

Create a mortgage application with EGL Rich UI

Contents

Create a mortgage application with EGL

Rich UI 1

Introduction	1
Lesson 1: Plan the mortgage application	4
Sketch the interface	4
Identify the application structure	4
Lesson checkpoint	5
Lesson 2: Set up the workspace	5
Create an EGL service project.	6
Create an EGL Rich UI project	8
Import the EGL Dojo widgets sample.	10
Change your build path for MortgageUIProject	11
Lesson checkpoint	12
Lesson 3: Create the mortgage calculation service.	12
Create a Service part	12
Create a Record part	14
Lesson checkpoint	15
Lesson 4: Create the user interface for the calculator	15
Create a Rich UI handler	16
Construct the user interface	17
Lesson checkpoint	27
Lesson 5: Add code to the mortgage calculator handler.	27
Create an EGL library	28
Change the code in the handler.	29
Complete the inputRec_form_Submit function.	30
Add the showProcessImage function	30
Add the hideProcessImage function	31
Add the calculateMortgage function	31
Add the displayResults function	31
Write the exception handler	32
Test the calculator	32
Lesson checkpoint	33
Lesson 6: Create the calculation results handler	34
Publish the service results	34
Create the CalculationResultsHandler handler.	35
Test the pie chart	37
Lesson checkpoint	37
Lesson 7: Create the main Rich UI handler	38
Create the MainHandler handler	38
Test the portal	40
Lesson checkpoint	42
Lesson 8: Create the calculation history handler	42
Create the handler	42
Lesson checkpoint	47
Lesson 9: Embed the calculation history handler in the application	47
Change the results portlet	47
Change the main portal	47
Test the portal	48

Lesson checkpoint	51
Lesson 10: Create the map locator handler	52
Create records for the Interface file	52
Create the Local Search Interface	54
Create the MapLocatorHandler handler	55
Lesson checkpoint	58
Lesson 11: Add code to the map locator handler	58
Finish the source code for MapLocatorHandler.egl	59
Test the new portlet	61
Lesson checkpoint	63
Lesson 12: Embed the map locator handler in the application	63
Change the main portal	63
Test the portal	63
Lesson checkpoint	64
Lesson 13: Install Apache Tomcat	64
Download and access the server	64
Lesson checkpoint	66
Lesson 14: Deploy and test the mortgage application	67
Edit the deployment descriptor.	67
Deploy the Rich UI application.	69
Run the generated code	70
Lesson checkpoint	73
Summary	73
Resources	73
Finished code for MortgageCalculationService.egl after Lesson 3	74
Finished code for MortgageCalculatorHandler.egl after Lesson 4	74
Finished code for MortgageCalculatorHandler.egl after Lesson 5	77
Finished code for CalculationResultsHandler.egl after Lesson 6	80
Finished code for MainHandler.egl after Lesson 7	80
Finished code for CalculationHistoryHandler.egl after Lesson 8	81
Finished code for MainHandler.egl after Lesson 9	82
Finished code for GooglePlaceRecords.egl after Lesson 10	83
Finished code for GooglePlacesService.egl after Lesson 10	84
Finished code for MapLocatorHandler.egl after Lesson 10	84
Finished code for MapLocatorHandler.egl after Lesson 11	85
Finished code for MainHandler.egl after Lesson 12	86

Index 89

Create a mortgage application with EGL Rich UI

Create a Rich UI application so that the user can do the following tasks: calculate monthly mortgage payments, given a loan amount and interest rate; display a pie chart that tells the interest and principal for the life of the mortgage; retrieve any of the payment calculations that were provided earlier to the same user; and display a map of the mortgage lenders who have offices in one or another U.S. zip code.

Learning objectives

In this tutorial, you will complete these tasks:

- Plan the application and design the interface.
- Import a widget to control the different sections of the interface.
- Write a service to calculate mortgage payments.
- Request output from the service and display the results.
- Create a pie chart.
- Pass data between sections of the interface.
- Create an internal table that lists all calculations.
- Access an existing service to find mortgage lenders and to map their locations.
- Install and configure the Apache Tomcat web server.
- Deploy the web page to the web server and test the application.

Time required

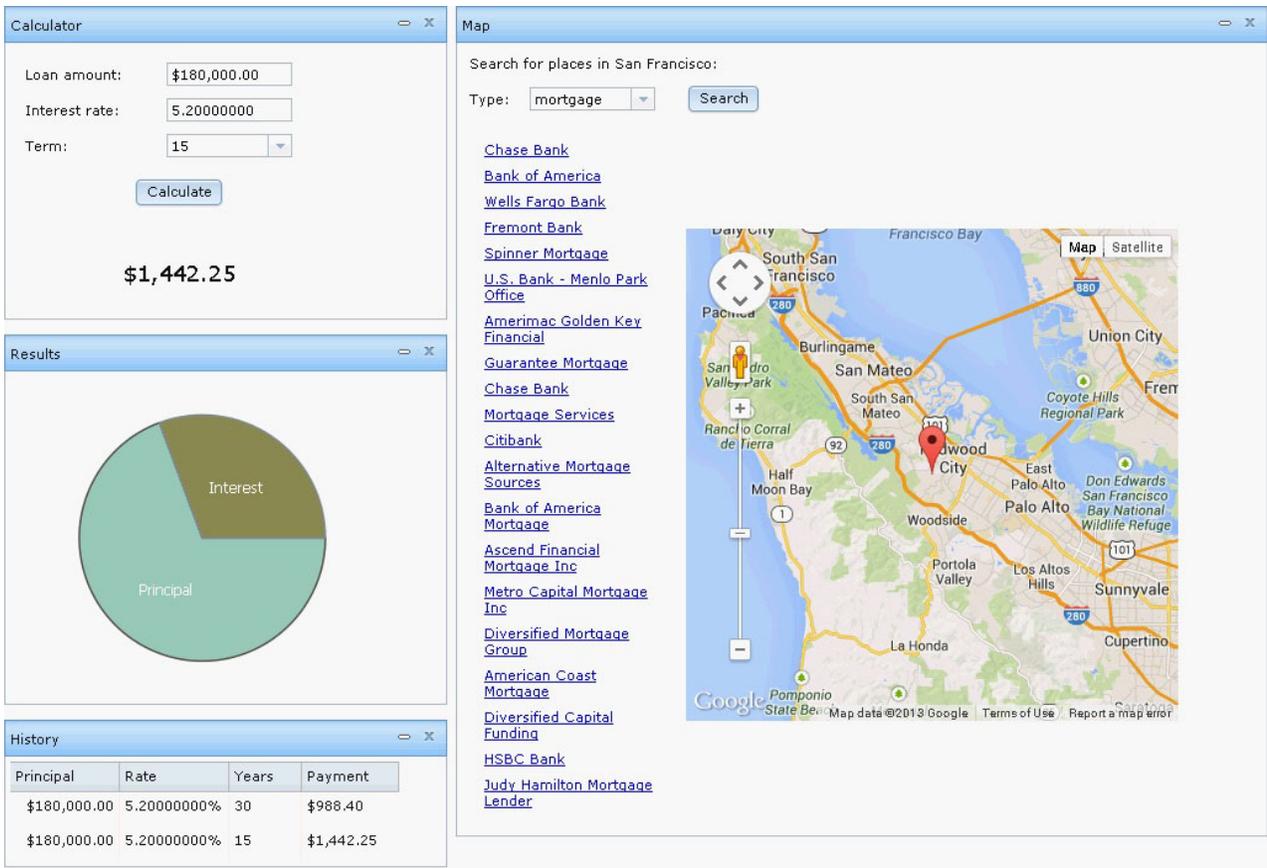
90 minutes

The tutorial in HTML format:

 ["Create a mortgage application with EGL Rich UI" at http://wilson.boulder.ibm.com/infocenter/rbdhelp/v8r0m0](http://wilson.boulder.ibm.com/infocenter/rbdhelp/v8r0m0)

Introduction

The following image shows the application that you will create:



At run time, the user interacts with the Rich UI application. It was deployed to a server, was transmitted to the user's browser, and is running in that browser. From the browser, the Rich UI application accesses services, each of which runs remotely on a server and returns data to the application.

The use of different kinds of logic helps to provide a main benefit of Rich UI: Users can interact with a responsive, local-running web application even as the services do background work such as calculating mortgage payments.

In general, a web service is deployed as a SOAP service or REST service. For further details on the distinction between the two, see "Architectural styles in web services" at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>.

In this tutorial, you access two services:

- A remote SOAP service finds addresses of mortgage lenders and identifies the locations on a map.
- A second service is written by you and is deployed along with the Rich UI application. This kind of service is called an EGL *dedicated service*, and in this case it calculates the mortgage payments.

In general, you can use a dedicated service to do tasks that other EGL-generated Java™ services can do, such as accessing a database or file system. However, the dedicated service is not available to other code unless you redeploy it as an EGL-generated web service.

The benefit of a dedicated service results from its shared deployment with the Rich UI application. If a Rich UI application accesses a web service, your deployment of the application typically requires that you specify the service

location. However, if a Rich UI application accesses a dedicated service, your deployment of the application does not require the location detail. Instead, the service will be available wherever you deploy the Rich UI application.

You can run the Rich UI application and access the service even before you deploy the application internally to a web project. That internal deployment creates the HTML file and embeds that file with others in a web archive (WAR) file, which is a compressed resource like a .zip file. After the Rich UI application and the dedicated service are deployed internally in this way, you deploy them to a server.

Note: Invocation of a dedicated service is slow in the Rich UI editor, but access is much faster when the application and services are deployed to a server.

Learning objectives

The learning objectives are as described in “Create a mortgage application with EGL Rich UI,” on page 1.

Time required

This tutorial takes about 2 hours to finish. If you explore other concepts related to this tutorial, it might take longer to complete.

You can create the EGL files you need for this application in one of the following ways:

- **Line by line (most helpful):** Complete the individual lessons to explore the code in small, manageable chunks, learning important keywords and concepts. This method also requires the greatest time commitment.
- **Finished code files:** At the end of each lesson in which you create a file, you can link to the completed code, which you can copy into the Rich UI editor.

Skill level

Introductory

Audience

This tutorial is designed for people who know the basic concepts of programming and want experience with EGL Rich UI.

System requirements

To complete this tutorial, you must have the following tools and components installed on your computer:

- Rational® Business Developer Version 8.0.1.2 or higher.
- A working Internet connection.

Prerequisites

You do not need any experience with EGL to complete this tutorial.

Expected results

You will create a working Rich UI application that calculates mortgages and finds mortgage lenders in a specified area of the United States.

Lesson 1: Plan the mortgage application

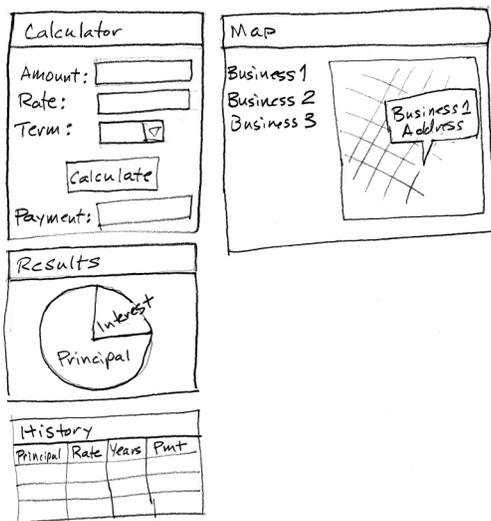
Design your application on paper before you begin coding.

When you plan an application, do as follows:

- List your objectives, as this tutorial did earlier.
- Sketch the interface.
- Identify the application structure.

Sketch the interface

Use this sketch as a guide when you create the components of the interface:



Identify the application structure

When you write a complex Rich UI application, you write code in several Rich UI handlers, each of which corresponds to a web page or to a section of a web page. As noted earlier, the handlers can access services, some of which you might develop by using an EGL Service part.

Whenever possible, use preexisting resources. For one example, your Rich UI application will access a service hosted by Google to retrieve a list of mortgage lenders within a specified zip code. For a second example, you will use the following EGL projects that are provided with the product:

com.ibm.egl.rui.dojo.samples

Provides the following kinds of code and more:

- Widgets that divide the interface into sections, for flexibility at development time
- Logic that creates the dialog boxes with which you notify the user who provides invalid data
- Google map widgets

com.ibm.egl.rui.dojo.widgets

Provides the following widget types for this tutorial:

- DojoButton
- DojoComboBox
- DojoCurrencyTextBox

- DojoPieChart
- DojoTextField
- PieChartData

All those widget types are based on Dojo, as are many other widgets that are available to you. For background details on that technology, see Dojo toolkit (<http://dojotoolkit.org>).

com.ibm.egl.rui

Provides the EGL Infobus, which provides communication between the Rich UI handlers that contribute to the interface. The project also provides the following widget types for this tutorial:

- Box
- DataGrid
- GridLayout
- HyperLink
- Image
- TextField
- TextLabel

You will develop the following logic:

MortgageCalculationService

A dedicated service that calculates monthly payments

MortgageLib

A library that provides code to several handlers

MainHandler

A handler that declares other handlers, each of which either controls a section of the web page or does other work in the background

MortgageCalculatorHandler

A handler that calculates monthly payments

CalculationHistoryHandler

A handler that displays an interactive list of previous payment calculations

CalculationResultsHandler

A handler that displays a pie chart of interest payments and principal

MapLocatorHandler

A handler that displays the locations of mortgage lenders

Lesson checkpoint

In this lesson, you completed the following tasks:

- Sketched the application interface
- Identified the application structure

In the next lesson, you import the Dojo samples project and create two EGL projects to hold your code.

Lesson 2: Set up the workspace

Before you write your logic, create two EGL projects and import the Dojo samples.

An EGL application is organized in one or more *projects*, each of which is a physical folder in the workspace. A project contains an EGL source folder that is provided for you, and that folder contains one or more *packages*, which in turn contain EGL source files. This hierarchy is basic to your work in EGL: a project, then an EGL source folder, then a package with EGL source files.

The EGL source files include EGL *parts*, which are type definitions that you create. For example, a Service part contains logic, and a Record part can be the basis of a variable that you declare in your Service part.

Packages are important because they separate parts into different contexts, or *namespaces*:

- A part name might be duplicated in two different packages, and any EGL source code can reference each part precisely. The main benefit of namespaces is that different teams can develop different EGL parts without causing name collisions.
- Each part name in a given package is unique within that package:
 - A part in one package can easily reference another part in the same package by specifying the part name. For example, here is a declaration of a record that is based on the Record part MyRecordPart:

```
myRecord MyRecordPart{};
```
 - A part in one package can also reference a part in a second package by giving the package name and part name, or by a shortcut that involves importing the part. This tutorial gives examples.

One project can reference the parts in a second project, but only if the EGL build path of the referencing project identifies the referenced project. Again, this tutorial gives examples. However, in all cases, avoid using the same package name in different projects, as that usage can cause problems in name resolution.

Your next task in this tutorial is to create the following projects:

MortgageServiceProject

Holds an EGL Service part and related definitions

MortgageUIProject

Holds the Rich UI handlers and related definitions

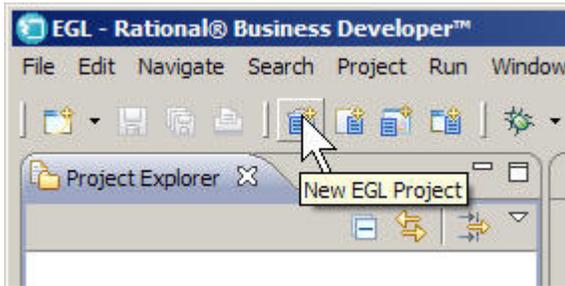
You can include all your code in a single project, but the separation shown here lets you easily deploy the two kinds of code in different ways.

Create an EGL service project

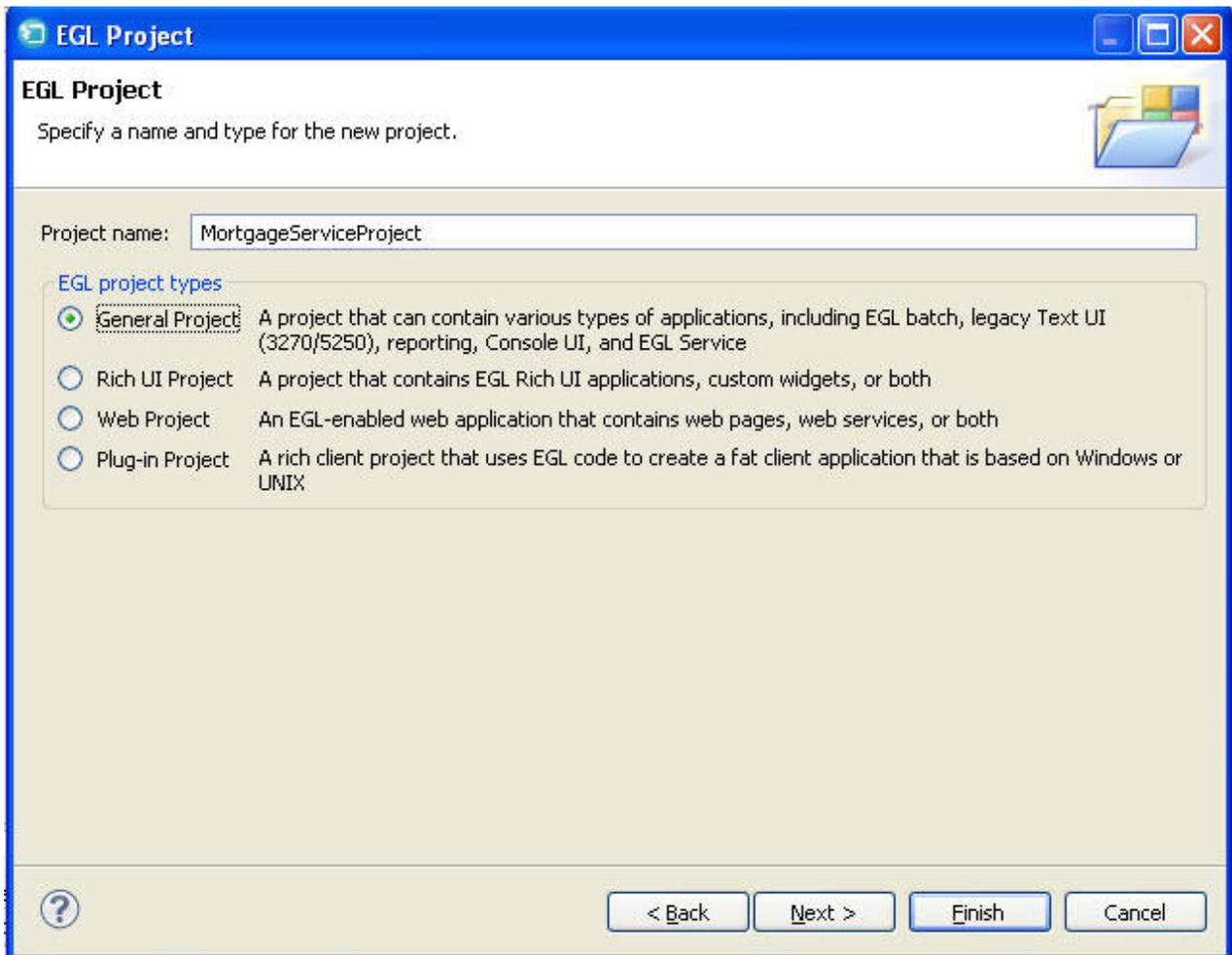
1. If you are in a workbench perspective other than EGL, change to the EGL perspective by clicking **Window > Open Perspective > Other > EGL**. The perspective icon is in the upper-right corner of the workbench.



2. Click **File > New > EGL Project**, or click the **New EGL Project** icon on the menu bar.



3. In the EGL Project window, enter the following information:
 - a. In the **Project name** field, type the following name:
MortgageServiceProject
 - b. In the **EGL Project Types** section, click **General Project**.



- c. Click **Next**.
4. In the second EGL Project window, the defaults that EGL provides should be correct. Verify the following information:

- a. The **Target runtime platform** is Java. This setting indicates that EGL generates Java source code from your EGL Service part.
 - b. Under **Build descriptor options**, the **Create a build descriptor** radio button is selected. Build descriptors control the generation process. Because you are creating a separate project for your service, you can use the default build descriptor that EGL creates for you.
5. Click **Finish**.

EGL creates a project named MortgageServiceProject. Note the folders inside the directory:

EGLSource

Put your packages and source files here.

EGLGen/JavaSource

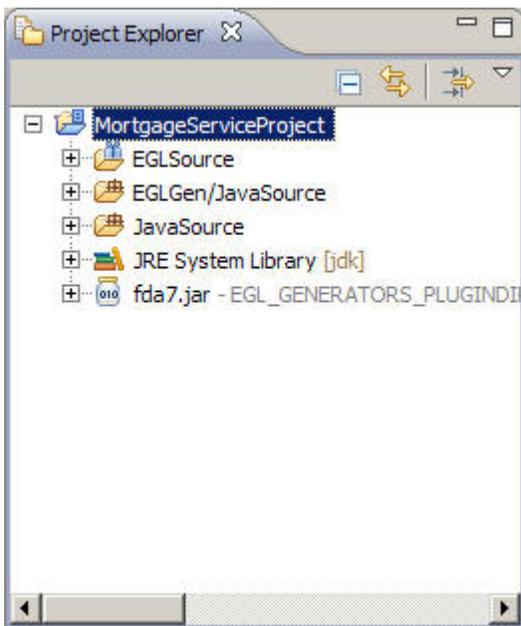
EGL places the Java files it generates here.

JavaSource

Put any custom Java source files here. These files are not overwritten during the generation process.

JRE System Library

EGL uses this folder for JAR files that support the Java Runtime Environment.



Related reference

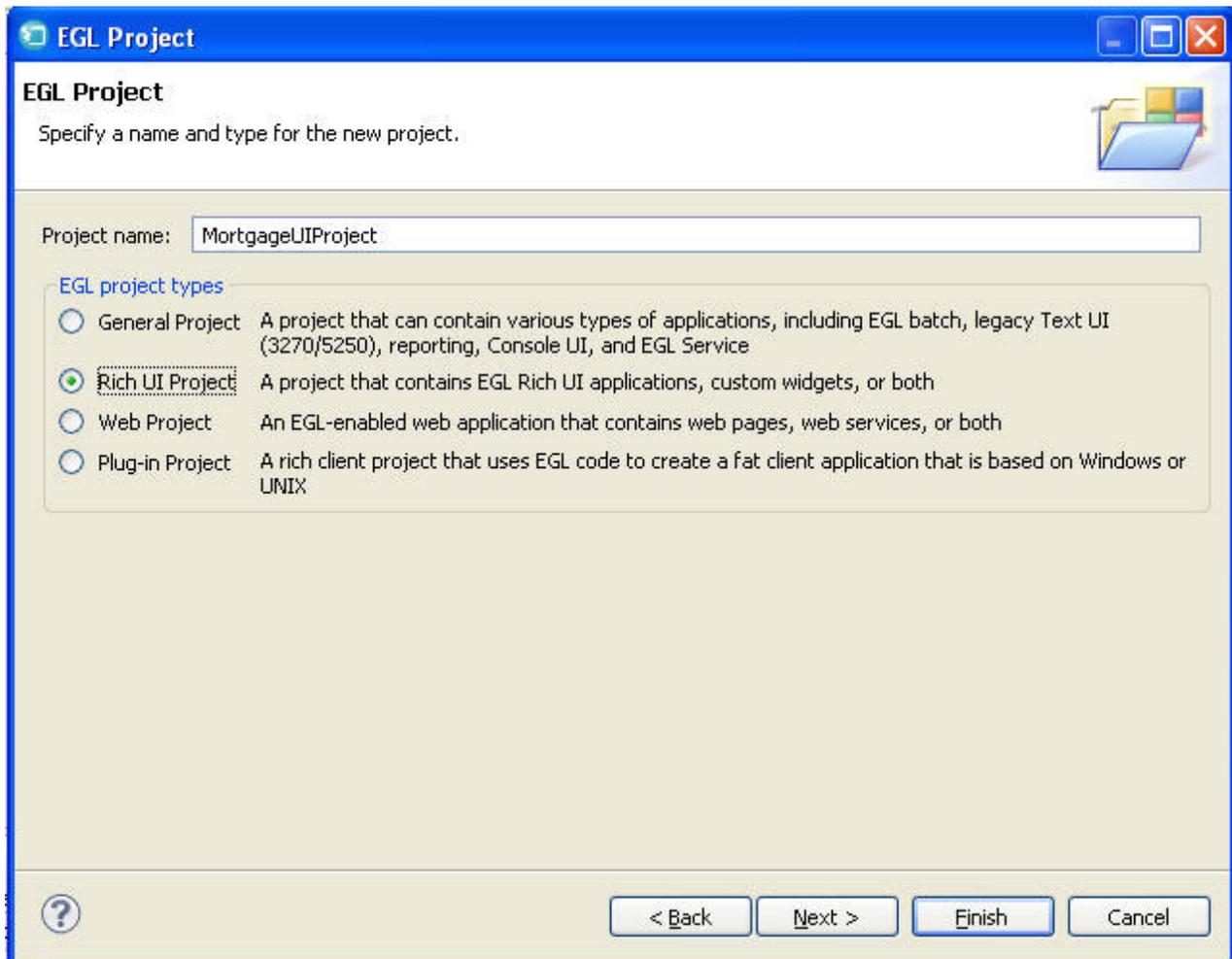
 "Default build descriptors" at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

Create an EGL Rich UI project

1. Click the **New EGL Project** icon on the menu bar.
2. In the New EGL Project window, enter the following information:
 - a. In the **Project name** field, type the following name:

MortgageUIProject

- b. In the **EGL Project Types** section, click **Rich UI Project**.

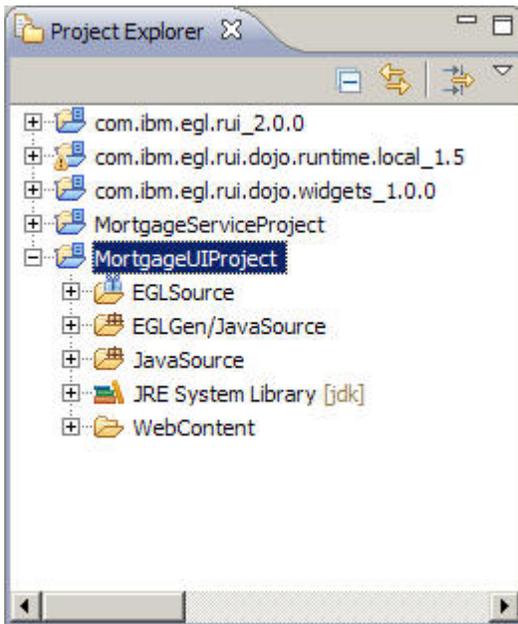


- c. Click **Next**.
3. In the second EGL Project window, the defaults that EGL provides should be correct. Verify the following information:
 - a. **Use the default location for the project** is selected.
 - b. The **Widget libraries** list contains the following projects:
 - EGL Rich UI widgets
 - EGL Dojo widgets
 - c. In the **EGL project features** group, **Create an EGL deployment descriptor** is selected.
4. Click **Next**.
5. On the **EGL Settings** page, select **MortgageServiceProject**. The service project is added to the build path for the new project, so that the UI project can use parts that are defined in the service project.
6. Click **Finish**.

EGL creates a project named MortgageUIProject and adds support projects to the workspace for Rich UI, Dojo Widgets, and the Dojo runtime library. In addition to the directories that EGL created for the General project, a Rich UI project includes the following directory:

WebContent

Contains support files, such as cascading style sheets (CSS) and images.



Import the EGL Dojo widgets sample

1. From the top menu of the workbench, click **Help > Help Contents**.
2. In the Help contents, expand **Samples > Technology samples > EGL** and click **EGL Dojo widgets**.



3. In the Content pane, click **Get the sample**.

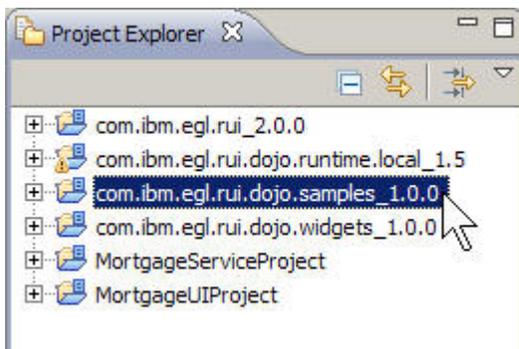
EGL Dojo widgets

You can use this collection of samples to practice w

- 📄 Setup instructions
- 🟡 Get the Dojo widget dependency project
- 🟡 Get the Rich UI widget dependency project
- 🟡 Get the sample
- 🟡 Get the local Dojo Toolkit project
- 🟡 Get the Google provider project
- 🟡 Get the AOL provider project

4. In the Import window, the default values are correct. Click **Finish**.

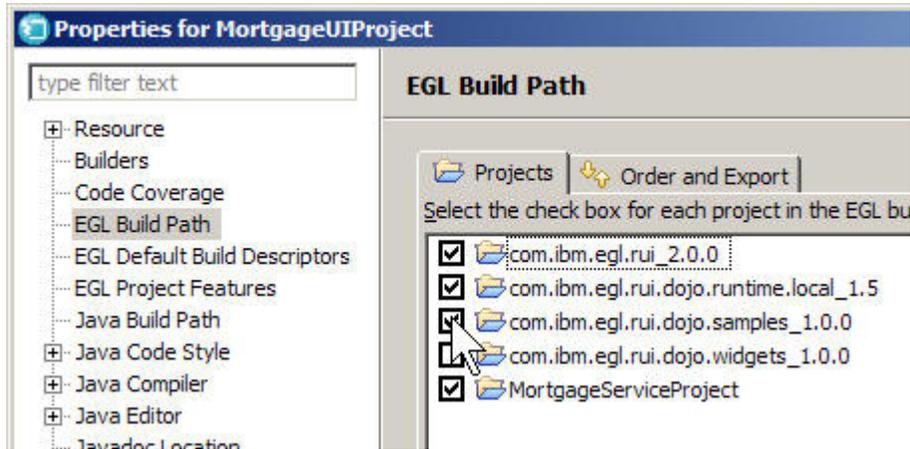
The latest version of the `com.ibm.egl.rui.dojo.samples` project is added to your workspace.



Change your build path for MortgageUIProject

The EGL build path determines the projects that EGL examines when trying to resolve references in your logic. To add the project that you just imported:

1. In the Project Explorer view, right-click **MortgageUIProject**, and then click **Properties**. On the left side of the Properties for MortgageUIProject window, click **EGL Build Path**. EGL displays a list of the projects in your workspace.
2. Select the **com.ibm.egl.dojo.samples** project. You do not need to select the `com.ibm.egl.dojo.widgets` project because it is already in the build path of the `com.ibm.egl.dojo.runtime.local` project. The finished build path window should look like the following image:



These selections mean that when you organize the import statements that provide other details to your programs, EGL will look in all of the selected projects to resolve references.

3. Click **OK**.

Related reference

 “The EGL build path” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

Lesson checkpoint

In this lesson, you completed the following tasks:

- Created an EGL project for the mortgage service
- Created an EGL project for the Rich UI application
- Imported the EGL Dojo samples project
- Adjusted the EGL build path for the second project

In the next lesson, you create a dedicated service to calculate a monthly mortgage payment.

Lesson 3: Create the mortgage calculation service

Create a dedicated service to calculate monthly payments.

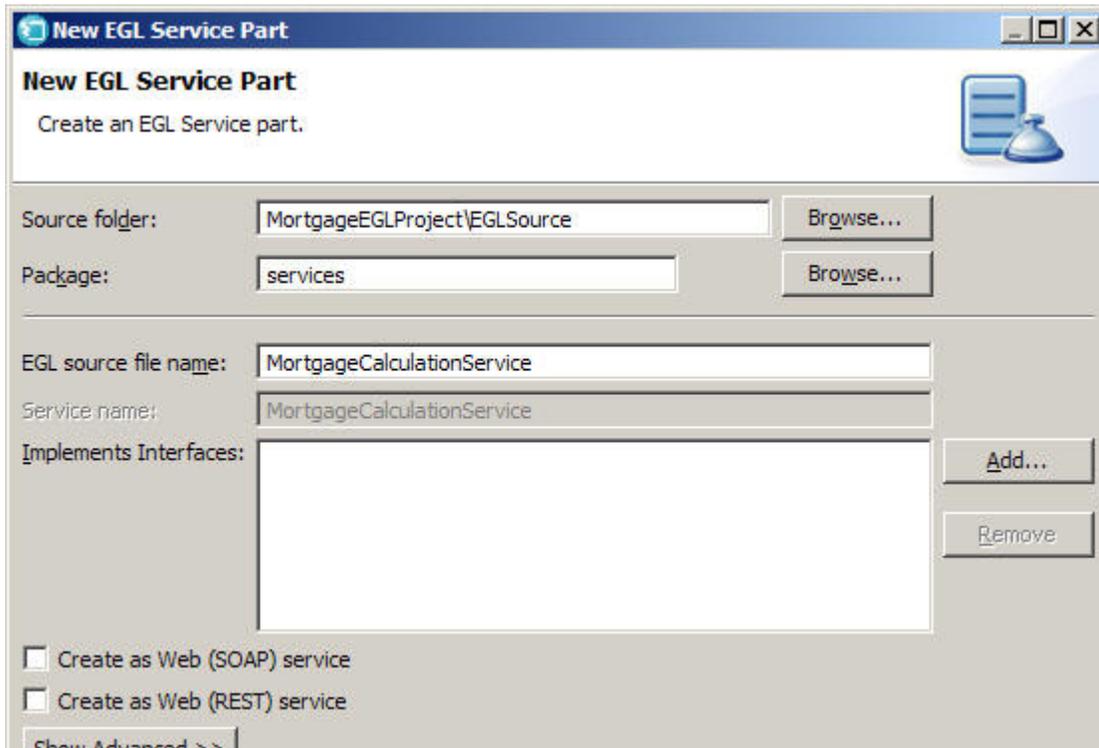
In this lesson, you create an EGL Service part, which is a generatable part. You must place each generatable part in a separate source file, and the name of the part must be the same as the name of the file.

Create a Service part

1. In the Project Explorer window, right-click **MortgageServiceProject**, and then click **New > Service**.
2. In the New EGL Service Part window, enter the following information:
 - a. In the **Package** field, enter the following name:
services
 - b. In the **EGL source file name** field, enter the following name:
MortgageCalculationService

EGL adds the .egl file extension.

- c. Verify that **Create as web (SOAP) service** and **Create as web (REST) service** are cleared, and leave the **Implements Interfaces** field empty.



3. Click **Finish**. EGL opens the new Service part in the editor.
4. Remove the code from the file, leaving only the following lines:

```
package services;

service MortgageCalculationService

end
```

5. Add the following function before the **end** statement:

```
function amortize(inputData MortgageCalculationResult inOut)
    amt MONEY = inputData.loanAmount;
    // convert to monthly rate
    rate DECIMAL(10, 8) = (1 + inputData.interestRate / 1200);
    // convert to months
    term INT = (inputData.term * 12);

    // calculate monthly payment amount
    pmt MONEY = (amt * (rate - 1) * Mathlib.pow(rate, term)) /
        (MathLib.pow(rate, term) - 1);
    totalInterest MONEY = (pmt * term) - amt;

    // update result record
    inputData.monthlyPayment = pmt;
    inputData.interest = totalInterest;
end
```

When you paste code from these instructions, the formatting might change. Press **Ctrl+Shift+F** to reformat the code. You can change the formatting rules by clicking **Window > Preferences > EGL > Editor > Formatter**.

Note:

EGL marks any code errors with a red X in the left margin and a wavy red line under the error. Move your cursor over the X to see an error message.

```
package services;

service MortgageCalculationService

onResult inOut)
    amt MONEY = inputData.loanAmount;
    // convert to monthly rate
    rate DECIMAL(10, 8) = (1 + inputData.interestRate / 1200);
    // convert to months
    term INT = (inputData.term * 12);

    // calculate monthly payment amount
    pmt MONEY = (amt * (rate - 1) * MathLib.pow(rate, term)) /
                (MathLib.pow(rate, term) - 1);
    totalInterest MONEY = (pmt * term) - amt;

    // update result record
    inputData.monthlyPayment = pmt;
    inputData.interest = totalInterest;
end
end
```

Because you have not yet defined a type named `MortgageCalculationResult`, EGL cannot create the `inputData` variable based on that type. When you create this Record type in the next exercise, EGL will remove the error markers from the display.

6. Save the file by clicking **File > Save**.

Create a Record part

The `amortize()` function uses a `MortgageCalculationResult` record. You can define this record in the same file as the Service.

To create the Record part:

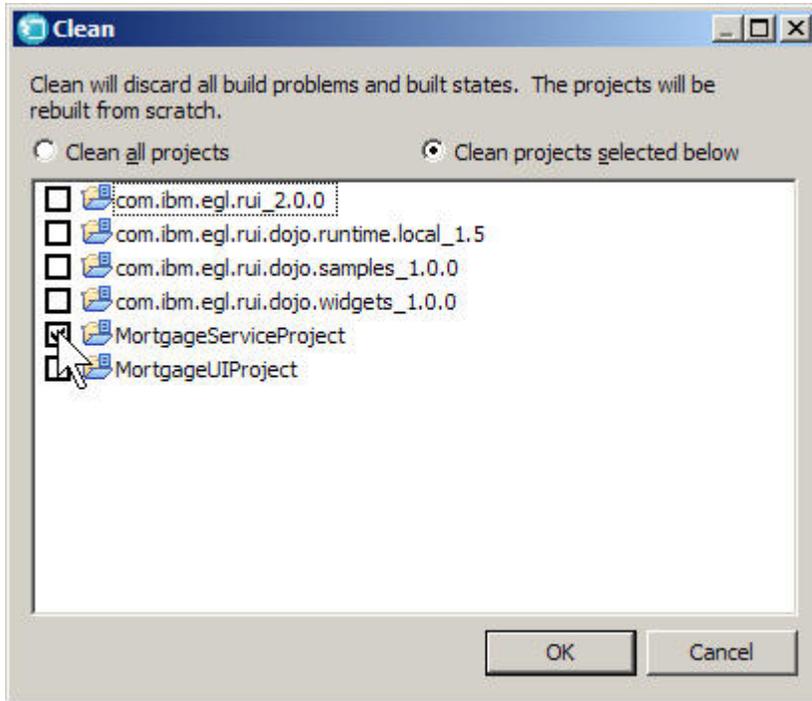
1. Add the following code after the `amortize()` function in the `MortgageCalculationService.egl` file. The Record is a part, so you define it outside the Service part, after the final **end** statement in the file:

```
record MortgageCalculationResult
    // user input
    loanAmount MONEY;
    interestRate DECIMAL(10,8);
    term INT;

    // calculated fields
    monthlyPayment MONEY;
    interest MONEY;
end
```

2. Save the file. EGL should not display error markers in the code. If you see errors in your source file, compare your code to the file contents in “Finished code for `MortgageCalculationService.egl` after Lesson 3” on page 74. As you

work through the tutorial, you might see red Xs next to the project or next to one of the folders below it, yet not see any errors in the file itself. If you encounter this situation, resolve it by clicking **Project > Clean**. In the Clean window, click **Clean projects selected below** and then click the appropriate project, such as **MortgageServiceProject**.



Click **OK**. EGL rebuilds the selected project and the red X is removed from the Project Explorer view.

3. Close the file by clicking the X next to the file name in the tab at the top of the editor or by clicking **File > Close**.

Lesson checkpoint

You learned how to complete the following tasks:

- Create an EGL Service part
- Create an EGL Record part and add it to the source file for the Service
- Check for errors in your code

In the next lesson, you create the user interface for the first application portlet.

Related reference

 “Services: a top-level overview” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

Lesson 4: Create the user interface for the calculator

Start to build the calculator by using EGL wizards and then the Rich UI editor.

You can add widgets to a web page by dragging content to the Design surface of the Rich UI editor. The drag-and-drop and subsequent interaction with the editor updates the source code for the Rich UI handler that you are developing.

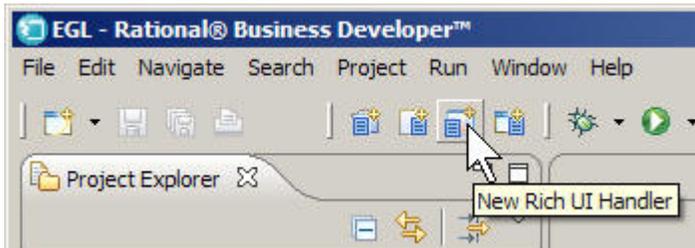
Two sources of drag-and-drop content are available:

- A palette of widget types
- The EGL Data view, which provides data-type definitions such as EGL Record parts. You first drag content from this view and then choose from among the widget types that can display the type of data you selected.

By default, the widget palette is at the right of the editor, and the Data view is at the lower left of the workbench.

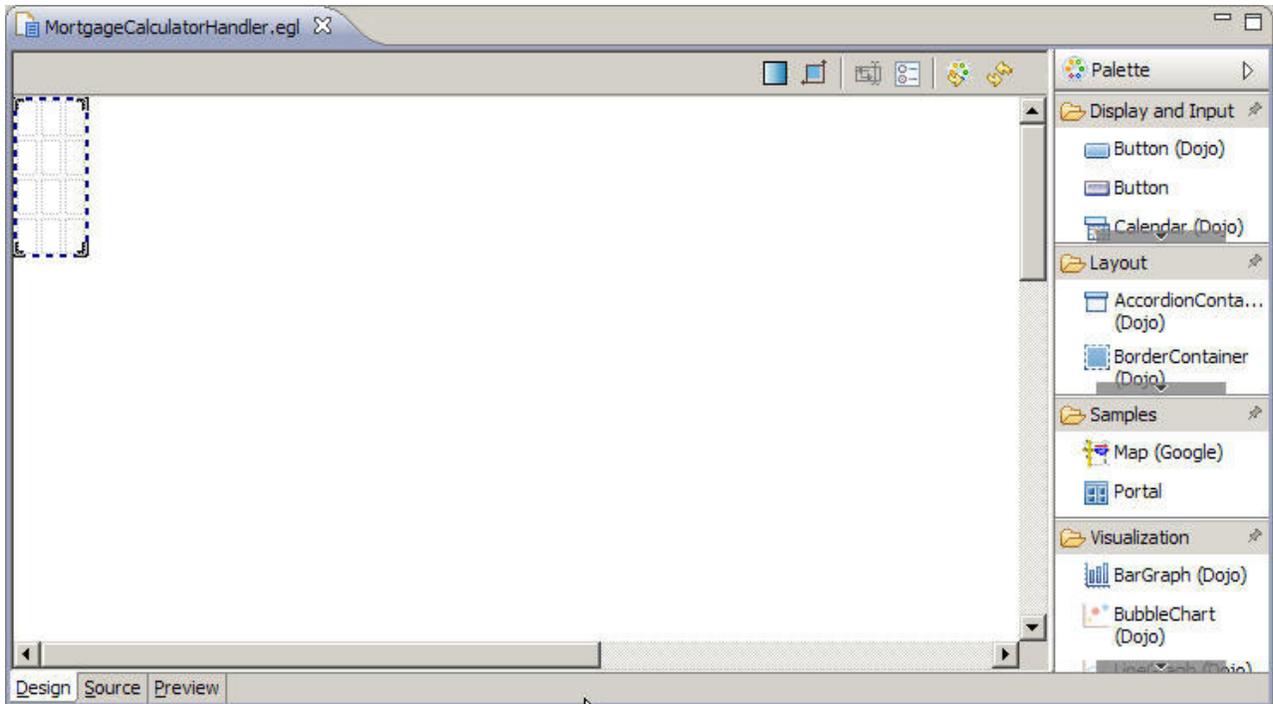
Create a Rich UI handler

1. In the **MortgageUIProject** project, select the **EGLSource** folder and click the **New Rich UI Handler** button on the menu bar.



2. In the “New Rich UI Handler part” window, enter the following information:
 - a. In the **Package** field, enter the following name:
handlers
 - b. In the **EGL source file name** field, enter the following name:
MortgageCalculatorHandler
 - c. Click **Finish**.

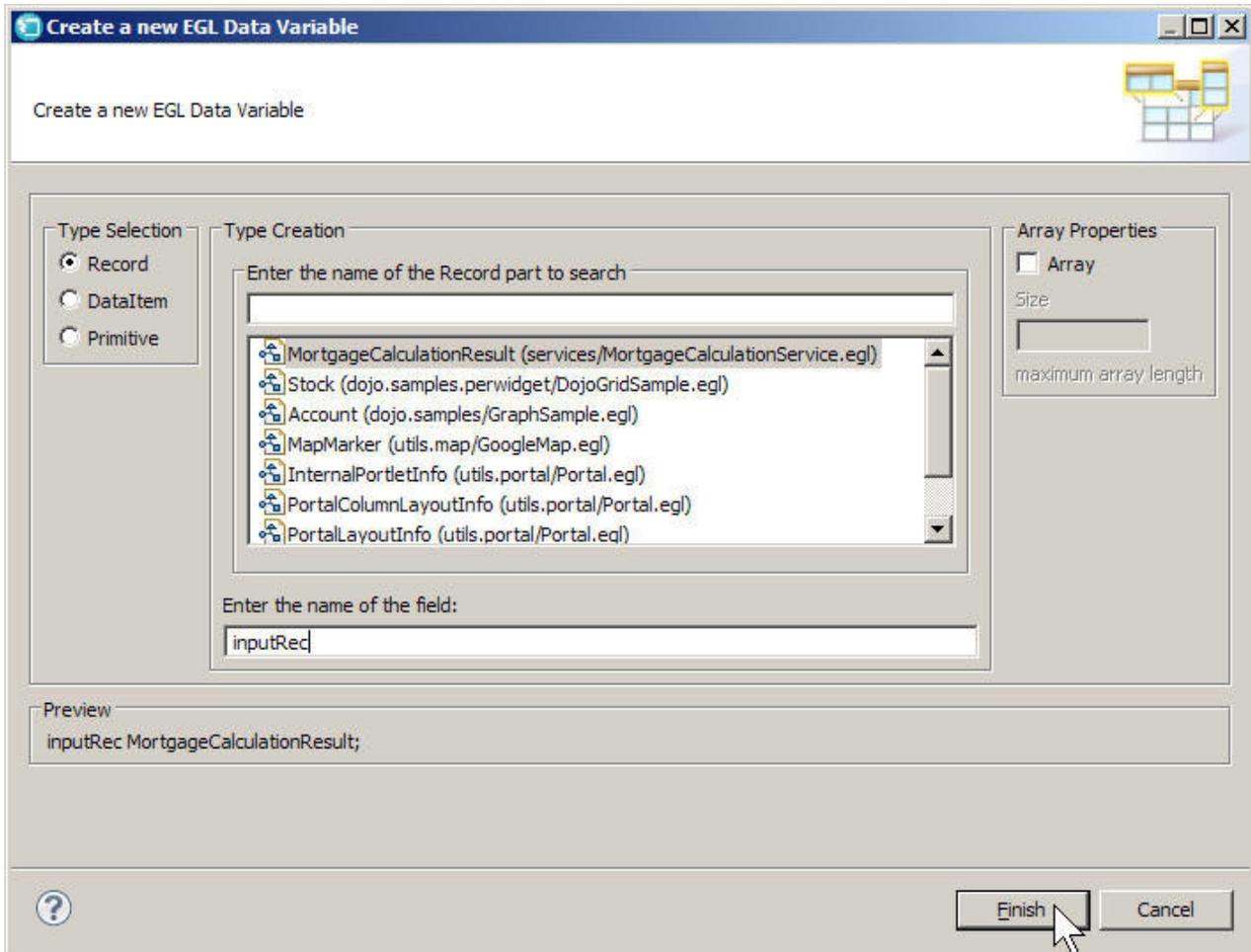
The new Handler opens in Design view in the Rich UI editor. EGL creates the **handlers** package for you in the **EGLSource** folder.



Construct the user interface

To construct the UI for the calculator:

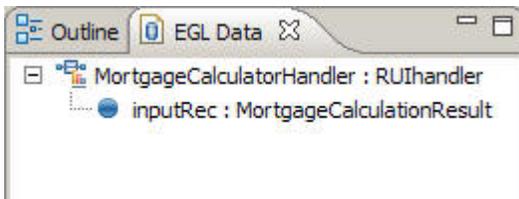
1. Create a record variable.
 - a. The EGL Data view, which is located by default in the lower-left corner of the workbench, lists all primitive and record variables for the handler that is currently open in the editor. Right-click the empty space below the entry for the MortgageCalculatorHandler handler. Click **New > EGL Variable**.
 - b. In the Create a new EGL Data Variable wizard, request a new record variable based on the MortgageCalculationResult Record part:
 - Make sure **Type Selection** is set to **Record**.
 - Select the MortgageCalculationResult record. This should be the first type in the list.
 - In the **Array Properties** section, leave **Array** cleared.
 - In the **Enter the name of the field** field, enter the following text:
inputRec
 - Click **Finish**.



This process creates the following record declaration in the source code for the handler:

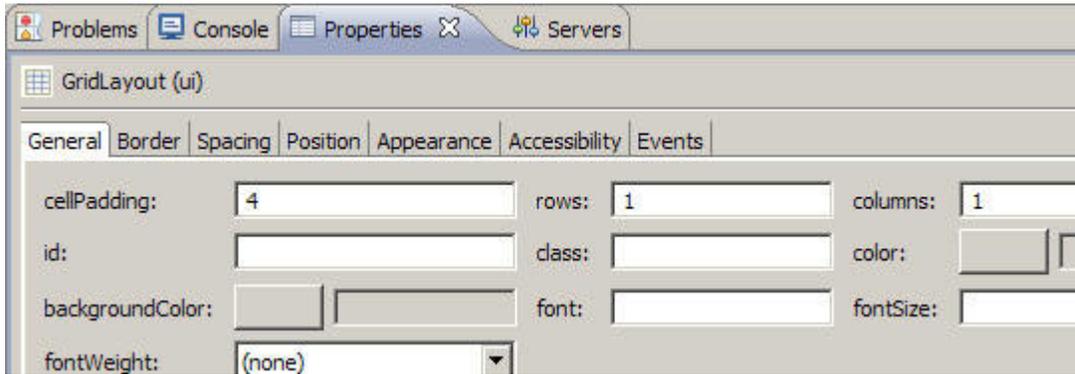
```
inputRec MortgageCalculationResult;
```

The process also shows the new record variable in the EGL Data view so that you can drag the variable onto the editor.



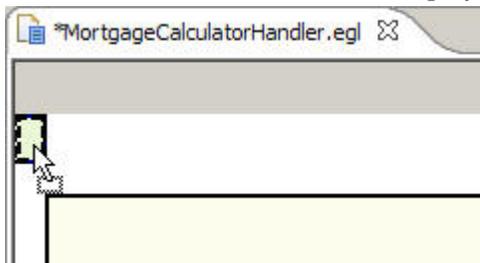
2. EGL automatically created a GridLayout widget for you as your initial UI. By default, this widget has four rows and three columns. You can use different mechanisms to vary the number of rows and columns, as demonstrated here:
 - a. Right-click the GridLayout widget to highlight a cell.
 - b. Click **Delete > Row**
 - c. Again, right-click the GridLayout widget to highlight a cell.
 - d. Click **Delete > Column**
 - e. Now go to the Properties view, which by default is one of the tabbed pages below the editor pane.

- f. On the General page, set the **rows** property to 1 and the **columns** property to 1.



The modified GridLayout widget remains the initial UI for the web page, but now has a single cell into which you will add other content.

3. Click the **inputRec** variable in the EGL Data view, and drag that variable from the EGL Data view to the one cell of the GridLayout widget in the editor. EGL displays the **Configure data widgets** page of the Insert Data wizard. Use

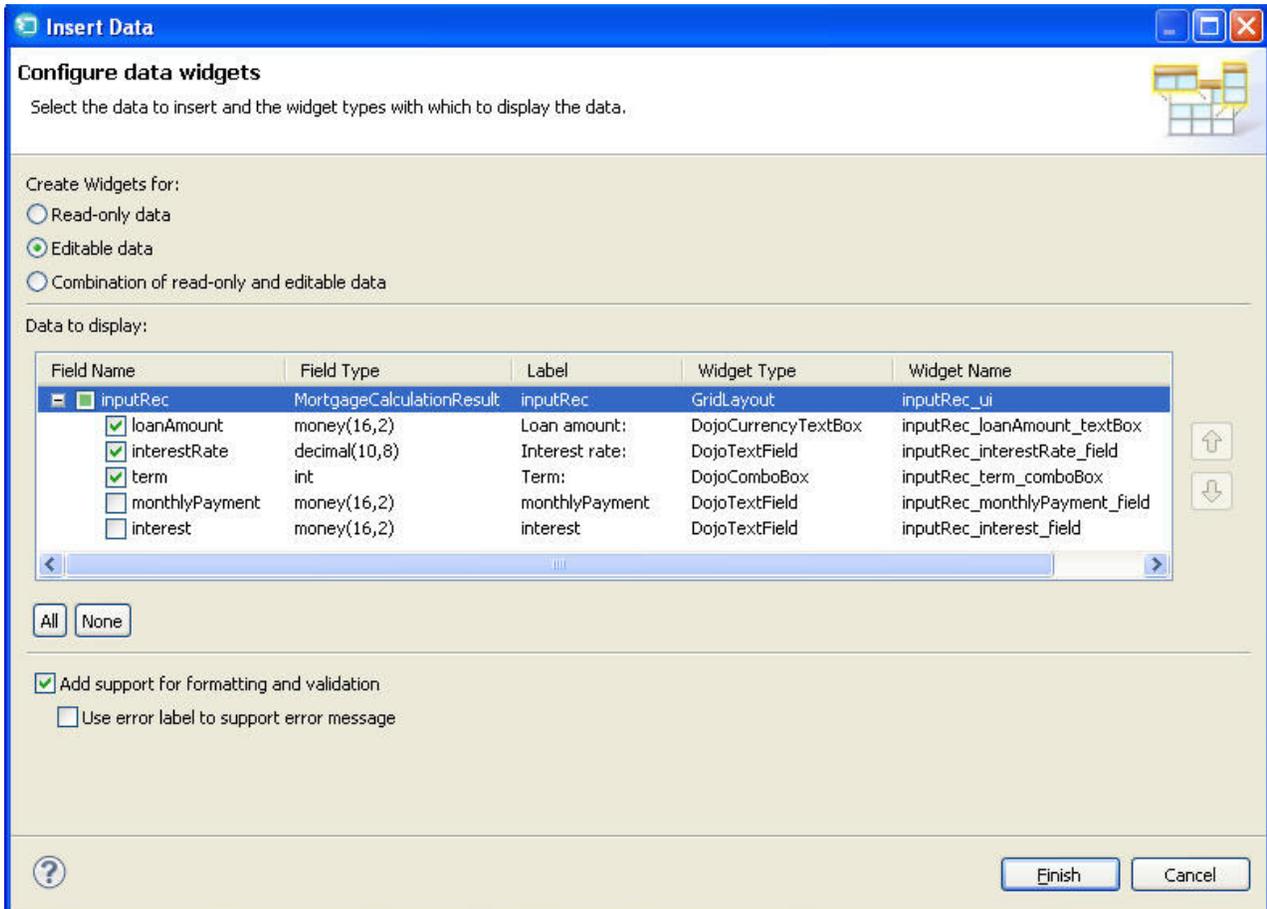


this page to configure the widgets that EGL creates based on the fields in the record you dragged onto the editor.

4. On the **Configure data widgets** page, make the following changes:
 - a. In the **Create Widgets for** section, click **Editable data**. This option determines the default widgets that EGL displays in the wizard, none of which have the read-only restriction.
 - b. In the Widget Type column for the **loanAmount** field, click the current value in the Widget Type column, click the down arrow, and select **DojoCurrencyTextBox**. This widget provides formatting and validation for money amounts.
 - c. Leave the default **DojoTextField** widget for the interestRate widget.
 - d. In the Widget Type column for the **term** field, click the current value in the Widget Type column, click the down arrow, and select **DojoComboBox**.
 - e. Clear the check box for the **monthlyPayment** and **interest** fields. You will add a widget for the monthly payment field later; the interest field is not part of this user interface.
 - f. Change the values in the Label column as follows, including the colons:
 - Change "loanAmount" to "Loan amount:"
 - Change "interestRate" to "Interest rate:"
 - Change "term" to "Term:"

You just customized the prompts for each of the first data-entry fields on the form being built.

- g. Ensure that **Add support for formatting and validation** is selected. This option adds a label and controller for each widget, along with a form manager and related functions that apply to all the content in the new grid layout. Do not select **Use error label to support error message**. This option provides an error label for any controller-specific error message, but this tutorial uses other mechanisms to indicate that an error has occurred.
- h. View the following image. When your page is essentially the same, click **Finish**.



The Design view is displayed, including a new, inner grid layout that is within the original grid layout and that contains the new content.



- 5. To save your work, press **Ctrl+S**.
- 6. Adjust the size of the second and third input fields for a more uniform appearance:

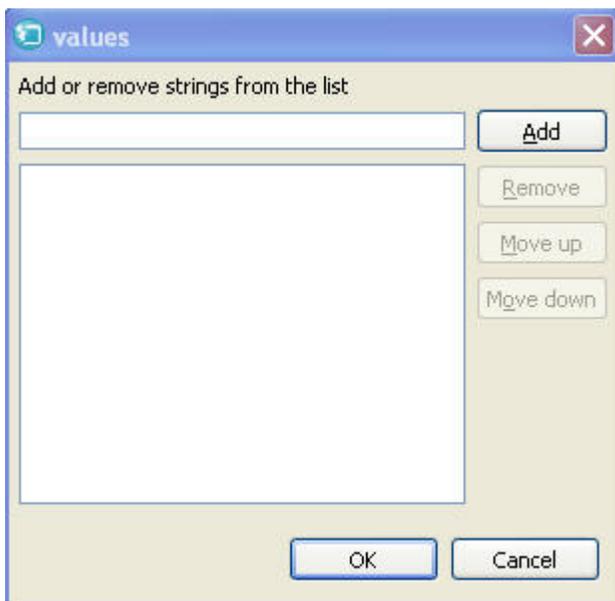
- a. Highlight the input field for the interest rate. The dotted line should enclose only that field.

- b. In the Properties view, on the Position page, enter the following value for the **width** property:

100

This value is the same as the default width for the DojoCurrencyTextBox widget that you used for the loan amount.

- c. Repeat steps a and b for the **Term** field. Click the Display surface to see the change.
7. Set the valid values and the default value for the Dojo combo box:
 - a. With the **Term** field highlighted, click the General page of the Properties view.
 - b. Next to the **values** property, click the ellipsis (...) to display the values window.



- c. Type the following number in the **Add or remove strings from the list** field:

5

- d. Click **Add**.

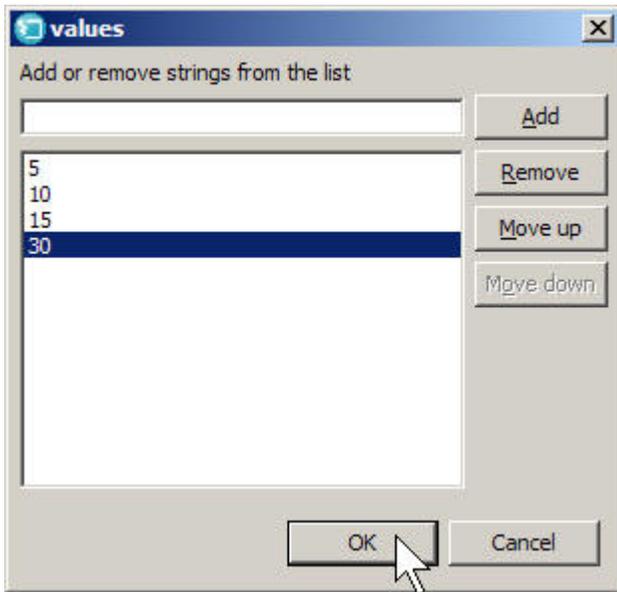
- e. Type each of the following numbers, clicking **Add** after each addition:

10

15

30

The values window should look like this image:



- f. Click **OK**.
8. To ensure that the initial display of the combo box includes the value 30, do as follows:

- a. Click the **Source** tab of the Rich UI editor.
- b. Set a default value for the term. Go to the line where the **inputRec** record is declared and add a set-values block to the declaration, as shown here:

```
inputRec MortgageCalculationResult {term = 30};
```

You are embedding the default value in the declaration, as is easiest. However, you could have updated the `start()` function as follows, with the same effect:

```
function start()  
    inputRec.term = 30;  
end
```

9. While you are looking at the source code, review the following controller declaration, which relates the `inputRec.term` variable with the Dojo combo box:

```
inputRec_term_controller Controller  
    { @MVC {model = inputRec.term, view = inputRec_term_comboBox as Widget}};
```

The controller declaration ensures that the value you assigned to the `inputRec.term` variable will be used to initialize the combo box.

10. To save your work, press **Ctrl+S**.
11. Click the Rich UI editor **Preview** tab. The web page shows the runtime display and should look like this image:

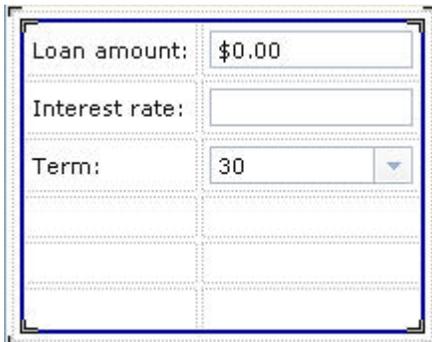
Loan amount:	<input type="text" value="\$0.00"/>
Interest rate:	<input type="text"/>
Term:	<input type="text" value="30"/> ▾

If you need to refresh the display, click the rightmost icon on the Rich UI toolbar, which is on the upper right of the Preview page and is shown here:

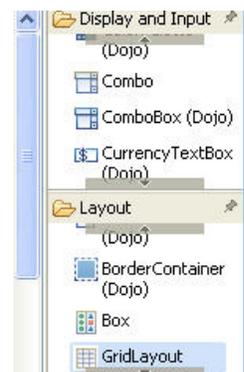
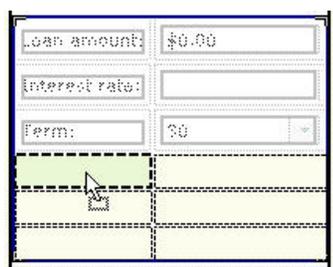


12. Add new content to the inner GridLayout widget, which holds the record detail:
 - a. Click the **Design** tab of the Rich UI editor.
 - b. Right-click the inner GridLayout widget, which is named `inputRec_ui`. That name is displayed at the top of the Properties view.
 - c. On the General page of the Properties view, change the **rows** property to the following value:
6

After you click the Design surface, the new rows are displayed underneath the Term row.

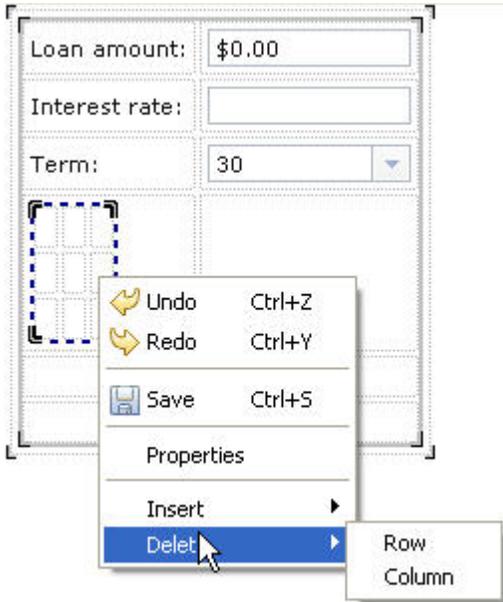


13. Add a second inner GridLayout widget to hold a submit button.
 - a. From the **Layout** drawer of the palette, drag a GridLayout widget to the first cell in the first blank row.

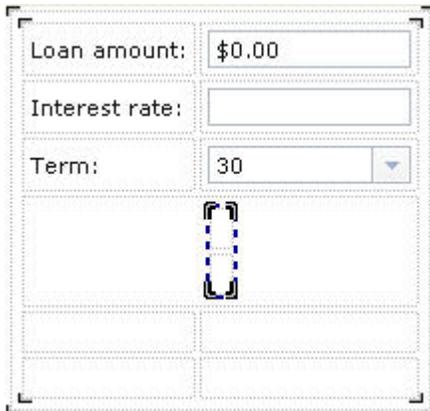


You will use the GridLayout widget to position the submit button and an animated processing image. Give the widget the following name:
`buttonLayout`

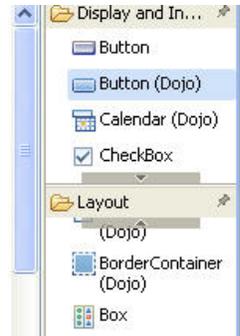
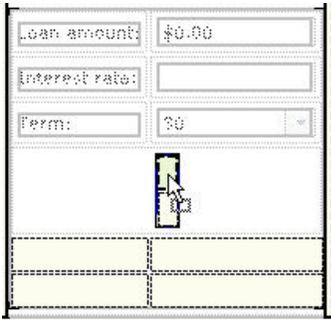
- b. Right-click the new widget to display the menu shown here.



- c. Select **Delete** and then click **Row**. The General page of the Properties view now indicates that the grid layout has 2 rows.
- d. Use either the Properties view or the menu that you just used to modify `buttonLayout` to have one column. If you make a mistake and want to revert to an earlier display, press `Ctrl+Z`.
- e. When `buttonLayout` is active, go to the Layout page of the Properties view. Do as follows so that the layout is centered in an otherwise blank row:
 - Change the **horizontalAlignment** property to `CENTER`.
 - Change the **horizontalSpan** property to `2`.



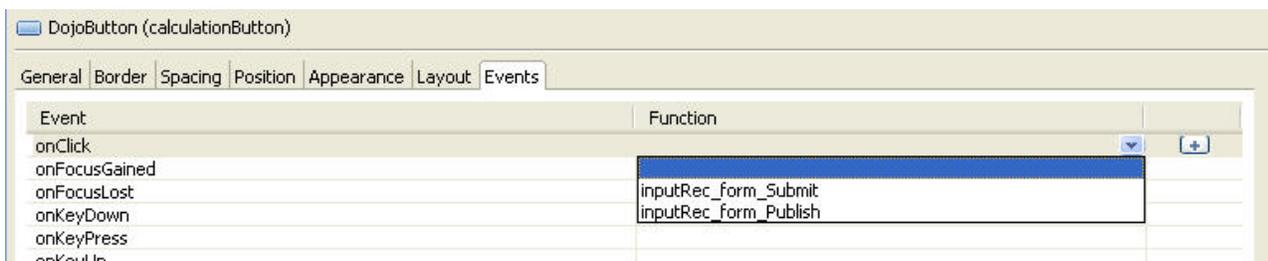
14. To save your work, press `Ctrl+S`.
15. Create a submit button and bind it to a stub function:
 - a. From the **Display and Input** drawer of the palette, drag a Dojo Button onto the upper cell of `buttonLayout`.



- b. Give the button the following name:
calculationButton
- c. On the General page of the Properties view, change the **text** property to the following label:
Calculate

Next, you must bind the button to a function that was created for you when you dragged the `inputRec` variable onto the editor.

- d. On the **Events** tab, select the row with the `onClick` event. Click the blank space in the Function column to display an arrow button. Click the arrow button and select `inputRec_form_Submit`. You will complete the code for this function in the next lesson.
Note the plus sign in the last column of the row. You could have clicked



this button to create a new function to bind to the **Calculate** button. The workbench would have brought you to the Source view and automatically created a *stub function*, which is a function that has no code but is ready for your input.

16. Add an animated image that indicates that a mortgage calculation is in process.
 - a. From the **Display and Input** drawer of the palette, drag an Image widget to the empty cell below the **Calculate** button.
 - b. In the New Variable window, give the image the following name:
processImage
 - c. In the Properties view, on the General page, assign a source for the image in the **src** field:
tools/spinner.gif
The image is located in the `com.ibm.egl.rui.dojo.samples/WebContent` folder. The development environment treats the WebContent folders of all of the projects in your workspace as a single virtual folder.
 - d. Also in the Properties view, on the Appearance page, clear the **visible** check box. The image remains hidden until the **Calculate** button is clicked.

- e. Also in the Properties view, on the Layout page, set the **horizontalAlignment** property to CENTER.
17. To save your work, press Ctrl+S.
18. Add a widget to display the results of the calculation.
 - a. Drag a TextLabel widget from the **Display and Input** drawer of the palette to the first cell of the fifth row, which is below the new graphic. Assign the widget the following name:

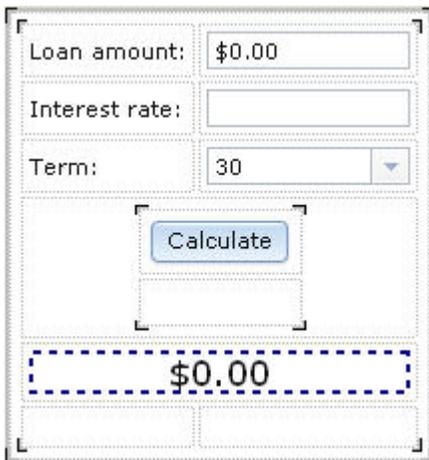

```
paymentLabel
```

You use a label widget here because the user does not change this field. The application calculates and updates the contents.

- b. In the Properties view, on the General page, enter the following value for the **text** property (including the dollar sign):


```
$0.00
```
- c. On the same page, set the **fontSize** property to 18.
- d. Also in the Properties view, on the Layout page, set the following properties:
 - Set **horizontalAlignment** to CENTER.
 - Set **horizontalSpan** to 2.

When you click the Design surface, the web page now looks like this image:

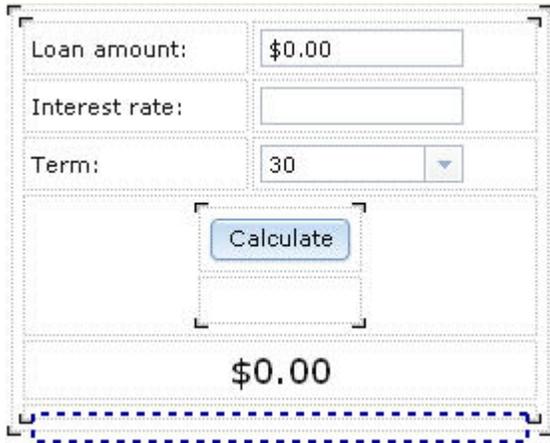


19. Add an error field for general errors, such as problems connecting to the service.
 - a. Drag a TextLabel widget from the **Display and Input** drawer of the palette to the first cell of the last empty row and assign the following name:


```
errorLabel
```
 - b. Change the following properties for the TextLabel widget:
 - On the General page, delete the default value of the **text** property that reads "TextLabel".
 - Also on the General page, click the blank button next to the **color** property field, and then click **Name format** and scroll down the list and select **Red**. Click **OK**. This sets the font color for any messages displayed in the label.
 - On the Position page, set the **width** property to 250.
 - On the Layout page, change the **horizontalSpan** property to 2.

- c. Click anywhere in the Design area and press Ctrl+S to save the handler.

The completed interface should look like the following image:



To review the source code, click the **Source** tab at the bottom of the editor pane. The code should match the file contents in “Finished code for MortgageCalculatorHandler.egl after Lesson 4” on page 74.

Related reference

 “Rich UI overview” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

 “Rich UI validation and formatting” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

Lesson checkpoint

You learned how to perform the following tasks:

- Create a Rich UI Handler.
- Create a variable in the EGL Data view.
- Update a user interface by dragging a record variable onto the Rich UI editor.
- Use the EGL Rich UI editor to change an interface.
- Use the Properties view to format the interface.

In the next lesson, you add code to support the interface that you just created.

Lesson 5: Add code to the mortgage calculator handler

Add functions to the handler that you created in the previous lesson.

In this lesson, you work directly with EGL source code, starting with an EGL library that you write. A library can contain constants, variables, and functions, any of which can be accessed by the different units of logic in your application. An important characteristic of a library is that changes to a variable are available to any unit of logic that accesses the library. However, the focus in this tutorial is on functions, which you place in a library to avoid having to maintain the same, widely used logic in multiple places.

To handle some commonplace issues, you can use the EGL Model View Controller (MVC) framework, which is provided by the `com.ibm.egl.rui` project. Although the initials "MVC" typically describe the different components of an enterprise application, the MVC framework in Rich UI concerns only the components of a user interface. Here, the *model* is a variable or record field, the *view* is a widget, and the *controller* is a declaration that oversees the transfer of data between the model and view. That transfer of data is sometimes automatic and is sometimes a response to a function invocation, as shown later.

The drag-and-drop actions of the previous lesson added not only controller declarations, but a *form manager*, which is a declaration that lets you treat other declarations as components of a single form. A form manager includes a set of form fields, each of which can include a label, a controller, and an error field.

Create an EGL library

Create an EGL library to hold a function that provides U.S. currency formatting for a MONEY variable. A more complex version might allow for local currency symbols and separators and different input types. However, for this tutorial, a simple version will suffice.

1. Select the MortgageUIProject in Project Explorer.



2. Click the **New EGL Source File** button.



3. In the New EGL part window, specify the package as follows:
libraries
4. Name the file as follows:
MortgageLib
5. Click **Finish**. The new source file opens in the EGL editor.
6. Remove the comment from the file and add the following code:

```
library MortgageLib
function formatMoney(amount STRING in) returns(STRING)
    len int = strlib.characterLen(amount);
    index int = len - 6; // 2 dec places + decimal + 3 chars before separator
    while(index > 0)
        amount = amount[1 : index] :: "," :: amount[index + 1 : len];
        index -= 3;
        len += 1;
    end
    return("$" :: amount);
end
end
```

As shown, you specify the library name and embed a new function, `formatMoney()`, which adds commas and a dollar sign to an input string. The assumption here is that the input field contains a money value with two decimal places.

7. Save the file. The library is automatically generated once the file is saved.

The `MortgageLib` library is now ready to use.

Change the code in the handler

Before you add new functions, you must make a few minor changes:

1. Click the **Source** tab for `MortgageCalculatorHandler`.
2. Set default values for the **Loan amount** and **Interest** fields. Go to the line where the `inputRec` record is declared and add the two assignments to the declaration, as shown here:

```
inputRec MortgageCalculationResult
    {term = 30, loanAmount=180000, interestRate = 5.2};
```

Save the file and click the **Preview** tab. The interface now shows the default values, including the value that you specified in the previous lesson, for the term field:

Loan amount:

Interest rate:

Term:

\$0.00

3. On the first line of the handler, before the declaration of the ui widget, declare a variable that the code uses to access the dedicated service:

```
mortService MortgageCalculationService{@dedicatedService};
```

```
handler MortgageCalculatorHandler type RUIhandler
{initialUI =[ui], onConstructionFunction = start,
  cssFile = "css/MortgageUIProject.css", title = "MortgageCalculatorHandler" }

mortService MortgageCalculationService{@dedicatedService};

ui GridLayout{columns = 1, rows = 1, cellPadding = 4, children =[inputRec_ui]};
inputRec MortgageCalculationResult{loanAmount = 180000, interestRate = 5.2};
```

4. Hover your cursor over either the red x or the red underline to learn that the cause of the error is as follows: "The type `MortgageCalculationService` cannot be resolved." The error might be puzzling, given that you earlier set the EGL

build path for the project in which you are working so that the project can access the one that contains the service. The problem here is that the source file is not importing the Service part.

5. To remove the error, type the following **import** statement after the other **import** statements in the file:

```
import services.MortgageCalculationService;
```

After a few seconds, the error marks are removed even though you did not save the file.

6. In many cases, the development environment can insert a missing **import** statement for you. To see this convenience in action, remove the **import** statement that you just typed. When you see the error marks return, type Ctrl+Shift+O to reinsert the statement.
7. Save the file.

Complete the inputRec_form_Submit function

EGL created a stub `inputRec_form_Submit` function. The intent is for the function to validate all the fields on the form and to “commit” them. The commit is part of the MVC implementation and means that the `inputRec` record is updated with the values in the widgets.

You will add code to call two other functions. The first function makes the `processImage` image visible and in this way tells the user that the application is working. The second function calls the service to calculate the mortgage payment.

To complete the `inputRec_form_Submit` function, update the `if` statement so that it reads as follows:

```
if(inputRec_form.isValid())
    inputRec_form.commit();
    showProcessImage();
    calculateMortgage();
else
    errorLabel.text = "Input form validation failed.";
end
```

Do not worry about the code format; a later section handles the issue. Also, the next sections remove the error marks that are shown here:

```
function inputRec_form_Submit(event Event in)
    if(inputRec_form.isValid())
        inputRec_form.commit();
        showProcessImage();
        calculateMortgage();
    else
        errorLabel.text = "Input form validation failed.";
    end
end
```

Add the showProcessImage function

You need the `showProcessImage` function to make the `processImage` image visible. To code the function in the Rich UI editor, add the following lines before the final `end` statement in the handler:

```
function showProcessImage()
    processImage.visible = yes;
end
```

Note: The **visible** property is part of any Image widget. You changed the initial value of this property in the previous lesson, when you cleared the **visible** check box for the processImage image.

Add the hideProcessImage function

You need the hideProcessImage function to hide the image when necessary. Add the following lines after the showProcessImage function:

```
function hideProcessImage()
    processImage.visible = no;
end
```

Add the calculateMortgage function

The calculateMortgage function calls a service to do a mortgage calculation based on the values displayed in the UI. To code the function in the Rich UI editor, add the following lines after the hideProcessImage function and ignore the error marks:

```
function calculateMortgage()
    call mortService.amortize(inputRec)
        returning to displayResults
        onException handleException;
end
```

Note:

1. The **call** statement in Rich UI is a variation used only to access services. The runtime communication in this case is asynchronous, which means that the user can continue to interact with the handler while the service is responding.
2. A service requester often passes a record to the service being accessed. In this tutorial, the handler passes the global inputRec variable and receives, in the same record, the value returned from the service.

Add the displayResults function

The displayResults function is a callback function, which runs immediately after the service successfully returns business data to the Rich UI handler. To code the function, add the following lines after the calculateMortgage function:

```
function displayResults(retResult MortgageCalculationResult in)
    paymentLabel.text = MortgageLib.formatMoney(retResult.monthlyPayment as STRING);
    inputRec_form.publish();
    hideProcessImage();
end
```

Note:

- You use the formatMoney function from your new EGL Library part to add commas and a dollar sign to the payment amount. Because you created the paymentLabel variable without involving the MVC framework, you must handle the formatting yourself.
- As noted earlier, a form manager includes form fields that in turn can include controllers and other declarations. The publish function is available in any form manager and invokes the controller-specific publish functions, one after the next, to do as follows:
 1. Retrieve the data from the variable that serves as the controller model.
 2. Format that data.

3. Write the formatted data to the widget that serves as the controller view. Given that sequence of events, the form-level `publish` function is often invoked as it is here: in a callback function that receives data from a service.

Write the exception handler

If the service invocation results in an error, the usual callback function is not invoked. However, if you arranged for the use of an exception handler, as in this case, that function is invoked.

Do as follows:

1. After the `displayResults` function, add the following code:

```
// catch-all exception handler
private function handleException(ae AnyException in)
  errorLabel.text = "Error calling service: " + ae.message;
end
```

Functions with the **private** modifier can only be called by the EGL part where the function resides; in this case, by the embedding handler. The function places text in the `errorLabel` widget that you created in the previous lesson.

2. Update the `calculateMortgage` function immediately before the **call** statement, as follows:

```
errorLabel.text = "";
```

In this way, you clear the `errorLabel` field before you invoke the service that does a mortgage calculation. If you do not add that code, an error such as the temporary loss of service connection will display an error message, even after the service is invoked successfully.

3. Right-click an empty space in the editor. Click **Organize imports**. EGL adds **import** statements for all the undefined symbols that it can.
4. Save the file. If you see errors in your source file, compare your code to the file contents in “Finished code for `MortgageCalculatorHandler.egl` after Lesson 5” on page 77.
5. After you resolve any errors, reformat your input by pressing `Ctrl+Shift+F` and save the file again.

Test the calculator

You are now ready to test the calculator.

1. Change to Preview view by clicking the **Preview** tab at the bottom of the editor. You can fully test your application in the Preview view, including the user interface and services. You can do your testing whether the services are deployed on a server or are available only as EGL source code. EGL displays the default values that you entered when you created the handler.
2. Click **Calculate**. EGL displays the monthly payment.

Loan amount:

Interest rate:

Term:

\$988.40

3. Type a letter in the first field. As is true of several EGL Dojo widgets, a red mark is displayed as soon as an error occurs, and an error message is displayed in an adjacent tooltip.

Loan amount:  

Interest rate:

Term:

\$988.40

When the widget loses focus, the tooltip is hidden, and when the widget regains focus, the tooltip is shown.

4. Change the values for any of the three fields and click **Calculate** again. The **Payment** field changes accordingly.

Related reference

 "Rich UI validation and formatting" at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

 "Form processing with Rich UI" at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

Lesson checkpoint

You learned how to complete the following tasks:

- Work in the Rich UI editor **Source** tab
- Create an EGL Library part for reusable functions
- Use EGL Model-View-Controller framework
- Call an EGL Service from a function
- Catch and handle errors

In the next lesson, you create a pie chart to compare the total principal to the total interest in a given calculation.

Lesson 6: Create the calculation results handler

The next handler that you create, `CalculationResultsHandler`, creates a pie chart to illustrate details that are issued by the previously created logic, `MortgageCalculatorHandler`.

The code that acts as an intermediary between the two handlers is an *Infobus*, which is an EGL library in the `com.ibm.egl.rui` project.

The Infobus works as follows:

- A handler such as `CalculationResultsHandler` subscribes to an event of a specified name. At the time of subscription, the handler also gives the name of a function that will receive data when the specified event occurs. As a result of this subscription, the Infobus registers the function, maintaining the detail necessary to invoke the function later.
- At the right moment, the same or a different handler publishes the event. This handler specifies both the event name and event-specific data and directs the Infobus to invoke the registered function.

You begin this lesson by dealing with the second of those two steps. You update the previously written `MortgageCalculatorHandler` handler to invoke the Infobus `publish` function when a new calculation is returned from the remote service. Then, you ensure that the `CalculationResultsHandler` handler has subscribed to the event.

The publish-and-subscribe makes possible the pie-chart display.

Publish the service results

1. Find the `displayResults()` function that you created in the previous lesson. Add the following line before the **end** statement:

```
InfoBus.publish("mortgageApplication.mortgageCalculated", retResult);
```

The first argument is the event name, and the second is the record being passed to the functions that are registered for that event. Recall that the structure of that record is as follows:

```
record MortgageCalculationResult
    // user input
    loanAmount money;
    interestRate decimal(10, 8);
    term int;

    // calculated fields
    monthlyPayment money;
    interest money;
end
```

An error mark is displayed because the Infobus library is not being imported. To add the required **import** statement, press `Ctrl+Shift+O`. To remove the error mark, save the file.

The `displayResults()` function now looks as follows:

```

function displayResults(retResult MortgageCalculationResult in)
    paymentLabel.text = MortgageLib.formatMoney(retResult.monthlyPayment as STRING);
    inputRec_form.publish();
    hideProcessImage();
    InfoBus.publish("mortgageApplication.mortgageCalculated", retResult);
end

```

As before, this code shows the payment amount in the payment field and uses the Rich UI MVC mechanism to publish the results of the calculation in the `retResult` record. The new statement involves a different kind of publish, making the record available to any widget that subscribes to the `mortgageApplication.mortgageCalculated` event.

Note: The Infobus event names are case-sensitive. For example, “mortgageApplication” is different from “MortgageApplication.”

2. Save and close the file.

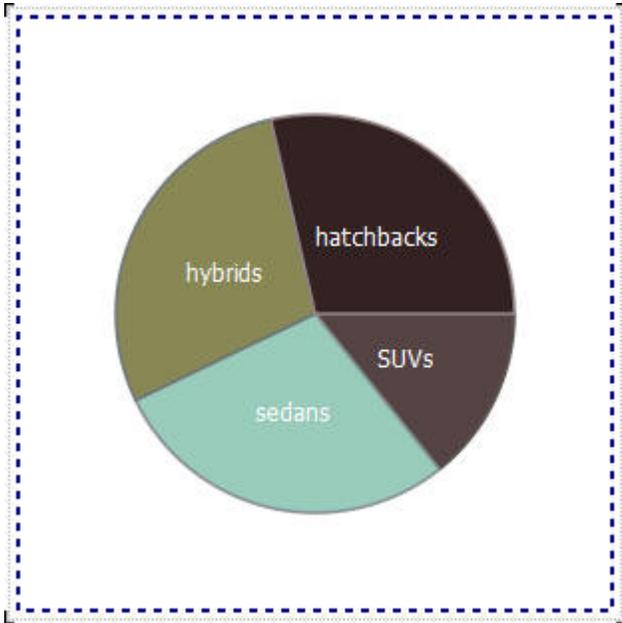
Related reference

 “Rich UI Infobus” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

Create the CalculationResultsHandler handler

1. In the MortgageUIProject project, right-click the **handlers** package and click **New > Rich UI handler**. These actions ensure that the **New EGL Rich UI Handler** page will reference the package.
2. Specify the source file name as `CalculationResultsHandler` and click **Finish**. The handler opens in the Design view of the Rich UI editor.
3. As you did when you coded the calculator, reduce the size of the initial `GridLayout` widget to a single cell. On the General page of the Properties view, change the **rows** property to 1 and the **columns** property to 1.
4. Drag a `PieChart` widget from the **Visualization** drawer of the palette onto the single cell of the grid layout and give the widget the following name:
`interestPieChart`

EGL displays a default pie chart.



5. At the bottom of the editor, click the **Source** tab.
6. In the interestPieChart widget declaration, change the height property to 250.
7. You need only two sections in the pie chart. In the interestPieChart declaration, in the data field, replace the four lines that declare PieChartData records. Here is the new code:

```
new PieChartData{ y=1, text="Principal", color="#99ccbb"},
new PieChartData{ y=0, text="Interest", color="#888855"}
```

To calculate how much of the pie chart is taken by a given record, divide the y-field value for that record by the sum of y-field values. In this case, the division is 1/1, and the initial display shows the mortgage principal at 100%. The display at development time is only a placeholder until the application handles the first, default calculation at run time.

8. Subscribe to the Infobus event mentioned earlier by adding the following line to the start function:

```
InfoBus.subscribe("mortgageApplication.mortgageCalculated", displayChart);
```

This code ensures that the Infobus invokes the displayChart function whenever the specified event occurs. The next step will remove the error marks.

9. After the start function, add the displayChart function as follows and then organize the import statements by typing Ctrl+Shift+O:

```
function displayChart(eventName STRING in, dataObject ANY in)
    localPieData PieChartData[2];

    resultRecord MortgageCalculationResult =
        dataObject as MortgageCalculationResult;
    localPieData = interestPieChart.data;
    localPieData[1].y = resultRecord.loanAmount;
    localPieData[2].y = resultRecord.interest;
    interestPieChart.data = localPieData;
end
```

When the event occurs, the `displayChart` function receives the input data into the `dataObject` parameter. The use of `ANY` as the parameter type ensures that you can use the InfoBus mechanism to transfer any type of record.

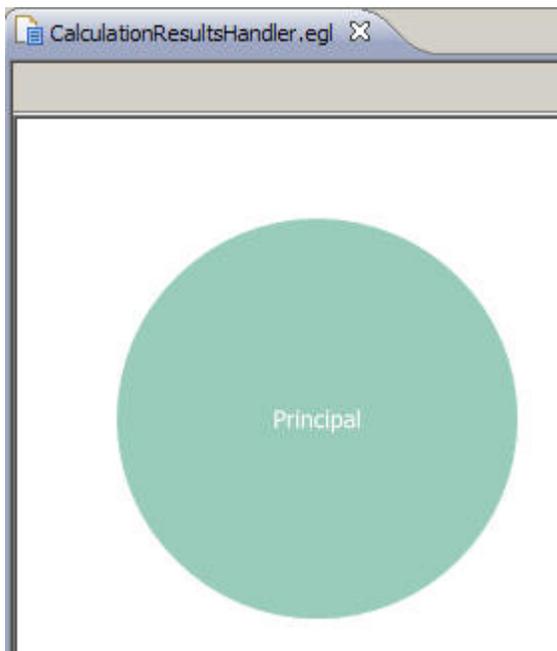
Next, the function acts as follows:

- Creates the `localPieData` array, which is of type `PieChartData[]`, as is appropriate for the `data` property of the pie chart.
 - Assigns the received value to a record of type `MortgageCalculationResult`, in a statement that casts the second input parameter to the data type that is appropriate to your use of the InfoBus:

```
resultRecord MortgageCalculationResult =
    dataObject as MortgageCalculationResult;
```
 - Assigns the `data` property of the pie chart, including color detail, to the new `localPieData` array.
 - Assigns the received loan amount and interest value to that array.
 - Forces the pie chart to refresh by updating its `data` property. Specifically, you assign the `localPieData` array to that property.
10. Save the file. If you see errors in your source file, compare your code to the file contents in “Finished code for `CalculationResultsHandler.egl` after Lesson 6” on page 80.

Test the pie chart

1. Change to Preview view. EGL displays a default pie chart showing 100% principal.



2. Close the file.

Lesson checkpoint

You learned how to complete the following tasks:

- Use the InfoBus library to pass information between handlers.
- Create a pie chart.

In the next lesson, you create the main handler, which uses the others.

Lesson 7: Create the main Rich UI handler

The main page uses the EGL portal widget to manage communication between different handlers.

As noted in an earlier lesson, Rich UI gives a new meaning to the long-standing notion of Model View Controller (MVC), which is redefined specifically for logic that runs in the browser. Similarly, Rich UI gives a new meaning to the words *portal* and *portlet*.

In general, a portal is a web page that controls independent UI components called portlets. In the traditional use of these terms, a portal is server-side code. The portlets embedded by the portal are web-page snippets, each of which might be stored in a different remote location. The web page is constructed on the server where the portal code resides, and the completed web page is transferred from the server to the browser.

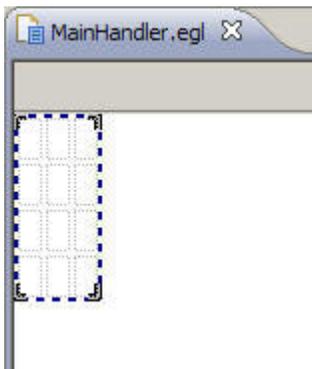
In contrast, a Rich UI portal is a widget that runs in the browser and that references a set of portlet widgets, each of which references a Rich UI handler. The next sections demonstrate how to code a portal and portlets in Rich UI.

Create the MainHandler handler

1. Create a new Rich UI Handler in the **handlers** package of the MortgageUIProject project, as you did in the previous lesson. The handler name in this case is MainHandler. The Handler opens in the Design view of the Rich UI editor.
2. If you do not see the **Samples** drawer in the Palette view, click the **Refresh palette** button to the left of the Palette view.



3. Select the initial GridLayout widget that was created for the handler. Make sure the entire widget is surrounded by the dotted line.



4. Press the Delete key to remove the widget.

- From the **Samples** drawer of the palette, drag a portal widget onto the editor and give it the following name:
mortgagePortal
- At the bottom of the editor, click the **Source** tab.
- In the mortgagePortal declaration, change the number of columns to 2, and set the column widths to 350 and 650. The content of the file is as shown here:

```

package handlers;

// RUI Handler

import com.ibm.egl.rui.widgets.GridLayout;
import utils.portal.Portal;

//
//
handler MainHandler type RUIhandler {initialUI = [ mortgagePortal ],
    onConstructionFunction = start,
    cssFile="css/MortgageUIProject.css", title="MainHandler"}

    mortgagePortal Portal{ columns = 2, columnWidths = [ 350, 650 ] };

function start()
end
end

```

- After the mortgagePortal declaration, skip a line and add the following declarations:

```

calculatorHandler MortgageCalculatorHandler{};
resultsHandler CalculationResultsHandler{};

```

These statements declare two variables, each of which is based on a Handler part; in this case, a Handler part developed in this tutorial.

- Skip a line and add the following code:

```

calculatorPortlet Portlet{children = [calculatorHandler.ui],
    title = "Calculator"};
resultsPortlet Portlet{children = [resultsHandler.ui],
    title = "Results", canMove = TRUE, canMinimize = TRUE};

```

Each new portlet variable is declared with a reference to the initial UI in the embedded handler.

- To remove error marks, press Ctrl+Shift+O.
- Code the start function as follows:

```

function start()
    mortgagePortal.addPortlet(calculatorPortlet, 1);
    mortgagePortal.addPortlet(resultsPortlet, 1);

    // Subscribe to calculation events
    InfoBus.subscribe("mortgageApplication.mortgageCalculated", restorePortlets);

    // Initial state is minimized
    resultsPortlet.minimize();
end

```

The code acts as follows:

- Adds the previously declared portlets to the portal: one to show the calculator, and another to show the pie chart.

- Ensures that the main handler is subscribed to the same event as CalculationResultsHandler, in this case to ensure that the restorePortlets function runs when the remote service returns a mortgage calculation.
 - Minimizes the second portlet so that at run time, the pie chart is not visible initially.
12. After the start function, add the function that will be invoked when the service returns a calculation:

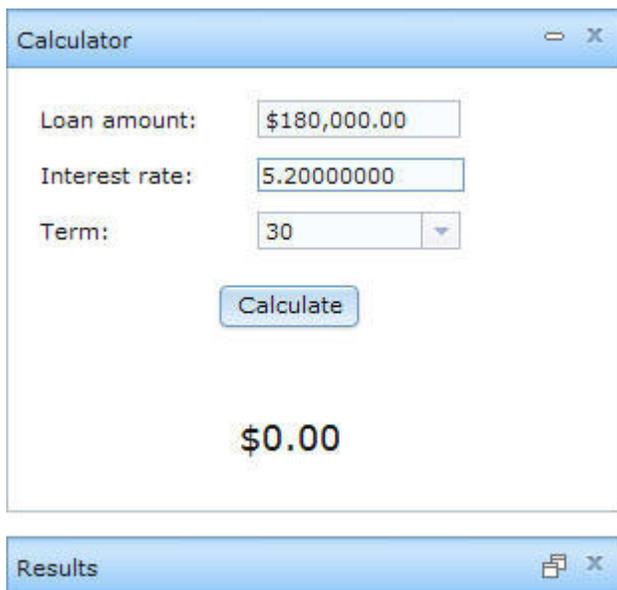

```
function restorePortlets(eventName STRING in, dataObject ANY in)
  if(resultsPortlet.isMinimized())
    resultsPortlet.restore();
  end
end
```

The portlet-specific restore function causes the pie chart to be displayed.
 13. To remove the error marks, press Ctrl+Shift+O, and save the file. If you see errors in your source file, compare your code to the file contents in “Finished code for MainHandler.egl after Lesson 7” on page 80.

Test the portal

Test the main portal to make sure that the results portlet receives changes from the calculation portlet.

1. At the bottom of the editor, click **Preview**. EGL displays the main handler, where the portal is declared. The handler displays the two subordinate portlets.



2. Click **Calculate**. The animated image indicates that processing is in progress. When the calculation finishes, the pie chart is displayed.

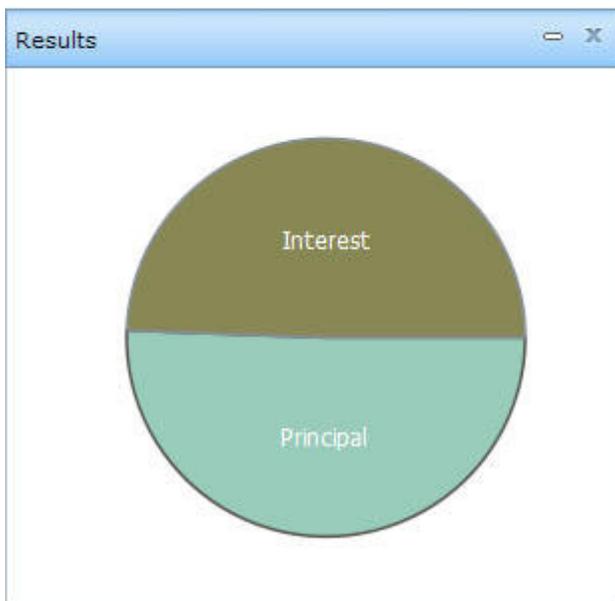
Calculator

Loan amount:

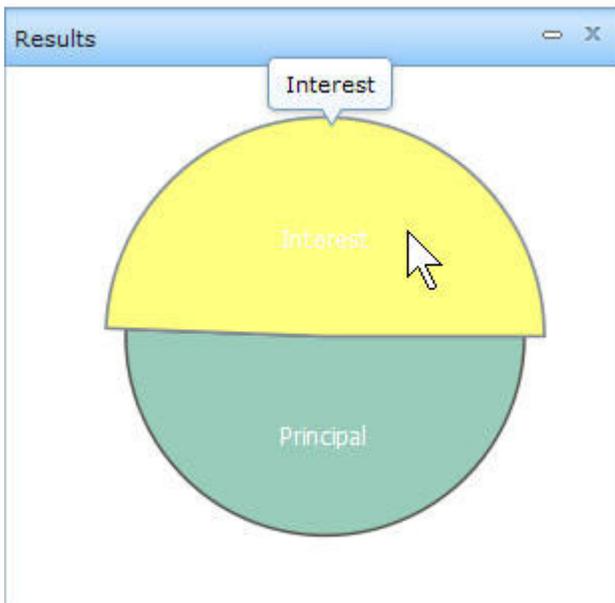
Interest rate:

Term:

\$988.40



3. Move your cursor over one of the pie chart sections for an expanded view.



4. Change any of the calculation values and click **Calculate** again. The pie chart reflects the changes in the proportion of principal to interest.
5. Close the file.

Lesson checkpoint

You learned how to complete the following tasks:

- Create a portal widget.
- Add portlets to the portal and, in this way, make available the handlers that you created in previous lessons.

In the next lesson, you add a portlet to list your mortgage calculations.

Lesson 8: Create the calculation history handler

Create a table in which you can click a row to display a previous calculation.

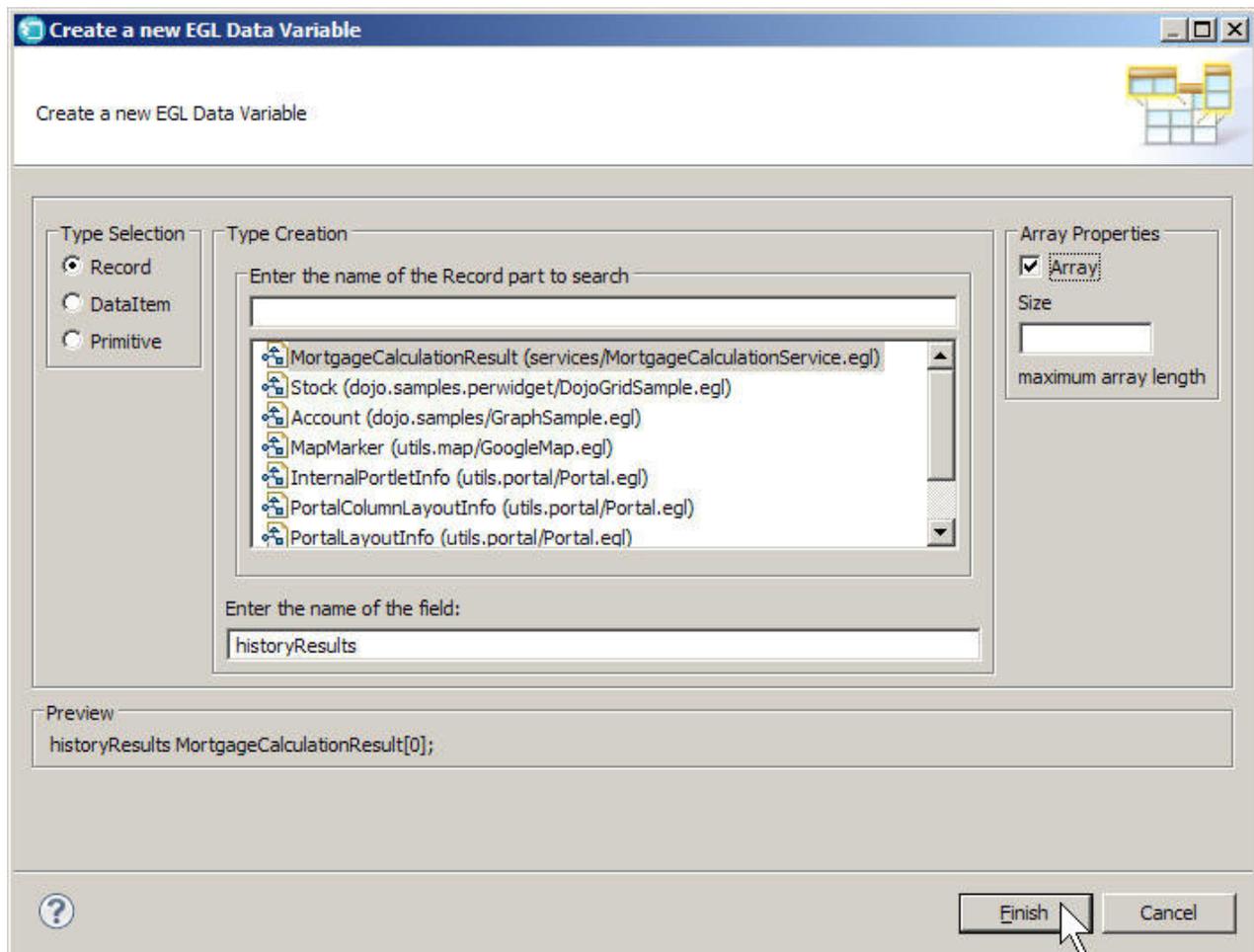
In this lesson, you use the DataGrid widget to create a table. The DataGrid widget has advanced capabilities for interaction and visual presentation that make it preferable to the GridLayout widget for displaying an array of records.

In lesson 4, you dragged a record variable onto the editor to create a GridLayout widget. In this lesson, you drag an array of records onto the editor, which by default creates a DataGrid widget.

Create the handler

1. In the **handlers** package, create a Rich UI handler named CalculationHistoryHandler. The Handler opens in the Design view of the Rich UI editor.
2. Delete the default GridLayout widget.
3. Create a variable to hold an array of MortgageCalculationResult records.

- a. Right-click the background of EGL Data view, and then click **New > EGL Variable**.
- b. In the Create a new EGL Data Variable wizard, in the **Type Creation** section, select the **MortgageCalculationResult** record, as you did in Lesson 4.
- c. For **Enter the name of the field**, enter the following name:
historyResults
- d. Under **Array Properties**, select the **Array** check box. Do not specify a size.

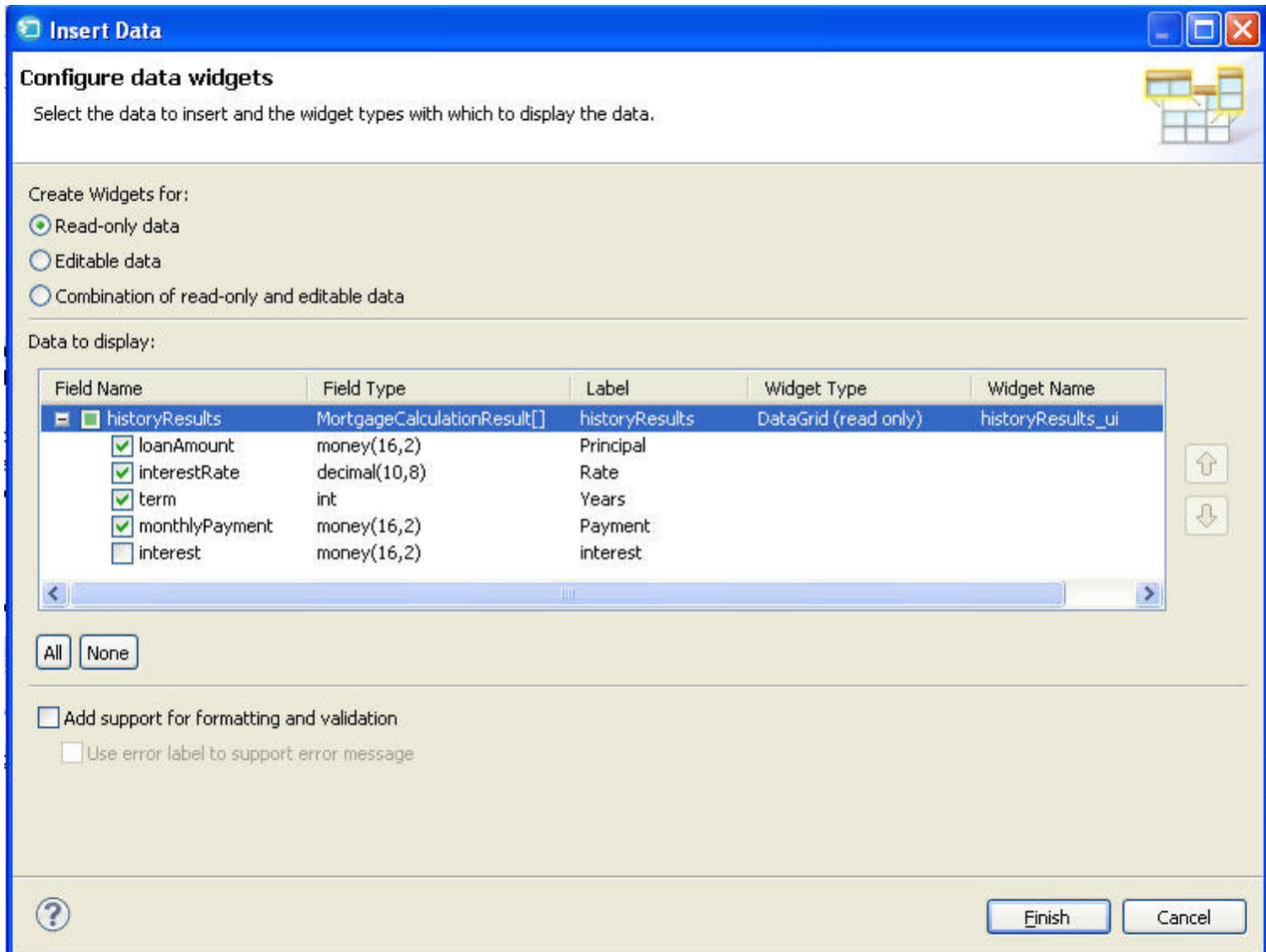


- e. Click **Finish**.
4. Drag the new variable to the Display surface in the Rich UI editor. EGL displays the Insert Data wizard. This wizard is the same wizard that you saw in Lesson 4, though with different defaults because the variable that you dragged onto the editor is a dynamic array.
 5. Make the following changes in the Insert Data wizard:
 - a. Under **Create Widgets for**, leave the default value of **Read-only data**.
 - b. Clear the check box for the **interest** field.
 - c. Change the labels for the remaining fields as follows:
 - Change **loanAmount** to Principal.
 - Change **interestRate** to Rate.
 - Change **term** to Years.

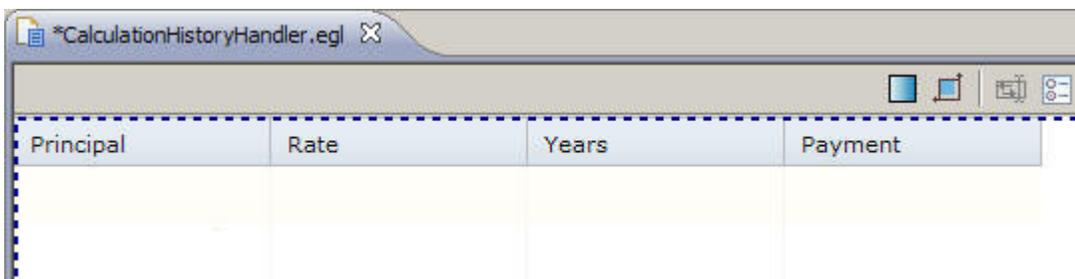
- Change **monthlyPayment** to **Payment**.

The wizard uses these labels as column headers for the grid.

- d. Clear the **Add support for formatting and validation** check box. The completed wizard looks like the following image:



- e. Click **Finish**. The web page looks as follows.



You will code the remainder of the calculation history handler in Source view.

6. At the bottom of the editor, click the **Source** tab.
7. In the declaration for the `historyResults_ui` `DataGrid` widget, add the following content before the `columns` property:


```
selectionMode = DataGridLib.SINGLE_SELECTION,
```

The specified value ensures that the user can select only one row of the grid rather than multiple rows.

8. On the line after you set `selectionMode`, type the following code:

```
selectionListeners ::= cellClicked,
```

You just updated a *listener property*, which takes an array of functions that run in array-element order. In particular, you appended a function to the array of functions associated with the `selectionListeners` property. You will code the new function later in this lesson.

The listener functions run in response to a user action, such as a click or, in some cases, in response to a function call that selects or deselects a row or that updates a check box.

9. Change the default widths of the columns so they will fit in the smaller portlet window:

- Set the width of the Principal column to 80.
- Set the width of the Rate column to 80.
- Set the width of the Years column to 50.
- Set the width of the Payment column to 70.

10. After each of the width values you just specified, in the same set-values block (the area with the curly brackets), set an **alignment** property to right-align the numbers in each column:

```
, alignment = DataGridLib.ALIGN_RIGHT
```

For example, the declaration for the Principal column now looks like the following code:

```
new DataColumn {name = "loanAmount", displayName = "Principal", width = 80,  
                alignment = DataGridLib.ALIGN_RIGHT},
```

11. Add the **formatter** property to three of the column declarations, as follows:
 - a. For the Principal column, reference the custom `formatDollars` function, which you will write later in this lesson:

```
, formatters = [ formatDollars ]
```

The entire declaration now looks like the following code:

```
new DataColumn {name = "loanAmount", displayName = "Principal", width = 80,  
                alignment = DataGridLib.ALIGN_RIGHT, formatters = [ formatDollars ]},
```

- b. Add the following code for the Rate column:

```
, formatters = [ DataGridFormatters.percentage ]
```
 - c. You do not need special formatting for the Years column.
 - d. Add the following code for the Payment column:

```
, formatters = [ formatDollars ]
```

The code now has the following content:

```

handler CalculationHistoryHandler type RUIHandler {initialUI = [ historyResults_ui ],
    onConstructionFunction = start,
    cssFile="css/MortgageUIProject.css",
    title="CalculationHistoryHandler"}

historyResults MortgageCalculationResult[0];

historyResults_ui DataGrid {
selectionMode = DataGridLib.SINGLE_SELECTION,
selectionListeners := cellClicked,
columns = [
    new DataGridColumn {name = "loanAmount", displayName = "Principal", width = 80,
        alignment = DataGridLib.ALIGN_RIGHT, formatters = [ formatDollars ]},
    new DataGridColumn {name = "interestRate", displayName = "Rate", width = 80,
        alignment = DataGridLib.ALIGN_RIGHT, formatters = [ DataGridFormatters.percentage ]},
    new DataGridColumn {name = "term", displayName = "Years", width = 50, alignment = DataGridLib.ALIGN_RIGHT},
    new DataGridColumn {name = "monthlyPayment", displayName = "Payment", width = 70,
        alignment = DataGridLib.ALIGN_RIGHT, formatters = [ formatDollars ]}
],
data = historyResults as any[]
};

function start()
end
end

```

In general, the **formatters** property takes an array of function names. The functions can be predefined, or you can write custom functions. For example, the percentage function is provided in the DataGridFormatters library that is included in the com.ibm.egl.rui.widgets project.

12. Add the following code to the start function:

```
InfoBus.subscribe("mortgageApplication.mortgageCalculated", addResultRecord);
```

As before, you use the InfoBus to invoke a function when the service returns a new calculation.

13. Add the addResultRecord function after the start() function:

```
// Update the grid to include the latest mortgage calculation
function addResultRecord(eventName STRING in, dataObject ANY in)
    resultRecord MortgageCalculationResult = dataObject as MortgageCalculationResult;
    historyResults.appendElement(resultRecord);
    historyResults_ui.data = historyResults as ANY[];
end
```

Here, you cast an incoming value to a MortgageCalculationResult record. You then append the new results to array of results and update the **data** property. That update causes the widget to refresh.

14. Add the following listener function:

```
// Publish an event to the InfoBus whenever the user selects an old calculation
function cellClicked(myGrid DataGrid in)
    updateRec MortgageCalculationResult = myGrid.getSelection()[1]
        as MortgageCalculationResult;
    InfoBus.publish("mortgageApplication.mortgageResultSelected", updateRec);
end
```

The function retrieves the data-grid row selected by the user and provides that row to the InfoBus. The InfoBus in turn invokes a function in any handler that has subscribed to the event named "mortgageApplication.mortgageResultSelected."

15. Add the following function to format monetary amounts:

```
function formatDollars(class string, value string, rowData any in)
    value = mortgageLib.formatMoney(value);
end
```

The value of the second parameter is available to the EGL runtime code because the parameter modifier is **InOut** by default.

Note that you are reusing the `formatMoney` function from the `mortgageLib` library.

16. Reformat the file by pressing `Ctrl+Shift+F`. Then, remove the error marks by pressing `Ctrl+Shift+O`, and save the file. If you see errors in your source file, compare your code to the file contents in “Finished code for `CalculationHistoryHandler.egl` after Lesson 8” on page 81.
17. Close the file.

Lesson checkpoint

You learned how to complete the following tasks:

- Drag and drop an array of records to create a data grid.
- Trigger an event when a cell of the data grid is clicked.
- Format columns in the data grid.

In the next lesson, you integrate this handler with the rest of the application.

Lesson 9: Embed the calculation history handler in the application

To add the calculation history handler to your page, you must change the results portlet and the main portal.

Change the results portlet

As of the end of the previous lesson, the `CalculationResultsHandler` handler subscribes to a single event: `mortgageApplication.mortgageCalculated`. When that event occurs, the handler updates and re-displays the pie chart. However, the user might select a row in the history portlet and cause a different event to be published: `mortgageApplication.mortgageResultSelected`. If `CalculationResultsHandler` subscribes to that event, too, the handler can respond to the user's selection in the same way, by updating and re-displaying the pie chart. The simplest way to subscribe to both events is to use the asterisk (*), which is a wildcard character that represents any event in a set of events. Do as follows:

1. In the Rich UI editor, open the `CalculationResultsHandler.egl` file and switch to the Source view.
2. In the `start()` function, find the following line:

```
InfoBus.subscribe("mortgageApplication.mortgageCalculated", displayChart);
```
3. Replace the lowest-level qualifier in the event name with the asterisk:

```
InfoBus.subscribe("mortgageApplication.*", displayChart);
```

EGL now calls the `displayChart` function whenever an event occurs if the name of the event begins with `mortgageApplication..`

4. Save and close the file.

Change the main portal

For the history portlet, add lines that are similar to the lines for the other two portlets:

1. In the Rich UI editor, open the `MainHandler.egl` file and click the **Source** tab.
2. Immediately below the `resultsHandler` declaration, add a similar declaration for `historyHandler`:

```
historyHandler CalculationHistoryHandler{};
```

3. Immediately below the resultsPortlet declaration, add a similar declaration for historyPortlet:

```
historyPortlet Portlet{children = [historyHandler.historyResults_ui],  
    title = "History", canMove = TRUE, canMinimize = TRUE};
```

4. In the start function, below the existing calls to addPortlet, add the new portlet to the portal:

```
mortgagePortal.addPortlet(historyPortlet, 1);
```

5. As you did with resultsPortlet, set historyPortlet to be minimized initially:

```
historyPortlet.minimize();
```

6. Add code for historyPortlet to the end of restorePortlets() function:

```
if(historyPortlet.isMinimized())  
    historyPortlet.restore();  
end
```

7. Save the file. If you see errors in your source file, compare your code to the file contents in “Finished code for MainHandler.egl after Lesson 9” on page 82.

Test the portal

Test the main portal to make sure that the new history portlet is displayed and works correctly.

1. At the bottom of the editor, click **Preview**. EGL displays the main portal and the three subsidiary portlets.
2. Click **Calculate**. An animated image indicates that work is in progress. When the calculation finishes, the pie chart and history are displayed.

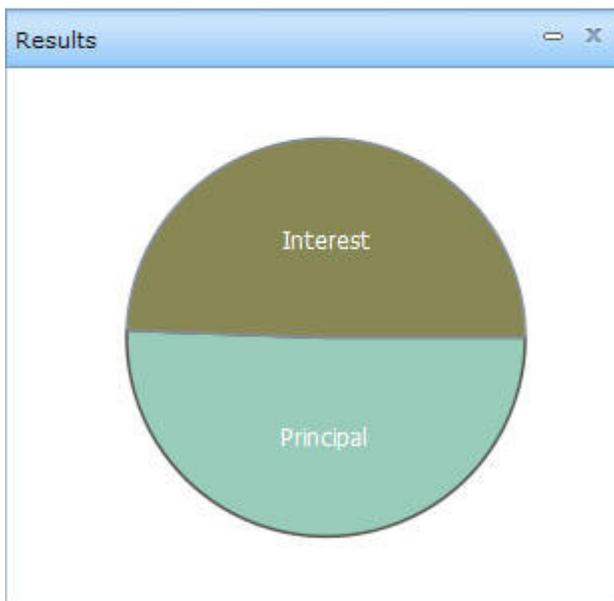
Calculator

Loan amount:

Interest rate:

Term:

\$988.40



History

Principal	Rate	Years	Payment
\$180,000.00	5.20000000%	30	\$988.40

3. Change the term of the mortgage to 5 years and click **Calculate** again. A row is added to the history list.
4. Click a cell in the first row of the history list.

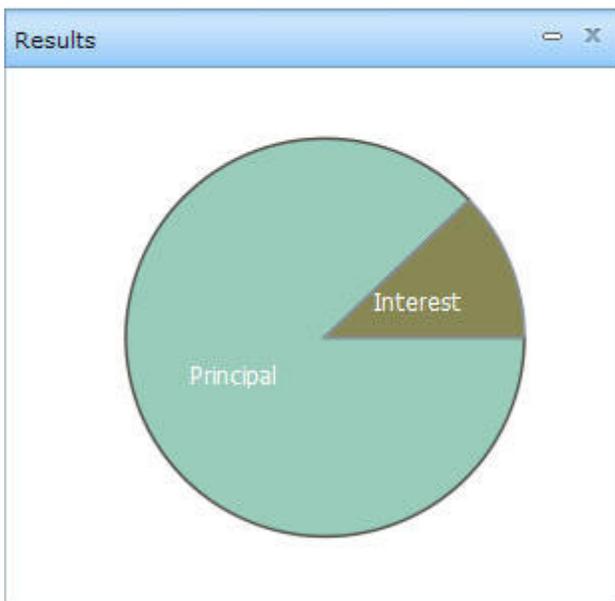
Calculator

Loan amount:

Interest rate:

Term:

\$3,413.34



History

Principal	Rate	Years	Payment
\$180,000.00	5.20000000%	30	\$988.40
\$180,000.00	5.20000000%	5	\$3,413.34

5. The pie chart displays the values for the row selected in the history list.

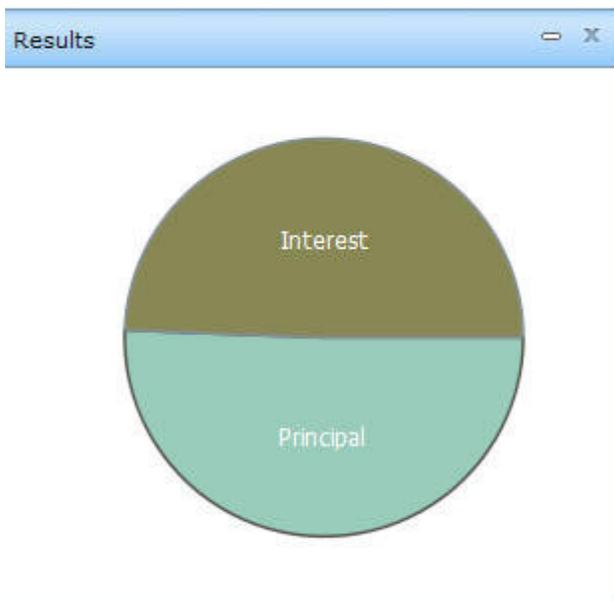
Calculator

Loan amount:

Interest rate:

Term:

\$3,413.34



History

Principal	Rate	Years	Payment
\$180,000.00	5.20000000%	30	\$988.40
\$180,000.00	5.20000000%	5	\$3,413.34

Lesson checkpoint

You learned how to subscribe to multiple, similarly named events.

In the next lesson, you add a portlet to display a map of mortgage companies that are in a specified area of the United States.

Lesson 10: Create the map locator handler

Begin to create a portlet where you can enter a mortgage code and see a list of mortgage places and a map. Click the name of a place in the list, and the map displays a location marker

This lesson relies on capabilities from two external web sites:

- The Google Places Service provides information about businesses in or near a specific place.
- The Google Maps API provides a map that you can embed in your UI to show the location of a business.

If now or later you would like to learn more about some technologies used in lessons 10 and 11, see the following sources:

- “REST for the developer” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v9r0m0>.
- “Correspondence between an XML string and an EGL variable” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/9r0m0>.
- Google Places API (<https://developers.google.com/places/documentation/details>).
- “Understanding how browsers handle a Rich UI application” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v9r0m0>.

Create records for the Interface file

To use the Google Places Service, you will create the following EGL parts:

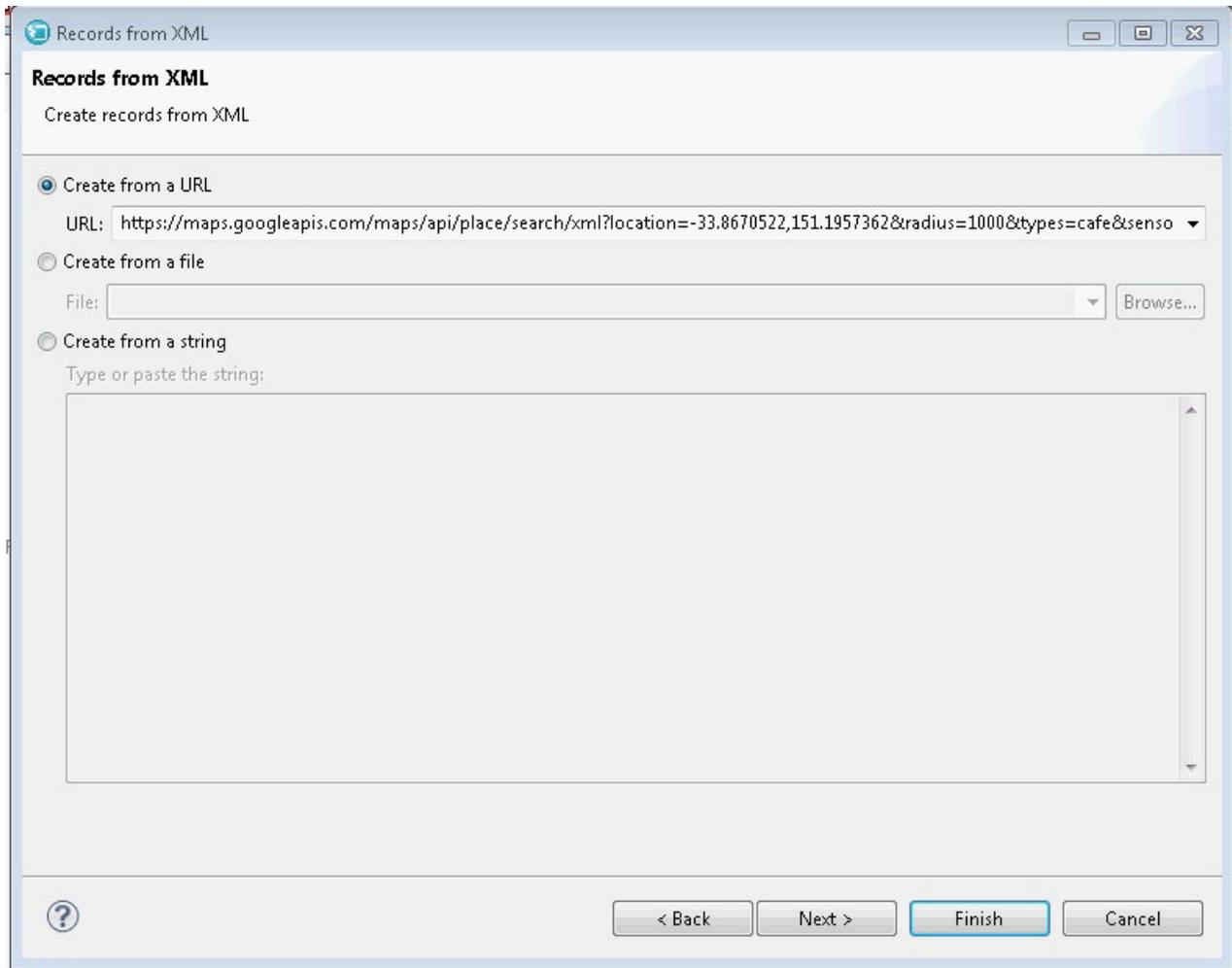
- A set of Record parts. Each definition is the basis of a variable that will be used to receive data from the service.
- An Interface part. This definition is the basis of a service-access variable, which is used in the **call** statement that invokes the service.

You can create the Record parts in various ways, but in this lesson you will access a REST service on the web and include, in the web address, the details necessary to retrieve data from the service. The New EGL Record wizard will create the Record parts that correspond to the data that is retrieved at development time. Do as follows:

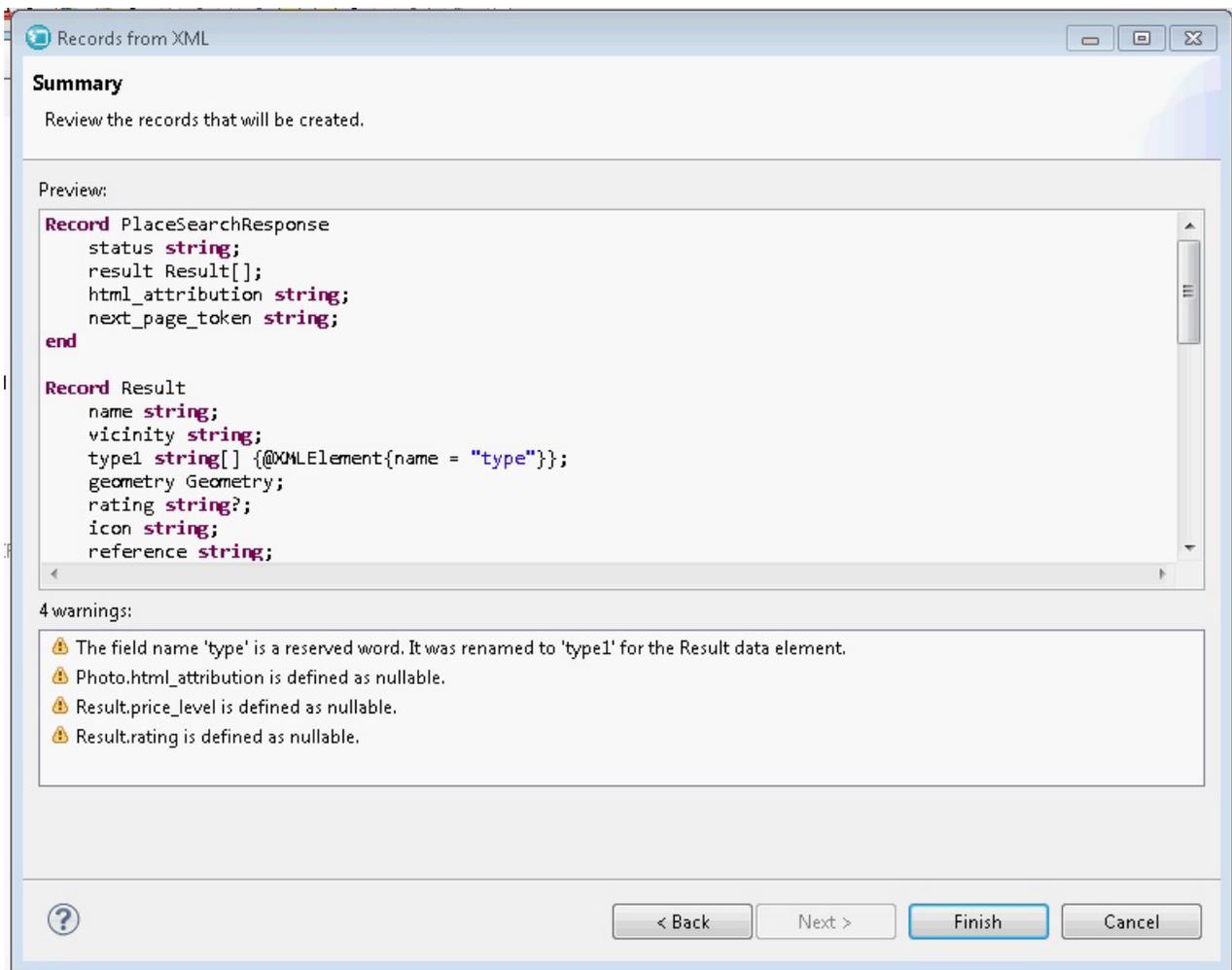
1. In the MortgageServiceProject project, in the EGLSource folder, right-click the services package and click **New > Record**.
2. In the first page of the New EGL Record wizard, accept the details about the source folder and package and type the following name for the new source file:
GooglePlaceRecords
3. Click **Next**.
4. In the Templates page, click **Records from XML**. Click **Next**.
5. In the Records from XML window, click **Create from a URL** and paste the following URL into the **URL** field:

```
https://maps.googleapis.com/maps/api/place/search/xml?location=-33.8670522,151.1957362  
&radius=1000&types=cafe&sensor=false&key=AIzaSyD_K9zveT6jhXgCApduywa0TuD5FiQFgpI&language=en
```

Combine the two lines into a single-line URL with no spaces.



6. Click **Next**. The wizard displays a Summary page that previews the code it will place in the new file.



Note: The Google API Service sometimes returns no data, in which case only the ResultSet Record part is created. The lack of data is most likely on Sundays. If the service is unavailable or does not return data, click **Cancel** and wait for a later time to complete the tutorial.

7. Click **Finish**, which saves the file.
8. If you see errors in your source file, compare your code to the file contents in .
9. Close the file.

Create the Local Search Interface

When you use an external web service, you create an Interface part that identifies the service operations that will be accessed. The Interface part is used by the requesting code and is not a component of the service itself.

Do as follows:

1. Create a new Interface part by right-clicking **MortgageUIProject** and clicking **New > Interface**.
2. In the New EGL Interface Part window, complete the following fields:
 - a. In the **Package** field, enter the following name:
interfaces
 - b. In the **EGL source file name** field, enter the following name:
GooglePlaceRecords

c. Click **Finish**.

3. Replace the contents of the file with the following code:

```
package interfaces;

// interface
interface GooglePlacesService
function getSearchResults( typeName string? in )
returns(PlaceSearchResponse)
  {@GetRest{uriTemplate =
  "https://maps.googleapis.com/maps/api/place/search/xml?location
  =37.47,-122.26&radius=50000&sensor=false&key=AIzaSyD_K9zveT6jhx
  gCApduywa0TuD5FiQFgpI&language=en&keyword={typeName}",
  responseFormat = XML}}; end
```

After you paste the code, do as follows:

- Remove extra lines so that the `uriTemplate` value is on a single line, without spaces.
- Press Ctrl-Shift-O to include the **import** statement necessary to resolve the reference to the `PlaceSearchResponse` Record part.
- Save the file.

The `getSearchResults` function prototype ensures that when the requester invokes the service, the requester's type argument is used in place of the bracketed elements in the `uriTemplate` value. The EGL runtime code uses the completed URI to access the service. The URI specifies that the service is to return 10 results, at most:

- The URI includes the keyword "mortgage," which is used by the service to search for data.
 - The URI specifies that the service is to return 10 results, at most.
- If you see errors in your source file, compare your code to the file contents in "Finished code for `GooglePlacesService.egl` after Lesson 10" on page 84.
 - In the absence of errors, close the file.

Related tasks

 "Creating an Interface part to access a REST service" at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v9r0m0>

Create the `MapLocatorHandler` handler

To create the `MapLocatorHandler` handler:

- In the `MortgageUIProject/EGLSource` folder, in the **handlers** package, create a Rich UI Handler part as you did in lesson 4. Give the handler the following name:

```
MapLocatorHandler
```

The handler opens in the Design view of the Rich UI editor.

- Click inside the grid layout, right click the cell you selected, and click **Delete > Row**. The grid layout has three rows.
- Create a line of introductory text:
 - From the **Display and Input** drawer of the palette, drag a `TextLabel` widget to the first cell of the `GridLayout` widget and give it the following name:

```
introLabel
```
 - In the Properties view, make the following changes:
 - On the General page, change the **text** property to the following phrase:
Search for places in San Francisco:

- On the Layout page, set the **horizontalSpan** property to 3.
 - To save the file, press Ctrl-S.
4. Create a label for the type code input field:
 - a. Drag a TextLabel widget into the first cell of the second row and assign the following name:
typeLabel
 - b. In the Properties view, set the **text** property as follows:
Type:
 5. Create a text field where the user can enter a type code:
 - a. Drag a DojoComboBox widget into the second cell of the second row and assign the following name:
typeComboBox
 - b. In the Properties view, on the Position page, set the **width** property to 100.
 - c. On the Events page, click the row for the onChange event. Click the plus sign (+) to add a function for the event. The New Event Handler dialog is displayed.
 - d. On the Properties page, click the “...” button around the values, and assign below options to it in the popped dialog. ["bar", "food", "restaurant", "cafe", "movie_theater", "mortgage", "bank", "atm"]
 - e. Enter the following name for the new function:
checkForEnter
 - f. Click **OK**. EGL switches to Source view and displays the checkForEnter function. Notice two other recent additions:
 - In the declaration of the typeComboBox widget, the **onChange** property is set to checkForEnter.
 - The following **import** statement resolves the reference to the DojoComboBox widget type:
import dojo.widgets.DojoComboBox;
 - g. Click **Design** to return to the Design view. The checkForEnter function name is now displayed next to the onChange event. The function is said to be *bound* to the typeField field. You will add the code for this function in the next lesson.
 - h. To save the file, press Ctrl-S.
 6. Add a button to initiate the search for the specified type code:
 - a. Drag a Button (Dojo) widget from the **Display and Input** drawer of the palette to the third cell in the second row. Assign the following name:
typeButton
 - b. In the Properties view, on the Events page, click the row for the onClick event. Click the plus sign (+) to add a function for the event.
 - c. In the New Event Handler window, enter the following name for the new function:
buttonClicked
 - d. Click **OK**. EGL switches to Source view and displays the buttonClicked function.
 - e. Click **Design** to return to the Design view. The buttonClicked function name is now displayed next to the onClick event. The function is bound to the typeButton button. You will add the code for this function in the next lesson.

- f. On the General page, change the **text** property for the button to the following name:
Search
- g. To save the file, press Ctrl-S.
7. Create a box to contain the list of mortgage companies.
 - a. From the **Layout** drawer of the palette, drag a Box widget onto the first cell of the third row and give it the following name:
listingBox
 - b. In the Properties view, on the General page, set the **columns** property to 1.
 - c. On the Position page, set the **width** property to 150.
 - d. On the Layout page, set the **verticalAlignment** property to TOP and the **horizontalSpan** property to 2.

At this point, the UI looks like the following picture:

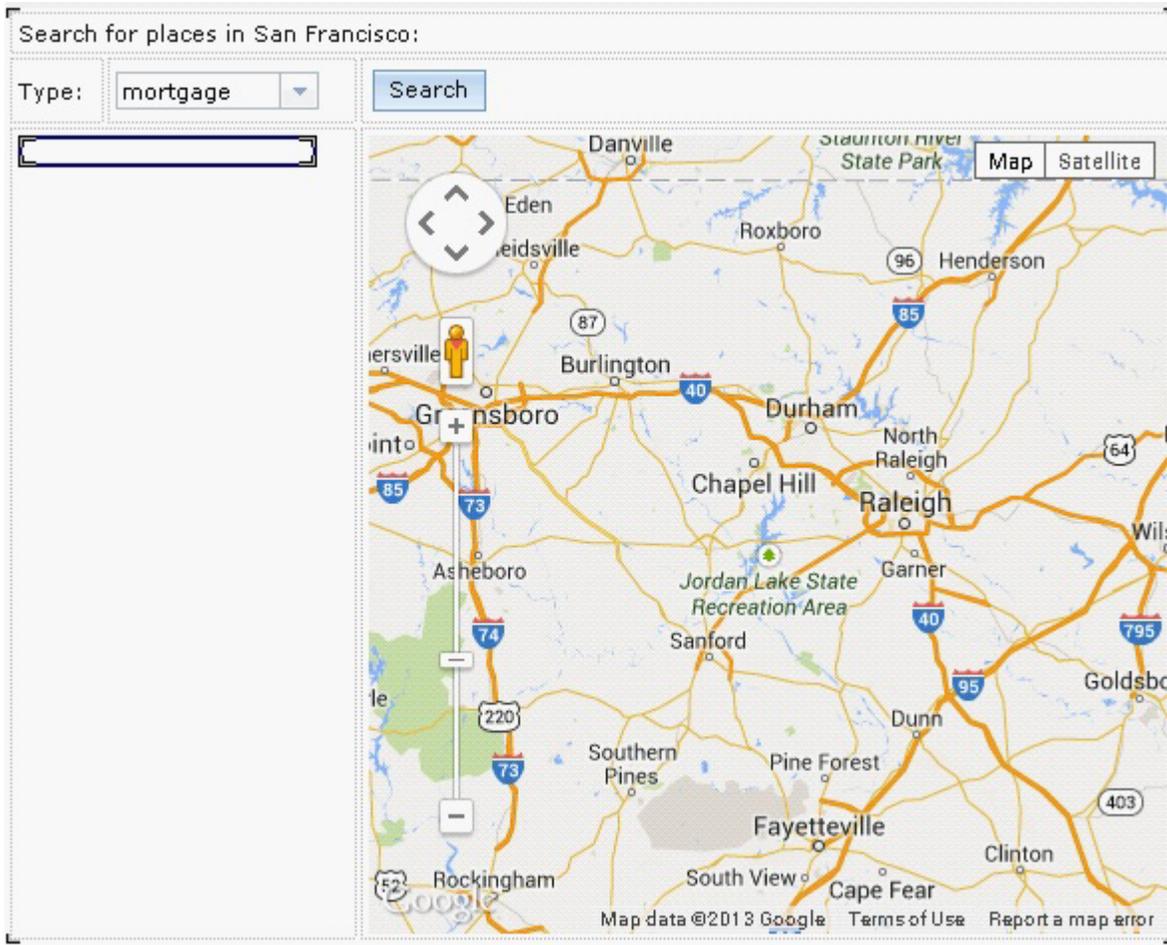


8. From the **Samples** drawer of the palette, drag a Google Map widget to the last cell in the third row, next to the listingBox widget, and give it the following name:
localMap

Refresh the Design view of the Rich UI editor by clicking the refresh button on the upper right of the Design view, not the upper right of Project Explorer.



The Design surface looks as follows:



9. To save the file, press Ctrl-S.

You have finished working in Design view.

If you click the **Source** tab, you can see code that the EGL Rich UI editor created.

Lesson checkpoint

You learned how to complete the following tasks:

- Create Record definitions from the data retrieved from a service.
- Configure an Interface part with details that allow for service access.
- Create a user interface that includes a Google map.

In the next lesson, you add source code to complete the MapLocatorHandler handler.

Lesson 11: Add code to the map locator handler

Complete the background code for the user interface that you created in the previous lesson.

Finish the source code for MapLocatorHandler.egl

1. Make sure that the MapLocatorHandler.egl file is open in the Rich UI editor. If you are in the Design view, click the **Source** tab.
2. Add a blank line just before the start function and declare a variable that is based on the Interface part you created:

```
lookupService GooglePlacesService{@restbinding};
```

The **@restbinding** property indicates that service-access details are in your code rather than in the EGL deployment descriptor. The decision is convenient but inflexible. A change in the service location requires that you change the source code. Lesson 14 introduces the EGL deployment descriptor, where you are likely to put service-access details in most of your development work.

3. To resolve a reference to the Interface part, press Ctrl-Shift-O. The next steps will add new red error marks, and you will not remove them until late in the lesson.
4. Do not add content to the start function.
5. Complete the checkForEnter function as follows:

```
function checkForEnter(event Event in)
    if(event.ch == 10)
        search();
    end
end
```

Consider the following background detail: The EGL runtime code invokes the checkForEnter function and passes an *event object*, which is a memory structure that includes details about the event. In this case, the event that caused the invocation is onKeyDown, and the event object includes the ASCII character that represents the user's keystroke. Specifically, the number 10 is the decimal value for the Carriage Return (the ENTER key) in the ASCII table, as noted here: ASCII table and description (<http://www.asciitable.com>).

The checkForEnter function is invoked only if the user presses a key such as Tab or ENTER when the text field has focus. The function in turn invokes the search function only if the key was ENTER. You will create the search function soon.

6. Complete the buttonClicked() function:

```
function buttonClicked(event Event in)
    search();
end
```

The **buttonClicked** function and its relationship to the button-specific **onClick** property ensures that the user's clicking the **Search** button invokes the search function.

7. To add the search function, place the following code at the end of the handler, before the final **end** statement in the file:

```
function search()
    localMap.zoom = 10;
    localMap.removeAllMarkers();
    // show an initial marker, as necessary to display the map at all
    localMap.addMarker(new MapMarker{ latitude = "37.47", longitude = "-122.26",
    address = "I am here!", description = "San Francisco"});

    // Call the remote Google service, passing the type value
    call lookupService.getSearchResults( typeComboBox.value ) returning to showResults
    onException displayError;
end
```

Note the following details:

- The EGL Google map widget includes the **zoom** property, which specifies the scale of the map. Rather than specifying the large scale used for the default map of North Carolina, where the **zoom** value was 8, set the **zoom** value to 10, which produces a map that shows individual city.
- The EGL Google map widget also includes the `addMarker` function, which accepts a record of type `MapMarker` and identifies the map location of an input address.

In this initial display for a search result set, the only detail that you provide to the `localMap.addMarker()` function is the city location marker.

8. Next, add the `showResults` function that is invoked if access of the Google Places service succeeds without error. Place the following code after the search function, before the last **end** statement in the file:

```
linkListing HyperLink[0];

for(i int from 1 to retResult.result.getSize() by 1)
  newLink HyperLink{padding = 4, text = retResult.result[i].name, href = "#"};
  newLink.setAttribute("title", retResult.result[i].vicinity );
  newLink.setAttribute("lat",
    retResult.result[i].geometry.location.lat);
  newLink.setAttribute("lng",
    retResult.result[i].geometry.location.lng);
  newLink.onClick ::= mapAddress;
  linkListing.appendElement(newLink);
end
listingBox.setChildren(linkListing);
end
```

Your call to the service returns an array of places details. Consider these aspects of the `showResults` function:

- Each element comprises a “title” (that is, a place name).
- The `showResults` function creates an array of hyperlink widgets and reads through the input array. For each element in the input array, the function creates an element in the array of hyperlink widgets.
- As shown by the following declaration, each hyperlink widget has displayable text and padding and includes a placeholder (#) instead of a web address:


```
newLink HyperLink{padding = 4, text = retResult.result[i].title, href = "#"};
```

 The hyperlink will cause the invocation of code rather than a web address. However, the presence of the placeholder ensures that the hyperlink shows text in a familiar way, with an underscore and in color, as if the user's clicking the hyperlink opens a web site.
- The function invokes the **setAttribute** function to place a value in the DOM tree, in an area of memory that is specific to the widget. In particular, the function stores a latitude and longitude for retrieval by another function.
- In relation to each hyperlink widget, the `showResults` function sets up a runtime behavior by assigning the `mapAddress` function to the `onClick` event.
- The complete array of hyperlink widgets is assigned as the only child of the listing box.

9. Place the following function after the `showResults` function:

```
function mapAddress(e Event in)

  // Show the marker when the user clicks the link
  businessAddress string = e.widget.getAttribute("address") as string;
  businessName string = e.widget.getAttribute("title") as string;
  lat string = e.widget.getAttribute("lat") as string;
```

```

    lng string = e.widget.getAttribute("lng") as string;
    localMap.addMarker( new MapMarker{ latitude = lat,
        longitude = lng, address = businessAddress, description = businessName});
End

```

When the user clicks the hyperlink at run time, the `mapAddress` function retrieves the attributes that were set in the `showResults` function and sets a marker on the displayed map.

10. You now add the exception handler that receives data if access of the Google Places service fails. Place the following code after the `mapAddress` function, before the last **end** statement in the file:

```

function displayError(ex AnyException in)
    DojoDialogLib.showError("Google Service",
        "Cannot invoke Google Places service: " + ex.message, null);
end

```

`DojoDialogLib` is a Library part in the `com.ibm.egl.rui.dojo.samples` project that you added to your workspace in Lesson 2. The `showError` function in that library displays information in a dialog. The function invocation includes a string named **message**, which is in the exception record that the EGL runtime code passes to the `displayError` function.

11. Format your code by pressing Ctrl-Shift-F, resolve the references by pressing Ctrl+Shift+O, and save the file. If you see errors in your source file, compare your code to the file contents in “Finished code for `MapLocatorHandler.egl` after Lesson 11” on page 85.

Related concepts

 “Understanding how browsers handle a Rich UI application”

Related reference

 “Event handling in Rich UI” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

 “The Exception stereotype” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

Test the new portlet

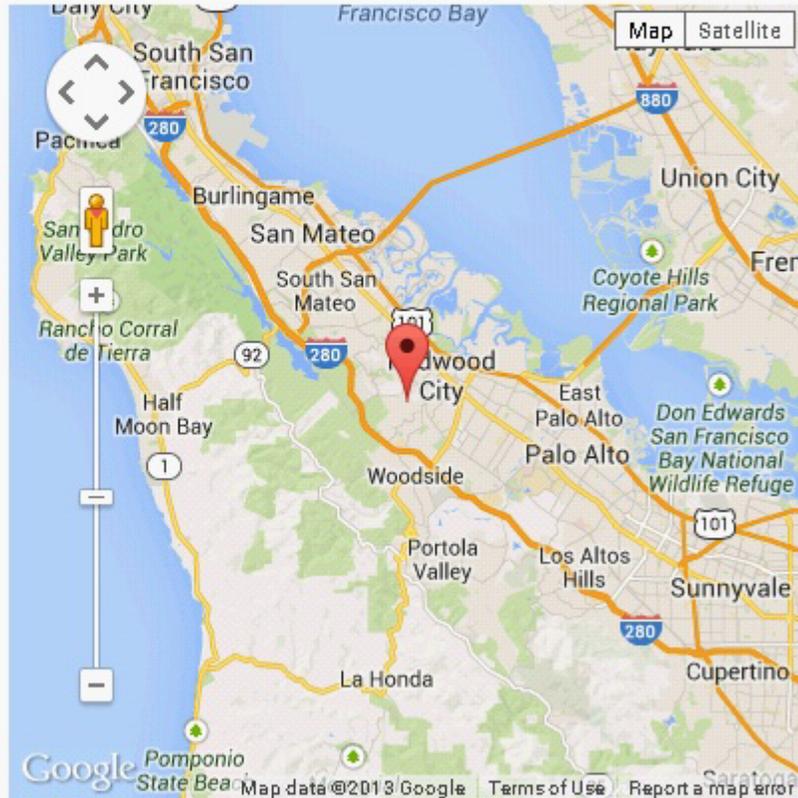
Because this portlet works independently, you can test it separately.

1. Make sure to save the file, and then click **Preview**. EGL displays the entry form with the default North Carolina map.
2. Select mortgage in the selection list for type.
3. Press the Enter key or click the **Search** button. A list of mortgage places is displayed down the left side of the screen. On the right is a map of San Francisco.

Search for places in San Francisco:

Type:

- [Chase Bank](#)
- [Bank of America](#)
- [Wells Fargo Bank](#)
- [Fremont Bank](#)
- [Spinner Mortgage](#)
- [U.S. Bank - Menlo Park Office](#)
- [Amerimac Golden Key Financial](#)
- [Guarantee Mortgage](#)
- [Chase Bank](#)
- [Mortgage Services](#)
- [Citibank](#)
- [Alternative Mortgage Sources](#)
- [Bank of America Mortgage](#)
- [Ascend Financial Mortgage Inc](#)
- [Metro Capital Mortgage Inc](#)
- [Diversified Mortgage Group](#)
- [American Coast Mortgage](#)
- [Diversified Capital Funding](#)
- [HSBC Bank](#)
- [Judy Hamilton Mortgage Lender](#)



Note: Note: The Google Places Search Service sometimes returns no data, in which case an error message is displayed because of a “null exception.” If the service is unavailable or does not return data, click Cancel and wait for a later time to complete the tutorial.

4. Click any of the names in the left column. The map displays a marker that shows the location of the business. If you hover over the marker, the name of the name is displayed.
5. Redo the same search or search on a nearby cafe. The markers that you placed on the map remain there.
6. If you want to remove all markers before each search, click the **Source** tab and add the following line at the top of the showResults function:

```
localMap.removeAllMarkers();
```
7. Test your work in the **Preview** tab.
8. Close the file.

Lesson checkpoint

You learned how to complete the following tasks:

- Create and use a variable that is based on the Local Search service.
- Respond to user keystrokes.
- Use the DOM tree to pass values from one function to another.
- Begin to use a map widget.

In the next lesson, you embed the new handler in the application.

Lesson 12: Embed the map locator handler in the application

Add a new portlet to the main portal.

Change the main portal

Add lines for the map portlet that are similar to the lines for the other three:

1. In the Rich UI editor, open the `MainHandler.egl` file and click the **Source** tab.
2. Immediately below the `historyHandler` declaration, add a similar declaration for `mapHandler`:

```
mapHandler MapLocatorHandler{};
```

3. Immediately below the `historyPortlet` declaration, add a similar declaration for `mapPortlet`:

```
mapPortlet Portlet{children = [mapHandler.ui],  
  title = "Map", canMove = FALSE, canMinimize = TRUE};
```

4. In the `start()` function, below the existing calls to `addPortlet()`, add the new portlet to the portal:

```
mortgagePortal.addPortlet(mapPortlet, 2);
```

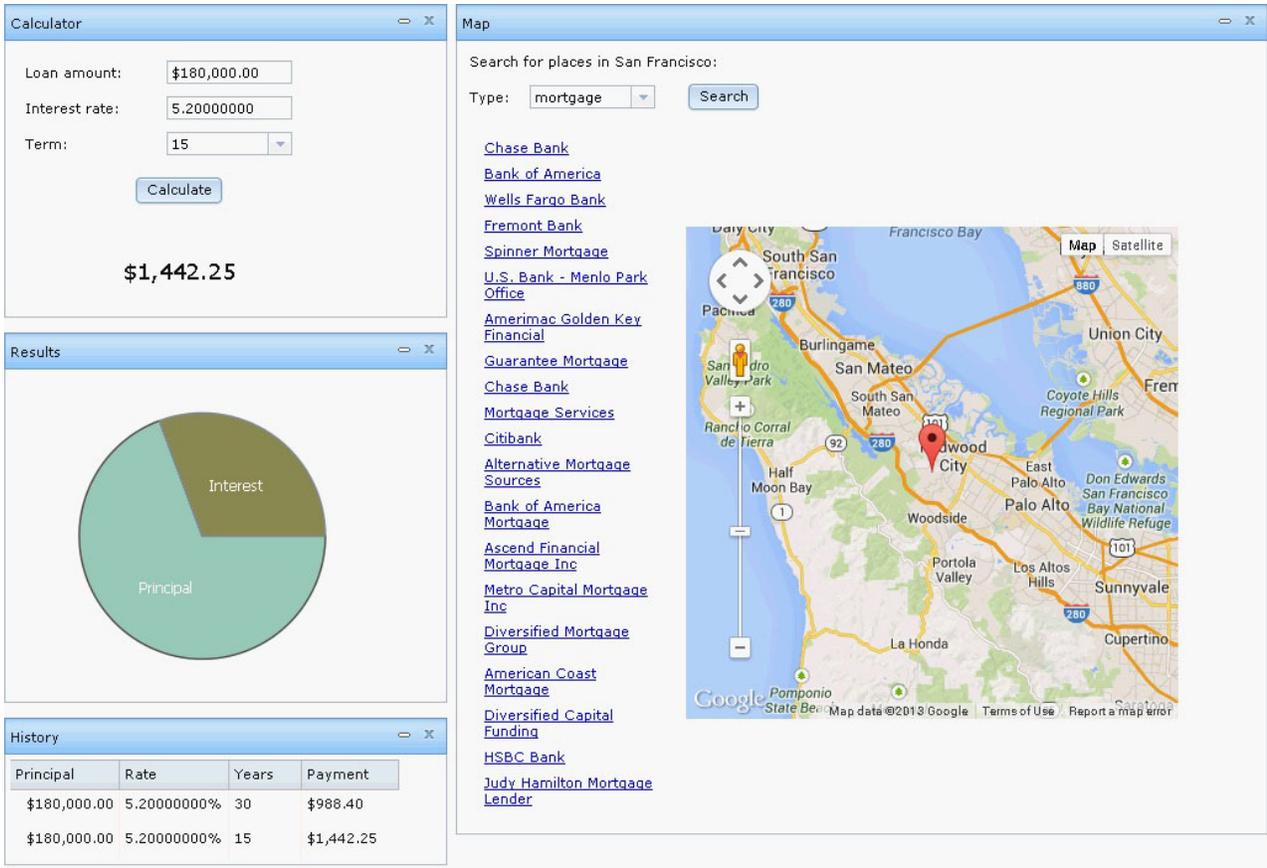
In this case, you are adding the map portlet to the second, wider column.

5. Save the file. If you see errors in your source file, compare your code to the file contents in “Finished code for `MainHandler.egl` after Lesson 12” on page 86.

Test the portal

Test the main portal to make sure that the new Map portlet is displayed and that all portlets work correctly.

1. At the bottom of the editor, click **Preview**. EGL displays the main portal and the four subsidiary portlets.
2. Calculate at least two different mortgages and search for mortgages in the Map portlet.



3. Close the file.

Lesson checkpoint

There were no new tasks in this lesson.

In the next lesson, you install Apache Tomcat on your system so that you can run your application on an application server.

Lesson 13: Install Apache Tomcat

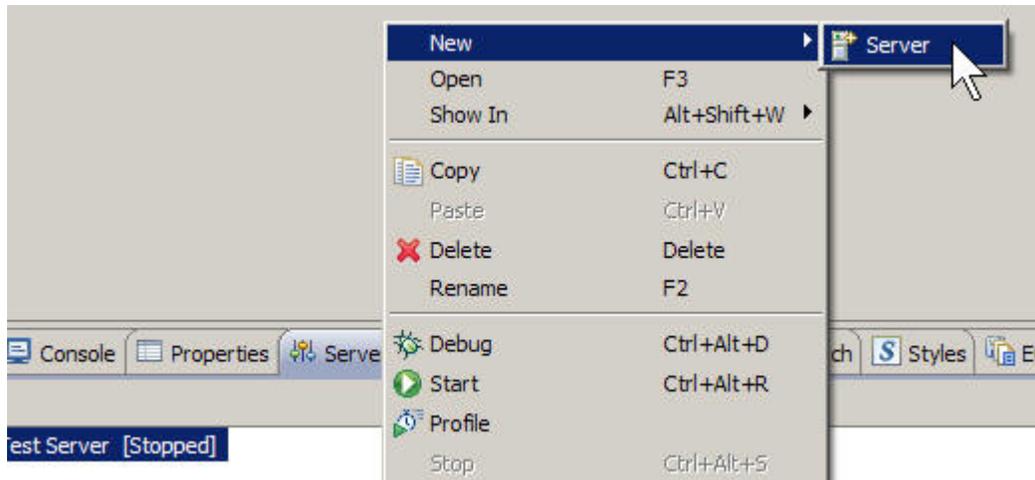
You can use Apache Tomcat to display the web page and to run the EGL-generated service.

Download and access the server

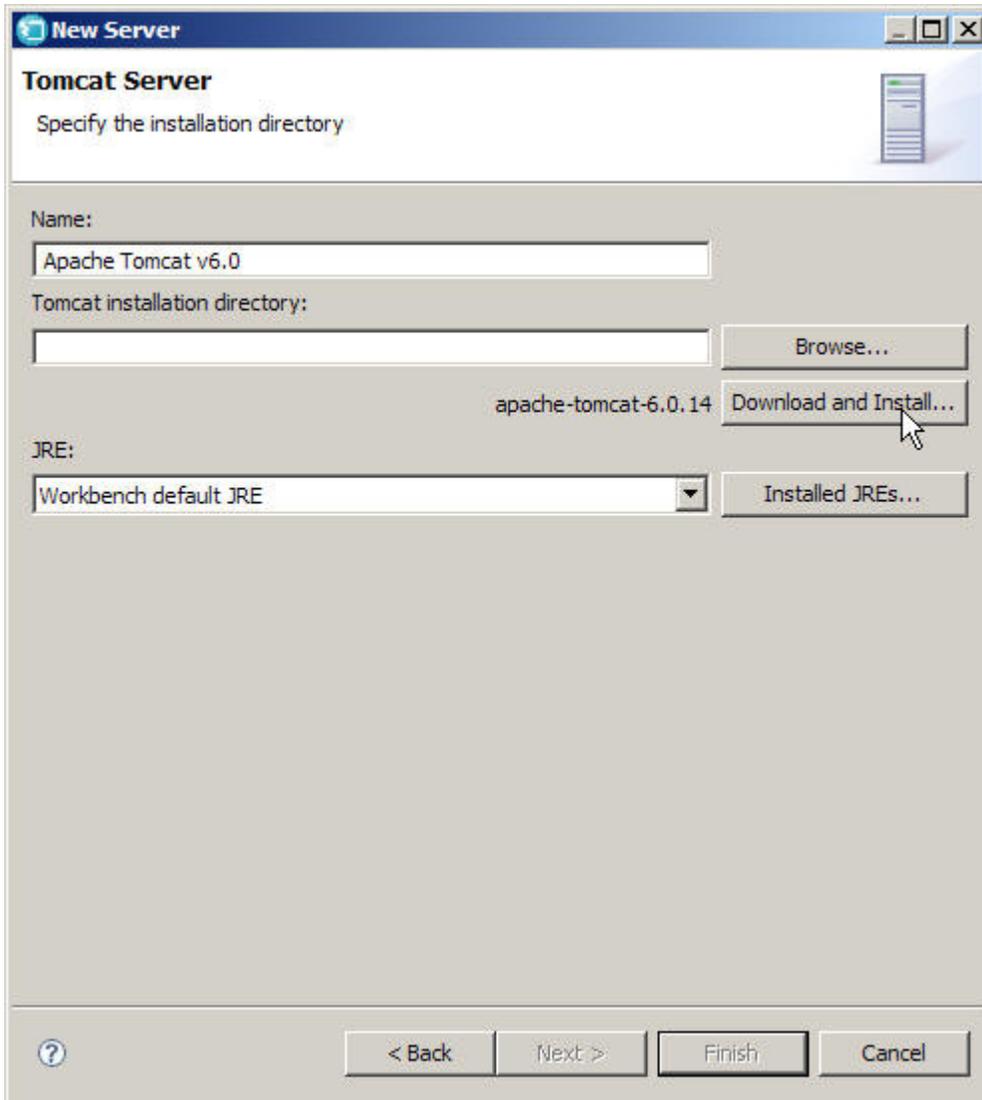
If you have IBM® WebSphere® Application Server installed, you can skip to the next lesson. In any case, you can download Apache Tomcat, if necessary, and make it available in the workbench.

To gain access to server:

1. Locate the Servers view, which is by default at the lower right of the workbench. EGL created an AJAX Test Server by default. Right-click the empty space and click **New > Server**.



2. In the Define a New Server window, expand **Apache** and click **Tomcat v6.0 Server with EGL debugging support**. Accept the default values for the other fields. Click **Next**.
3. In the Tomcat Server window, access the open-source software either by using the **Browse** to find an existing installation directory (for example, apache-tomcat-6.0.26) on your machine; or click **Download and Install**. If you found an existing installation directory, click **Finish** and continue the lesson at step 5 on page 66.



Accept the terms of the license agreement. Browse to a directory for the application files, such as C:\Program Files\Apache. While the workbench completes the installation, the Define a New Server window is displayed with the installation directory specified. Progress is shown at the lower right of the workbench.

4. When the installation is completed, click **Finish**.
5. Start the server by highlighting the server name and clicking the green **Start** icon at the top of the Server view.



Lesson checkpoint

In this lesson, you completed the following tasks:

- Downloaded Apache Tomcat, if necessary
- Started the server.

In the next lesson, you deploy the application to a server and run it there.

Lesson 14: Deploy and test the mortgage application

During the deployment process, EGL creates HTML files and server-specific code to match your target environment.

Deployment is a two stage process:

1. Internal deployment, when you deploy your handlers to a web project.
2. External deployment, when you deploy the web project to an application server.

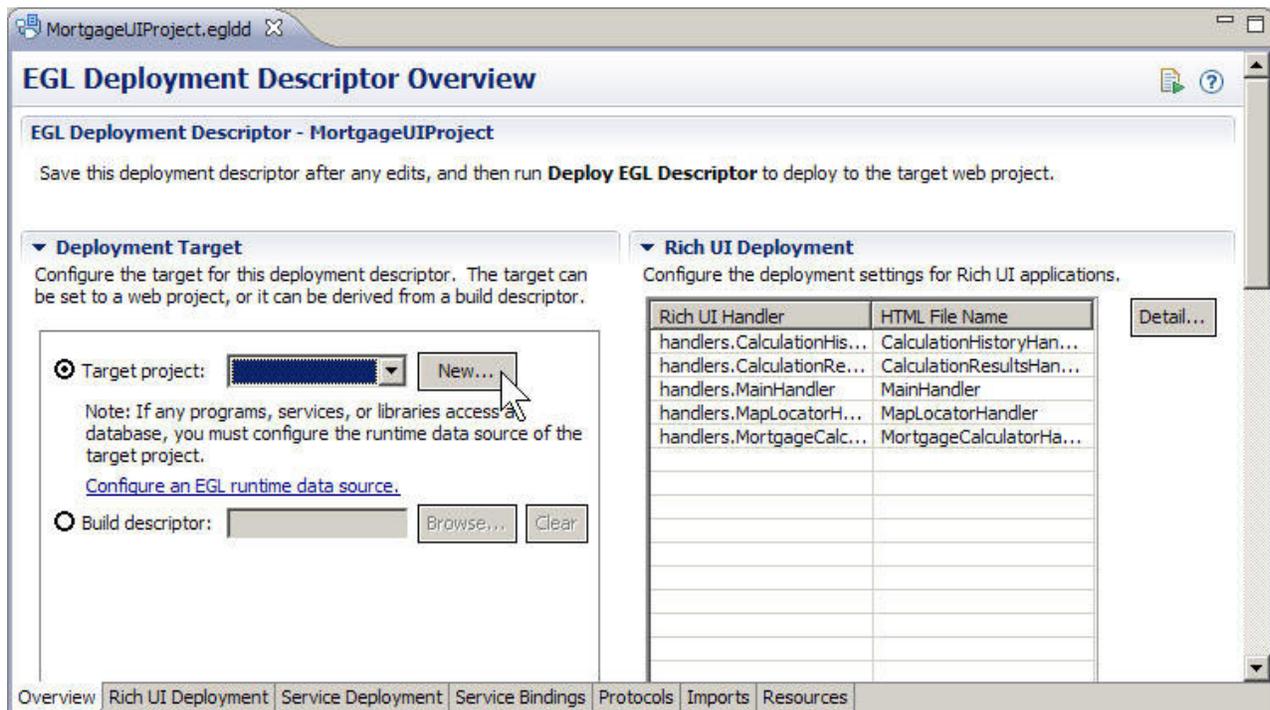
After you deploy the tutorial application internally, you can run it on an application server in the workbench.

Edit the deployment descriptor

The EGL deployment descriptor manages the internal deployment and is created automatically in each **EGLSource** folder. The main handler is in the MortgageUIProject, and you use the EGL deployment descriptor in the MortgageUIProject/EGLSource folder.

To edit the EGL deployment descriptor:

1. In the **EGLSource** folder, double-click the MortgageUIProject.egldd file. The EGL deployment descriptor opens in the Deployment Descriptor editor. EGL automatically added the embedded handlers to the list of Rich UI Handlers to deploy.
2. Because you are using a dedicated service for one service, and provided service-binding details in the code for another, you do not need to add information to the **Service Bindings Configuration** section. The list is empty.
3. Under **Deployment Target**, next to the **Target project** field, click **New**. The Dynamic Web Project wizard opens.



4. In the **Project Name** field, enter the following name:

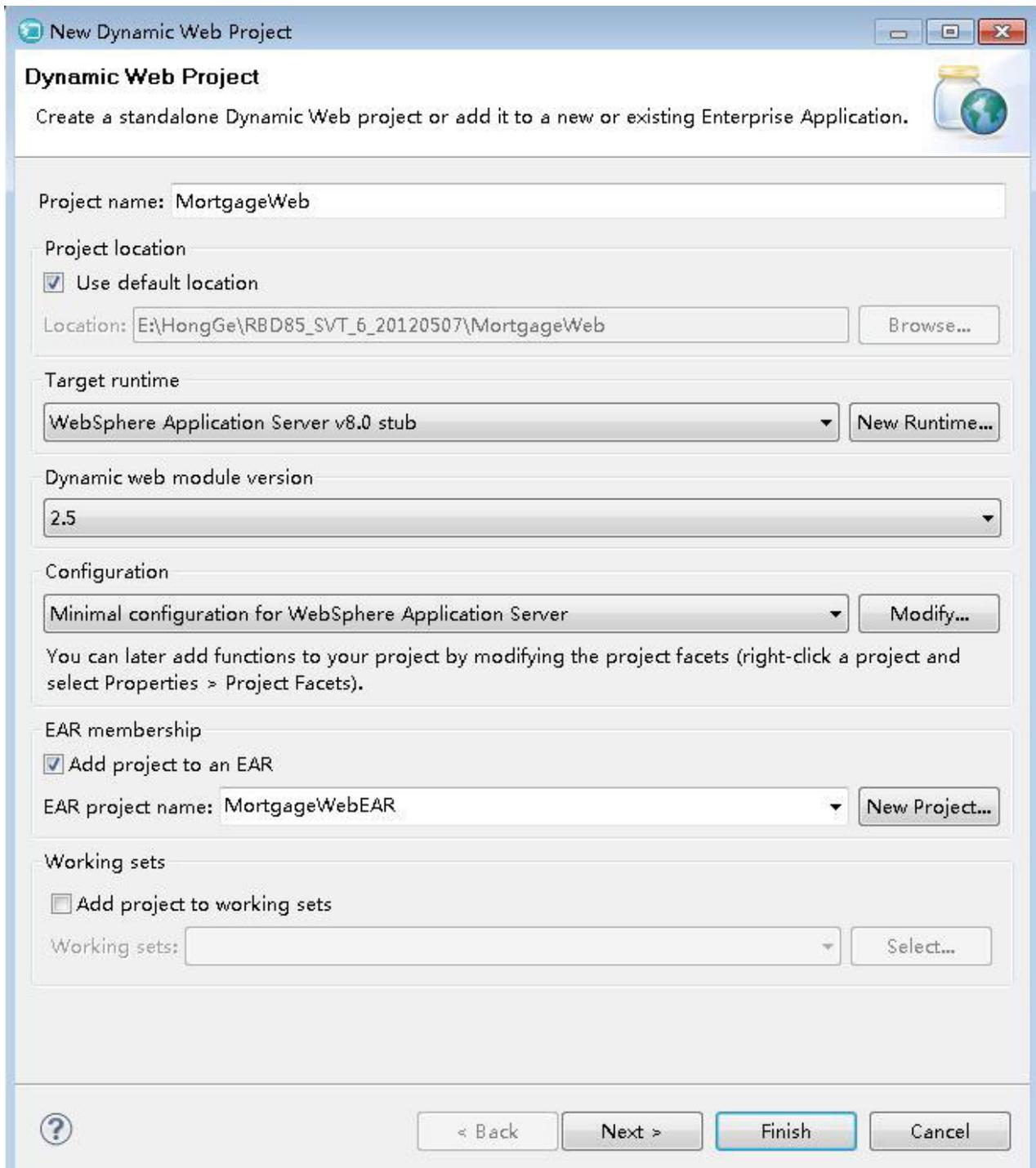
MortgageWeb

Any web project is acceptable. You are creating a simple one for the purposes of this tutorial.

5. For the Target runtime, select one of the following options from the list:
 - **Apache Tomcat v6.0**
 - **WebSphere Application Server *vn.n***

The value of the Configuration field changes automatically to match the new runtime environment.

6. If you are deploying to a WebSphere Application Server runtime, select **Add project to an EAR**, which is underneath **EAR membership**. If you add the project to an EAR, accept the default name that the wizard displays. For Apache Tomcat, ensure that the **Add project to an EAR** check box is clear.



7. Click **Finish**. EGL creates the web project and re-displays the deployment descriptor.
8. Save and close the deployment descriptor.

Deploy the Rich UI application

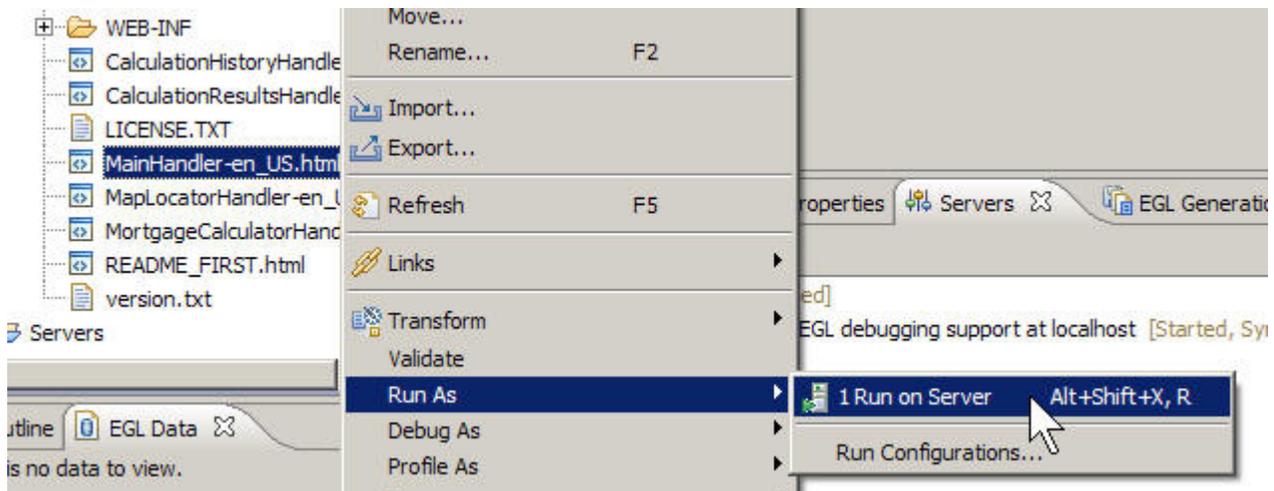
After you set the target project in the deployment descriptor, you can launch the deployment process:

1. In the **EGLSource** folder, right-click the `MortgageUIProject.egl` file.

2. Click **Deploy EGL Descriptor**. The deployment process requires no further action on your part. The process copies many files and might take several minutes.
3. If the Tomcat server shows a status of “Restart”, consider that statement a directive: restart the server by clicking the green **Start** icon in the upper right of the Servers view . Alternatively, you can right-click the server name and click **Restart**.
When the server has restarted, the status is “Started, Synchronized”.
4. If the Tomcat server shows a status of “Stopped”, start the server by clicking the green **Start** icon in the upper right of the Servers view . Alternatively, you can right-click the server name and click **Start**.
When the server has started, the status is “Started, Synchronized”.

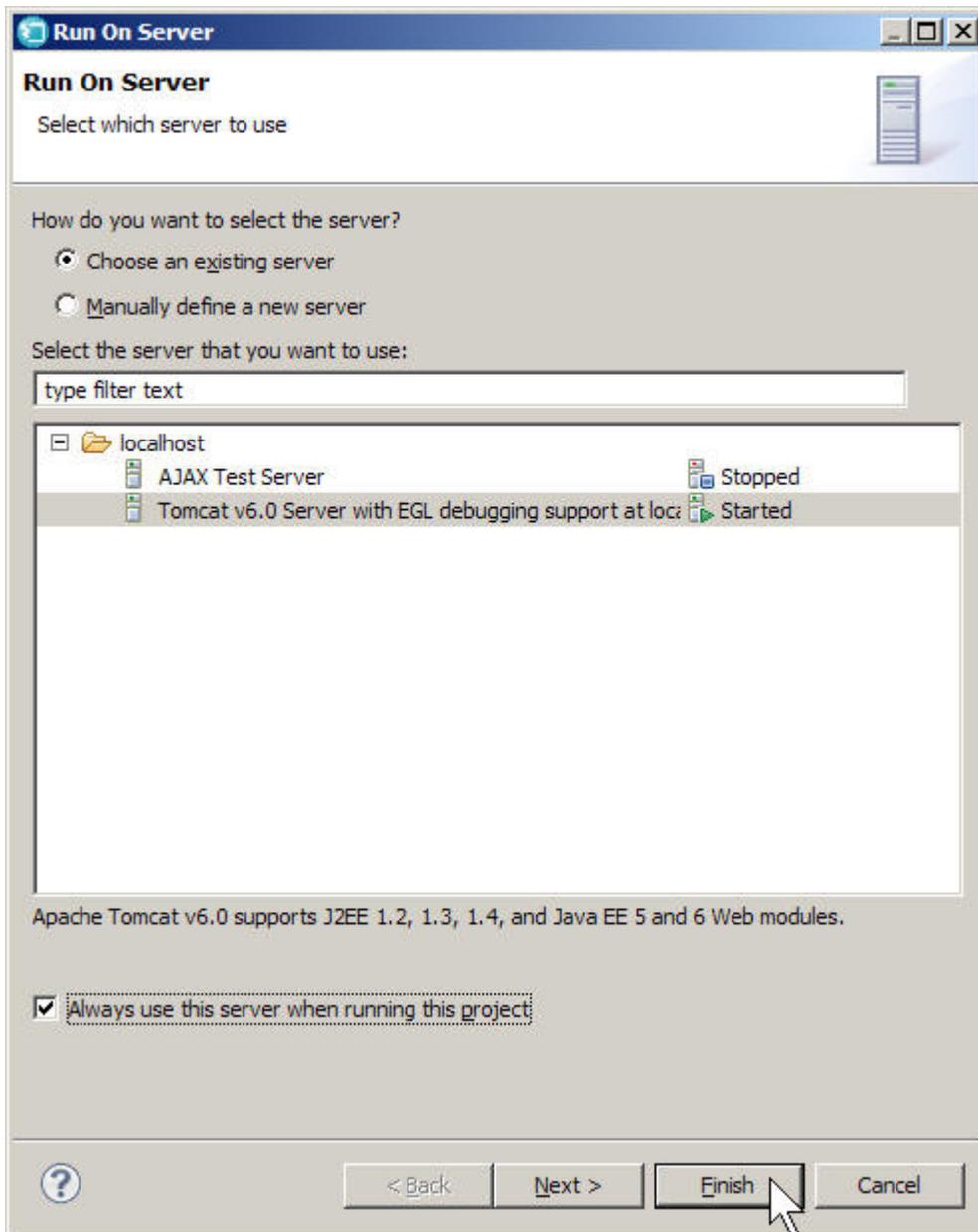
Run the generated code

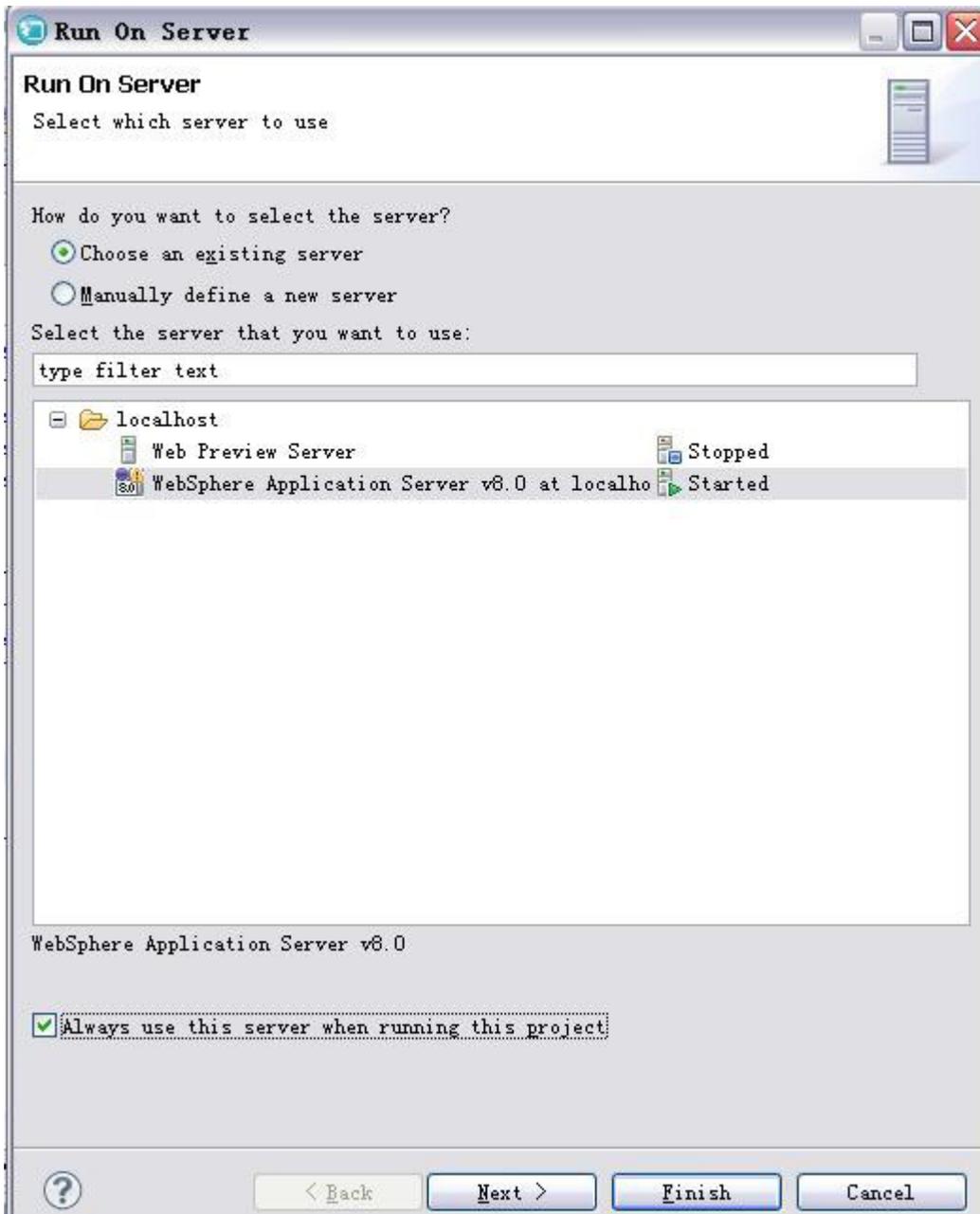
1. To run the internally deployed code, focus your attention on the target project, MortgageWeb. In the MortgageWeb/WebContent folder, find MainHandler-en_US.html.
2. Right-click the file name and click **Run As > Run on Server**



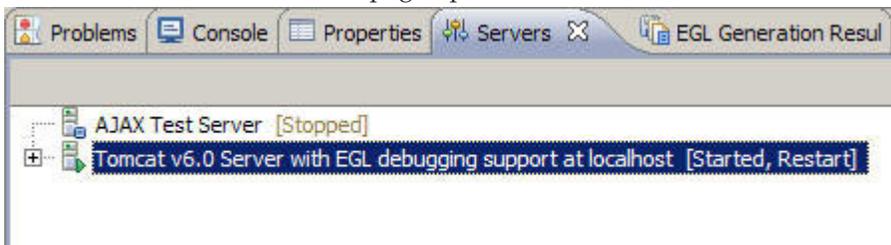
The Run On Server window opens.

3. In the Run On Server window, select the appropriate server and click **Always use this server when running this project**. Click **Finish**.





4. If you are using Tomcat and see a page not found error (404), check if the server is showing a Restart status. If so, restart the server and refresh the page. The page opens.



5. Test the application by calculating mortgages that are based on different rates, amounts, and terms. Verify that clicking a row in the history portlet displays

the appropriate information in the results portlet. Change the zip code in the map portlet and make sure the links cause the map to update.

Related concepts

“Introduction to EGL generation and deployment” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

Lesson checkpoint

You learned how to complete the following tasks:

- Edit a deployment descriptor to deploy a Rich UI handler
- Run the application on an application server

Summary

You have completed the *Create a mortgage application with EGL Rich UI* tutorial.

You practiced the following skills:

- Designing a complex Rich UI application.
- Creating and deploying a service.
- Creating a Rich UI web page by dragging variables and widget types and by writing source code.
- Accessing a service you wrote as well as services from other providers.

Resources

A variety of resources are available.

- Completed tutorial code is here:
 - “Finished code for MortgageCalculationService.egl after Lesson 3” on page 74
 - “Finished code for MortgageCalculatorHandler.egl after Lesson 4” on page 74
 - “Finished code for MortgageCalculatorHandler.egl after Lesson 5” on page 77
 - “Finished code for CalculationResultsHandler.egl after Lesson 6” on page 80
 - “Finished code for MainHandler.egl after Lesson 7” on page 80
 - “Finished code for CalculationHistoryHandler.egl after Lesson 8” on page 81
 - “Finished code for MainHandler.egl after Lesson 9” on page 82
 - “Finished code for GooglePlaceRecords.egl after Lesson 10” on page 83
 - “Finished code for GooglePlacesService.egl after Lesson 10” on page 84
 - “Finished code for MapLocatorHandler.egl after Lesson 10” on page 84
 - “Finished code for MapLocatorHandler.egl after Lesson 11” on page 85
 - “Finished code for MainHandler.egl after Lesson 12” on page 86
- The following help topics are of particular interest, and each has additional links:
 - “Overview of EGL Rich UI” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>
 - “Services: a top-level overview” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>
 - “Introduction to EGL generation and deployment” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>
 - “Correspondence between an XML string and an EGL variable” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

- "The EGL build path" at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

EGL Rich UI follows the Visual Formatting Model of the World Wide Web Consortium (W3C). For details, go to the W3C web site (<http://www.w3.org>) and search for "Visual formatting model."

Finished code for MortgageCalculationService.egl after Lesson 3

The following code is the text of the MortgageCalculationService.egl file at the end of Lesson 3.

```
package services;

service MortgageCalculationService

    function amortize(inputData MortgageCalculationResult inOut)
        amt MONEY = inputData.loanAmount;

        // convert to monthly rate
        rate DECIMAL(10, 8) = (1 + inputData.interestRate / 1200);

        // convert to months
        term INT = (inputData.term * 12);

        // calculate monthly payment amount
        pmt MONEY = (amt * (rate - 1) * MathLib.pow(rate, term))
            / (MathLib.pow(rate, term) - 1);
        totalInterest MONEY = (pmt * term) - amt;

        // update result record
        inputData.monthlyPayment = pmt;
        inputData.interest = totalInterest;
    end
end

record MortgageCalculationResult

    // user input
    loanAmount MONEY;
    interestRate DECIMAL(10,8);
    term INT;

    // calculated fields
    monthlyPayment MONEY;
    interest MONEY;
end
```

Related tasks

"Lesson 3: Create the mortgage calculation service" on page 12

Finished code for MortgageCalculatorHandler.egl after Lesson 4

The following code is the text of the MortgageCalculatorHandler.egl file after Lesson 4.

```
package handlers;

import com.ibm.egl.rui.widgets.GridLayout;
import services.MortgageCalculationResult;
import com.ibm.egl.rui.widgets.GridLayoutData;
import com.ibm.egl.rui.widgets.TextLabel;
import dojo.widgets.DojoCurrencyTextBox;
```

```

import com.ibm.egl.rui.mvc.Controller;
import com.ibm.egl.rui.mvc.MVC;
import com.ibm.egl.rui.mvc.FormField;
import dojo.widgets.DojoTextField;
import dojo.widgets.DojoComboBox;
import com.ibm.egl.rui.mvc.FormManager;
import com.ibm.egl.rui.widgets.GridLayoutLib;
import dojo.widgets.DojoButton;
import com.ibm.egl.rui.widgets.Image;

handler MortgageCalculatorHandler type RUIHandler {
    initialUI = [ ui ], onConstructionFunction = start,
    cssFile="css/MortgageUIProject.css", title="MortgageCalculatorHandler"

    ui GridLayout{ columns = 1, rows = 1,
        cellPadding = 4, children = [ inputRec_ui ] };

    inputRec MortgageCalculationResult {term = 30};

    inputRec_ui GridLayout {
        layoutData = new GridLayoutData{ row = 1, column = 1 },
        rows = 6, columns = 2, cellPadding = 4,
        children = [
            errorLabel, paymentLabel, buttonLayout, inputRec_loanAmount_nameLabel,
            inputRec_loanAmount_textBox, inputRec_interestRate_nameLabel,
            inputRec_interestRate_field, inputRec_term_nameLabel, inputRec_term_comboBox ] };

    inputRec_loanAmount_nameLabel TextLabel {
        text="Loan amount:", layoutData = new GridLayoutData { row = 1, column = 1 } };

    inputRec_loanAmount_textBox DojoCurrencyTextBox {
        currency = "USD", value = inputRec.loanAmount, width = "100",
        errorMessage="Amount is not valid." ,
        layoutData = new GridLayoutData { row = 1, column = 2 } };

    inputRec_loanAmount_controller Controller {
        @MVC {model = inputRec.loanAmount,
            view = inputRec_loanAmount_textBox as Widget},
        validStateSetter = handleValidStateChange_inputRec};

    inputRec_loanAmount_formField FormField {
        controller = inputRec_loanAmount_controller,
        nameLabel = inputRec_loanAmount_nameLabel};

    inputRec_interestRate_nameLabel TextLabel {
        text="Interest rate:" ,
        layoutData = new GridLayoutData { row = 2, column = 1 } };

    inputRec_interestRate_field DojoTextField {
        layoutData = new GridLayoutData { row = 2, column = 2}, width = "100" };

    inputRec_interestRate_controller Controller {
        @MVC {model = inputRec.interestRate,
            view = inputRec_interestRate_field as Widget},
        validStateSetter = handleValidStateChange_inputRec};

    inputRec_interestRate_formField FormField {
        controller = inputRec_interestRate_controller,
        nameLabel = inputRec_interestRate_nameLabel};

    inputRec_term_nameLabel TextLabel {
        text="Term:" , layoutData = new GridLayoutData { row = 3, column = 1 } };

    inputRec_term_comboBox DojoComboBox {
        values = ["5", "10", "15", "30"] ,
        layoutData = new GridLayoutData { row = 3, column = 2}, width = "100" };

```

```

inputRec_term_controller Controller {
    @MVC {model = inputRec.term,
        view = inputRec_term_comboBox as Widget},
    validStateSetter = handleValidStateChange_inputRec};

inputRec_term_formField FormField {
    controller = inputRec_term_controller, nameLabel = inputRec_term_nameLabel};

inputRec_form FormManager {
    entries = [ inputRec_loanAmount_formField,
        inputRec_interestRate_formField,
        inputRec_term_formField ] };

buttonLayout GridLayout{
    layoutData = new GridLayoutData{
        row = 4, column = 1,
        horizontalAlignment = GridLayoutLib.ALIGN_CENTER,
        horizontalSpan = 2 },
    cellPadding = 4, rows = 2, columns = 1,
    children = [ processImage, calculationButton ] };

calculationButton DojoButton{
    layoutData = new GridLayoutData{ row = 1, column = 1 },
    text = "Calculate", onClick ::= inputRec_form_Submit };

processImage Image{
    layoutData = new GridLayoutData{
        row = 2, column = 1,
        horizontalAlignment = GridLayoutLib.ALIGN_CENTER },
    src = "tools/spinner.gif",
    visible = false};

paymentLabel TextLabel{
    layoutData = new GridLayoutData{
        row = 5, column = 1,
        horizontalAlignment = GridLayoutLib.ALIGN_CENTER,
        horizontalSpan = 2 },
    text = "$0.00", fontSize = "18" };

errorLabel TextLabel{ layoutData = new GridLayoutData{
    row = 6, column = 1,
    horizontalSpan = 2 },
    color = "Red", width = "250" };

function start()
end

function inputRec_form_Submit(event Event in)
    if(inputRec_form.isValid())
        inputRec_form.commit();
    end
end

function inputRec_form_Publish(event Event in)
    inputRec_form.publish();
    inputRec_form_Validate();
end

function inputRec_form_Validate()
    inputRec_form.isValid();
end

function handleValidStateChange_inputRec(view Widget in, valid boolean in)
    for (n int from inputRec_form.entries.getSize() to 1 decrement by 1)
        entry FormField = inputRec_form.entries[n];
        if (entry.controller.view == view)

```

```

        if(valid)
            // TODO: handle valid value
        else
            msg String? = entry.controller.getErrorMessage();
            // TODO: handle invalid value
        end
    end
end
end
end
end

```

Related tasks

“Lesson 4: Create the user interface for the calculator” on page 15

Finished code for MortgageCalculatorHandler.egl after Lesson 5

The following code is the text of the MortgageCalculatorHandler.egl file after Lesson 5.

```

package handlers;

import com.ibm.egl.rui.widgets.GridLayout;
import services.MortgageCalculationResult;
import com.ibm.egl.rui.widgets.GridLayoutData;
import com.ibm.egl.rui.widgets.TextLabel;
import dojo.widgets.DojoCurrencyTextBox;
import com.ibm.egl.rui.mvc.Controller;
import com.ibm.egl.rui.mvc.MVC;
import com.ibm.egl.rui.mvc.FormField;
import dojo.widgets.DojoTextField;
import dojo.widgets.DojoComboBox;
import com.ibm.egl.rui.mvc.FormManager;
import com.ibm.egl.rui.widgets.GridLayoutLib;
import dojo.widgets.DojoButton;
import com.ibm.egl.rui.widgets.Image;
import services.MortgageCalculationService;
import libraries.MortgageLib;

handler MortgageCalculatorHandler type RUIhandler {
    initialUI = [ ui ], onConstructionFunction = start,
    cssFile="css/MortgageUIProject.css", title="MortgageCalculatorHandler"}

    mortService MortgageCalculationService{@dedicatedService};

    ui GridLayout{ columns = 1, rows = 1,
        cellPadding = 4, children = [ inputRec_ui ] };

    inputRec MortgageCalculationResult{
        term = 30, loanAmount=180000, interestRate = 5.2};

    inputRec_ui GridLayout{
        layoutData = new GridLayoutData{ row = 1, column = 1 },
        rows = 6, columns = 2, cellPadding = 4,
        children = [
            errorLabel, paymentLabel, buttonLayout, inputRec_loanAmount_nameLabel,
            inputRec_loanAmount_textBox,inputRec_interestRate_nameLabel,
            inputRec_interestRate_field,inputRec_term_nameLabel,inputRec_term_comboBox ] };

    inputRec_loanAmount_nameLabel TextLabel {
        text="Loan amount:" , layoutData = new GridLayoutData { row = 1, column = 1 } };

    inputRec_loanAmount_textBox DojoCurrencyTextBox {
        currency = "USD", value = inputRec.loanAmount, width ="100",
        errorMessage="Amount is not valid." ,
        layoutData = new GridLayoutData { row = 1, column = 2 } };

```

```

inputRec_loanAmount_controller Controller {
    @MVC {model = inputRec_loanAmount,
        view = inputRec_loanAmount_textBox as Widget},
    validStateSetter = handleValidStateChange_inputRec};

inputRec_loanAmount_formField FormField {
    controller = inputRec_loanAmount_controller,
    nameLabel = inputRec_loanAmount_nameLabel};

inputRec_interestRate_nameLabel TextLabel {
    text="Interest rate:" ,
    layoutData = new GridLayoutData { row = 2, column = 1 } };

inputRec_interestRate_field DojoTextField {
    layoutData = new GridLayoutData { row = 2, column = 2, width = "100" };

inputRec_interestRate_controller Controller {
    @MVC {model = inputRec_interestRate,
        view = inputRec_interestRate_field as Widget},
    validStateSetter = handleValidStateChange_inputRec};

inputRec_interestRate_formField FormField {
    controller = inputRec_interestRate_controller,
    nameLabel = inputRec_interestRate_nameLabel};

inputRec_term_nameLabel TextLabel {
    text="Term:" , layoutData = new GridLayoutData { row = 3, column = 1 } };

inputRec_term_comboBox DojoComboBox {
    values = ["5","10","15","30"] ,
    layoutData = new GridLayoutData { row = 3, column = 2, width = "100" };

inputRec_term_controller Controller {
    @MVC {model = inputRec_term,
        view = inputRec_term_comboBox as Widget},
    validStateSetter = handleValidStateChange_inputRec};

inputRec_term_formField FormField {
    controller = inputRec_term_controller, nameLabel = inputRec_term_nameLabel};

inputRec_form FormManager {
    entries = [ inputRec_loanAmount_formField,
        inputRec_interestRate_formField,
        inputRec_term_formField ] };

buttonLayout GridLayout{
    layoutData = new GridLayoutData{
        row = 4, column = 1,
        horizontalAlignment = GridLayoutLib.ALIGN_CENTER,
        horizontalSpan = 2 },
    cellPadding = 4, rows = 2, columns = 1,
    children = [ processImage, calculationButton ] };

calculationButton DojoButton{
    layoutData = new GridLayoutData{ row = 1, column = 1 },
    text = "Calculate", onClick ::= inputRec_form_Submit };

processImage Image{
    layoutData = new GridLayoutData{
        row = 2, column = 1,
        horizontalAlignment = GridLayoutLib.ALIGN_CENTER },
    src = "tools/spinner.gif",
    visible = false};

paymentLabel TextLabel{
    layoutData = new GridLayoutData{
        row = 5, column = 1,

```

```

        horizontalAlignment = GridLayoutLib.ALIGN_CENTER,
        horizontalSpan = 2 },
    text = "$0.00", fontSize = "18" };

errorLabel TextLabel{ layoutData = new GridLayoutData{
    row = 6, column = 1,
    horizontalSpan = 2 },
    color = "Red", width = "250" };

function start()
end

function inputRec_form_Submit(event Event in)
    if(inputRec_form.isValid())
        inputRec_form.commit();
        showProcessImage();
        calculateMortgage();
    else
        errorLabel.text = "Input form validation failed.";
    end
end

function inputRec_form_Publish(event Event in)
    inputRec_form.publish();
    inputRec_form_Validate();
end

function inputRec_form_Validate()
    inputRec_form.isValid();
end

function handleValidStateChange_inputRec(view Widget in, valid boolean in)
    for (n int from inputRec_form.entries.getSize() to 1 decrement by 1)
        entry FormField = inputRec_form.entries[n];
        if (entry.controller.view == view)

            if(valid)
                // TODO: handle valid value
            else
                msg String? = entry.controller.getErrorMessage();
                // TODO: handle invalid value
            end
        end
    end
end

function showProcessImage()
    processImage.visible = yes;
end

function hideProcessImage()
    processImage.visible = no;
end

function calculateMortgage()
    errorLabel.text = "";
    call mortService.amortize(inputRec)
        returning to displayResults
        onException handleException;
end

function displayResults(retResult MortgageCalculationResult in)
    paymentLabel.text = MortgageLib.formatMoney(retResult.monthlyPayment as STRING);
    inputRec_form.publish();
    hideProcessImage();
end

```

```

// catch-all exception handler
private function handleException(ae AnyException in)
    errorLabel.text = "Error calling service: " + ae.message;
end

end

```

Related tasks

“Lesson 5: Add code to the mortgage calculator handler” on page 27

Finished code for CalculationResultsHandler.egl after Lesson 6

The following code is the text of the CalculationResultsHandler.egl file after Lesson 6, which also made small but important changes to the CalculationResultsHandler.egl file.

```

package handlers;

import com.ibm.egl.rui.infobus.InfoBus;
import com.ibm.egl.rui.widgets.GridLayout;
import com.ibm.egl.rui.widgets.GridLayoutData;
import dojo.widgets.DojoPieChart;
import dojo.widgets.PieChartData;
import services.MortgageCalculationResult;

handler CalculationResultsHandler type RUIHandler{
    initialUI =[ui], onConstructionFunction = start,
    cssFile = "css/MortgageUIProject.css", title = "CalculationResultsHandler"}

    ui GridLayout{columns = 1, rows = 1, cellPadding = 4,
        children =[interestPieChart]};
    interestPieChart DojoPieChart{layoutData =
        new GridLayoutData{row = 1, column = 1},
        radius = 100, width = "300", height = "250", labelOffset = 50,
        fontColor = "white",
        data =[
            new PieChartData{y = 1, text = "Principal", color = "#99ccbb"},
            new PieChartData{y = 0, text = "Interest", color = "#888855"}
        ]};

    function start()
        InfoBus.subscribe("mortgageApplication.mortgageCalculated", displayChart);
    end

    function displayChart(eventName string in, dataObject any in)
        localPieData PieChartData[2];

        resultRecord MortgageCalculationResult =
            dataObject as MortgageCalculationResult;
        localPieData = interestPieChart.data;
        localPieData[1].y = resultRecord.loanAmount;
        localPieData[2].y = resultRecord.interest;
        interestPieChart.data = localPieData;
    end
end

```

Related tasks

“Lesson 6: Create the calculation results handler” on page 34

Finished code for MainHandler.egl after Lesson 7

The following code is the text of the MainHandler.egl file after Lesson 7.

```

package handlers;

import com.ibm.egl.rui.infobus.InfoBus;
import utils.portal.Portal;
import utils.portal.Portlet;

handler MainHandler type RUIhandler
{initialUI = [ mortgagePortal ],
  onConstructionFunction = start,
  cssFile="css/MortgageUIProject.css",
  title="MainHandler"}

mortgagePortal Portal{ columns = 2, columnWidths = [ 350, 650 ] };

calculatorHandler MortgageCalculatorHandler{};
resultsHandler CalculationResultsHandler{};

calculatorPortlet Portlet
{children = [calculatorHandler.ui], title = "Calculator"};
resultsPortlet Portlet{children = [resultsHandler.ui],
  title = "Results", canMove = TRUE, canMinimize = TRUE};

function start()
  mortgagePortal.addPortlet(calculatorPortlet, 1);
  mortgagePortal.addPortlet(resultsPortlet, 1);

  // Subscribe to calculation events
  InfoBus.subscribe("mortgageApplication.mortgageCalculated", restorePortlets);

  // Initial state is minimized
  resultsPortlet.minimize();
end

function restorePortlets(eventName STRING in, dataObject ANY in)
  if(resultsPortlet.isMinimized())
    resultsPortlet.restore();
  end
end
end

```

Related tasks

“Lesson 7: Create the main Rich UI handler” on page 38

Finished code for CalculationHistoryHandler.egl after Lesson 8

The following code is the text of the CalculationHistoryHandler.egl file after Lesson 8.

```

package handlers;

import com.ibm.egl.rui.infobus.InfoBus;
import com.ibm.egl.rui.widgets.DataGrid;
import com.ibm.egl.rui.widgets.DataGridColumn;
import com.ibm.egl.rui.widgets.DataGridFormatters;
import com.ibm.egl.rui.widgets.DataGridLib;
import libraries.MortgageLib;
import services.MortgageCalculationResult;

handler CalculationHistoryHandler type RUIhandler
{initialUI = [ historyResults_ui ],
  onConstructionFunction = start,
  cssFile="css/MortgageUIProject.css",
  title="CalculationHistoryHandler"}

historyResults MortgageCalculationResult[0];

```

```

historyResults_ui DataGrid {
    selectionMode = DataGridLib.SINGLE_SELECTION,
    selectionListeners ::= cellClicked,
    columns = [
        new DataGridColumn {name = "loanAmount", displayName = "Principal",
            width = 80, alignment = DataGridLib.ALIGN_RIGHT,
            formatters = [ formatDollars ]},
        new DataGridColumn {name = "interestRate", displayName = "Rate",
            width = 80, alignment = DataGridLib.ALIGN_RIGHT,
            formatters = [ DataGridFormatters.percentage ]},
        new DataGridColumn {name = "term", displayName = "Years",
            width = 50, alignment = DataGridLib.ALIGN_RIGHT},
        new DataGridColumn {name = "monthlyPayment", displayName = "Payment",
            width = 70, alignment = DataGridLib.ALIGN_RIGHT,
            formatters = [ formatDollars ]}
    ],
    data = historyResults as any[] };

function start()
    InfoBus.subscribe("mortgageApplication.mortgageCalculated", addResultRecord);
end

// Update the grid to include the latest mortgage calculation
function addResultRecord(eventName STRING in, dataObject ANY in)
    resultRecord MortgageCalculationResult =
        dataObject as MortgageCalculationResult;
    historyResults.appendElement(resultRecord);
    historyResults_ui.data = historyResults as ANY[];
end

// Publish an event to the InfoBus whenever the user selects an old calculation
function cellClicked(myGrid DataGrid in)
    updateRec MortgageCalculationResult =
        myGrid.getSelection()[1] as MortgageCalculationResult;
    InfoBus.publish("mortgageApplication.mortgageResultSelected", updateRec);
end

function formatDollars(class string, value string, rowData any in)
    value = mortgageLib.formatMoney(value);
end
end

```

Related tasks

“Lesson 8: Create the calculation history handler” on page 42

Finished code for MainHandler.egl after Lesson 9

The following code is the text of the MainHandler.egl file at the end of Lesson 9, which also made small but important changes to CalculationResultsHandler.egl.

```

package handlers;

import com.ibm.egl.rui.infobus.InfoBus;
import utils.portal.Portal;
import utils.portal.Portlet;

handler MainHandler type RUIhandler{initialUI =[mortgagePortal ],
    onConstructionFunction = start,
    cssFile = "css/MortgageUIProject.css",
    title = "MainHandler"}

mortgagePortal Portal{columns = 2, columnWidths =[350, 650]};

calculatorHandler MortgageCalculatorHandler{};
resultsHandler CalculationResultsHandler{};
historyHandler CalculationHistoryHandler{};

```

```

calculatorPortlet Portlet{children =[calculatorHandler.ui ], title = "Calculator"};
resultsPortlet Portlet{children =[resultsHandler.ui ],
    title = "Results", canMove = true, canMinimize = true};
historyPortlet Portlet{children = [historyHandler.historyResults_ui],
    title = "History", canMove = TRUE, canMinimize = TRUE};

function start()
    mortgagePortal.addPortlet(calculatorPortlet, 1);
    mortgagePortal.addPortlet(resultsPortlet, 1);
    mortgagePortal.addPortlet(historyPortlet, 1);

    // Subscribe to calculation events
    InfoBus.subscribe("mortgageApplication.mortgageCalculated", restorePortlets);

    // Initial state is minimized
    resultsPortlet.minimize();
    historyPortlet.minimize();
end

function restorePortlets(eventName string in, dataObject any in)

    if(resultsPortlet.isMinimized())
        resultsPortlet.restore();
    end

    if(historyPortlet.isMinimized())
        historyPortlet.restore();
    end
end
end

```

Related tasks

“Lesson 9: Embed the calculation history handler in the application” on page 47

Finished code for GooglePlaceRecords.egl after Lesson 10

The following code is the text of the GooglePlaceRecords.egl file at the end of Lesson 10.

```

package services;

Record PlaceSearchResponse
    status string;
    result Result[];
    html_attribution string;
    next_page_token string;
end

Record Result
    name string;
    vicinity string;
    type1 string[] {@XMLElement{name = "type"}};
    geometry Geometry;
    rating string?;
    icon string;
    reference string;
    id string;
    opening_hours Opening_hours?;
    photo Photo?;
    price_level string?;
end

Record Geometry
    location Location;
end

Record Location

```

```

lat string;
lng string;
end

Record Opening_hours
  open_now string;
end

Record Photo
  photo_reference string;
  width string;
  height string;
  html_attribution string?;
end

```

Related tasks

“Lesson 10: Create the map locator handler” on page 52

Finished code for GooglePlacesService.egl after Lesson 10

The following code is the text of the GooglePlacesService.egl file at the end of Lesson 10.

```

package interfaces;

import services.PlaceSearchResponse;

// interface
interface GooglePlacesService

  function getSearchResults( typeName string? in ) returns(PlaceSearchResponse)
    @GetRest{uriTemplate = "https://maps.googleapis.com/maps/api/place/search/xml?location=37.47,

```

Related tasks

“Lesson 10: Create the map locator handler” on page 52

Finished code for MapLocatorHandler.egl after Lesson 10

The following code is the text of the MapLocatorHandler.egl file at the end of Lesson 10.

```

package handlers;

import com.ibm.egl.rui.widgets.Box;
import com.ibm.egl.rui.widgets.GridLayout;
import com.ibm.egl.rui.widgets.GridLayoutData;
import com.ibm.egl.rui.widgets.GridLayoutLib;
import com.ibm.egl.rui.widgets.HyperLink;
import com.ibm.egl.rui.widgets.TextLabel;
import egl.ui.rui.Event;
import dojo.widgets.DojoButton;
import dojo.widgets.DojoComboBox;
import interfaces.GooglePlacesService;
import services.PlaceSearchResponse;
import utils.dialog.DojoDialogLib;
import utils.map.GoogleMap;
import utils.map.MapMarker;

handler MapLocatorHandler type RUIHandler{initialUI =[ui
  ], onConstructionFunction = start, cssFile = "css/MortgageUIProject.css", title = "MapLo

  ui GridLayout{columns = 3, rows = 3, cellPadding = 4, children =[localMap,
    listingBox, typeButton, typeComboBox, typeLabel, introLabel
  ]};
  introLabel TextLabel{layoutData = new GridLayoutData{row = 1, column = 1, horizontalSpan = 3}, t
  typeLabel TextLabel{layoutData = new GridLayoutData{row = 2, column = 1}, text = "Type:"};
  typeComboBox DojoComboBox{layoutData = new GridLayoutData{row = 2, column = 2}, value = "mortgage
    "bar", "food", "restaurant", "cafe", "movie_theater", "mortgage",

```

```

        "bank", "atm"]];

typeButton DojoButton{layoutData = new GridLayoutData{row = 2, column = 3}, text = "Search", c
listingBox Box{layoutData = new GridLayoutData{row = 3, column = 1, verticalAlignment = GridLa
localMap GoogleMap{layoutData = new GridLayoutData{row = 3, column = 3}, width = 400, height =

function start()
end

function checkForEnter(event Event in)

end

function buttonClicked(event Event in)

end
end
end

```

Related tasks

“Lesson 10: Create the map locator handler” on page 52

Finished code for MapLocatorHandler.egl after Lesson 11

The following code is the text of the MapLocatorHandler.egl file at the end of Lesson 11.

```

package handlers;

import com.ibm.egl.rui.widgets.Box;
import com.ibm.egl.rui.widgets.GridLayout;
import com.ibm.egl.rui.widgets.GridLayoutData;
import com.ibm.egl.rui.widgets.GridLayoutLib;
import com.ibm.egl.rui.widgets.HyperLink;
import com.ibm.egl.rui.widgets.TextLabel;
import egl.ui.rui.Event;
import dojo.widgets.DojoButton;
import dojo.widgets.DojoComboBox;
import interfaces.GooglePlacesService;
import services.PlaceSearchResponse;
import utils.dialog.DojoDialogLib;
import utils.map.GoogleMap;
import utils.map.MapMarker;

handler MapLocatorHandler type RUIHandler{initialUI =[ui
    ], onConstructionFunction = start, cssFile = "css/MortgageUIProject.css", title = "Map

    ui GridLayout{columns = 3, rows = 3, cellPadding = 4, children =[localMap,
        listingBox, typeButton, typeComboBox, typeLabel, introLabel
    ]];
    introLabel TextLabel{layoutData = new GridLayoutData{row = 1, column = 1, horizontalSpan = 3},
    typeLabel TextLabel{layoutData = new GridLayoutData{row = 2, column = 1}, text = "Type:"};
    typeComboBox DojoComboBox{layoutData = new GridLayoutData{row = 2, column = 2}, value = "mortg
        "bar", "food", "restaurant", "cafe", "movie_theater", "mortgage",
        "bank", "atm"]];

    typeButton DojoButton{layoutData = new GridLayoutData{row = 2, column = 3}, text = "Search", c
    listingBox Box{layoutData = new GridLayoutData{row = 3, column = 1, verticalAlignment = GridLa
    localMap GoogleMap{layoutData = new GridLayoutData{row = 3, column = 3}, width = 400, height =

    lookupService GooglePlacesService{@restbinding};

    function start()
    end

    function checkForEnter(event Event in)

```

```

        if(event.ch == 13)
            search();
        end
    end

    function buttonClicked(event Event in)
        search();
    end

    function search()
        localMap.zoom = 10;
        localMap.removeAllMarkers();
        // show an initial marker, as necessary to display the map at all
        localMap.addMarker(new MapMarker{ latitude = "37.47", longitude = "-122.26", address = "I am

        // Call the remote Google service, passing the type value
        call lookupService.getSearchResults( typeComboBox.value ) returning to showResults
        onException displayError;
    end

    function showResults(retResult PlaceSearchResponse in)
        linkListing HyperLink[0];

        for(i int from 1 to retResult.result.getSize() by 1)
            newLink HyperLink{padding = 4, text = retResult.result[i].name, href = "#"};
            newLink.setAttribute("title", retResult.result[i].vicinity );
            newLink.setAttribute("lat",
                retResult.result[i].geometry.location.lat);
            newLink.setAttribute("lng",
                retResult.result[i].geometry.location.lng);
            newLink.onClick ::= mapAddress;
            linkListing.appendElement(newLink);
        end
        listingBox.setChildren(linkListing);
    end

    function mapAddress(e Event in)

        // Show the marker when the user clicks the link
        businessAddress string = e.widget.getAttribute("address") as string;
        businessName string = e.widget.getAttribute("title") as string;
        lat string = e.widget.getAttribute("lat") as string;
        lng string = e.widget.getAttribute("lng") as string;
        localMap.addMarker( new MapMarker{ latitude = lat,
            longitude = lng, address = businessAddress, description = businessName});
    end

    function displayError(ex AnyException in)
        DojoDialogLib.showError("Google Service",
            "Cannot invoke Google Places service: " + ex.message, null);
    end
end

```

Related tasks

“Lesson 11: Add code to the map locator handler” on page 58

Finished code for MainHandler.egl after Lesson 12

The following code is the text of the MainHandler.egl file at the end of Lesson 12.

```

package handlers;

import com.ibm.egl.rui.infobus.InfoBus;
import egl.ui.columns;
import utils.portal.Portal;
import utils.portal.Portlet;

```

```

handler MainHandler type RUIhandler{initialUI =[mortgagePortal
    ], onConstructionFunction = start, cssFile = "css/MortgageUIProject.css", title = "Ma

    mortgagePortal Portal{columns = 2, columnWidths =[350, 650]};

    calculatorHandler MortgageCalculatorHandler{};
    resultsHandler CalculationResultsHandler{};
    historyHandler CalculationHistoryHandler{};
    mapHandler MapLocatorHandler{};

    historyPortlet Portlet{children =[historyHandler.historyResults_ui
        ], title = "History", canMove = true, canMinimize = true};
    calculatorPortlet Portlet{children =[calculatorHandler.ui
        ], title = "Calculator"};
    resultsPortlet Portlet{children =[resultsHandler.ui
        ], title = "Results", canMove = true, canMinimize = true};
    mapPortlet Portlet{children = [mapHandler.ui],
        title = "Map", canMove = FALSE, canMinimize = TRUE};

    function start()
        mortgagePortal.addPortlet(calculatorPortlet, 1);
        mortgagePortal.addPortlet(resultsPortlet, 1);
        mortgagePortal.addPortlet(historyPortlet, 1);
    mortgagePortal.addPortlet(mapPortlet, 2);

        historyPortlet.minimize();

        // Subscribe to calculation events
        InfoBus.subscribe("mortgageApplication.mortgageCalculated",
            restorePortlets);

        // Initial state is minimized
        resultsPortlet.minimize();
    end

    function restorePortlets(eventName string in, dataObject any in)
        if(resultsPortlet.isMinimized())
            resultsPortlet.restore();
        end
        if(historyPortlet.isMinimized())
            historyPortlet.restore();
        end
    end
end
end

```

Related tasks

“Lesson 12: Embed the map locator handler in the application” on page 63

Index

C

calculateMortgage() function 31
CalculationHistoryHandler.egl
 source code 81
CalculationResultsHandler.egl
 source code 80

D

displayResults() function 31

G

GooglePlacesService.egl
 source code
 lesson 10 84

H

hideProcessImage() function 31

I

inputRec_ui_Submit() function 30

M

MainHandler.egl
 source code
 lesson 12 86
 lesson 7 81
 lesson 9 82
MapLocatorHandler.egl
 source code
 lesson 10 83, 84
 lesson 11 85
MortgageCalculationService.egl
 source code
 Lesson 3 74
MortgageCalculatorHandler.egl
 source code
 lesson 4 74
 lesson 5 77

P

portals
 definitions 1
portlet
 definitions 1

R

REST protocols
 definitions 1

S

Service part
 creating 12
showProcessImage() function 30
SOAP protocols
 definitions 1