openlaszlo/lzproject folder. The base name for the resource files is *LzTrackMessages.properties*.
- Code within the REST controller servlets, the model classes and the JSP pages to access the resource bundles and insert the messages into Log4J output and the response JSP pages.

Let's take a more detailed look at the code used here.

I18NFilter

Within the doFilter() method of the I18N Filter the following actions are executed:

- Check if a HTTP request paramater with the name *language* is part of the request.
- Lookup a possible locale object in the HTTP session
- If there was no locale in the session, call the findBundle() method to automatically determine the best resource bundle to be used, or
- If a language code was passed to the filter, look for a resource bundle matching this language by calling the setNewLocale() method. If a corresponding resource bundle can be found, store the new locale in the session.

```
// A locale can be manually selected by passing a request paramater
// with the variable name language to the filter containing a
// ISO-639 two letter country code
String languageCode = httpRequest.getParameter("language");

// Check if we have a locale in the session
this.locale = (Locale) session.getAttribute("myLocale");

// If no locale is defined in the session, automatically find the best
// match
if (this.locale == null) {
  this.findBundle();
}

// Check if user manually selected a new language for the app
if (languageCode != null && languageCode.length() == 2
      && languageCode != this.locale.getLanguage()) {
  this.setNewLocale(languageCode);
}
filterChain.doFilter(request, response);
```

I18NFilter – process of selecting the right resource bundle.

Within the REST servlet controllers or the JSP pages the locale is then read out of the HTTP session and used to load the resource bundle.

```
protected ResourceBundle loadResourceBundle(HttpServletRequest request) {
  // Extract locale from session to load resource bundle
  HttpSession session = request.getSession();
```

```
  Locale locale = (Locale) session.getAttribute("myLocale");
  // Load the message bundle for this locale and pass it to the user
  // object
  return ResourceBundle.getBundle(BaseRestController.RESOURCE_BASE_NAME,
        locale);
}
```

Method loadResourceBundle() of the BaseRestController class.

The resource bundle could just be saved in the session. As soon as the resource property files become larger and the session is replicated across multiple servers it is not advisable to store the resource bundle in the session.

Within the REST controller the resource bundle is then used to extract the messages from the resource property file. The example shows how the generation of a response for a deleteTaskById() call.

```
if (lines == 0) {
  Object[] args = { new Integer(this.id) };
  result.setMessage(MessageFormat.format(bundle
        .getString("task.unknownTaskId"), args));
} else {
  Object[] args = { new Integer(this.id) };
  result.setMessage(MessageFormat.format(bundle
        .getString("task.delete.success"), args));
}
```

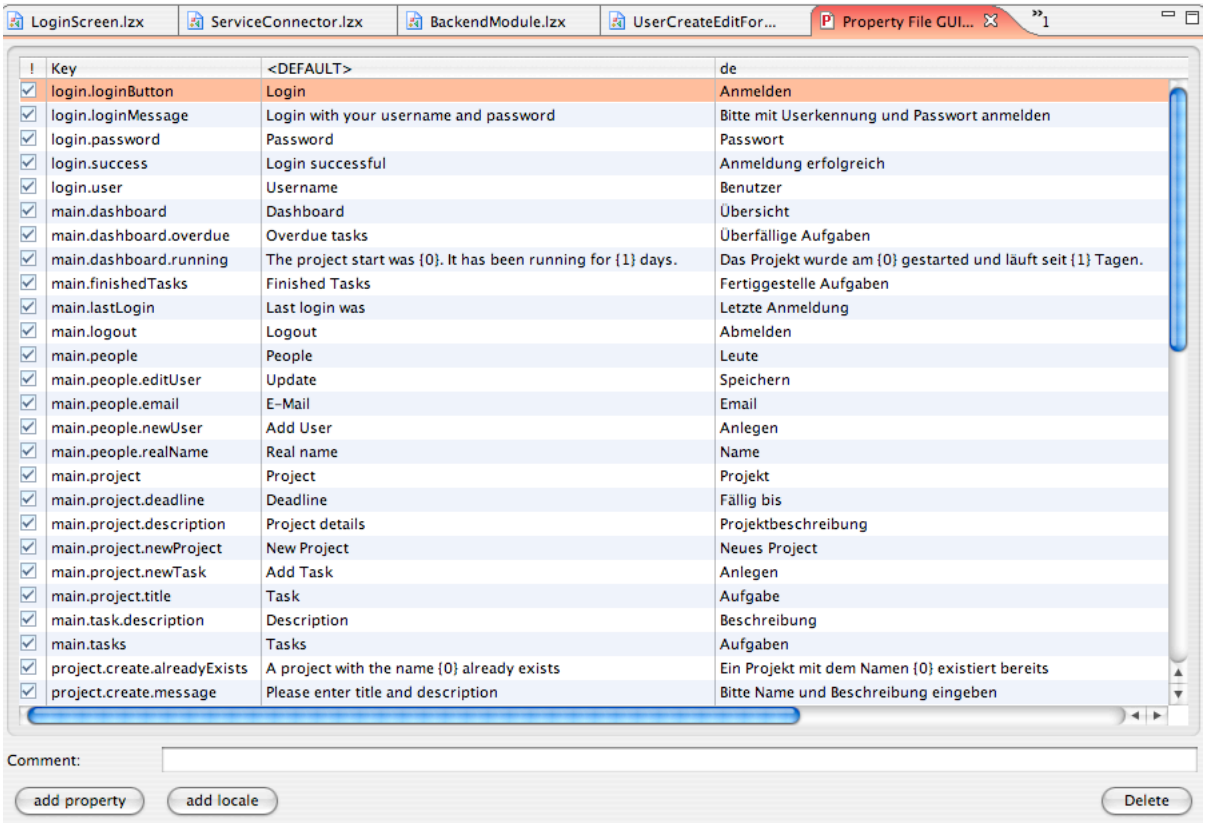
Method loadResourceBundle() of the BaseRestController class.

Keys denote messages like *task.unknownTaskId*, which in turn correspond to the keys used in the resource property files.

```
task.markAsFinished.success=Task with ID  {0} marked as finished
task.unknownTaskId=Unknown task ID {0}
task.delete.missingParameter=Missing parameters
task.delete.success=Task with ID {0} deleted
```

Excerpt from the English resource property file showing the messages task management.

Messages can be comfortably managed through tools like Property File GUI Editor, a free Eclipse plug-in.

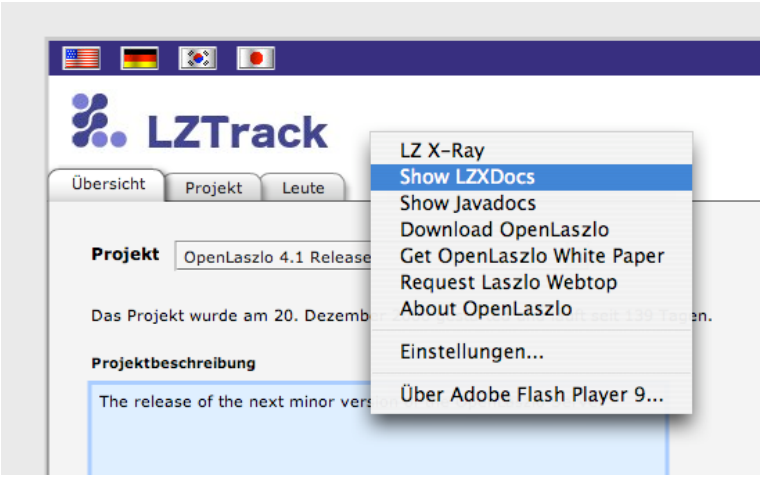


!	Key	<DEFAULT>	de
☑	login.loginButton	Login	Anmelden
☑	login.loginMessage	Login with your username and password	Bitte mit Userkennung und Passwort anmelden
☑	login.password	Password	Passwort
☑	login.success	Login successful	Anmeldung erfolgreich
☑	login.user	Username	Benutzer
☑	main.dashboard	Dashboard	Übersicht
☑	main.dashboard.overdue	Overdue tasks	Überfällige Aufgaben
☑	main.dashboard.running	The project start was {0}. It has been running for {1} days.	Das Projekt wurde am {0} gestarted und läuft seit {1} Tagen.
☑	main.finishedTasks	Finished Tasks	Fertiggestelle Aufgaben
☑	main.lastLogin	Last login was	Letzte Anmeldung
☑	main.logout	Logout	Abmelden
☑	main.people	People	Leute
☑	main.people.editUser	Update	Speichern
☑	main.people.email	E-Mail	Email
☑	main.people.newUser	Add User	Anlegen
☑	main.people.realName	Real name	Name
☑	main.project	Project	Projekt
☑	main.project.deadline	Deadline	Fällig bis
☑	main.project.description	Project details	Projektbeschreibung
☑	main.project.newProject	New Project	Neues Project
☑	main.project.newTask	Add Task	Anlegen
☑	main.project.title	Task	Aufgabe
☑	main.task.description	Description	Beschreibung
☑	main.tasks	Tasks	Aufgaben
☑	project.create.alreadyExists	A project with the name {0} already exists	Ein Projekt mit dem Namen {0} existiert bereits
☑	project.create.message	Please enter title and description	Bitte Name und Beschreibung eingeben

Editing localization messages inside Eclipse with Property File GUI Editor

Documenation – Javadocs and LZXDocs

Similar to the Javadoc tool for Java there is an LZXDoc tool for generating LZXdocs, Javadoc like HTML documentation pages. The LZXDoc tool can be found in the OpenLaszlo.org Wiki. The LZProject application contains the Javadoc documentation as well as the LZDocs. You'll find them in the context menu of LZProject, just select either one of them to open the documentation.



Open the Javadocs or the LZXDocs through the context menu

Development environment

What is the best environment for OpenLaszlo development? While it's practically possible to use any kind of text editor to code LZX, there are a number of tools which are particularly useful. Eclipse still is the de-facto standard for Java development and a good choice for LZX coding. While there are no visual tools available right now for interface layout, Spket IDE supports LZX code completion and is available as an Eclipse plug-in as well as a stand-alone application.

The following technologies and tools were used in the development process of the LZProject application.

- LZX and Java coding
    - Eclipse IDE 3.2
    - Spket Eclipse plug-in
    - Subclipse plug-in for Subversion
    - Eclipse plug-in for Derby (derby_core and derby_ui)
    - Property File GUI Editor plug-in
    - Sysdeo Tomcat plug-in
    - Jakarta Taglib I18N
- Build tools
    - Apache Ant
    - XDoclet
    - LZXDoc
- Java Servlet container
    - Apache Tomcat 5.0.28
- Embedded SQL database
    - Apache Derby, version 10.2.2.0