

Rational Business Developer



# Access a database with EGL Rich UI

*Version 8 Release 5*



Rational Business Developer



# Access a database with EGL Rich UI

*Version 8 Release 5*

**Note**

Before using this information and the product it supports, read the information in “Notices,” on page 87.

This edition applies to version 8.5 of Rational Business Developer and to all subsequent releases and modifications until otherwise indicated in new editions.

© **Copyright IBM Corporation 2009, 2012.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# Contents

## Access a database with EGL Rich UI . . . 1

Introduction . . . . .	1
Lesson 1: Plan the application . . . . .	3
Sketch the interface . . . . .	3
Consider the application flow. . . . .	3
Identify the application structure . . . . .	4
Lesson checkpoint . . . . .	5
Lesson 2: Connect to a new Derby database . . . . .	5
Create an SQL database connection. . . . .	5
Switch to the Data perspective . . . . .	7
Create a table . . . . .	8
Lesson checkpoint . . . . .	9
Lesson 3: Set up the projects and use the EGL SQL retrieve feature . . . . .	9
Create the PaymentService project. . . . .	10
Create the PaymentClient project . . . . .	12
Edit the build descriptor for the PaymentService project . . . . .	14
Use the EGL SQL retrieve feature to create a Record part . . . . .	15
Lesson checkpoint . . . . .	17
Lesson 4: Create the Rich UI handler . . . . .	17
Create the initial layout . . . . .	17
Create a data grid to hold the content of a set of database rows . . . . .	19
Add the first set of buttons . . . . .	24
Add a variable and layout to handle a single row . . . . .	26
Add the second set of buttons . . . . .	31
Lesson checkpoint . . . . .	33
Lesson 5: Create the service . . . . .	33
Create a Service part . . . . .	34
Lesson checkpoint . . . . .	35
Lesson 6: Add code for the service functions . . . . .	36
Add a payment record . . . . .	36
Read all database records. . . . .	38
Replace a record. . . . .	39
Delete a record . . . . .	39
Create test data . . . . .	40
Lesson checkpoint . . . . .	41
Lesson 7: Create a library of reusable functions . . . . .	41
Create a Library part . . . . .	41
Create the categories array . . . . .	42
Create the get functions for categories . . . . .	42
Lesson checkpoint . . . . .	43
Lesson 8: Add variables and functions to the Rich UI handler. . . . .	43

Add code to support the data grid . . . . .	43
Code the function that responds when the user clicks the data grid . . . . .	44
Format column values in the grid . . . . .	45
Test the formatting of the data grid and the transfer of data to the single-record layout . . . . .	45
Comment the prototype data . . . . .	46
Declare a service-access variable . . . . .	47
Create functions that use the service-access variable to invoke the service . . . . .	47
Update the start function to initialize the data grid with database rows . . . . .	49
Complete the callback functions . . . . .	49
Test the interface . . . . .	50
Lesson checkpoint . . . . .	52
Lesson 9: Complete the code that supports the user interface . . . . .	52
Complete the layout that displays a single row . . . . .	53
Test the new code . . . . .	53
Complete the code for the second set of buttons . . . . .	54
Test the new code . . . . .	55
Lesson checkpoint . . . . .	58
Lesson 10: Install Apache Tomcat . . . . .	58
Download and access the server . . . . .	58
Lesson checkpoint . . . . .	59
Lesson 11: Deploy and test the payment application . . . . .	60
Edit the deployment descriptor. . . . .	60
Set the data source for the new project . . . . .	62
Deploy the Rich UI application. . . . .	63
Run the generated code . . . . .	64
Lesson checkpoint . . . . .	67
Summary . . . . .	67
Resources . . . . .	67
Code for PaymentFileMaintenance.egl after lesson 4. . . . .	68
Finished code for SQLService.egl after lesson 6 . . . . .	73
Finished code for PaymentLib.egl after lesson 7 . . . . .	74
Code for PaymentFileMaintenance.egl after lesson 8. . . . .	74
Finished code for PaymentFileMaintenance.egl . . . . .	80

## Appendix. Notices . . . . . 87

Trademarks . . . . .	89
----------------------	----



---

## Access a database with EGL Rich UI

In this tutorial, you create a Rich UI application so that the user can access rows in an SQL database.

### Learning objectives

In this tutorial, you will complete these tasks:

- Plan the application and design the interface.
- Create a Derby database.
- Write a data-access service that interacts with the database tables.
- Create a web application that accesses the service, displays the retrieved data, and processes the user's updates.
- Install and configure the Apache Tomcat web server.
- Deploy the web application and service.

### Time required

About 3 hours

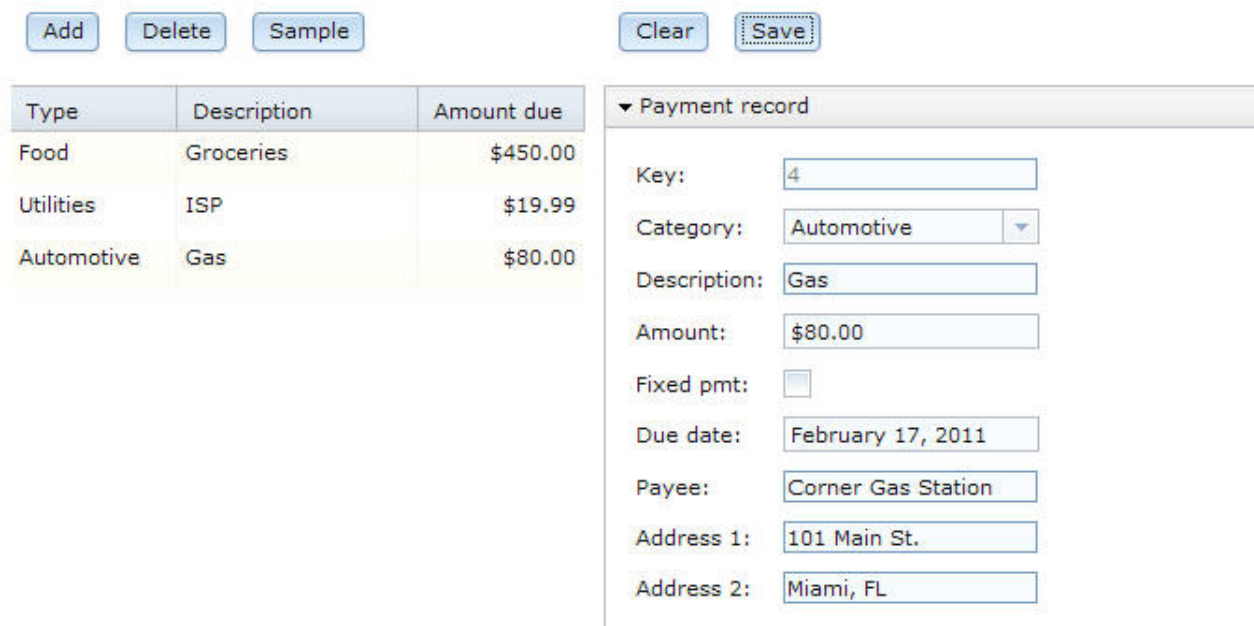
**The tutorial in HTML format:**

 “Access a database with EGL Rich UI” at <http://wilson.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

---

## Introduction

The following image shows the main page of the application that you will create:



The screenshot displays the main interface of the EGL Rich UI application. At the top, there are five buttons: "Add", "Delete", "Sample", "Clear", and "Save". Below these buttons is a table with three columns: "Type", "Description", and "Amount due". The table contains three rows of data. To the right of the table is a form titled "Payment record" with a dropdown arrow. The form contains several input fields: "Key" (with value 4), "Category" (with a dropdown menu showing "Automotive"), "Description" (with value "Gas"), "Amount" (with value "\$80.00"), "Fixed pmt" (with an unchecked checkbox), "Due date" (with value "February 17, 2011"), "Payee" (with value "Corner Gas Station"), "Address 1" (with value "101 Main St."), and "Address 2" (with value "Miami, FL").

Type	Description	Amount due
Food	Groceries	\$450.00
Utilities	ISP	\$19.99
Automotive	Gas	\$80.00

▼ Payment record

Key:

Category:

Description:

Amount:

Fixed pmt: ☐

Due date:

Payee:

Address 1:

Address 2:

The web page displays all rows in a database table and lets the user update each one. In addition, the user can add and delete rows.

The technology for developing the web page with EGL Rich UI involves several steps:

1. You write the code.
2. You generate the code and deploy it to another project in the workbench. At that point, the code that is destined for a browser is an HTML and JavaScript format; but other code is in Java format, as described later.
3. You deploy all the code to a server such as Apache Tomcat.
4. The server transmits the HTML and JavaScript code to the user's browser.
5. The application both presents data to the user and accesses services that run remotely on a server.

A main benefit of EGL Rich UI is that users can interact with a responsive, local-running web application even as services do background work such as accessing a database.

In this tutorial, the Rich UI application accesses a service that you write and deploy along with the Rich UI application. This kind of service is called an EGL *dedicated service*. 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.

**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 described in “Access a database with EGL Rich UI,” on page 1.

## Time required

This tutorial takes about 3 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 develop logic, 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 and database-access service.

---

## Lesson 1: Plan the 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.
- Consider the flow of events.
- Identify the application structure.

## Sketch the interface

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

Type	Description	Amt due
xxx	xxxxxxx	\$nnn.nn
xxxxx	xy xxxx	\$n,nnn.nn

Payment record	
Fld 1	<input type="text"/> [error]
Fld 2	<input type="text"/> [error]
Fld 3	<input type="text"/> [error]

At the left are three buttons (**Add**, **Delete**, and **Sample**) and a data grid; on the right are two buttons (**Clear** and **Save**) and a single-record layout.

## Consider the application flow

At run time, the user can do as follows:

- Click the **Sample** button to delete all rows from the database table, to add sample rows, and to display the sample rows in the data grid.
- Click the **Add** button to add an almost empty row to the database and to display that data.
- Click the **Delete** button to delete, from the database, the data that was displayed in the currently selected row of the data grid.
- Click the **Clear** button to remove content from the single-record layout.
- Click a row of the data grid to copy the details of that row to the single-record layout.
- Change the details in the single-record layout and click the **Save** button to update the related database row.

The reader might disagree with this flow of events. For example, why not have the user clear the single-record layout, type data into the layout, and click the **Add** button to create a database row that has useful data from the start? That change is one of many options, and a good learning strategy is to follow the steps of this tutorial and to use the lessons learned for a production-level application.

## 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. However, in this tutorial you develop only one handler. As noted earlier, a handler can access services, some of which you might develop by using an EGL Service part.

Whenever possible, use preexisting resources. Your Rich UI application will use the following EGL projects that are provided with the product:

### **com.ibm.egl.rui.dojo.widgets**

Provides the following widget types for this tutorial:

- DojoButton
- DojoCheckbox
- DojoComboBox
- DojoCurrencyTextBox
- DojoDateTextBox
- DojoTitlePane

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 following widget types for this tutorial:

- DataGrid
- GridLayout
- TextField
- TextLabel

You will develop the following logic:

### **SQLService**

A dedicated service that interacts with a database table.

### **PaymentLib**

A library that can provide code to several handlers

### **PaymentFileMaintenanceHandler**

The handler that defines the web application.

## **Lesson checkpoint**

In this lesson, you completed the following tasks:

- Sketched the application interface
- Considered the runtime flow of events.
- Identified the application structure

In the next lesson, you create a Derby database and a table.

---

## **Lesson 2: Connect to a new Derby database**

Use the Derby open source database manager to handle the data store for the application.

This tutorial uses the open source Derby database. In this chapter, you connect to a Derby database and create the table to be accessed. Alternatively, you can connect to a database of one of the following kinds: Cloudscape, DB2<sup>®</sup> UDB, Informix<sup>®</sup>, Oracle, or SQL Server. If you prefer to use one of those databases, review the following help topic: “Creating an SQL database connection” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>. In any case, create the table described in this lesson.

Follow these steps to set up the Derby database:

1. Create an SQL database connection through the EGL Preferences.
2. Use the Data perspective to create and connect to the database.
3. Write an SQL script to create a table within the database.
4. Disconnect from the database, as is necessary because Derby allows only one connection, which you will need during code development.

### **Create an SQL database connection**

1. In the top menu of the EGL workbench, click **Window** and then click **Preferences > EGL > SQL Database Connections**.
2. Next to the list of connection details, click **New**.
3. In the Connection Profile window, complete these steps:
  - a. Under **Connection Profile Types**, click **Derby**.
  - b. In the **Name** field, type the following string:  
Derby Database Connection
  - c. Click **Next**.
4. In the Specify a Driver and Connection Details window, specify the following information:
  - a. From the **Drivers** list, select **Derby Embedded JDBC Driver 10.1 Default**.
  - b. For the **Database location** field, enter a simple path:  
C:\databases\PaymentDB

The final element in the path is the name of a folder that does not yet exist.

- c. Specify generic login information:
  - In the **User name** field, enter admin
  - In the **Password** field, also enter admin
- d. Select the **Create database (if required)** check box.
- e. Select the **Save password** check box. When you work with live data, you might prefer not to select this option, but it simplifies the tutorial.
- f. Make sure that **Connect when the wizard completes** is selected and that **Connect every time the workbench is started** is cleared.

**New Derby Connection Profile**

**Specify a Driver and Connection Details**

Select a driver from the drop-down and provide login details for the connection.

Drivers: Derby Embedded JDBC Driver 10.1 Default

**Properties**

**General** | Optional

Database location: C:\databases\PaymentDB Browse...

User name: admin

Password: •••••

URL: jdbc:derby:C:\databases\PaymentDB;create=true

☒ Create database (if required)

☐ Upgrade database to current version

☒ Save password

☒ Connect when the wizard completes Test Connection

☐ Connect every time the workbench is started

? < Back Next > Finish Cancel

- g. Click **Test Connection**. You should see a message that says “Ping succeeded!” Click **OK** to close the message window. If the test failed, get more information by clicking **Details** on the failure message.
- h. Click **Finish**.

5. In the Preferences window, make sure that **Derby Database Connection** is highlighted, then click **OK**.

## Switch to the Data perspective

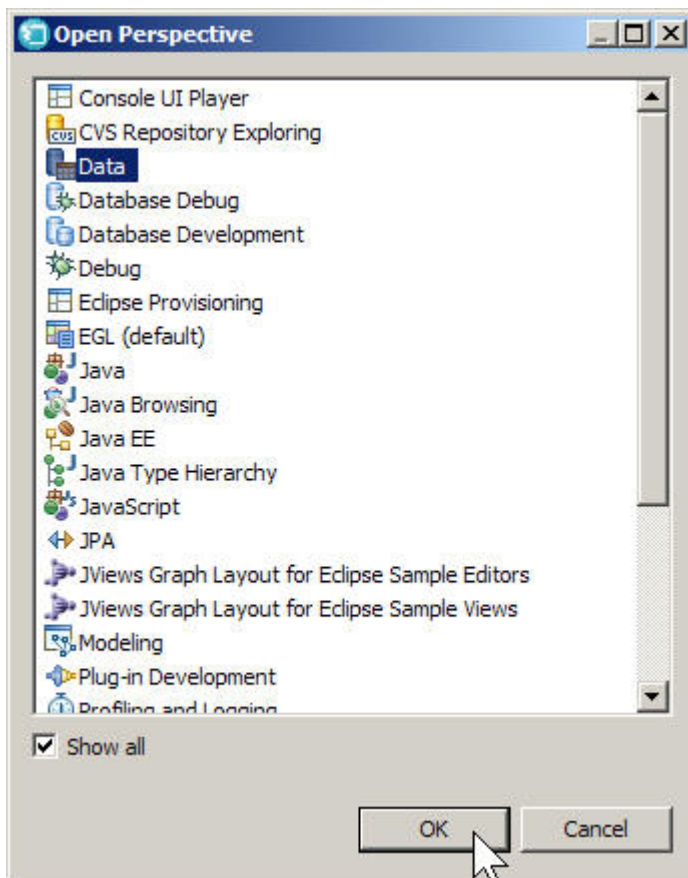
To set up the Derby database for your application, use the Data perspective, which is a workbench perspective and different from the EGL Data view.

To connect to the database:

1. Change to the Data perspective as follows:
  - a. Click the Open Perspective button, which is located by default in the right side of the navigation bar.



- b. If the Data perspective is not shown on the menu, click **Other**.
- c. If you still do not see the Data perspective, select **Show All** at the bottom of the wizard. Click **Data** and then click **OK**.



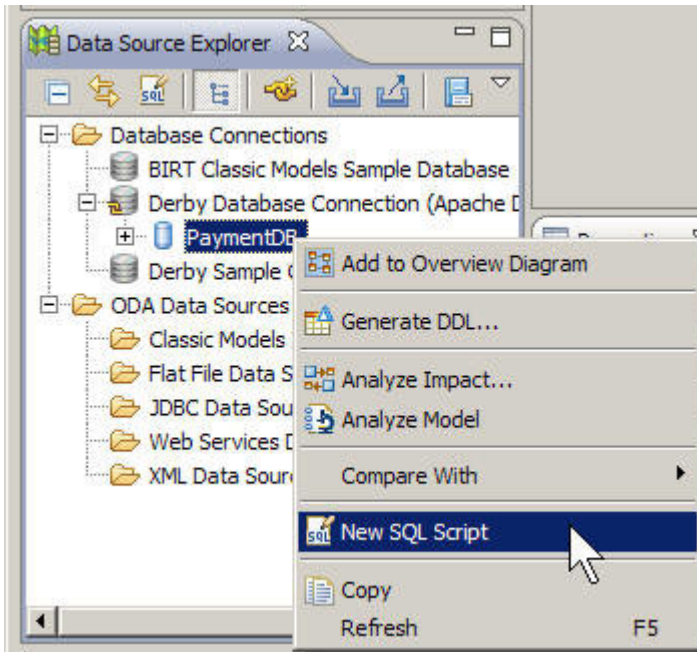
2. Locate the Data Source Explorer view, by default in the lower left corner of the workbench; and under **Database Connections**, right-click **Derby Database Connection**. Click the **Connect** option. The option was enabled because you set

the following check boxes when you created the connection: **Create database (if required)** and **Connect when the wizard completes**.

## Create a table

While in the Data perspective, you can write an SQL script to create a table in the database.

1. In the Data Source Explorer view, expand **Derby Database Connection**. Right-click the **PaymentDB** database name and click **New SQL Script**. A new script file opens in the editor.



2. Copy the following SQL code into the script file:

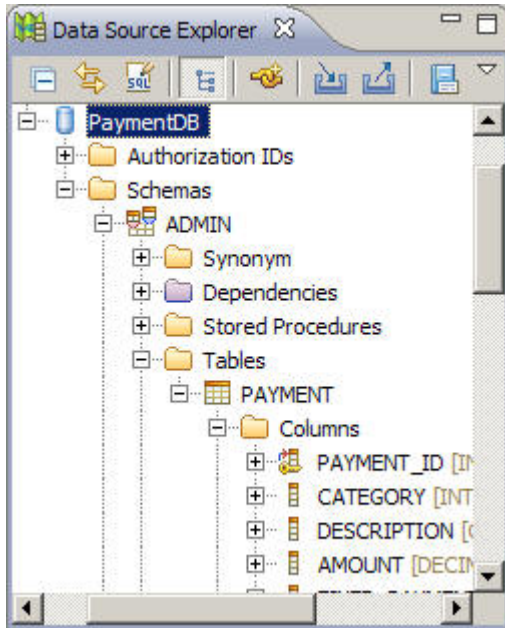
```
CREATE TABLE PAYMENT(  
    PAYMENT_ID INT PRIMARY KEY NOT NULL  
        GENERATED ALWAYS AS IDENTITY  
        (START WITH 1, INCREMENT BY 1),  
    CATEGORY INT,  
    DESCRIPTION CHAR(30),  
    AMOUNT DECIMAL(10,2),  
    FIXED_PAYMENT SMALLINT,  
    DUE_DATE DATE,  
    PAYEE_NAME CHAR(30),  
    PAYEE_ADDRESS1 CHAR(30),  
    PAYEE_ADDRESS2 CHAR(30));
```

In the next step, you run this code to create a table named PAYMENT.

### Note:

- a. The PAYMENT\_ID column is an identity column, which means that Derby will place a unique value into that column whenever the user creates a record. Each value is one more than the last.
- b. The names of Derby tables and columns are always in uppercase regardless of the case of names that are in the CREATE TABLE statement.

3. Right-click anywhere in the background of the editor pane, and then click **Run SQL**. The SQL Results view, which is by default at the bottom center of the workbench, should show the “create table” operation and a status of “Succeeded”. You can now expand the **PaymentDB** entry in the Data Source Explorer and see the columns for the new table:



4. Close the script file. You do not need to save the file, as you will not need it again.
5. You cannot access the database from EGL source code while the Data view is using the connection. Right-click **Derby database connection** and click **Disconnect**.

## Lesson checkpoint

In this lesson, you completed the following tasks:

- Created an EGL database connection
- Created a database named PaymentDB
- Created a database table named PAYMENT

In the next lesson, you start writing application code.

---

## Lesson 3: Set up the projects and use the EGL SQL retrieve feature

Before you write your logic, create two EGL projects, as well as a Record part that is based on the database table.

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.

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:

#### **PaymentService**

Holds an EGL Service part and related definitions

#### **PaymentClient**

Holds the Rich UI handlers and related definitions

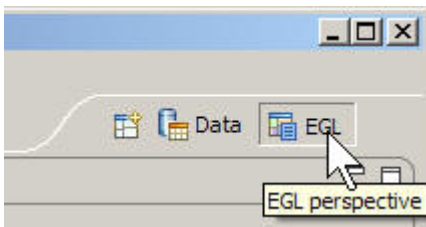
You can include all your code in a single project, but the separation shown here lets you easily deploy the two kinds of code in different ways.

Parts in one project can use parts in a different project. EGL uses a *build path* to search for unresolved references. Later in this lesson, you will add the **PaymentService** project to the build path for the **PaymentClient** project.

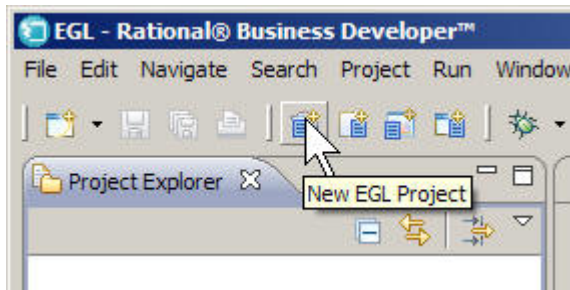
## Create the PaymentService project

To create an EGL project to contain the service:

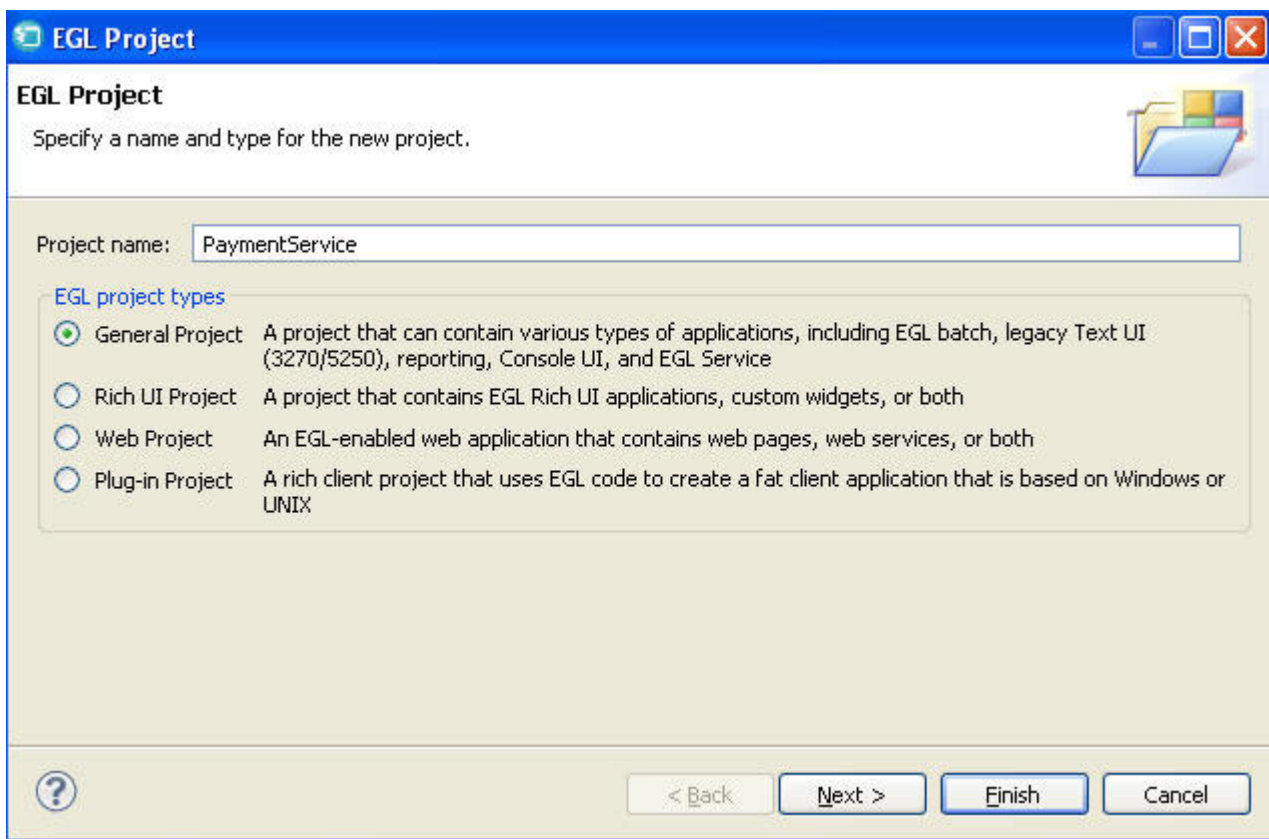
1. Change back to the EGL perspective by clicking the **EGL** button in the upper right of the workbench.



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



3. In the New EGL Project window, enter the following information:
  - a. In the **Project name** field, type the following name:  
PaymentService
  - 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 `PaymentService`. Note the folders inside the directory:

**EGLSource**

Put your packages and source files here.

**EGLGen/JavaSource**

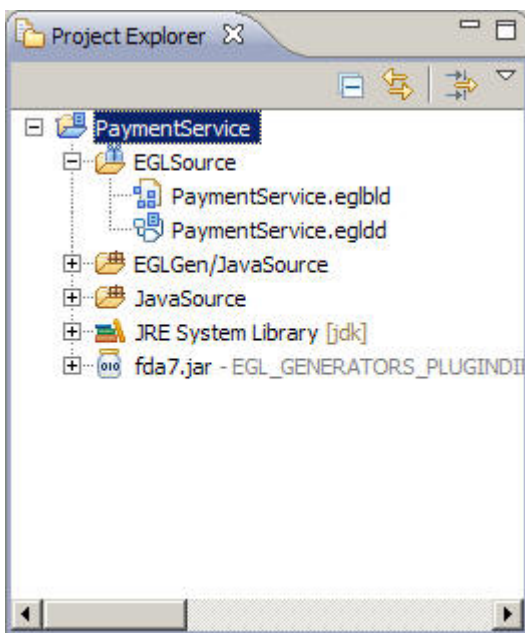
EGL places the Java files it generates here.

**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>

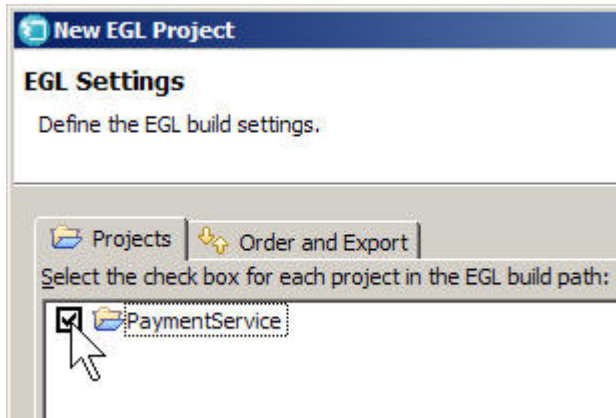
## Create the PaymentClient project

An EGL Rich UI project includes many shortcuts to speed the development of a user interface for the web.

To create an EGL Rich UI project:

1. Click the **New EGL Project** icon on the menu bar.
2. In the EGL Project window, enter the following information:
  - a. In the **Project name** field, type the following name:  
PaymentClient
  - b. Under **EGL project types**, 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 build settings page, select **PaymentService**.



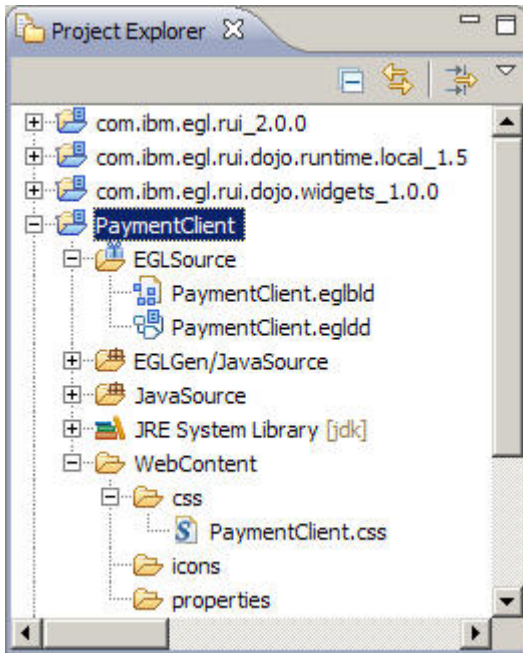
The PaymentService project is added to the build path for the project being created so that the Rich UI handler can use parts that are defined in PaymentService.

6. Click **Finish**.

EGL creates a project named PaymentClient 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.



When you first add a Rich UI project to your workspace, three other projects are added automatically:

- com.ibm.egl.rui
- com.ibm.egl.rui.dojo.runtime.local
- com.ibm.egl.rui.dojo.widgets

These three projects contain widgets and other support files that you use in creating a Rich UI application.

## Edit the build descriptor for the PaymentService project

The EGL build file has the extension `.eglbld` and contains *build parts*, which are XML definitions that are inputs for the EGL generator.

A build file typically includes multiple build descriptors so that you can generate EGL code in multiple ways. For example, EGL created the following build descriptors in the `PaymentService.eglbld` file:

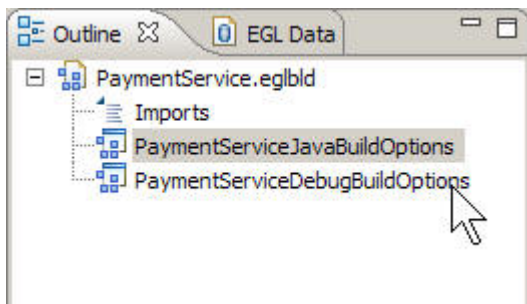
- `PaymentServiceJavaBuildOptions`
- `PaymentServiceDebugBuildOptions`

You must change the build descriptors for the project so that the logic you write there can access a database.

To edit the build descriptor:

1. In the `PaymentService` project, expand the `EGLSource` folder. Double-click the `PaymentService.eglbld` file. The Build Parts editor opens.
2. In the **Load DB options using Connection** field, click the down arrow and select the **Derby Database Connection**.
3. Find the Outline view, located by default in the lower left corner of the workbench. The `PaymentServiceJavaBuildOptions` build descriptor should currently be highlighted. Double-click **PaymentServiceDebugBuildOptions** and

repeat step 2 for the other build descriptor, which is used by the EGL debugger.



4. Save and close the build descriptor file.

## Use the EGL SQL retrieve feature to create a Record part

You can automatically retrieve the fields for a Record part that corresponds to the PAYMENT table in the PaymentDB database. The column names are the basis of the field names in the Record part.

1. Change preferences for the SQL retrieve feature:
  - a. From the top menu in the workbench, click **Window > Preferences > EGL > SQL**.
  - b. In the third group of options, **Case control rules for naming structure items**, select **Change to lower case and capitalize first letter after underscore**. Field names will be in mixed case.
  - c. In the fourth group of options, **Underscore control rules for naming structure items**, select **Remove underscores**. Field names will not include the underscores in column names.
  - d. In the last group of options, make sure that **Retrieve primary key information from the system catalog** is selected. One or more fields in the Record part will correspond to key fields in the SQL table, as is useful when you rely on default EGL SQL processing.
  - e. Clear **Prompt for SQL user ID and password when needed**. You prevent a dialog window from opening each time you access the database.
  - f. Click **OK**.
2. In the Project Explorer view, right-click **PaymentService** and then click **New > Record**
3. In the New EGL Source Record window, enter the following information:
  - a. In the **EGL source file name** field, enter the following name:  
ServiceRecords  
  
EGL adds the .egl file extension automatically.
  - b. In the **Package** field, enter the following name:  
records
  - c. Click **Finish**.  
  
EGL creates the records directory and the ServiceRecords.egl file and then opens the file in the EGL editor.
4. Replace the contents of the file by copying and pasting the following code:

```

package records;

record paymentRec type SQLRecord {tableNames = [{"PAYMENT"}]}

end

```

**Note:**

- a. Your use of a Record part that is labeled with **SQLRecord** means that, in the following case, the EGL generator will create code that is appropriate for SQL I/O:
    - You code an I/O statement such as **add**, as in this example:
 

```
add mySQLRecord;
```
    - The record on which the I/O statement operates (in the example, mySQLRecord) is an SQL record; that is, the record is based on a Record part that is labeled with **SQLRecord**.
  - b. The SQL Record part uses several properties such as **tableNames**, in most cases to change the output of the EGL generator and in this way to change runtime behavior.
  - c. Table names are specified as two-dimensional arrays because you might have reason to specify a *table label* (an SQL alias), which is useful when you write custom SQL statements. Here is a **tableNames** property setting that includes a table label:
 

```
tableNames=[{"PAYMENT", "A"}]
```
  - d. You can avoid setting the **tableNames** property if you are creating a Record part that corresponds to a single database table and if that Record part has the same name as the database table. In this tutorial, the Record part has a different name, and the **tableNames** property is required.
5. Right-click anywhere in the Record statement and click **SQL record > Retrieve SQL**. EGL automatically creates fields for the Record part in accordance with information that is provided by the database management system. You specified little more than the table name, and here is the result:

```

package records;

record paymentRec type SQLRecord {tableNames = [{"PAYMENT"}],
keyItems=[paymentId], fieldsMatchColumns = yes}

    paymentId int           {column="PAYMENT_ID"};
    category int           {column="CATEGORY", isSqlNullable=yes};
    description string      {column="DESCRIPTION", isSqlNullable=yes, maxLen=30};
    amount decimal(10,2)    {column="AMOUNT", isSqlNullable=yes};
    fixedPayment smallInt   {column="FIXED_PAYMENT", isSqlNullable=yes};
    dueDate date            {column="DUE_DATE", isSqlNullable=yes};
    payeeName string        {column="PAYEE_NAME", isSqlNullable=yes, maxLen=30};
    payeeAddress1 string    {column="PAYEE_ADDRESS1", isSqlNullable=yes, maxLen=30};
    payeeAddress2 string    {column="PAYEE_ADDRESS2", isSqlNullable=yes, maxLen=30};

end

```

6. Change the type for amount from decimal(10,2) to money. The change provides additional options when you drag-and-drop the Record part from the EGL data view to the Rich UI Design surface, as shown later.
7. Change the type for fixedPayment to boolean. Again, the change is useful during a later drag-and-drop operation.

8. Save (Ctrl-S) and close the ServiceRecords.egl file.

## Lesson checkpoint

In this lesson, you completed the following tasks:

- Created an EGL project for developing a data-access service.
- Created an EGL project for developing a Rich UI application.
- Modified the build descriptors in a build file; specifically, by adding database information from a connection definition in the workbench.
- Set preferences for the EGL SQL retrieve feature.
- Created a Record part, retrieving most information from a database.

In the next lesson, you develop some of the Rich UI application and view your prototype code in action.

---

## Lesson 4: Create the Rich UI handler

Start to build the handler 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.

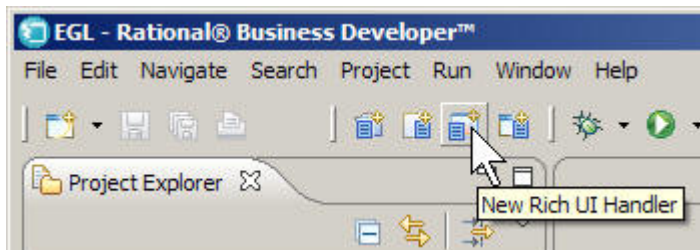
By default, the widget palette is at the right of the editor, and the Data view is at the lower left of the workbench.

In this lesson, you create a Rich UI Handler and add a data grid to display all rows in the database. Later, you will add a grid layout to display the fields in a selected record.

## Create the initial layout

To create the handler:

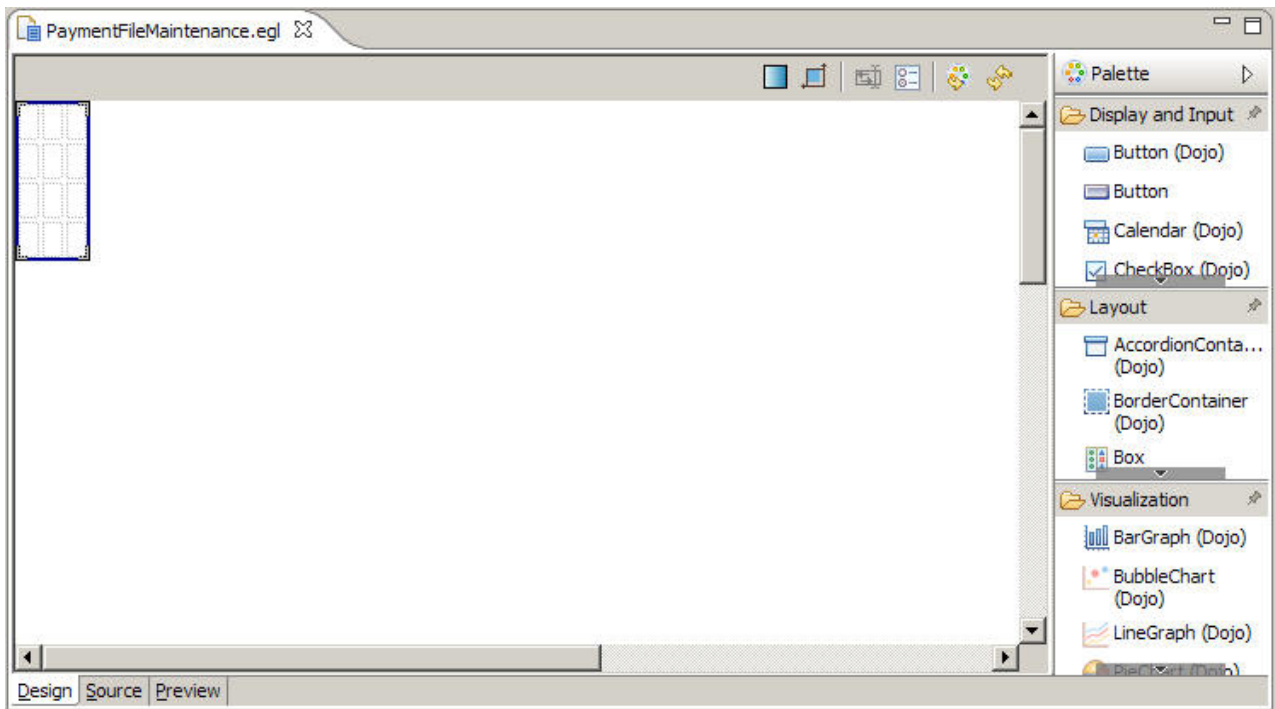
1. In the **PaymentClient** project, select the **EGLSource** folder and click **New > Rich UI Handler**.



2. In the New Rich UI Handler part window, enter the following information:

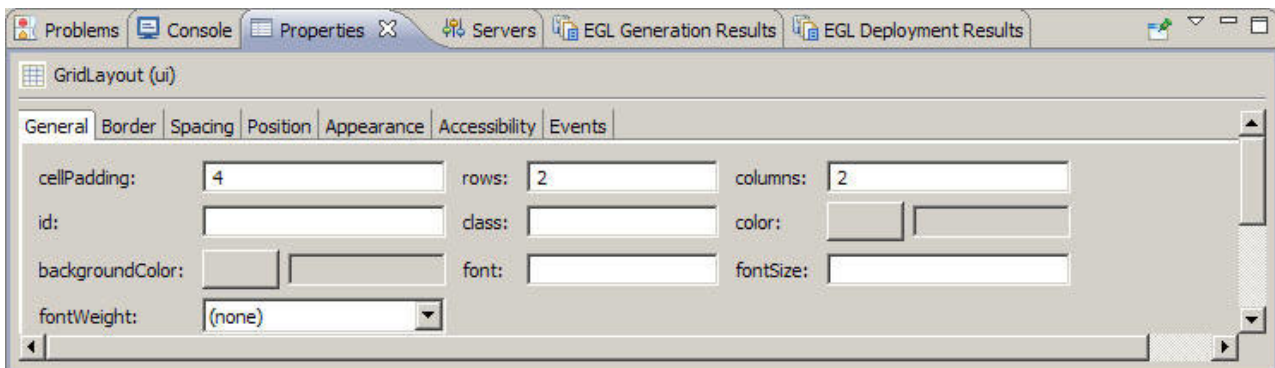
- a. In the **EGL source file name** field, enter the following name:  
PaymentFileMaintenance
- b. In the **Package** field, enter the following name:  
handlers
- c. Click **Finish**.

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



EGL automatically created a grid layout as your initial UI. By default, this widget has four rows and three columns. Compare this layout with the sketch in lesson 1, which uses only four cells.

3. To reduce the size of the layout, click into it and go to the Properties view, which by default is one of several tabbed pages below the editor pane. On the General page, set the rows property to 2 and the columns property to 2, and then click the Design surface.



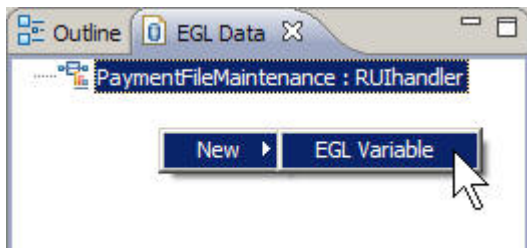
A later step demonstrates a different way to change the number of rows and columns in a grid layout. The main layout of this Rich UI handler now has a first row, where the handler will display two sets of buttons, and a second row, where the handler will display the following content: on the left, a list of records, and on the right, a layout for displaying the details of one record.

## Create a data grid to hold the content of a set of database rows

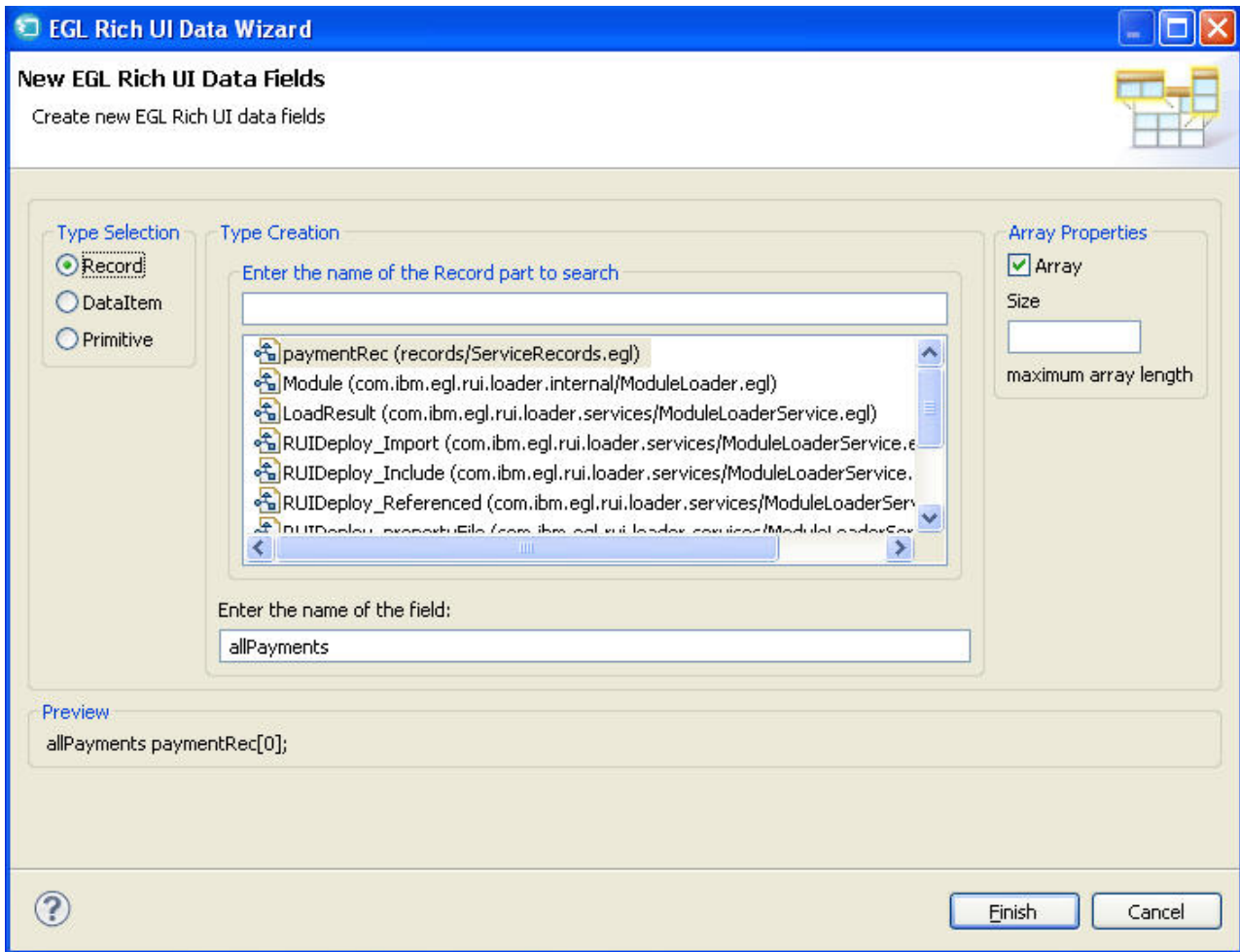
Create a data grid by dragging a record array variable onto the Rich UI editor.

To create the data grid:

1. Create a record array variable.
  - a. The EGL Data view, which is located by default in the lower left corner of the workbench, lists all of the primitive and record variables for the handler that is currently open in the editor. Right-click the empty space below the entry for the `PaymentFileMaintenance` file. Click **New > EGL Variable**.



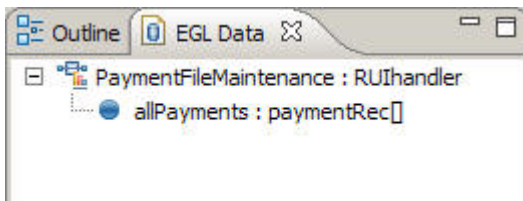
- b. In the New EGL Rich UI Data Fields wizard, request a new record variable based on the `paymentRec` record:
      - Make sure **Type Selection** is set to **Record**.
      - Select the `paymentRec` record. This record should be the only one in the list.
      - In the **Array Properties** section, select the **Array** check box. Leave the **Size** field blank.
      - For **Enter the name of the field**, enter the following name:  
`allPayments`
      - Click **Finish**.



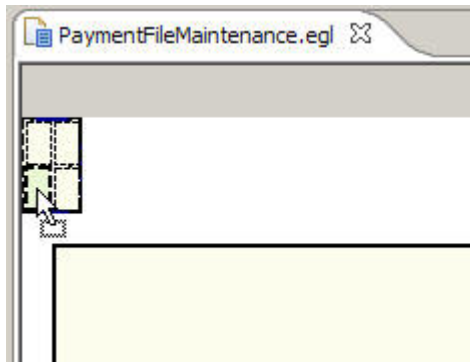
This process creates the following record declaration in the source code for the handler:

```
allPayments paymentRec[0];
```

In the EGL Data view is now a record variable that you can drag the variable onto the editor.



2. Drag the **allPayments** record variable from the EGL Data view to the lower left cell of the layout.

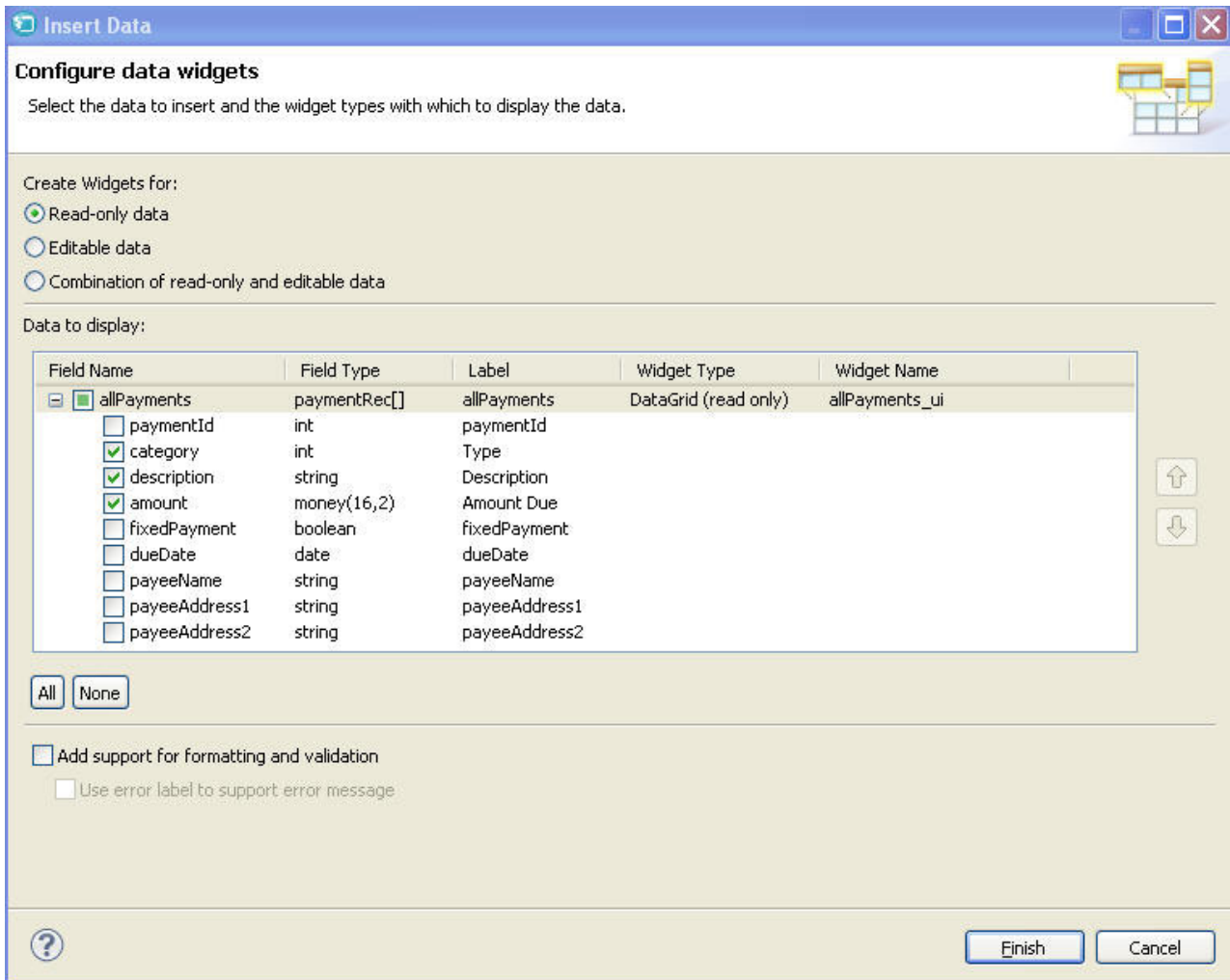


EGL displays the Configure data widgets page of the Insert Data wizard. Use this page to configure the widgets that EGL creates. The widget types depend on the type of fields in the record array that you dragged onto the Design surface.

3. Make the following changes in the Insert Data wizard:
  - a. Under **Create Widgets for**, leave the default value of **Read-only data**.
  - b. The check boxes under the `allPayments` variable indicate the fields that are to be used as columns in the display. Clear all the fields by clicking **None**.
  - c. Check the following fields:
    - category
    - description
    - amount
  - d. Change the labels for those fields:
    - Change **category** to Type.
    - Change **description** to Description.
    - Change **amount** to Amount due.

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

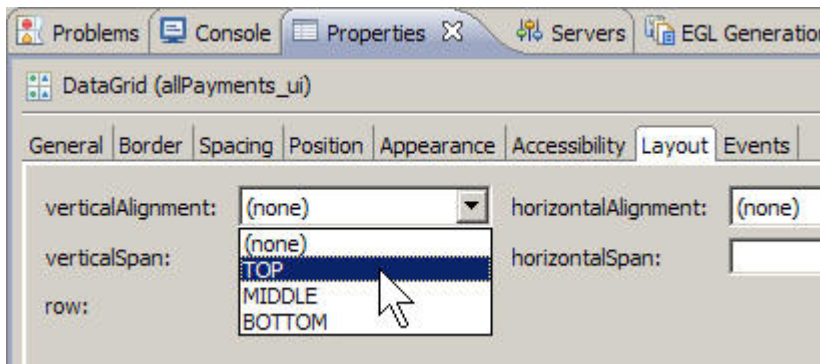
- e. Clear **Add support for formatting and validation**. Here are the completed settings:



f. Click **Finish**. The empty grid is displayed:

Type	Description	Amount due

- Click into the Properties view:
  - Ensure that the following title is displayed: **DataGrid (allPayments\_ui)**. If not, click into the data grid, ensure that the title is displayed, and click back to the Properties view.
  - On the **General** page, change the **selectionMode** property to **SINGLE**. This property indicates that the user can select only one row of the grid at a time.
  - On the **Layout** page, change the **verticalAlignment** property to **TOP**.



This property ensures that the allPayments\_ui data grid will line up with the detail grid you will add later.

- Click the **Source** tab at the bottom of the editor to see the code that you already created. Take this opportunity to reduce the width of two columns in the data grid. Specifically, consider the DataGridColumn declarations for the category and amount columns and change the width property from the default 120 pixels to 90 pixels. Here is the data grid declaration after your change:

```
allPayments_ui DataGrid {
    layoutData = new GridLayoutData
        {row = 2, column = 1
          verticalAlignment = GridLayoutLib.VALIGN_TOP},
    columns = [
        new DataGridColumn{name = "category",
                           displayName = "Type",
                           width = 90},
        new DataGridColumn{name = "description",
                           displayName = "Description",
                           width = 120},
        new DataGridColumn{name = "amount",
                           displayName = "Amount due",
                           width = 90}
    ],
    data = allPayments as any[],
    selectionMode = DataGridLib.SINGLE_SELECTION};
```

- Add prototype data in the start function, which is referenced in the **onConstructionFunction** property of the handler and which runs before the user first accesses the web page. Specifically, assign an array of records to the **data** property of the data grid:

```
function start()
    allPayments_ui.data =
    [
        new paymentRec{category = 1, description = "test01", amount = 100.00},
        new paymentRec{category = 2, description = "test02", amount = 200.00},
        new paymentRec{category = 3, description = "test03", amount = 300.00}
    ]; end
```

- To format the file, click Ctrl-Shift-F.
- Click the Preview tab.

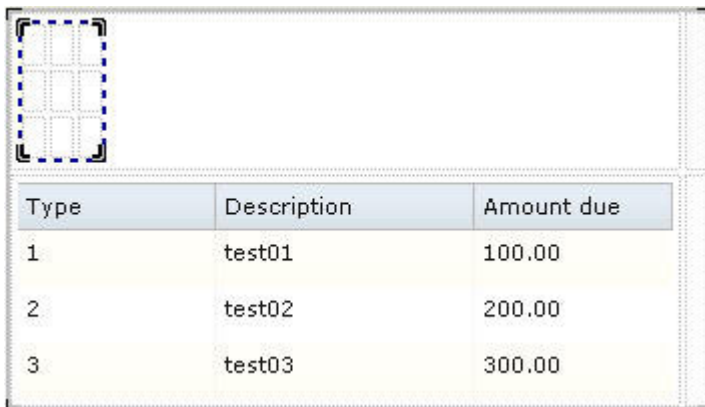
Type	Description	Amount due
1	test01	100.00
2	test02	200.00
3	test03	300.00

9. Save the file.

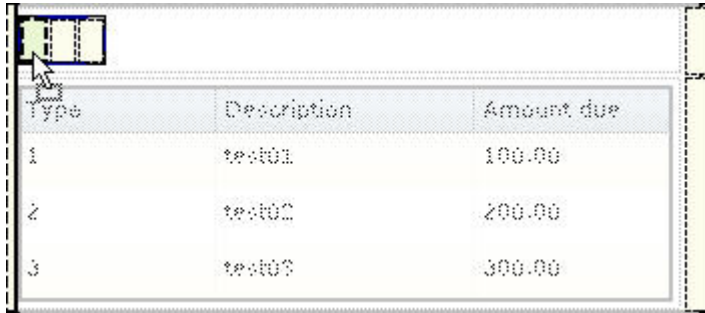
## Add the first set of buttons

To create the **Add**, **Delete**, and **Sample** buttons on the Design surface:

1. Click the Design tab.
2. In the Palette view, go to the Layout drawer and find the GridLayout widget type. Drag a new grid layout to the upper left corner of the main layout. Assign the following name to the new widget:  
buttonLayout



3. Click into the new layout, right click a cell, and notice that you can insert or delete content by using the menu.
4. Click **Delete > Row**.
5. Click again into the new layout, right click a cell, and click **Delete > Row**. A single row remains, with three columns.
6. Create the **Add** button:
  - a. In the Palette view, go to the Display and Input drawer and then to **Button (Dojo)**. Drag a Dojo Button widget to the leftmost cell of buttonLayout.



Type	Description	Amount due
1	test01	100.00
2	test02	200.00
3	test03	300.00

- b. Assign the following name to the button:

`addButton`

- c. Go to the Properties view:

- On the **General** page, change the **text** property to Add.
- On the **Events** page, select the row for the **onClick** event. A plus sign (+) is displayed at the far right of the line. Click the plus sign and specify the following name for a function that will be invoked when the user clicks the **Add** button:

`addRow`

The Source view opens to display the `addRow` function. Rather than complete the function now, finish laying out this section of the web page. Click the Design tab to return to the Design surface.

7. Create the **Delete** button:

- a. In the Palette view, go to the Display and Input drawer and then to **Button (Dojo)**. Drag a Dojo Button widget to the middle cell of `buttonLayout`.

- b. Assign the following name to the button:

`deleteButton`

- c. Go the Properties view for the button:

- On the **General** page, change the **text** property to Delete.
- On the **Events** page, assign the following function name to the **onClick** event:

`deleteRow`

- d. When the `deleteRow` function is displayed, click the Design tab.

8. Using the same process as in previous steps, create a Dojo button in the rightmost cell of `buttonLayout`. Name the button `sampleButton`, change the **text** property to Sample, and use the following name for the **onClick** function: `sampleData`. The `sampleData` function is displayed.

9. Inspect the source code, noting the code that was provided for each of the buttons.

10. Click the Preview tab.

Type	Description	Amount due
1	test01	100.00
2	test02	200.00
3	test03	300.00

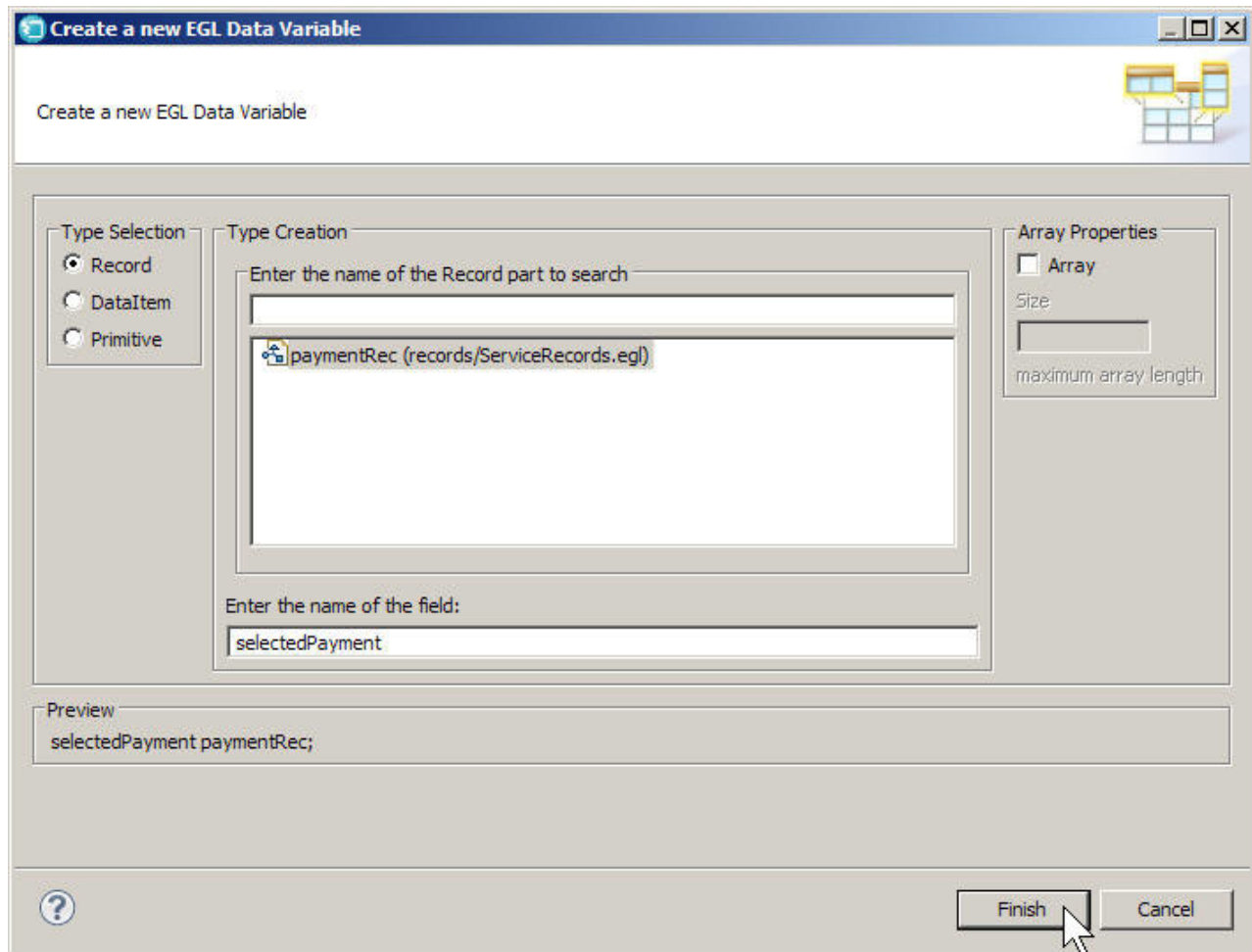
11. Save the file.

## Add a variable and layout to handle a single row

You previously created an array to hold the database rows. You now declare a variable for a single row and then drag that variable onto the Design surface to create a layout for displaying the row.

To create the variable:

1. Click the Design tab to display the Design surface.
2. Right-click the background of the EGL Data view, which is likely to be at the bottom left of the workbench. Click **New > Variable**.
3. In the Create a new EGL Data Variable wizard, request a new record variable based on the paymentRec record:
  - a. Make sure that **Type Selection** is set to **Record**.
  - b. Select the paymentRec record.
  - c. In the **Array Properties** section, make sure that the **Array** check box is cleared.
  - d. For **Enter the name of the field**, enter the following name:  
selectedPayment



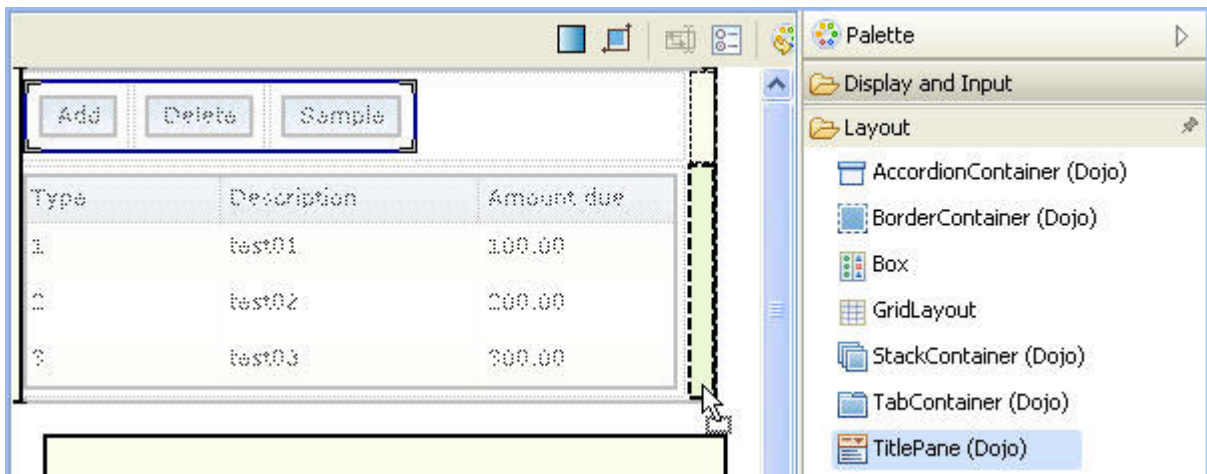
e. Click **Finish**.

As noted in the Preview section of the page shown, the following record declaration is created in the source code for the handler:

```
selectedPayment paymentRec;
```

To create the grid layout:

4. In the Palette view, go to the Layout drawer and find the TitlePane (Dojo) widget type. Drag a new title pane to the lower right cell of the main grid layout, next to the cell that holds the allPayments\_ui grid.



5. Assign the following name to the title pane:

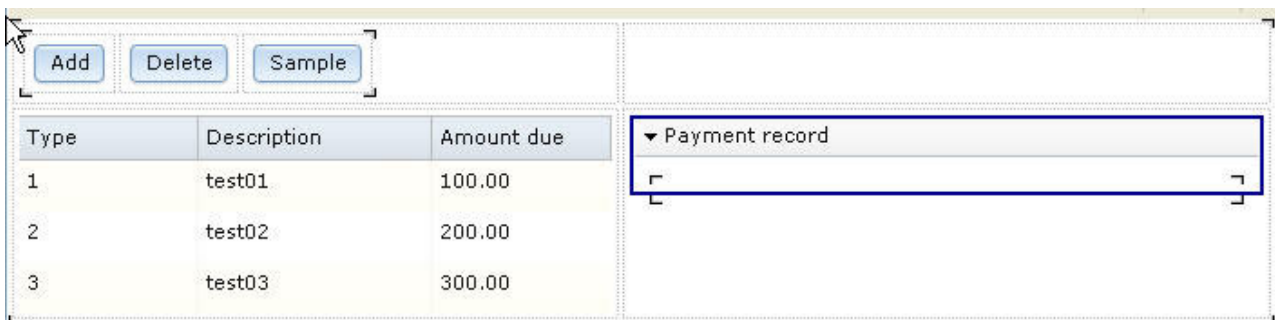
editPane

Click **OK**.

6. Make the following changes to the properties for the editPane widget:

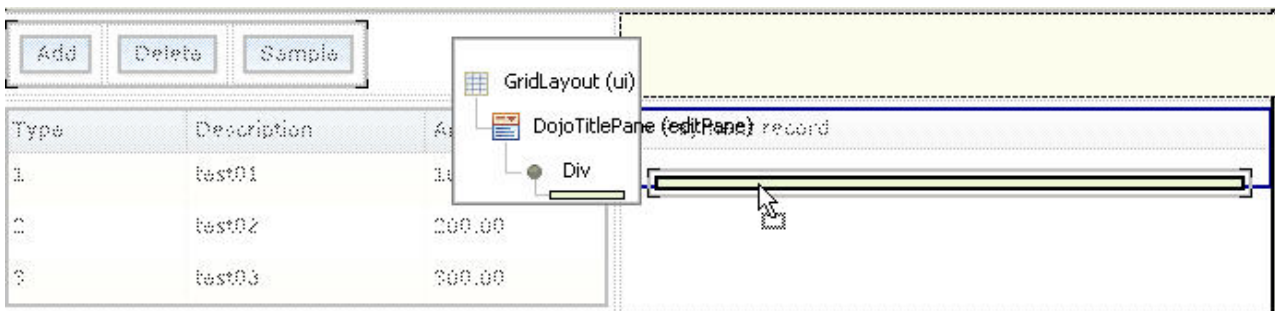
- On the **General** page, change the **title** property to Payment record
- On the **Position** page, change the **width** property to 350. This value leaves room for error messages.
- On the **Layout** page, change the **verticalAlignment** property to TOP.

The web page should now look like the following image:



7. Save the file.

8. From the EGL Data view, drag the selectedPayment variable to the bracketed area inside the payment record pane.



The Configure data widgets wizard is displayed.

9. Make the following changes:
  - a. Under **Create Widgets for**, select **Editable data**.
  - b. Make sure that the Widget Type for the selectedPayment record is GridLayout.
  - c. Change the **Label** fields as shown in the following table. These labels are used to identify the fields in the display:

*Table 1. Revised names for selectedPayment fields*

Default name	Revised name
paymentID	Key:
category	Category:
description	Description:
amount	Amount:
fixedPayment	Fixed pmt:
dueDate	Due date:
payeeName	Payee:
payeeAddress1	Address 1:
payeeAddress2	Address 2:

You must specify colons explicitly, as they are not added automatically to labels.

- d. For the category field, in the Widget Type column, click **DojoTextField**. A down arrow is displayed. Click the arrow, and then click **DojoComboBox**.
- e. For the amount field, in the Widget Type column, click **DojoTextField**. A down arrow is displayed. Click the arrow, and then click **DojoCurrencyTextBox**. This widget provides some basic formatting for currency.
- f. Ensure that **Add support for formatting and validation** is checked. The selection creates a Form Manager, which uses the EGL Rich UI Model-View-Controller (MVC) framework to manage Rich UI validation and formatting.

Here are the settings:

Insert Data

Configure data widgets

Select the data to insert and the widget types with which to display the data.

Create Widgets for:

☐ Read-only data
 ☒ Editable data
 ☐ Combination of read-only and editable data

Data to display:

Field Name	Field Type	Label	Widget Type	Widget Name
<input checked="" type="checkbox"/> selectedPayment	paymentRec	selectedPayment	GridLayout	selectedPayment_ui
<input checked="" type="checkbox"/> paymentId	int	Key:	DojoTextField	selectedPayment_paymentId_field
<input checked="" type="checkbox"/> category	int	Category:	DojoComboBox	selectedPayment_category_comboBox
<input checked="" type="checkbox"/> description	string	Description:	DojoTextField	selectedPayment_description_field
<input checked="" type="checkbox"/> amount	money(16,2)	Amount:	DojoCurrencyTextBox	selectedPayment_amount_textBox
<input checked="" type="checkbox"/> fixedPayment	boolean	Fixed pmt:	DojoCheckBox	selectedPayment_fixedPayment_checkBox
<input checked="" type="checkbox"/> dueDate	date	Due date:	DojoDateTextBox	selectedPayment_dueDate_textBox
<input checked="" type="checkbox"/> payeeName	string	Payee:	DojoTextField	selectedPayment_payeeName_field
<input checked="" type="checkbox"/> payeeAddress1	string	Address 1:	DojoTextField	selectedPayment_payeeAddress1_field
<input checked="" type="checkbox"/> payeeAddress2	string	Address 2:	DojoTextField	selectedPayment_payeeAddress2_field

☒ Add support for formatting and validation
 

☐ Use error label to support error message

g. Click **Finish**.

The new grid layout contains a form.

Key:

Category:

Description:

Amount:

\$0.00

Fixed pmt:

☐

Due date:

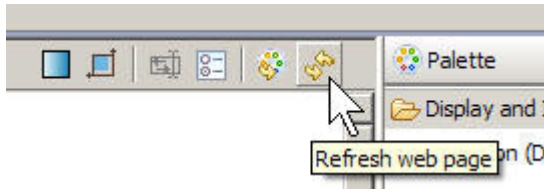
May 3, 2011

Payee:

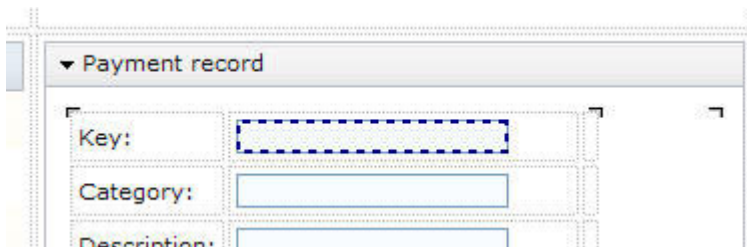
Address 1:

Address 2:

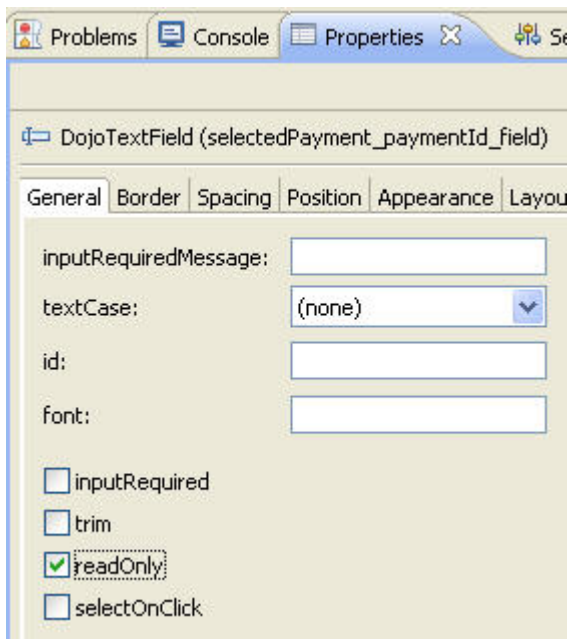
**Note:** You might need to click the Refresh button in the upper right corner of the Rich UI editor to see this change:



10. Make the Key field read-only:
  - a. Repeatedly click the Dojo text field next to the Key label until only that field is surrounded by a dotted line.



- b. In the Properties view, **General** page, select the **readOnly** check box.

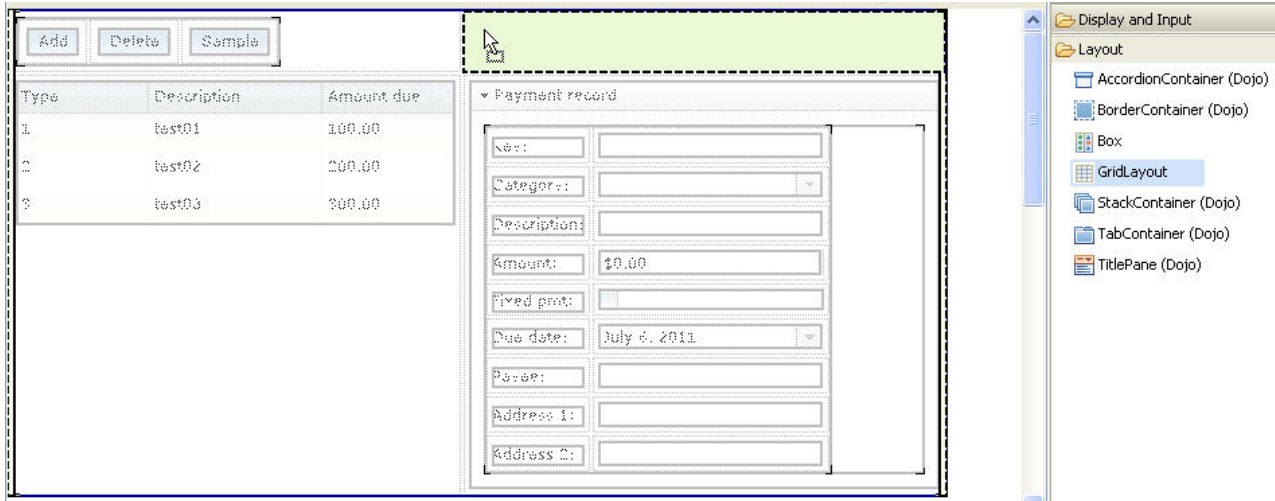


11. For a more uniform appearance, do as follows:
  - a. Click the DojoCurrencyTextBox widget for **Amount** until only that widget is surrounded by a dotted line.
  - b. On the **Position** page of the Properties view, set the **width** property to 166.

## Add the second set of buttons

To add the **Clear** and **Save** buttons:

1. In the Palette view, go to the Layout drawer and find the GridLayout widget type. Drag a new grid layout to the upper right corner of the main layout and assign the following name:  
detailButtonLayout



2. With the new layout selected, update the number of rows and columns in whichever way you prefer: in the Properties view, or by deleting the rows and columns, or by changing the source code. In any case, ensure that the layout has 1 row and 2 columns.
3. At the Design surface, create the **Clear** button:
  - a. In the Palette view, go to the Display and Input drawer and then to **Button (Dojo)**. Drag a Dojo button to the first cell of the new layout.
  - b. Using the same process as was used earlier, name the button `clearButton`, change the **text** property to `Clear`, and use the following name for the **onClick** function: `clearAllFields`. The `clearAllFields` function is displayed.
4. Create the **Save** button:
  - a. Click the Design tab.
  - b. In the Palette view, go to the Display and Input drawer and then to **Button (Dojo)**. Drag a Dojo button to the second cell of the new layout.
  - c. Name the button `saveButton` and change the **text** property to `Save`.
  - d. On the **Events** page, select the **onClick** event and click the down arrow in the *second* column to display the available function names. Click **selectedPayment\_form\_Submit**, which is a function that EGL created automatically when you dragged the `selectedPayment` record variable onto the user interface.
  - e. Click the Preview tab.

Add
Delete
Sample

Clear
Save

Type	Description	Amount due
1	test01	100.00
2	test02	200.00
3	test03	300.00

▼ Payment record

Key:

Category:

Description:

Amount:

Fixed pmt: ☐

Due date:






Payee:

Address 1:

Address 2:

- f. Save the file, which should match the finished code in “Code for PaymentFileMaintenance.egl after lesson 4” on page 68.

#### Related reference

-  “Rich UI overview” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>
-  “Rich UI DataGrid and DataGridTooltip” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>
-  “Rich UI GridLayout” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>
-  “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

In this lesson, you completed the following tasks:

- Created a Rich UI handler.
- Created variables in the EGL Data view.
- Created a data grid by dragging a record array variable onto the editor.
- Adjusted widgets in the Properties view and by using a menu.
- Worked in all three tabs of the Rich UI editor, updating the source and previewing the web page.

In the next lesson, you create the service that will access the database.

## Lesson 5: Create the service

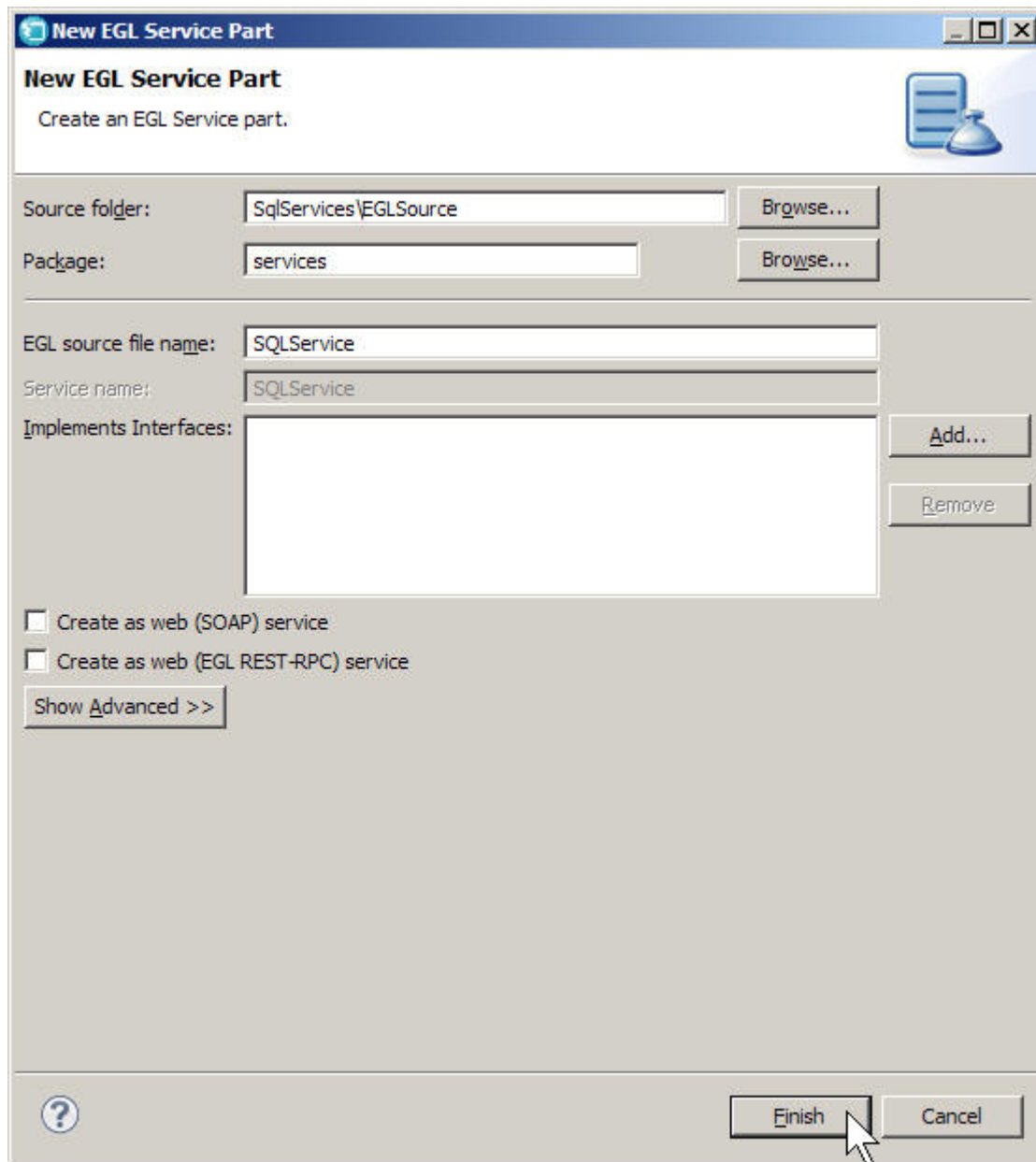
Create a dedicated service to access the database.

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

To create a Service part:

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

end
```
5. Save the file, but do not close it.

## Lesson checkpoint

You learned how to create an EGL Service part.

In the next lesson, you add code for the functions to SQLService.

### Related reference

## Lesson 6: Add code for the service functions

In EGL, I/O statements such as **add** and **get** access data that resides in different kinds of persistent data storage, from file systems to queues to databases. The coding is similar for the different cases.

In this lesson, you add functions that access rows in a relational database. Add the functions in order, before the final **end** statement in `SQLService.egl`.

### Add a payment record

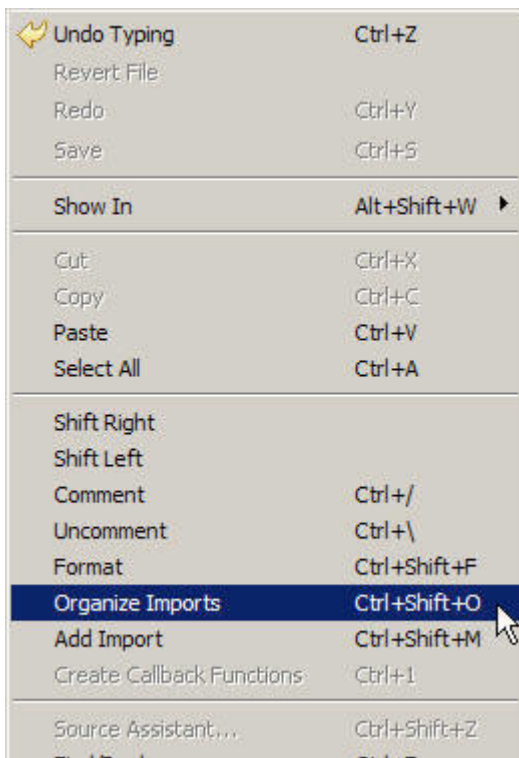
The `addPayment()` function adds a new row to the database.

To code the function:

1. In the EGL editor, copy and paste the following lines into `SQLService.egl` before the **end** statement:

```
function addPayment(newPayment paymentRec inOut)
  add newPayment;
end
```
2. Before you continue, you must resolve the reference to the `paymentRec` Record part. You can automatically create **import** statements by using the Organize Imports feature. Right-click any blank area in the editor and click **Organize Imports**.

EGL adds the following statement to the beginning of the file:

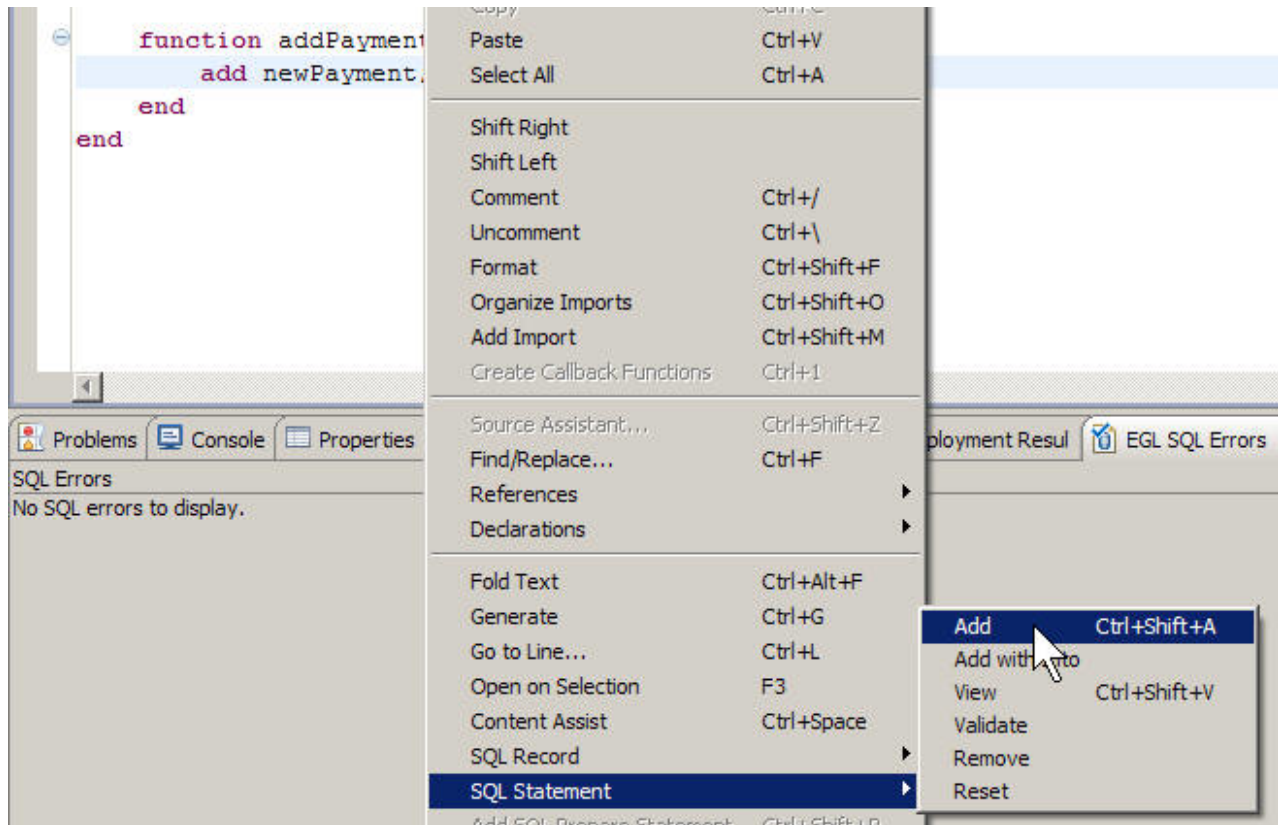


```
import records.paymentRec;
```

The reference is now resolved. You will use this feature often, whether by selecting the menu item or by pressing Ctrl-Shift-O.

3. Save the file (Ctrl-S), and then place your cursor anywhere in the **add** statement. Right-click and select **SQL Statement > Add**.

This feature changes the implicit SQL that underlies the EGL **add** statement



into embedded code that you can modify.

```
function addPayment(newPayment paymentRec inOut)
  add newPayment with
    #sql{
      insert into PAYMENT
        (PAYMENT_ID, CATEGORY, DESCRIPTION, AMOUNT,
        FIXED_PAYMENT, DUE_DATE, PAYEE_NAME, PAYEE_ADDRESS1,
        PAYEE_ADDRESS2)
      values
        (:newPayment.paymentId, :newPayment.category,
        :newPayment.description, :newPayment.amount,
        :newPayment.fixedPayment, :newPayment.dueDate,
        :newPayment.payeeName, :newPayment.payeeAddress1,
        :newPayment.payeeAddress2)
    };
end
```

4. Because the `paymentID` field is auto-generated, you must not overwrite it:
  - a. Delete `PAYMENT_ID` and subsequent comma from the INSERT list.

- b. Delete :newPayment.paymentId and subsequent comma from the VALUES list.

**Note:** In keeping with SQL terminology, each variable that is referenced in an SQL statement is called a *host variable*. The word *host* refers to the language that embeds the SQL statement; in this case, EGL. For example, the initial colon in :newPayment.paymentId indicates a host variable.

The revised **add** statement looks like the following image:

```
function addPayment(newPayment paymentRec inOut)
  add newPayment with
    #sql{
      insert into PAYMENT
        (CATEGORY, DESCRIPTION, AMOUNT,
        FIXED_PAYMENT, DUE_DATE, PAYEE_NAME, PAYEE_ADDRESS1,
        PAYEE_ADDRESS2)
      values
        (:newPayment.category,
        :newPayment.description, :newPayment.amount,
        :newPayment.fixedPayment, :newPayment.dueDate,
        :newPayment.payeeName, :newPayment.payeeAddress1,
        :newPayment.payeeAddress2)
    };
end
```

5. Save the file.

#### Related reference

- “add considerations for SQL” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>
- “Functions” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>
- “import” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>
- “SQL data access” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

## Read all database records

The getAllPayments function reads all of the records from the table and stores them in an array.

To code the function:


1. In the EGL editor, copy and paste the following lines into SQLService.egl before the **end** statement:

```
function getAllPayments() returns (paymentRec[])
  paymentArray paymentRec[];
  get paymentArray;
  return (paymentArray);
end
```

The EGL **get** statement generates an SQL SELECT statement to retrieve a result set. When the target of the **get** statement is a dynamic array of records, EGL retrieves all matching rows from the result set and inserts each successive row into the next array element.

2. Save the file.

#### Related reference

 “get considerations for SQL” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

## Replace a record

The `editPayment` function replaces an existing row in the database with an edited version. The function assumes that the user previously read the row from the database.

To code the function:


1. In the EGL editor, copy and paste the following lines into `SQLService.egl` before the **end** statement:

```
function editPayment(chgPayment paymentRec inOut)
    replace chgPayment nocursor;
end
```

The EGL **replace** statement generates an SQL UPDATE statement.

2. Save the file.

#### Related reference

 “replace considerations for SQL” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

## Delete a record

The `deletePayment` function deletes the specified record from the table.

To code the function:

1. In the EGL editor, copy and paste the following lines into `SQLService.egl` before the **end** statement:

```
function deletePayment(delPayment paymentRec inOut)

    try
        delete delPayment nocursor;

        onException(exception SQLException)
            if(SQLLib.sqlData.sqlState != "02000") // sqlState is of type CHAR(5)
                throw exception;
            end
        end
    end
end
```

The EGL **delete** statement generates an SQL DELETE statement. If no rows are present, the Derby database returns an SQLState value of "02000", and the EGL runtime code throws an exception that the function *catches*: that is, processes in some `onException` logic.


When a function catches but ignores an exception, processing continues without interruption. That rule applies to the preceding logic, when the value of SQLState is "02000". When a function uses the **throw** statement to *throw* an


exception, the exception stays active. That rule also applies to the preceding logic, when the value of `SQLState` is other than "02000".

At run time, if a service does not handle an exception, the service requester receives an exception of type `ServiceInvocationException`. Incidentally, if the service cannot be accessed, the requester receives an exception of type `ServiceInvocationException` or `ServiceBindingException`, depending on the details of the error.

2. Save the file.

#### Related reference

 "delete considerations for SQL" at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

 "Exception handling" at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

## Create test data

The `createDefaultTable` function creates a set of data for testing your completed application.

To code the function:

1. In the EGL editor, copy and paste the following lines into `SQLService.egl` before the **end** statement:


```
function createDefaultTable() returns (paymentRec[])  
  
    try  
        execute #sql{  
            delete from PAYMENT  
        };  
  
        onException(exception SQLException)  
  
            if (SQLLib.sqlData.sqlState != "02000") // sqlState is of type CHAR(5)  
                throw exception;  
            end  
        end;  
  
        ispDate DATE = dateTimeLib.dateValueFromGregorian(20110405);  
        addPayment(new paymentRec{category = 1, description = "Apartment",  
            amount = 880, fixedPayment = YES});  
        addPayment(new paymentRec{category = 2, description = "Groceries",  
            amount = 450, fixedPayment = NO});  
        addPayment(new paymentRec{category = 5, description = "ISP",  
            amount = 19.99, fixedPayment = YES, dueDate = ispDate });  
        return (getAllPayments());  
    end
```


The code acts as follows:

- The EGL **execute** statement runs a literal SQL statement that deletes all rows from the `PAYMENT` table.
- The `ispDate` variable receives a date value from the **dateTimeLib.dateValueFromGregorian()** system function. The content of the variable is then in a format that is appropriate for insertion into the `dueDate` field in the database.
- The `addPayment` function is repeatedly invoked to add new rows to the `PAYMENT` table.
- The call to the `getAllPayments` function returns an array of rows that were retrieved from the table.

2. Press Ctrl-Shift-F to format the code. If you see any red Xs, compare your code with the finished code in “Finished code for SQLService.egl after lesson 6” on page 73
3. Save and close the file.

#### Related reference

 “execute considerations for SQL” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

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

## Lesson checkpoint

You learned how to complete the following tasks:

- Add embedded SQL code to a program and modify that code
- Automatically create and organize **import** statements

In the next lesson, you will create a widget to hold the table of expense data.

---

## Lesson 7: Create a library of reusable functions

Create a library to format money values and to associate category numbers with descriptions.

Libraries contain functions, constants, and variables that you can use in multiple locations.

When you reference a declaration in the library from other logic such as a service or handler, you can include the library name as a prefix. For example, `MyLibrary.myLibraryVariable` is appropriate if the library name is `MyLibrary` and the library includes the `myLibraryVariable` variable. Alternatively, you can include the library name in a **use** statement in the other logic and avoid the need to qualify every reference. In that case, `myLibraryVariable` is sufficient to reference that variable.

## Create a Library part

To create a Library part:

1. Right-click the `PaymentClient` folder, then click **New > Library**.
2. In the New EGL Library window, enter the following information:
  - In the **EGL source file name** field, enter the following name:  
`PaymentLib`
  - In the **Package** field, enter the following name:  
`libraries`
  - Under **EGL Library Type**, leave the default value of **Basic** selected.

The new Library part opens in the EGL editor.

3. Replace the boilerplate code in the Library part with the following lines:

```
package libraries;

library PaymentLib type BasicLibrary {}

end
```

4. Save the file.

## Create the categories array

Add the following code before the final **end** statement:

```
categories STRING[] = [  
    "Rent",           // 1  
    "Food",           // 2  
    "Entertainment", // 3  
    "Automotive",     // 4  
    "Utilities",      // 5  
    "Clothes",        // 6  
    "Other"           // 7  
];
```

The value is an array, and as is true of all arrays in EGL, the index of the first element is 1, not 0.

The array is used in logic that acts as follows:

- Places an expense category into the database in integer form, to save space.
- Places the expense category onto the web page in string form, for clarity.

## Create the get functions for categories

The next functions convert between the following two formats for expense categories: integer and string.

1. Add the following code before the final **end** statement:

```
function getCategoryDesc(cat INT in) returns(STRING)  
    if(cat)           // the integer is not 0  
        return(categories[cat]);  
    else  
        return("");  
    end  
end
```

The function receives the integer format of an expense category and returns the related array element. If the input value is 0, the function returns an empty string.

2. Add the following code before the final **end** statement:

```
function getCategoryNum(desc STRING in) returns(INT)  
    for(i INT from 1 to categories.getSize())  
        if(categories[i] == desc)  
            return(i);  
        end  
    end  
    return(0); // no match  
end
```

This function receives the string format of an expense category and returns the integer format, if possible. If no match is found for the received string, the function returns 0.

3. Format the file.
4. Save and close the PaymentLib Library. If you see errors in your source file, compare your code to the file contents in “Finished code for PaymentLib.egl” after lesson 7” on page 74.

### Related reference

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

## Lesson checkpoint

You learned how to complete the following tasks:

- Create a Library part.
- Add functions and a variable to a library.

---

## Lesson 8: Add variables and functions to the Rich UI handler

Add source code that supports the user interface.

In lessons 8 and 9, you update the EGL source code directly and review changes in the Preview tab.

### Add code to support the data grid

Change the declaration of the data grid for two purposes: to cause the web page to react when the user selects a cell and to ensure that the grid output is formatted correctly.

1. In Project Explorer, open **PaymentClient > EGLSource > handlers** and double-click **PaymentFileMaintenance.egl**.

2. Click on the Source tab.

Make the following changes, ignoring the error marks:

3. In the `allPayments_ui` `DataGrid` declaration, add the following code immediately before the **columns** property:

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

The **selectionListeners** property specifies one or more functions that are called whenever the user selects a cell in the grid. In this case, you are appending a function name to a pre-existing array. You will write the `cellClicked` function later in this lesson.

4. Formatters are functions that change the appearance of the values in `DataGrid` columns. To demonstrate the feature, find the `DataGridColumn` declaration for `category`. To ensure that the user sees a category description rather than an integer, add this code after `width=90`:

```
, formatters = [ formatCategory ]
```

5. When you display dollar amounts in a column, you typically right-align the values. You do not need to code a function to cause right-alignment. Instead, add this code after the **width** entry for `amount`:

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

The `allPayments_ui` declaration is now as follows, with error marks shown for `cellClicked` and `formatCategory`:

```
allPayments_ui DataGrid {
    layoutData = new GridLayoutData
        {row = 2, column = 1,
         verticalAlignment = GridLayoutLib.VALIGN_TOP},
    selectionListeners ::= cellClicked,
    columns =[
        new DataGridColumn{name = "category",
                           displayName = "Type",
                           width = 90,
                           formatters = [formatCategory]},
        new DataGridColumn{name = "description",
                           displayName = "Description",
                           width = 120},
        new DataGridColumn{name = "amount",
                           displayName = "Amount due",
```

```

width = 90,
alignment = DataGridLib.ALIGN_RIGHT}
],
data = allPayments as any[],
selectionMode = DataGridLib.SINGLE_SELECTION);

```

6. Save the file.

## Code the function that responds when the user clicks the data grid

The `cellClicked` function is invoked when the user clicks a cell in the data grid.

Immediately below the `start` function, add the following lines:

```

function cellClicked(myGrid DataGrid in)
    selectedPayment = allPayments_ui.getSelection()[1] as paymentRec;
    selectedPayment_form.publish();
end

```

First, the `cellClicked` function updates the `selectedPayment` record with data from a single data-grid row. That row can include more fields than are displayed to the user. In this application, the single row in the data grid will have come from a single row in the database.

Second, the **publish** function causes the transfer of data from the `selectedPayment` record to the `selectedPayment_ui` layout. That transfer is made possible by code that was provided for you when you created the `selectedPayment_ui` layout, which is the single-record layout at the right of your web page. If you review the code, you can trace the relationships:

- A Form Manager declaration includes form fields.
- Each form field references a controller declaration.
- The controller declaration relates a model to a view; in this case, a field of the `selectedPayment` record to a child of the `selectedPayment_ui` layout.

The Form Manager provides various benefits but is essentially a collection of controllers.

Here is an explanation of two other issues—the use of the bracketed array index (`[1]`), and the use of the **as** operator:

- The **getSelection** function always returns a subset of the rows in the **data** array of the data grid. However, when you declared the data grid, you specified the following setting to indicate that the user can select only one row at a time: `selectionMode = DataGridLib.SINGLE_SELECTION`. When the user can select only one row, only one element is available.
  - Every element in the array returned by a **getSelection** function is of type ANY. You typically use the same Record part to process input to the grid and to process output from the grid, and in this tutorial, the Record part is `paymentRec`. The Record part has the following uses:
    - To be the basis of the array elements that you assign to the **data** property of the data grid, as shown in the following setting:
 

```
data = allPayments as any[]
```
    - To cast the array element that is returned by the **getSelection** function of the data grid, as shown here:
 

```
allPayments_ui.getSelection()[1] as paymentRec
```
- In each case, the **as** clause provides the necessary cast.

## Format column values in the grid

To add the formatter function:

1. Add the following code before the final **end** statement in the file:

```
function formatCategory(class string, value string, rowData any in)
    value = PaymentLib.getCategoryDesc(value as INT);
end
```

Formatters have the parameters shown. In this case, the formatter wraps a library function you created earlier.

2. Press Ctrl-Shift-O to organize the required import statements and save the file. All the error marks disappear.

## Test the formatting of the data grid and the transfer of data to the single-record layout

You can test your recent changes even before you gain access to the database.

1. Click the Preview tab and note that the categories are now descriptions (for example, "Rent" rather than "1").

Type	Description	Amount due
Rent	test01	\$100.00
Food	test02	\$200.00
Entertainment	test03	\$300.00

2. Click one or another row in the data grid and note that the single-row layout is updated appropriately. However, the formatter affected only the data grid, and the description field in the single-row layout contains a numeric. The tutorial will address that issue later.
3. Click the Source tab and change the start function so that the first record in the prototype data includes a value for payeeName, which is a paymentRec record field that is not displayed by the data grid:

```
function start()
    allPayments_ui.data =[
        new paymentRec{
            category = 1, description = "test01", amount = 100.00
            , payeeName = "Someone"},
        new paymentRec{category = 2, description = "test02", amount = 200.00},
        new paymentRec{category = 3, description = "test03", amount = 300.00}];
end
```

4. Click the Preview tab and click the first row in the data grid.

Add
Delete
Sample

Clear
Save

Type	Description	Amount due
Rent	test01	\$100.00
Food	test02	\$200.00
Entertainment	test03	\$300.00

▼ Payment record

Key:

Category:

Description:

Amount:

Fixed pmt: ☐

Due date:

Payee:

Address 1:

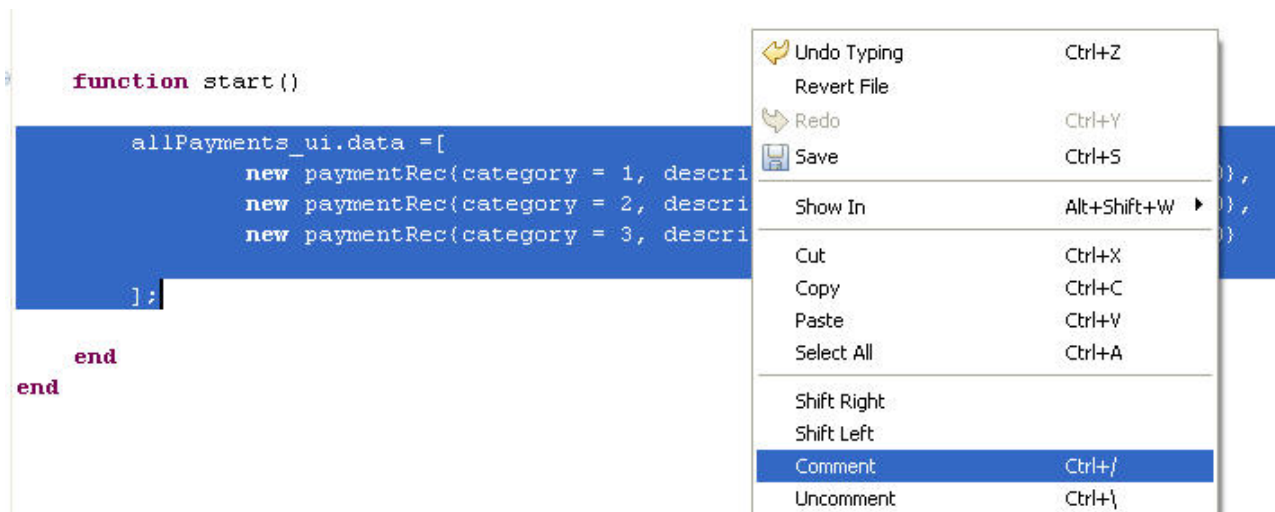
Address 2:

As shown, you can switch quickly from one tab in the Rich UI editor to another, to test even a small change.

## Comment the prototype data

You can comment or uncomment code quickly, as shown in this step.

1. Click the Source tab.
2. In the start function, select the complete assignment statement, right-click the area selected, and click Comment.



3. Comment marks (//) are now at the start of each line. You could remove the comments by repeating the task and clicking **Uncomment** instead of **Comment**. However, leave the comments in place. EGL also supports the use of slash asterisk (/\*) and asterisk slash (\*/) delimiters, as shown here:

```
/*
```

You can add comments in either of two ways.

```
*/
```

## Declare a service-access variable

You now declare a service-access variable, which will let you communicate with the service that you defined earlier.

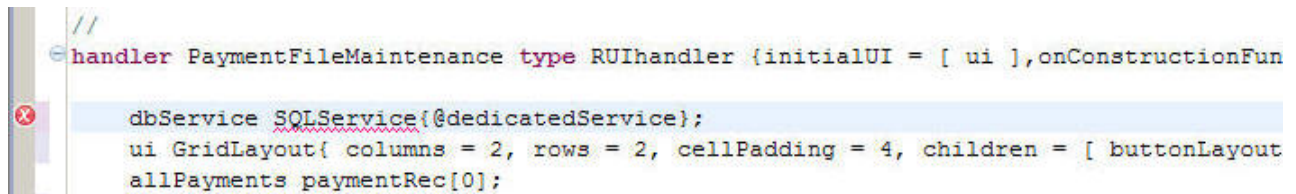
To create the variable:

1. Near the top of the EGL source code, find the handler declaration for `PaymentFileMaintenance`. Add a blank line, and immediately before the `ui GridLayout` declaration, add the following statement:

```
dbService SQLService{@dedicatedService};
```

The `@dedicatedService` property indicates that the service being referenced is a dedicated service, which will be deployed with the Rich UI handler.

In the following display, the red X in the margin indicates a problem in the code:



```
//
handler PaymentFileMaintenance type RUIhandler {initialUI = [ ui ],onConstructionFun

dbService SQLService{@dedicatedService};
ui GridLayout{ columns = 2, rows = 2, cellPadding = 4, children = [ buttonLayout
allPayments paymentRec[0];
```

To see the error message, move the cursor over the X.

2. Fix the “unresolved type” error by pressing `Ctrl+Shift+O`. The new **import** statement provides access to the `services` package, `SQLService` part, which is in the **PaymentService** project. The reference to `SQLService` is resolved because that project is on the EGL build path of the **PaymentClient** project.
3. Save the file.

## Create functions that use the service-access variable to invoke the service

You now create several functions to invoke different functions in the dedicated service. Once you understand how to set up one invocation, the others are straightforward.

Begin by creating the function that reads all data.

1. Leave a blank line after the `cellClicked` function and add the following code:

```
function readFromTable()
    call dbService.getAllPayments() returning to updateAll
    onException serviceLib.serviceExceptionHandler;
end
```

### Note:

- a. 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.

b. The asynchronous **call** statement includes two function names:

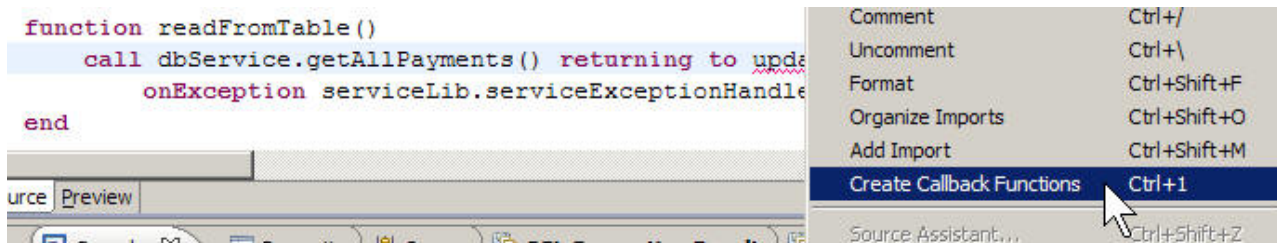
- updateAll
- serviceLib.serviceExceptionHandler

The two are *callback functions*, which are invoked by the EGL runtime code after the service responds or fails. If the service returns a value successfully, the updateAll function is invoked. If the call fails, the EGL runtime code invokes a function that is associated with the name serviceLib.serviceExceptionHandler.

By default, an error results in the display of error information in the Console view (at development time) or at the bottom of the web page (at run time). However, you can specify an error handler of your own, typically by assigning a function name in place of serviceLib.serviceExceptionHandler.

2. Click anywhere in the **call** statement, right-click, and click **Create Callback Functions**. Alternatively, you could click anywhere in the statement, held down the Ctrl key, and pressed 1.

EGL creates an empty updateAll function. An error handler would have been



created as well if you had specified a function name for **onException**, other than serviceLib.serviceExceptionHandler.

The parameter list in the created updateAll function is equivalent to the type of return value that is expected from the service. Here are the relationships that explain the behavior of the Rich UI editor:

- The parameter list in the callback function is correct because the getAllPayments function in the Service part is available to the editor.
- The function is available because you resolved the reference to the SQLService part in a previous step.

Next, create the function that adds sample data.

3. Click Ctrl-F to gain access to the Find/Replace dialog, type SampleData, and click **Find**.
4. Update the sampleData function so that the code is as follows:

```
function sampleData(event Event in)  
  call dbService.createDefaultTable() returning to updateAll  
  onException serviceLib.serviceExceptionHandler;  
end
```

You do not use the **Create Callback Functions** feature because the callback functions exist.

Next, create the function that adds data.

5. Update the addRow function so that the code is as follows:

```
function addRow(event Event in)  
  call dbService.addPayment(new paymentRec) returning to recordAdded  
  onException serviceLib.serviceExceptionHandler;  
end
```

- Click anywhere in the **call** statement, right-click, and click **Create Callback Functions**. EGL adds the recordAdded function.

Create the function that deletes data.

- Update the deleteRow function so that the code is as follows:

```
function deleteRow(event Event in)

    for(i INT from 1 to allPayments.getSize())
        if(allPayments[i].paymentID == selectedPayment.paymentID)
            allPayments.removeElement(i);
            exit for;
        end
    end

    call dbService.deletePayment(selectedPayment) returning to recordRevised
    onException serviceLib.serviceExceptionHandler;
end
```

The function acts as follows:

- Deletes the selected row from the local array of records
  - Calls the database service to delete the row from the database itself
- Click anywhere in the **call** statement, right-click, and click **Create Callback Functions**. EGL adds the recordRevised function.
  - Press Ctrl-Shift-F to format the code.
  - Save the file.

#### Related information:

 “Invoking a service asynchronously from a Rich UI application” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>

## Update the start function to initialize the data grid with database rows

To initialize the data grid, add the following code before the **end** statement of the start function:

```
readFromTable();
```

Although you could have assigned the readFromTable function directly to the **onConstructionFunction** property, you are advised to retain the start function as a separate unit of logic in case you later decide to add other code that runs before the web page is rendered.

Retain the commented code in the start function in case you need to test the web page without accessing the database. You can use the comment and uncomment capability of the Rich UI editor to quickly switch from the function call to the prototype data and back again.

## Complete the callback functions

You now complete the callback functions that were created automatically:

- updateAll
- recordAdded
- recordRevised

The updateAll function receives an array of paymentRec records from the dedicated service. The function is called in the following ways:

- As a callback function at startup, after the readFromTable function calls the service.
- As a callback function whenever the user clicks the **Sample** button to invoke the sampleData function.

1. Update the updateAll function so that the code is as follows:

```
function updateAll(retResult paymentRec[] in)
    allPayments = retResult;
    allPayments_ui.data = allPayments as any[];
end
```

The function updates the global array of payment records with the data received from the service and then refreshes the data grid.

The recordAdded function receives the record that was sent to and returned by the service function addPayment.

2. Update the recordAdded function so that the code is as follows:

```
function recordAdded(newPayment paymentRec in)
    readFromTable();
end
```

The function readFromTable reads all the rows from the database. The data stored by the grid can then contain the new row, including the paymentID value that was automatically generated by the database and that is otherwise unavailable to the grid.

The recordRevised function receives the record that was sent to and returned by the service function addPayment.

3. Update the recordRevised function so that the code is as follows:

```
function recordRevised(delPayment paymentRec in)
    allPayments_ui.data = allPayments as any[];
end
```

The function refreshes the data grid.

4. Press Ctrl-Shift-F to format the code. If you see errors in your source file, compare your code to the file contents in “Code for PaymentFileMaintenance.egl after lesson 8” on page 74.
5. Save the file.

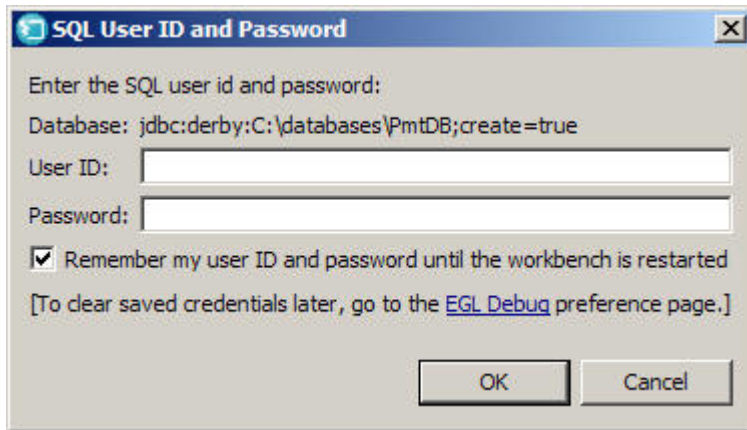
## Test the interface

Preview your work now that you are accessing a database.

1. Click the Preview tab. The data grid has no content because you commented out the prototype data, and the database has no rows.

Type	Description	Amount due
------	-------------	------------

2. Click **Sample** to create sample data.
3. If EGL requests a password, enter admin for both the **User ID** and **Password** fields. Select **Remember my user ID...** and click **OK**.



SQL User ID and Password

Enter the SQL user id and password:

Database: jdbc:derby:C:\databases\PmtDB;create=true

User ID:

Password:

☒ Remember my user ID and password until the workbench is restarted  
 [To clear saved credentials later, go to the [EGL Debug](#) preference page.]

OK Cancel

If you exit and restart the workbench before you complete this tutorial, this window might be re-displayed the next time you attempt to access the database. Eventually the grid is re-displayed with rows of sample data.

Add Delete Sample

Type	Description	Amount due
Rent	Apartment	\$880.00
Food	Groceries	\$450.00
Utilities	ISP	\$19.99

- Click the **Add** button. A new row with a single default value is displayed at the bottom of the grid.

Add Delete Sample

Type	Description	Amount due
Rent	Apartment	\$880.00
Food	Groceries	\$450.00
Utilities	ISP	\$19.99
		\$0.00

- Select the Apartment row and click **Delete**. The row is deleted from both the display and the database.

Type	Description	Amount due
Food	Groceries	\$450.00
Utilities	ISP	\$19.99
		\$0.00

6. Click the first row of the data grid.

Data from the database was transferred from the data grid to the single-record

Type	Description	Amount due
Food	Groceries	\$450.00
Utilities	ISP	\$19.99
		\$0.00

▼ Payment record

Key:

69

Category:

2

Description:

Groceries

Amount:

\$450.00

Fixed pmt:

☐

Due date:

July 7, 2011

Payee:

Address 1:

Address 2:

layout. Note that the value of the **Key** field reflects how many rows were added to the database and will probably not match the value on your web page.

## Lesson checkpoint

You learned how to complete the following tasks:

- To create formatters.
- To respond to the user's selection in a data grid.
- To transfer data from the data grid to a grid layout.
- To comment and uncomment code.
- To access services from a Rich UI application.

In the next lesson, you will complete the code for the Rich UI handler.

## Lesson 9: Complete the code that supports the user interface

Next, you will complete the single-row layout, as well as the code that supports the **Clear** and **Save** buttons.

## Complete the layout that displays a single row

To complete the single-row layout:

1. Click the Source tab, if necessary.
2. Locate the `selectedPayment_category_comboBox` declaration. In the set-values block, in place of the brackets for the **values** property, specify the `PaymentLib.categories` array. The list of values in the combo box will now be the values in the categories array that you created in the `PaymentLib` library. Here is the changed declaration:

```
selectedPayment_category_comboBox DojoComboBox{  values = PaymentLib.categories,  
    layoutData = new GridLayoutData{row = 2, column = 2}};
```

3. To set the value of that combo box to a category description rather than an integer, update the `cellClicked` function to access a library function that you coded earlier:

```
function cellClicked(myGrid DataGrid in)  
    selectedPayment = allPayments_ui.getSelection()[1] as paymentRec;  
    selectedPayment_form.publish();  
    selectedPayment_category_comboBox.value =  
        PaymentLib.getCategoryDesc(selectedPayment.category);  
end
```

4. Save the file, but do not close it.

## Test the new code

Review the effect of your last change.

1. Click the **Preview** tab.
2. Click the first line of sample data. The single-record layout now displays the category name rather than an integer.

AddDeleteSample

ClearSave

Type	Description	Amount due
Food	Groceries	\$450.00
Utilities	ISP	\$19.99
		\$0.00

▼ Payment record

Key: 69

Category: Food

Description: Groceries

Amount: \$450.00

Fixed pmt: ☐

Due date: July 7, 2011

Payee:

Address 1:

Address 2:

## Complete the code for the second set of buttons

When the user clicks **Clear** to remove nondefault content from the single-record layout, the `clearAllFields` function runs. The function sets up the layout so that when the user types data and clicks **Save**, the new typed data updates an existing database row.

1. Click the Source tab.
2. Find the `clearAllFields` function and make it as follows:

```
function clearAllFields(event Event in)
    saveID INT = selectedPayment.paymentID; // retain the key
    selectedPayment = new PaymentRec{};
    selectedPayment.paymentID = saveID;
    selectedPayment_form.publish();
end
```

The code retains the record key for use in a subsequent update of the database. The code then creates a record, assigns it to the `selectedPayment` variable, assigns the saved key value to that variable, and publishes the variable to the single-record layout.

3. Complete the function that is invoked when the user clicks **Save**:
  - a. Find the function, which is named `selectedPayment_form_Submit`.
  - b. Make the function as follows:

```
function selectedPayment_form_Submit(event Event in)
    selectedPayment_category_comboBox.value
        = PaymentLib.getCategoryNum(selectedPayment_category_comboBox.value);

    if (selectedPayment_form.isValid())
        selectedPayment_form.commit();
        selectedPayment_category_comboBox.value =
            PaymentLib.getCategoryDesc(selectedPayment_category_comboBox.value);

        // update allPayments with new version of selectedPayment
        for(i INT from 1 to allPayments.getSize())
            if(allPayments[i].paymentID == selectedPayment.paymentID)
                allPayments[i] = selectedPayment;
            exit for;
        end
    end

    call dbService.editPayment(selectedPayment)
        returning to recordRevised
        onException serviceLib.serviceExceptionHandler;
end
```

The following clause checks the validity of copying the widget content to the related field:

```
if (selectedPayment_form.isValid())
```

A problem arises with the Dojo combo box for **Description**, because the widget content is of type `STRING` and the related field is `selectedPayment.category`, which is of type `INT`. The validation of the Dojo combo box requires that combo box include either integers or strings, such as "1" or "20," that can be converted to integers.

To handle the issue, use an EGL combo widget or ensure that the Dojo combo box includes a valid integer before validation. The previous code demonstrates the second option, and begins by assigning the integer:

```
selectedPayment_category_comboBox.value
    = PaymentLib.getCategoryNum(selectedPayment_category_comboBox.value);
```

The function thereafter checks the validity of the data in the single-record layout and, if the data is valid, does as follows:

- 1) Commits the validated data to the `selectedPayment` record. This “commit” is part of MVC processing and has nothing to do with a database commit.
  - 2) Updates the Dojo combo box in the single-record layout so that the value of that field is again a string.
  - 3) Revises the `allPayments` array element that contains the saved key value. At that point, the array element includes a copy of the data that the user wants in the database.
  - 4) Calls the service to update a single row in the database. The related callback function assigns the `allPayments` array to the data array of the data grid, and that assignment re-renders the grid with the updated data. The grid will be re-rendered with data assigned in the `selectedPayment_form_Submit` function, not with data retrieved from the database.
4. Save the file, but do not close it. If you see errors in your source file, compare your code to the file contents in “Finished code for `PaymentFileMaintenance.egl`” on page 80.

## Test the new code

You can now test the completed application.

1. Click the **Preview** tab. The sample data that you entered earlier is displayed.
2. Select the blank record at the bottom of the sample data. You created this record in a previous lesson. The Payment record grid shows blank fields, with the following exceptions:
  - A key number is displayed.
  - The **Amount** field shows a zero value.
  - The current date is used as a default because the value of the `DATE` variable is null.

Add
Delete
Sample

Clear
Save

Type	Description	Amount due
Food	Groceries	\$450.00
Utilities	ISP	\$19.99
		\$0.00

▼ Payment record

Key:

Category:

Description:

Amount:

Fixed pmt: ☐

Due date:

Payee:

Address 1:

Address 2:

3. Complete the record with data such as the following:
  - For **Category**, enter Automotive.
  - For **Description**, enter Gas.
  - For **Amount**, enter \$80.00.
  - Leave the **Fixed pmt** check box clear.
  - Click the current date in the **Due Date** field and select a date from the displayed calendar.

Add
Delete
Sample

Type	Description	Amount due
Food	Groceries	\$450.00
Utilities	ISP	\$19.99
		\$0.00

Clear
Save

▼ Payment record

Key:
73

Category:
Automotive

Description:
Gas

Amount:
\$80.00

Fixed pmt:
☐

Due date:
July 7, 2011

Payee:

August

S M T W T F S

31 1 2 3 4 5 6

7 8 9 10 11 12 13

14 15 16 17 18 19 20

21 22 23 24 25 26 27

28 29 30 31 1 2 3

4 5 6 7 8 9 10

2010 2011 2012

Address 1:

Address 2:

- For **Payee**, enter Corner Gas Station.
  - For **Address 1**, enter 101 Main Street
  - For **Address 2**, enter Miami, FL.
4. Click **Save**. The new data is stored in the database and is displayed in the data grid.

Add
Delete
Sample

Type	Description	Amount due
Food	Groceries	\$450.00
Utilities	ISP	\$19.99
Automotive	Gas	\$80.00

Clear
Save

▼ Payment record

Key:
73

Category:
Automotive

Description:
Gas

Amount:
\$80.00

Fixed pmt:
☐

Due date:
August 18, 2011

Payee:
Corner Gas Station

Address 1:
101 Main Street

Address 2:
Miami, FL

- 5.
6. Click **Clear**. The single-record layout is reset to initial values.

## Lesson checkpoint

You learned how to complete the following tasks:

- Assign a preset string array as the set of values that are provided by a Dojo combo box.
- Use conversion functions if you need to relate a field of type INT to a Dojo combo box that contains strings.
- Use the Form Manager **isValid** and **commit** functions.

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

---

## Lesson 10: Install Apache Tomcat

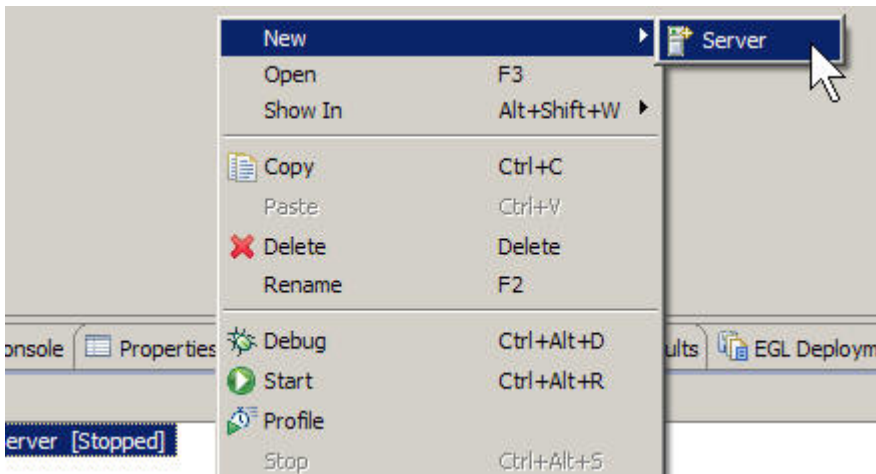
You can use Apache Tomcat to display the web page and to run the EGL-generated service.

### 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:

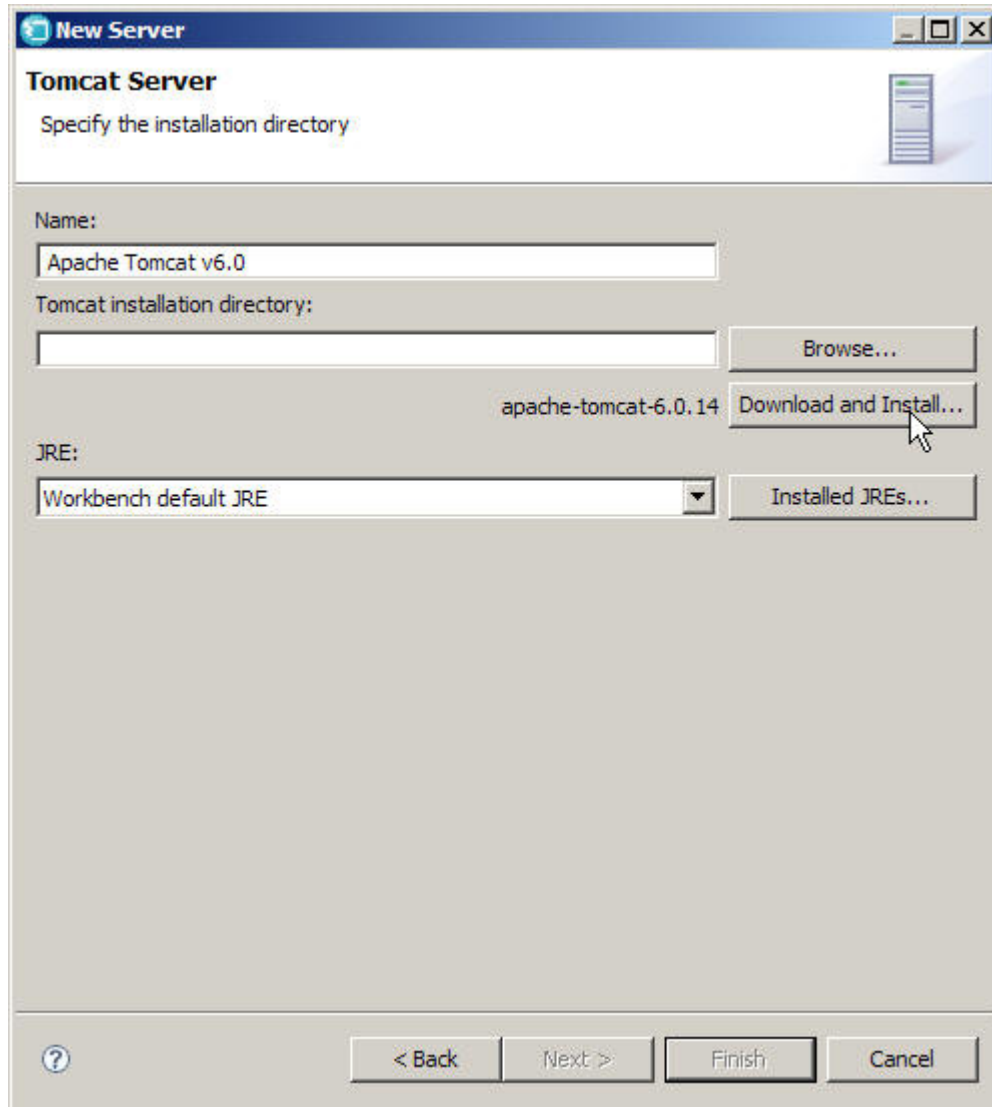
1. Locate the Servers view, which is by default at the lower right of the workbench. EGL created an AJAX Test Server by default. Right-click the empty space and click **New > Server**.



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.

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 11: Deploy and test the payment 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.

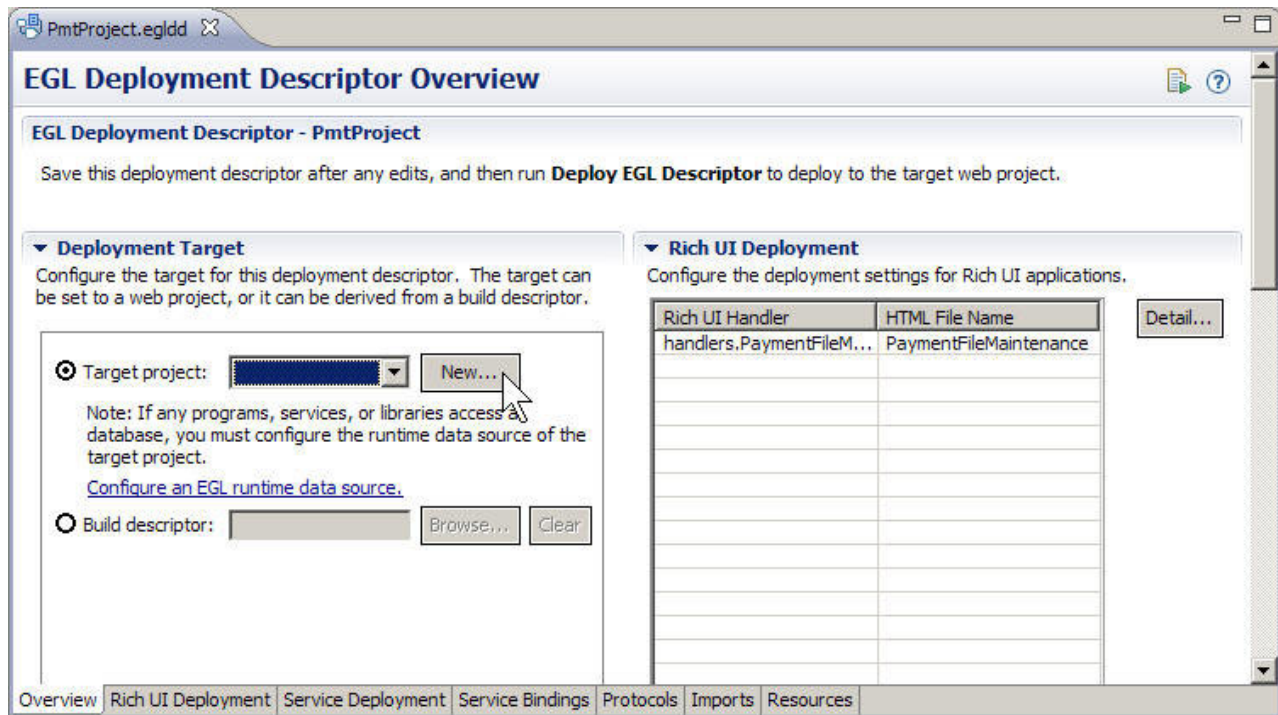
After you deploy the tutorial application internally, you can run it on an application server in the workbench.

### 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 `PaymentClient/EGLSource` folder.

To edit the EGL deployment descriptor:

1. In the **EGLSource** folder, double-click the `PaymentClient.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, 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:  
PaymentWeb

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

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

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.

**Dynamic Web Project**  
Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name:

Project location  
☒ Use default location  
 Location:

Target runtime

Dynamic web module version

Configuration  
   
 You can later add functions to your project by modifying the project facets (right-click a project and select Properties > Project Facets).

EAR membership  
☒ Add project to an EAR  
 EAR project name:

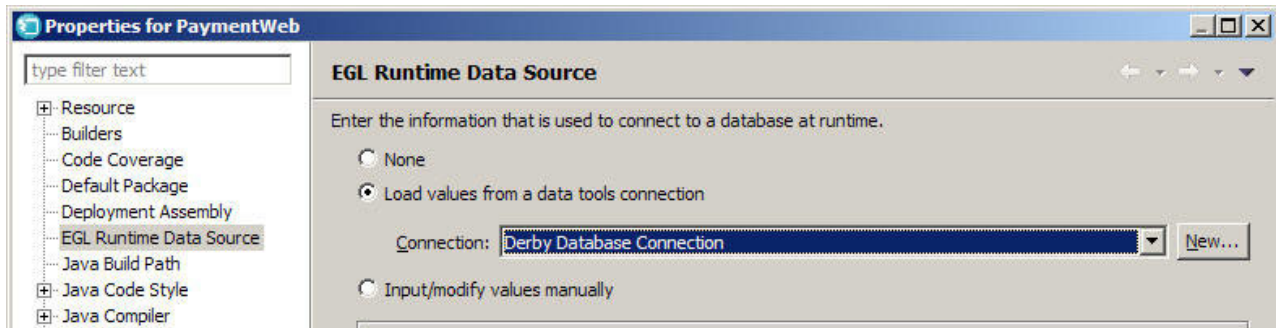
Working sets  
☐ Add project to working sets  
 Working sets:

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

## Set the data source for the new project

Before you can access the database from the new project, you must connect the project to the database.

1. In the Project Explorer view, right-click the **PaymentWeb** project and click **Properties > EGL Runtime Data Source**.
2. Click **Load values from a data tools connection**.
3. Click the down arrow next to the **Connection** field and select **Derby Database Connection**, which is the connection profile that you created in Lesson 2.



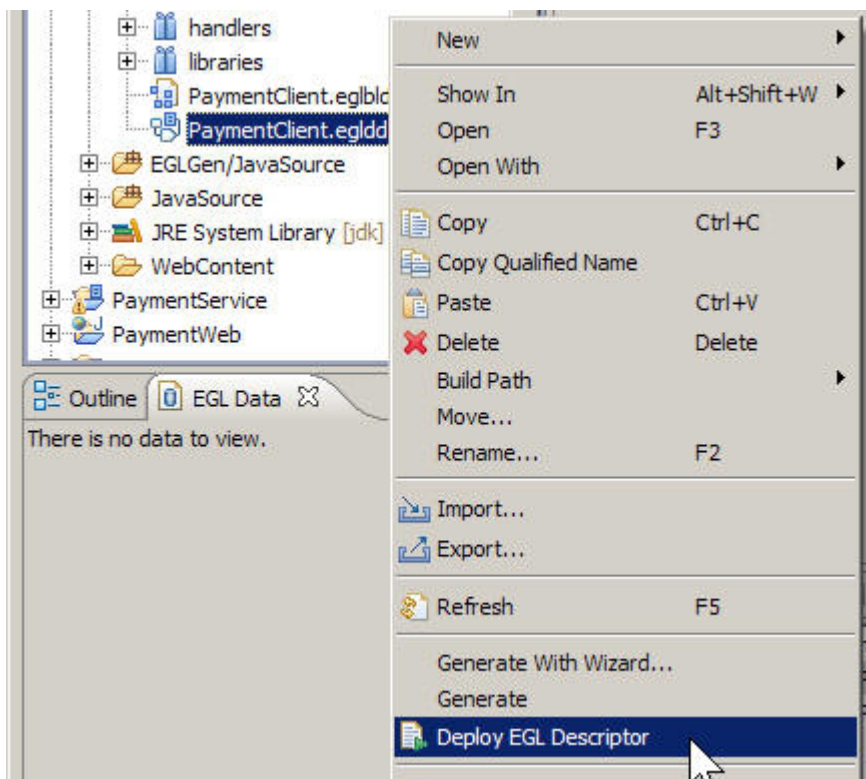
4. Click OK.

## Deploy the Rich UI application


You can now launch the deployment process:

1. In the EGLSource folder, right-click the PaymentClient.egldd file.
2. Click **Deploy EGL Descriptor**.

The deployment process requires no further action on your part. The process

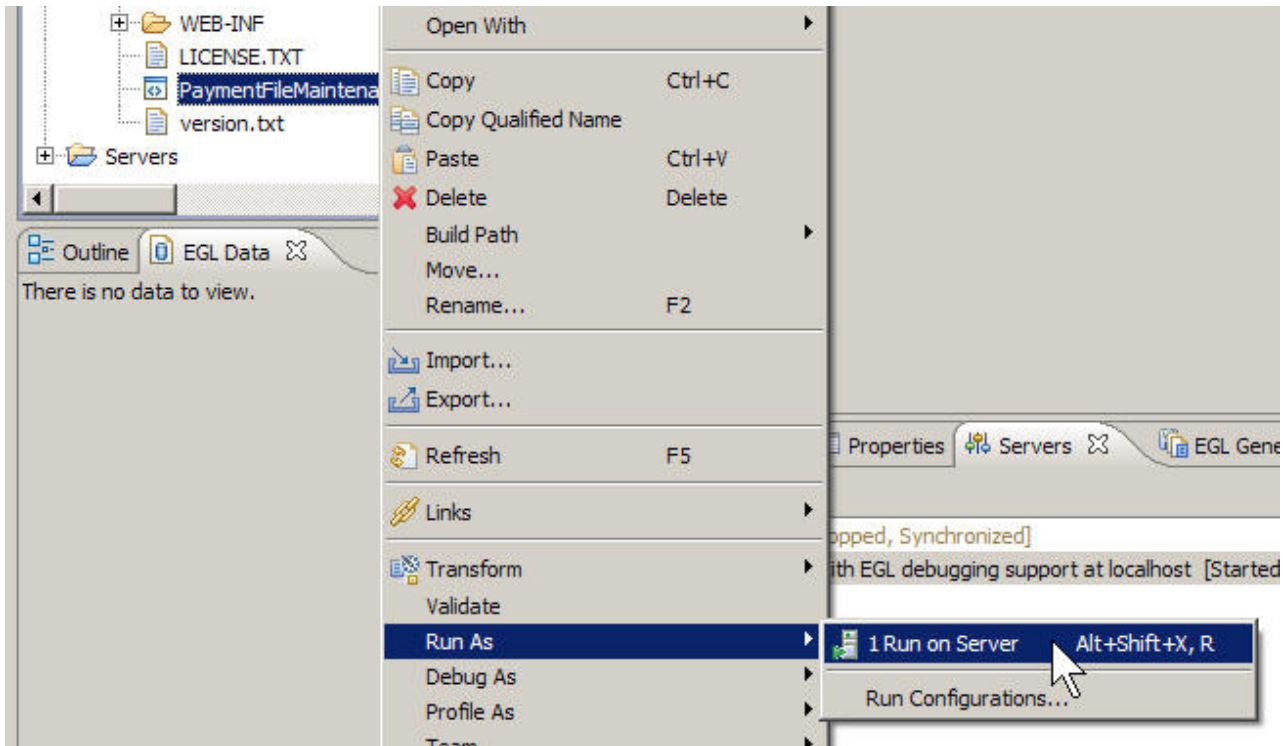


copies many files and might take several minutes.

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".

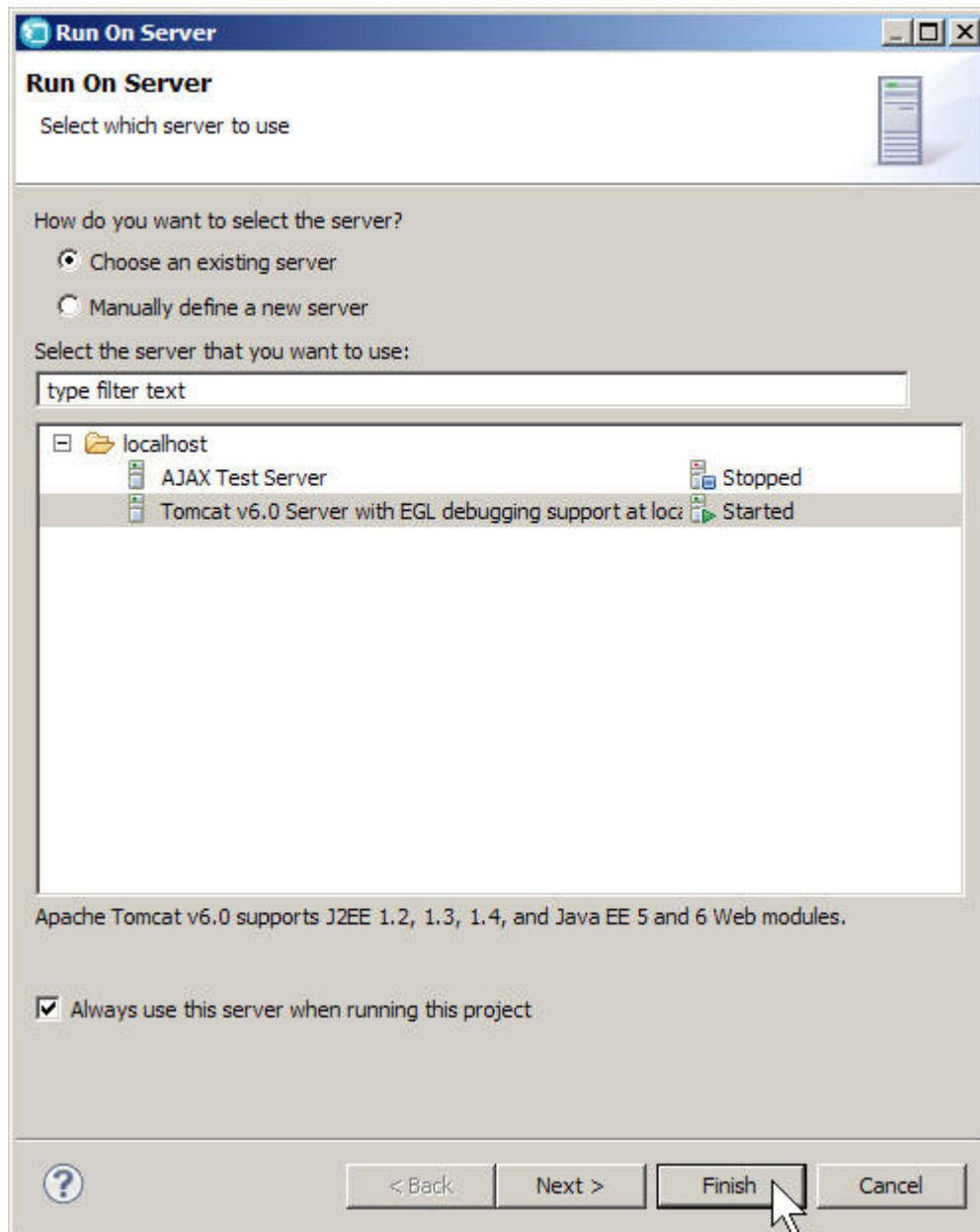
## Run the generated code

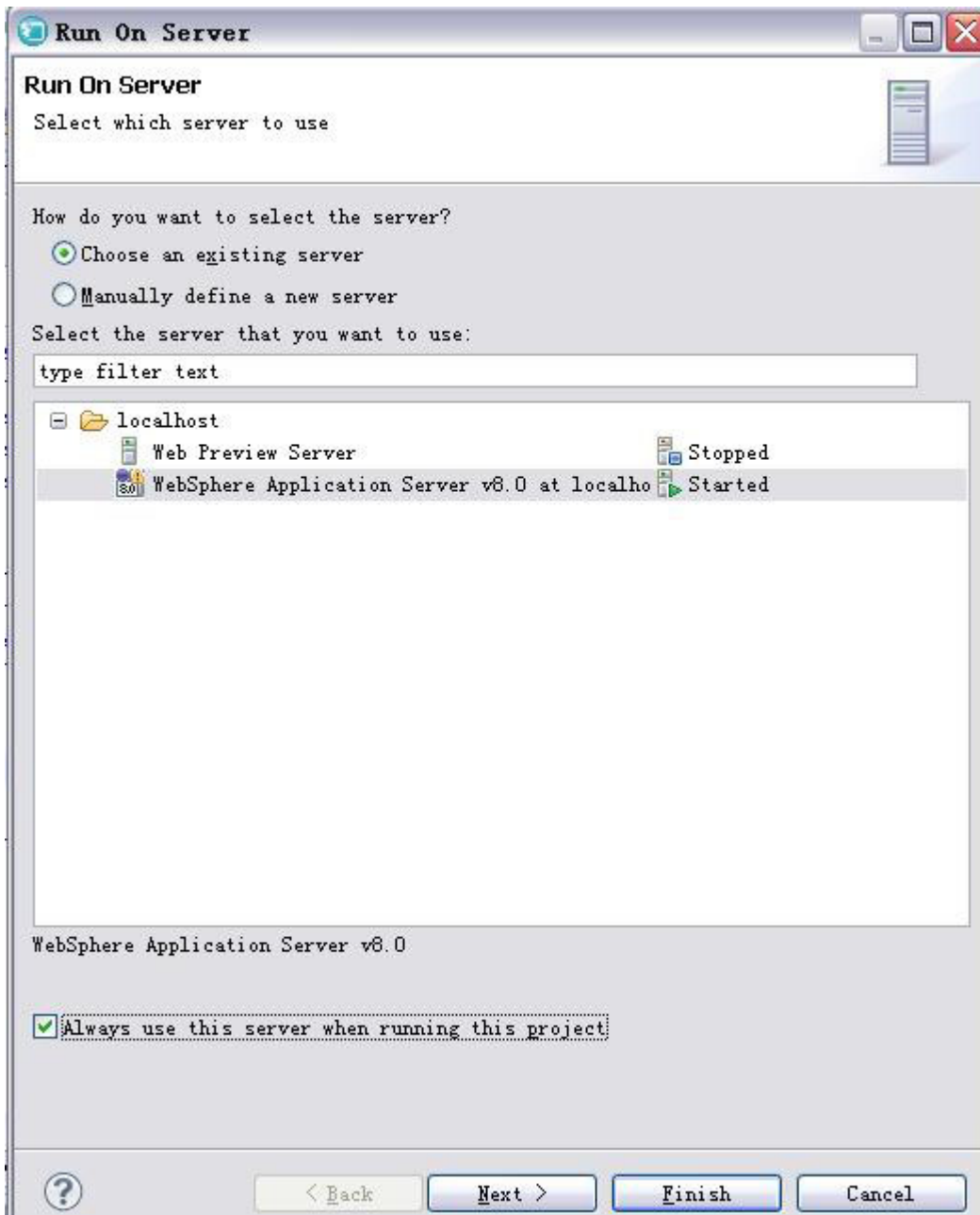
1. To run the internally deployed code, focus your attention on the target project, PaymentWeb. In the PaymentWeb/WebContent folder, find PaymentFileMaintenance-en\_US.html.
2. Right-click the file name and click **Run As > Run on Server**



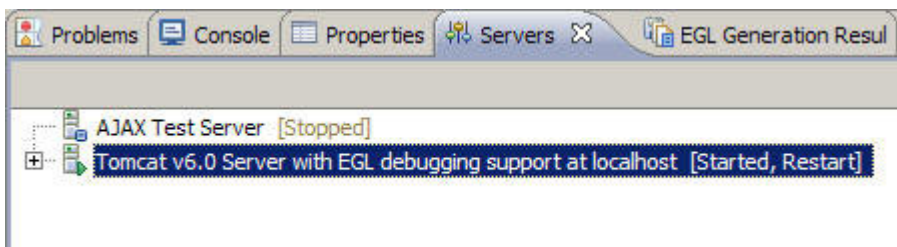
The Run On Server window opens.

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 whether the server is showing a Restart status. If so, restart the server and refresh the page.  
The page opens.



5. Test the application by adding, deleting, and modifying payment records.

### 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 completed the *Access a database with EGL Rich UI* tutorial.

You practiced the following skills:

- Creating and accessing a relational database.
- Creating a library of reusable functions.
- Designing and deploying a Rich UI application and a dedicated service.

---

## Resources

A variety of resources are available.

- Completed tutorial code is here:
  - “Code for PaymentFileMaintenance.egl after lesson 4” on page 68
  - “Finished code for SQLService.egl after lesson 6” on page 73
  - “Finished code for PaymentLib.egl after lesson 7” on page 74
  - “Code for PaymentFileMaintenance.egl after lesson 8” on page 74
  - “Finished code for PaymentFileMaintenance.egl” on page 80
- A complementary example of database processing is here:
  - “End-to-end processing with a UI program and a data grid” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>
- 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>
  - “SQL data” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>
  - “Introduction to EGL generation and deployment” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>
  - “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>
  - “Rich UI DataGrid and DataGridTooltip” at <http://publib.boulder.ibm.com/infocenter/rbdhelp/v8r0m0>
  - “Rich UI GridLayout” 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."

## Code for PaymentFileMaintenance.egl after lesson 4

The following code is the text of the PaymentFileMaintenance.egl file at the end of lesson 4.

```
package handlers;

import com.ibm.egl.rui.mvc.Controller;
import com.ibm.egl.rui.mvc.FormField;
import com.ibm.egl.rui.mvc.FormManager;
import com.ibm.egl.rui.mvc.MVC;
import com.ibm.egl.rui.widgets.DataGrid;
import com.ibm.egl.rui.widgets.DataGridColumn;
import com.ibm.egl.rui.widgets.DataGridLib;
import com.ibm.egl.rui.widgets.Div;
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.TextLabel;
import egl.io.sql.column;
import egl.ui.rui.Event;
import egl.ui.rui.Widget;
import dojo.widgets.DojoButton;
import dojo.widgets.DojoCheckBox;
import dojo.widgets.DojoComboBox;
import dojo.widgets.DojoCurrencyTextBox;
import dojo.widgets.DojoDateTextBox;
import dojo.widgets.DojoLib;
import dojo.widgets.DojoTextField;
import dojo.widgets.DojoTitlePane;
import records.paymentRec;

handler PaymentFileMaintenance type RUIhandler{
    initialUI =[ui], onConstructionFunction = start,
    cssFile = "css/PaymentClient.css", title = "PaymentFileMaintenance"

    ui GridLayout{columns = 2, rows = 2, cellPadding = 4,
        children =
            [detailButtonLayout, editPane, buttonLayout, allPayments_ui ]
    };

    allPayments paymentRec[0];

    allPayments_ui DataGrid{
        layoutData = new GridLayoutData{
            row = 2, column = 1,
            verticalAlignment = GridLayoutLib.VALIGN_TOP},
        columns =[
            new DataGridColumn{name = "category", displayName = "Type", width = 90},
            new DataGridColumn{name = "description", displayName = "Description",
                width = 120},
            new DataGridColumn{name = "amount", displayName = "Amount due", width = 90}
        ],
        data = allPayments as any[],
        selectionMode = DataGridLib.SINGLE_SELECTION};

    buttonLayout GridLayout{
        layoutData = new GridLayoutData{ row = 1, column = 1 },
        cellPadding = 4, rows = 1, columns = 3,
        children = [ sampleButton, deleteButton, addButton ] };

    addButton DojoButton{
        layoutData = new GridLayoutData{ row = 1, column = 1 },
```

```

        text = "Add", onClick ::= addRow };

deleteButton DojoButton{
    layoutData = new GridLayoutData{ row = 1, column = 2 },
    text = "Delete", onClick ::= deleteRow };

sampleButton DojoButton{
    layoutData = new GridLayoutData{ row = 1, column = 3 },
    text = "Sample", onClick ::= sampleData };

selectedPayment paymentRec;

editPane DojoTitlePane{
    layoutData = new GridLayoutData{ row = 2, column = 2,
                                     verticalAlignment = GridLayoutLib.VALIGN_TOP },

    title = "Payment record",
    isOpen=true, duration=1000, width = "350",
    children =
        [ new Div {children = [ selectedPayment_ui ]}]
};

selectedPayment_ui GridLayout {
    rows = 9, columns = 2, cellPadding = 4,
    children = [ selectedPayment_paymentId_nameLabel,
                 selectedPayment_paymentId_field,
                 selectedPayment_category_nameLabel,
                 selectedPayment_category_comboBox,
                 selectedPayment_description_nameLabel,
                 selectedPayment_description_field,
                 selectedPayment_amount_nameLabel,
                 selectedPayment_amount_textBox,
                 selectedPayment_fixedPayment_nameLabel,
                 selectedPayment_fixedPayment_checkBox,
                 selectedPayment_dueDate_nameLabel,
                 selectedPayment_dueDate_textBox,
                 selectedPayment_payeeName_nameLabel,
                 selectedPayment_payeeName_field,
                 selectedPayment_payeeAddress1_nameLabel,
                 selectedPayment_payeeAddress1_field,
                 selectedPayment_payeeAddress2_nameLabel,
                 selectedPayment_payeeAddress2_field ] };

selectedPayment_paymentId_nameLabel TextLabel {
    text="Key:" ,
    layoutData = new GridLayoutData { row = 1, column = 1 } };

selectedPayment_paymentId_field DojoTextField {
    layoutData = new GridLayoutData { row = 1, column = 2},
    readOnly = true };

selectedPayment_paymentId_controller Controller {
    @MVC {model = selectedPayment.paymentId,
          view = selectedPayment_paymentId_field as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_paymentId_formField FormField {
    controller = selectedPayment_paymentId_controller,
    nameLabel = selectedPayment_paymentId_nameLabel};

selectedPayment_category_nameLabel TextLabel {
    text="Category:",
    layoutData = new GridLayoutData { row = 2, column = 1 } };

selectedPayment_category_comboBox DojoComboBox {
    values = [],
    layoutData = new GridLayoutData { row = 2, column = 2 } };

```

```

selectedPayment_category_controller Controller {
    @MVC {model = selectedPayment.category,
        view = selectedPayment_category_comboBox as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_category_formField FormField {
    controller = selectedPayment_category_controller,
    nameLabel = selectedPayment_category_nameLabel};

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

selectedPayment_description_field DojoTextField {
    layoutData = new GridLayoutData { row = 3, column = 2 } };

selectedPayment_description_controller Controller {
    @MVC {model = selectedPayment.description,
        view = selectedPayment_description_field as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_description_formField FormField {
    controller = selectedPayment_description_controller,
    nameLabel = selectedPayment_description_nameLabel};

selectedPayment_amount_nameLabel TextLabel {
    text="Amount:",
    layoutData = new GridLayoutData { row = 4, column = 1 } };

selectedPayment_amount_textBox DojoCurrencyTextBox {
    currency = "USD", value = selectedPayment.amount, width = 166,
    errorMessage="Amount is not valid.",
    layoutData = new GridLayoutData { row = 4, column = 2 } };

selectedPayment_amount_controller Controller {
    @MVC {model = selectedPayment.amount,
        view = selectedPayment_amount_textBox as Widget},

    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_amount_formField FormField {
    controller = selectedPayment_amount_controller,
    nameLabel = selectedPayment_amount_nameLabel};

selectedPayment_fixedPayment_nameLabel TextLabel {
    text="Fixed pmt:" ,
    layoutData = new GridLayoutData { row = 5, column = 1 } };

selectedPayment_fixedPayment_checkBox DojoCheckBox {
    layoutData = new GridLayoutData { row = 5, column = 2 } };

selectedPayment_fixedPayment_controller Controller {
    @MVC {model = selectedPayment.fixedPayment,
        view = selectedPayment_fixedPayment_checkBox as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_fixedPayment_formField FormField {
    controller = selectedPayment_fixedPayment_controller,
    nameLabel = selectedPayment_fixedPayment_nameLabel};

selectedPayment_dueDate_nameLabel TextLabel {
    text="Due date:",
    layoutData = new GridLayoutData { row = 6, column = 1 } };

selectedPayment_dueDate_textBox DojoDateTextBox {
    formatLength = DojoLib.DATEBOX_FORMAT_LONG,
    value = selectedPayment.dueDate,

```

```

        layoutData = new GridLayoutData { row = 6, column = 2 } };

selectedPayment_dueDate_controller Controller {
    @MVC {model = selectedPayment.dueDate,
        view = selectedPayment_dueDate_textBox as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_dueDate_formField FormField {
    controller = selectedPayment_dueDate_controller,
    nameLabel = selectedPayment_dueDate_nameLabel};

selectedPayment_payeeName_nameLabel TextLabel {
    text="Payee:",
    layoutData = new GridLayoutData { row = 7, column = 1 } };

selectedPayment_payeeName_field DojoTextField {
    layoutData = new GridLayoutData { row = 7, column = 2 } };

selectedPayment_payeeName_controller Controller {
    @MVC {model = selectedPayment.payeeName,
        view = selectedPayment_payeeName_field as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_payeeName_formField FormField {
    controller = selectedPayment_payeeName_controller,
    nameLabel = selectedPayment_payeeName_nameLabel};

selectedPayment_payeeAddress1_nameLabel TextLabel {
    text="Address 1:" ,
    layoutData = new GridLayoutData { row = 8, column = 1 } };

selectedPayment_payeeAddress1_field DojoTextField {
    layoutData = new GridLayoutData { row = 8, column = 2 } };

selectedPayment_payeeAddress1_controller Controller {
    @MVC {model = selectedPayment.payeeAddress1,
        view = selectedPayment_payeeAddress1_field as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_payeeAddress1_formField FormField {
    controller = selectedPayment_payeeAddress1_controller,
    nameLabel = selectedPayment_payeeAddress1_nameLabel};

selectedPayment_payeeAddress2_nameLabel TextLabel {
    text="Address 2:" ,
    layoutData = new GridLayoutData { row = 9, column = 1 } };

selectedPayment_payeeAddress2_field DojoTextField {
    layoutData = new GridLayoutData { row = 9, column = 2 } };

selectedPayment_payeeAddress2_controller Controller {
    @MVC {model = selectedPayment.payeeAddress2,
        view = selectedPayment_payeeAddress2_field as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_payeeAddress2_formField FormField {
    controller = selectedPayment_payeeAddress2_controller,
    nameLabel = selectedPayment_payeeAddress2_nameLabel};

selectedPayment_form FormManager {
    entries = [ selectedPayment_paymentId_formField,
        selectedPayment_category_formField,
        selectedPayment_description_formField,
        selectedPayment_amount_formField,
        selectedPayment_fixedPayment_formField,
        selectedPayment_dueDate_formField,
        selectedPayment_payeeName_formField,

```

```

        selectedPayment_payeeAddress1_formField,
        selectedPayment_payeeAddress2_formField ] };

detailButtonLayout GridLayout{
    layoutData = new GridLayoutData{ row = 1, column = 2 },
    cellPadding = 4, rows = 1, columns = 2,
    children = [ saveButton, clearButton ] };

clearButton DojoButton{
    layoutData = new GridLayoutData{ row = 1, column = 1 },
    text = "Clear", onClick ::= clearAllFields };

saveButton DojoButton{
    layoutData = new GridLayoutData{ row = 1, column = 2 },
    text = "Save", onClick ::= selectedPayment_form_Submit };

function start()
    allPayments_ui.data =[
        new paymentRec{category = 1, description = "test01", amount = 100.00},
        new paymentRec{category = 2, description = "test02", amount = 200.00},
        new paymentRec{category = 3, description = "test03", amount = 300.00}];
end

function addRow(event Event in)
end

function deleteRow(event Event in)
end

function sampleData(event Event in)
end

function selectedPayment_form_Submit(event Event in)

    if(selectedPayment_form.isValid())
        selectedPayment_form.commit();
    end
end

function selectedPayment_form_Publish(event Event in)
    selectedPayment_form.publish();
    selectedPayment_form_Validate();
end

function selectedPayment_form_Validate()
    selectedPayment_form.isValid();
end

function handleValidStateChange_selectedPayment(view Widget in, valid boolean in)

    for (n int from selectedPayment_form.entries.getSize() to 1 decrement by 1)
        entry FormField = selectedPayment_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

```

```

        function clearAllFields(event Event in)
        end
    end
end

```

### Related tasks

“Lesson 4: Create the Rich UI handler” on page 17

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

## Finished code for SQLService.egl after lesson 6

The following code is the text of the SQLService.egl file after Lesson 6.

```

package services;

import records.paymentRec;

service SQLService
    function addPayment(newPayment paymentRec inOut)
        add newPayment with #sql{
            insert into PAYMENT
                (CATEGORY, DESCRIPTION, AMOUNT, FIXED_PAYMENT,
                 DUE_DATE, PAYEE_NAME, PAYEE_ADDRESS1, PAYEE_ADDRESS2)
            values
                (:newPayment.category, :newPayment.description, :newPayment.amount,
                 :newPayment.fixedPayment, :newPayment.dueDate, :newPayment.payeeName,
                 :newPayment.payeeAddress1, :newPayment.payeeAddress2)
        };
    end

    function getAllPayments() returns(paymentRec[])
        paymentArray paymentRec[];
        get paymentArray;
        return(paymentArray);
    end

    function editPayment(chgPayment paymentRec inOut)
        replace chgPayment nocursor;
    end

    function deletePayment(delPayment paymentRec inOut)
        try
            delete delPayment nocursor;

            onException(exception SQLException)
                if(SQLLib.sqlData.sqlState != "02000") // sqlState is of type CHAR(5)
                    throw exception;
                end
            end
        end

    function createDefaultTable() returns(paymentRec[])
        try
            execute #sql{
                delete from PAYMENT
            };

            onException(exception SQLException)
                if (SQLLib.sqlData.sqlState != "02000") // sqlState is of type CHAR(5)
                    throw exception;
                end
            end;

            ispDate date = dateTimeLib.dateValueFromGregorian(20110405);
            addPayment(new paymentRec

```

```

        {category = 1, description = "Apartment", amount = 880, fixedPayment = yes});
    addPayment (new paymentRec
        {category = 2, description = "Groceries", amount = 450, fixedPayment = no});
    addPayment(new paymentRec
        {category = 5, description = "ISP", amount = 19.99,
        fixedPayment = yes, dueDate = ispDate});
    return(getAllPayments());
end
end

```

### Related tasks

“Lesson 6: Add code for the service functions” on page 36

In EGL, I/O statements such as **add** and **get** access data that resides in different kinds of persistent data storage, from file systems to queues to databases. The coding is similar for the different cases.

## Finished code for PaymentLib.egl after lesson 7

The following code is the text of the PaymentLib.egl file after lesson 7.

```

package libraries;

library PaymentLib type BasicLibrary{}

    categories string[] =[
        "Rent",           // 1
        "Food",           // 2
        "Entertainment", // 3
        "Automotive",     // 4
        "Utilities",      // 5
        "Clothes",        // 6
        "Other"           // 7
    ];
    function getCategoryDesc(cat int in) returns(string)
        if(cat) // the integer is not 0
            return(categories[cat]);
        else
            return("");
        end
    end

    function getCategoryNum(desc string in) returns(int)
        for(i int from 1 to categories.getSize())
            if(categories[i] == desc)
                return(i);
            end
        end
        return(0); // no match
    end
end

```

### Related tasks

“Lesson 7: Create a library of reusable functions” on page 41

Create a library to format money values and to associate category numbers with descriptions.

## Code for PaymentFileMaintenance.egl after lesson 8

The following code is the text of the PaymentFileMaintenance.egl file at the end of lesson 8.

```

package handlers;

import com.ibm.egl.rui.mvc.Controller;
import com.ibm.egl.rui.mvc.FormField;
import com.ibm.egl.rui.mvc.FormManager;

```

```

import com.ibm.egl.rui.mvc.MVC;
import com.ibm.egl.rui.widgets.DataGrid;
import com.ibm.egl.rui.widgets.DataGridColumn;
import com.ibm.egl.rui.widgets.DataGridLib;
import com.ibm.egl.rui.widgets.Div;
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.TextLabel;
import egl.ui.rui.Event;
import egl.ui.rui.Widget;
import dojo.widgets.DojoButton;
import dojo.widgets.DojoCheckBox;
import dojo.widgets.DojoComboBox;
import dojo.widgets.DojoCurrencyTextBox;
import dojo.widgets.DojoDateTextBox;
import dojo.widgets.DojoLib;
import dojo.widgets.DojoTextField;
import dojo.widgets.DojoTitlePane;
import libraries.PaymentLib;
import records.paymentRec;
import services.SQLService;

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

    dbService SQLService{@dedicatedService};

    ui GridLayout{columns = 2, rows = 2, cellPadding = 4,
        children =
            [detailButtonLayout, editPane, buttonLayout, allPayments_ui ]
    };

    allPayments paymentRec[0];

    allPayments_ui DataGrid{
        layoutData = new GridLayoutData{
            row = 2, column = 1,
            verticalAlignment = GridLayoutLib.VALIGN_TOP},
        selectionListeners ::= cellClicked,
        columns =[
            new DataGridColumn{name = "category", displayName = "Type",
                width = 90, formatters = [ formatCategory ]},
            new DataGridColumn{name = "description", displayName = "Description",
                width = 120},
            new DataGridColumn{name = "amount", displayName = "Amount due",
                width = 90, alignment = DataGridLib.ALIGN_RIGHT}
        ],
        data = allPayments as any[],
        selectionMode = DataGridLib.SINGLE_SELECTION};

    buttonLayout GridLayout{
        layoutData = new GridLayoutData{ row = 1, column = 1 },
        cellPadding = 4, rows = 1, columns = 3,
        children = [ sampleButton, deleteButton, addButton ] };

    addButton DojoButton{
        layoutData = new GridLayoutData{ row = 1, column = 1 },
        text = "Add", onClick ::= addRow };

    deleteButton DojoButton{
        layoutData = new GridLayoutData{ row = 1, column = 2 },
        text = "Delete", onClick ::= deleteRow };

    sampleButton DojoButton{
        layoutData = new GridLayoutData{ row = 1, column = 3 },

```

```

        text = "Sample", onClick ::= sampleData };

selectedPayment paymentRec;

editPane DojoTitlePane{
    layoutData = new GridLayoutData{ row = 2, column = 2,
                                     verticalAlignment = GridLayoutLib.VALIGN_TOP },
    title = "Payment record",
    isOpen=true, duration=1000, width = "350",
    children =
        [ new Div {children = [ selectedPayment_ui ]}]
};

selectedPayment_ui GridLayout {
    rows = 9, columns = 2, cellPadding = 4,
    children = [ selectedPayment_paymentId_nameLabel,
                 selectedPayment_paymentId_field,
                 selectedPayment_category_nameLabel,
                 selectedPayment_category_comboBox,
                 selectedPayment_description_nameLabel,
                 selectedPayment_description_field,
                 selectedPayment_amount_nameLabel,
                 selectedPayment_amount_textBox,
                 selectedPayment_fixedPayment_nameLabel,
                 selectedPayment_fixedPayment_checkBox,
                 selectedPayment_dueDate_nameLabel,
                 selectedPayment_dueDate_textBox,
                 selectedPayment_payeeName_nameLabel,
                 selectedPayment_payeeName_field,
                 selectedPayment_payeeAddress1_nameLabel,
                 selectedPayment_payeeAddress1_field,
                 selectedPayment_payeeAddress2_nameLabel,
                 selectedPayment_payeeAddress2_field ] };

selectedPayment_paymentId_nameLabel TextLabel {
    text="Key:" ,
    layoutData = new GridLayoutData { row = 1, column = 1 } };

selectedPayment_paymentId_field DojoTextField {
    layoutData = new GridLayoutData { row = 1, column = 2},
    readOnly = true };

selectedPayment_paymentId_controller Controller {
    @MVC {model = selectedPayment.paymentId,
         view = selectedPayment_paymentId_field as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_paymentId_formField FormField {
    controller = selectedPayment_paymentId_controller,
    nameLabel = selectedPayment_paymentId_nameLabel};

selectedPayment_category_nameLabel TextLabel {
    text="Category:",
    layoutData = new GridLayoutData { row = 2, column = 1 } };

selectedPayment_category_comboBox DojoComboBox {
    values = [],
    layoutData = new GridLayoutData { row = 2, column = 2 } };

selectedPayment_category_controller Controller {
    @MVC {model = selectedPayment.category,
         view = selectedPayment_category_comboBox as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_category_formField FormField {
    controller = selectedPayment_category_controller,
    nameLabel = selectedPayment_category_nameLabel};

```

```

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

selectedPayment_description_field DojoTextField {
    layoutData = new GridLayoutData { row = 3, column = 2 } };

selectedPayment_description_controller Controller {
    @MVC {model = selectedPayment.description,
        view = selectedPayment_description_field as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_description_formField FormField {
    controller = selectedPayment_description_controller,
    nameLabel = selectedPayment_description_nameLabel};

selectedPayment_amount_nameLabel TextLabel {
    text="Amount:",
    layoutData = new GridLayoutData { row = 4, column = 1 } };

selectedPayment_amount_textBox DojoCurrencyTextBox {
    currency = "USD", value = selectedPayment.amount, width = 166,
    errorMessage="Amount is not valid.",
    layoutData = new GridLayoutData { row = 4, column = 2 } };

selectedPayment_amount_controller Controller {
    @MVC {model = selectedPayment.amount,
        view = selectedPayment_amount_textBox as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_amount_formField FormField {
    controller = selectedPayment_amount_controller,
    nameLabel = selectedPayment_amount_nameLabel};

selectedPayment_fixedPayment_nameLabel TextLabel {
    text="Fixed pmt:" ,
    layoutData = new GridLayoutData { row = 5, column = 1 } };

selectedPayment_fixedPayment_checkBox DojoCheckBox {
    layoutData = new GridLayoutData { row = 5, column = 2 } };

selectedPayment_fixedPayment_controller Controller {
    @MVC {model = selectedPayment.fixedPayment,
        view = selectedPayment_fixedPayment_checkBox as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_fixedPayment_formField FormField {
    controller = selectedPayment_fixedPayment_controller,
    nameLabel = selectedPayment_fixedPayment_nameLabel};

selectedPayment_dueDate_nameLabel TextLabel {
    text="Due date:",
    layoutData = new GridLayoutData { row = 6, column = 1 } };

selectedPayment_dueDate_textBox DojoDateTextBox {
    formatLength = DojoLib.DATEBOX_FORMAT_LONG,
    value = selectedPayment.dueDate,
    layoutData = new GridLayoutData { row = 6, column = 2 } };

selectedPayment_dueDate_controller Controller {
    @MVC {model = selectedPayment.dueDate,
        view = selectedPayment_dueDate_textBox as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_dueDate_formField FormField {

```

```

        controller = selectedPayment_dueDate_controller,
        nameLabel = selectedPayment_dueDate_nameLabel});

selectedPayment_payeeName_nameLabel TextLabel {
    text="Payee:",
    layoutData = new GridLayoutData { row = 7, column = 1 } };

selectedPayment_payeeName_field DojoTextField {
    layoutData = new GridLayoutData { row = 7, column = 2 } };

selectedPayment_payeeName_controller Controller {
    @MVC {model = selectedPayment.payeeName,
        view = selectedPayment_payeeName_field as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_payeeName_formField FormField {
    controller = selectedPayment_payeeName_controller,
    nameLabel = selectedPayment_payeeName_nameLabel};

selectedPayment_payeeAddress1_nameLabel TextLabel {
    text="Address 1:" ,
    layoutData = new GridLayoutData { row = 8, column = 1 } };

selectedPayment_payeeAddress1_field DojoTextField {
    layoutData = new GridLayoutData { row = 8, column = 2 } };

selectedPayment_payeeAddress1_controller Controller {
    @MVC {model = selectedPayment.payeeAddress1,
        view = selectedPayment_payeeAddress1_field as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_payeeAddress1_formField FormField {
    controller = selectedPayment_payeeAddress1_controller,
    nameLabel = selectedPayment_payeeAddress1_nameLabel};

selectedPayment_payeeAddress2_nameLabel TextLabel {
    text="Address 2:" ,
    layoutData = new GridLayoutData { row = 9, column = 1 } };

selectedPayment_payeeAddress2_field DojoTextField {
    layoutData = new GridLayoutData { row = 9, column = 2 } };

selectedPayment_payeeAddress2_controller Controller {
    @MVC {model = selectedPayment.payeeAddress2,
        view = selectedPayment_payeeAddress2_field as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_payeeAddress2_formField FormField {
    controller = selectedPayment_payeeAddress2_controller,
    nameLabel = selectedPayment_payeeAddress2_nameLabel};

selectedPayment_form FormManager {
    entries = [ selectedPayment_paymentId_formField,
        selectedPayment_category_formField,
        selectedPayment_description_formField,
        selectedPayment_amount_formField,
        selectedPayment_fixedPayment_formField,
        selectedPayment_dueDate_formField,
        selectedPayment_payeeName_formField,
        selectedPayment_payeeAddress1_formField,
        selectedPayment_payeeAddress2_formField ] };

detailButtonLayout GridLayout{
    layoutData = new GridLayoutData{ row = 1, column = 2 },
    cellPadding = 4, rows = 1, columns = 2,
    children = [ saveButton, clearButton ] };

```

```

clearButton DojoButton{
    layoutData = new GridLayoutData{ row = 1, column = 1 },
    text = "Clear", onClick ::= clearAllFields };

saveButton DojoButton{
    layoutData = new GridLayoutData{ row = 1, column = 2 },
    text = "Save", onClick ::= selectedPayment_form_Submit };

function start()
    // allPayments_ui.data = [
    //     new paymentRec{category = 1, description = "test01",
    //                     amount = 100.00, payeeName = "Someone"},
    //     new paymentRec{category = 2, description = "test02", amount = 200.00},
    //     new paymentRec{category = 3, description = "test03", amount = 300.00}];
    readFromTable();
end

function cellClicked(myGrid DataGrid in)
    selectedPayment = allPayments_ui.getSelection()[1] as paymentRec;
    selectedPayment_form.publish();
end

function readFromTable()
    call dbService.getAllPayments() returning to updateAll
    onException serviceLib.serviceExceptionHandler;
end

function updateAll(retResult paymentRec[] in)
    allPayments = retResult;
    allPayments_ui.data = allPayments as any[];
end

function addRow(event Event in)
    call dbService.addPayment(new paymentRec) returning to recordAdded
    onException serviceLib.serviceExceptionHandler;
end

function recordAdded(newPayment paymentRec in)
    readFromTable();
end

function deleteRow(event Event in)
    for(i INT from 1 to allPayments.getSize())
        if(allPayments[i].paymentID == selectedPayment.paymentID)
            allPayments.removeElement(i);
            exit for;
        end
    end

    call dbService.deletePayment(selectedPayment) returning to recordRevised
    onException serviceLib.serviceExceptionHandler;
end

function recordRevised(delPayment paymentRec in)
    allPayments_ui.data = allPayments as any[];
end

function sampleData(event Event in)
    call dbService.createDefaultTable() returning to updateAll
    onException serviceLib.serviceExceptionHandler;
end

function selectedPayment_form_Submit(event Event in)
    if(selectedPayment_form.isValid())
        selectedPayment_form.commit();
    end
end

```

```

        end
    end

    function selectedPayment_form_Publish(event Event in)
        selectedPayment_form.publish();
        selectedPayment_form_Validate();
    end

    function selectedPayment_form_Validate()
        selectedPayment_form.isValid();
    end

    function handleValidStateChange_selectedPayment(view Widget in, valid boolean in)

        for (n int from selectedPayment_form.entries.getSize() to 1 decrement by 1)
            entry FormField = selectedPayment_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 clearAllFields(event Event in)
    end

    function formatCategory(class string, value string, rowData any in)
        value = PaymentLib.getCategoryDesc(value as INT);
    end
end

```

### Related tasks

“Lesson 8: Add variables and functions to the Rich UI handler” on page 43  
Add source code that supports the user interface.

## Finished code for PaymentFileMaintenance.egl

The following code is the text of the PaymentFileMaintenance.egl file at the end of lesson 9.

```

package handlers;

import com.ibm.egl.rui.mvc.Controller;
import com.ibm.egl.rui.mvc.FormField;
import com.ibm.egl.rui.mvc.FormManager;
import com.ibm.egl.rui.mvc.MVC;
import com.ibm.egl.rui.widgets.DataGrid;
import com.ibm.egl.rui.widgets.DataGridColumn;
import com.ibm.egl.rui.widgets.DataGridLib;
import com.ibm.egl.rui.widgets.Div;
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.TextLabel;
import egl.ui.rui.Event;
import egl.ui.rui.Widget;
import dojo.widgets.DojoButton;
import dojo.widgets.DojoCheckBox;
import dojo.widgets.DojoComboBox;
import dojo.widgets.DojoCurrencyTextBox;
import dojo.widgets.DojoDateTextBox;

```

```

import dojo.widgets.DojoLib;
import dojo.widgets.DojoTextField;
import dojo.widgets.DojoTitlePane;
import libraries.PaymentLib;
import records.paymentRec;
import services.SQLService;

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

    dbService SQLService{@dedicatedService};

    ui GridLayout{columns = 2, rows = 2, cellPadding = 4,
        children =
            [detailButtonLayout, editPane, buttonLayout, allPayments_ui ]
    };

    allPayments paymentRec[0];

    allPayments_ui DataGrid{
        layoutData = new GridLayoutData{
            row = 2, column = 1,
            verticalAlignment = GridLayoutLib.VALIGN_TOP},
        selectionListeners ::= cellClicked,
        columns =[
            new DataGridColumn{name = "category", displayName = "Type",
                width = 90, formatters = [ formatCategory ]},
            new DataGridColumn{name = "description", displayName = "Description",
                width = 120},
            new DataGridColumn{name = "amount", displayName = "Amount due",
                width = 90, alignment = DataGridLib.ALIGN_RIGHT}
        ],
        data = allPayments as any[],
        selectionMode = DataGridLib.SINGLE_SELECTION};

    buttonLayout GridLayout{
        layoutData = new GridLayoutData{ row = 1, column = 1 },
        cellPadding = 4, rows = 1, columns = 3,
        children = [ sampleButton, deleteButton, addButton ] };

    addButton DojoButton{
        layoutData = new GridLayoutData{ row = 1, column = 1 },
        text = "Add", onClick ::= addRow };

    deleteButton DojoButton{
        layoutData = new GridLayoutData{ row = 1, column = 2 },
        text = "Delete", onClick ::= deleteRow };

    sampleButton DojoButton{
        layoutData = new GridLayoutData{ row = 1, column = 3 },
        text = "Sample", onClick ::= sampleData };

    selectedPayment paymentRec;

    editPane DojoTitlePane{
        layoutData = new GridLayoutData{ row = 2, column = 2,
            verticalAlignment = GridLayoutLib.VALIGN_TOP },
        title = "Payment record",
        isOpen=true, duration=1000, width = "350",
        children =
            [ new Div {children = [ selectedPayment_ui ]}]
    };

    selectedPayment_ui GridLayout {
        rows = 9, columns = 2, cellPadding = 4,
        children = [ selectedPayment_paymentId_nameLabel,

```

```

        selectedPayment_paymentId_field,
        selectedPayment_category_nameLabel,
        selectedPayment_category_comboBox,
        selectedPayment_description_nameLabel,
        selectedPayment_description_field,
        selectedPayment_amount_nameLabel,
        selectedPayment_amount_textBox,
        selectedPayment_fixedPayment_nameLabel,
        selectedPayment_fixedPayment_checkBox,
        selectedPayment_dueDate_nameLabel,
        selectedPayment_dueDate_textBox,
        selectedPayment_payeeName_nameLabel,
        selectedPayment_payeeName_field,
        selectedPayment_payeeAddress1_nameLabel,
        selectedPayment_payeeAddress1_field,
        selectedPayment_payeeAddress2_nameLabel,
        selectedPayment_payeeAddress2_field ] };

selectedPayment_paymentId_nameLabel TextLabel {
    text="Key:" ,
    layoutData = new GridLayoutData { row = 1, column = 1 } };

selectedPayment_paymentId_field DojoTextField {
    layoutData = new GridLayoutData { row = 1, column = 2},
    readOnly = true };

selectedPayment_paymentId_controller Controller {
    @MVC {model = selectedPayment.paymentId,
        view = selectedPayment_paymentId_field as Widget},
        validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_paymentId_formField FormField {
    controller = selectedPayment_paymentId_controller,
    nameLabel = selectedPayment_paymentId_nameLabel};

selectedPayment_category_nameLabel TextLabel {
    text="Category:",
    layoutData = new GridLayoutData { row = 2, column = 1 } };

selectedPayment_category_comboBox DojoComboBox {
    values = PaymentLib.categories,
    layoutData = new GridLayoutData { row = 2, column = 2 } };

selectedPayment_category_controller Controller {
    @MVC {model = selectedPayment.category,
        view = selectedPayment_category_comboBox as Widget},
        validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_category_formField FormField {
    controller = selectedPayment_category_controller,
    nameLabel = selectedPayment_category_nameLabel};

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

selectedPayment_description_field DojoTextField {
    layoutData = new GridLayoutData { row = 3, column = 2 } };

selectedPayment_description_controller Controller {
    @MVC {model = selectedPayment.description,
        view = selectedPayment_description_field as Widget},
        validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_description_formField FormField {
    controller = selectedPayment_description_controller,
    nameLabel = selectedPayment_description_nameLabel};

```

```

selectedPayment_amount_nameLabel TextLabel {
    text="Amount:",
    layoutData = new GridLayoutData { row = 4, column = 1 } };

selectedPayment_amount_textBox DojoCurrencyTextBox {
    currency = "USD", value = selectedPayment.amount, width = 166,
    errorMessage="Amount is not valid.",
    layoutData = new GridLayoutData { row = 4, column = 2 } };

selectedPayment_amount_controller Controller {
    @MVC {model = selectedPayment.amount,
        view = selectedPayment_amount_textBox as Widget},

validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_amount_formField FormField {
    controller = selectedPayment_amount_controller,
    nameLabel = selectedPayment_amount_nameLabel};

selectedPayment_fixedPayment_nameLabel TextLabel {
    text="Fixed pmt:" ,
    layoutData = new GridLayoutData { row = 5, column = 1 } };

selectedPayment_fixedPayment_checkBox DojoCheckBox {
    layoutData = new GridLayoutData { row = 5, column = 2 } };

selectedPayment_fixedPayment_controller Controller {
    @MVC {model = selectedPayment.fixedPayment,
        view = selectedPayment_fixedPayment_checkBox as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_fixedPayment_formField FormField {
    controller = selectedPayment_fixedPayment_controller,
    nameLabel = selectedPayment_fixedPayment_nameLabel};

selectedPayment_dueDate_nameLabel TextLabel {
    text="Due date:",
    layoutData = new GridLayoutData { row = 6, column = 1 } };

selectedPayment_dueDate_textBox DojoDateTextBox {
    formatLength = DojoLib.DATEBOX_FORMAT_LONG,
    value = selectedPayment.dueDate,
    layoutData = new GridLayoutData { row = 6, column = 2 } };

selectedPayment_dueDate_controller Controller {
    @MVC {model = selectedPayment.dueDate,
        view = selectedPayment_dueDate_textBox as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_dueDate_formField FormField {
    controller = selectedPayment_dueDate_controller,
    nameLabel = selectedPayment_dueDate_nameLabel};

selectedPayment_payeeName_nameLabel TextLabel {
    text="Payee:",
    layoutData = new GridLayoutData { row = 7, column = 1 } };

selectedPayment_payeeName_field DojoTextField {
    layoutData = new GridLayoutData { row = 7, column = 2 } };

selectedPayment_payeeName_controller Controller {
    @MVC {model = selectedPayment.payeeName,
        view = selectedPayment_payeeName_field as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_payeeName_formField FormField {

```

```

        controller = selectedPayment_payeeName_controller,
        nameLabel = selectedPayment_payeeName_nameLabel};

selectedPayment_payeeAddress1_nameLabel TextLabel {
    text="Address 1:" ,
    layoutData = new GridLayoutData { row = 8, column = 1 } };

selectedPayment_payeeAddress1_field DojoTextField {
    layoutData = new GridLayoutData { row = 8, column = 2 } };

selectedPayment_payeeAddress1_controller Controller {
    @MVC {model = selectedPayment.payeeAddress1,
        view = selectedPayment_payeeAddress1_field as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_payeeAddress1_formField FormField {
    controller = selectedPayment_payeeAddress1_controller,
    nameLabel = selectedPayment_payeeAddress1_nameLabel};

selectedPayment_payeeAddress2_nameLabel TextLabel {
    text="Address 2:" ,
    layoutData = new GridLayoutData { row = 9, column = 1 } };

selectedPayment_payeeAddress2_field DojoTextField {
    layoutData = new GridLayoutData { row = 9, column = 2 } };

selectedPayment_payeeAddress2_controller Controller {
    @MVC {model = selectedPayment.payeeAddress2,
        view = selectedPayment_payeeAddress2_field as Widget},
    validStateSetter = handleValidStateChange_selectedPayment};

selectedPayment_payeeAddress2_formField FormField {
    controller = selectedPayment_payeeAddress2_controller,
    nameLabel = selectedPayment_payeeAddress2_nameLabel};

selectedPayment_form FormManager {
    entries = [ selectedPayment_paymentId_formField,
        selectedPayment_category_formField,
        selectedPayment_description_formField,
        selectedPayment_amount_formField,
        selectedPayment_fixedPayment_formField,
        selectedPayment_dueDate_formField,
        selectedPayment_payeeName_formField,
        selectedPayment_payeeAddress1_formField,
        selectedPayment_payeeAddress2_formField ] };

detailButtonLayout GridLayout{
    layoutData = new GridLayoutData{ row = 1, column = 2 },
    cellPadding = 4, rows = 1, columns = 2,
    children = [ saveButton, clearButton ] };

clearButton DojoButton{
    layoutData = new GridLayoutData{ row = 1, column = 1 },
    text = "Clear", onClick ::= clearAllFields };

saveButton DojoButton{
    layoutData = new GridLayoutData{ row = 1, column = 2 },
    text = "Save", onClick ::= selectedPayment_form_Submit };

function start()
    // allPayments_ui.data =[
    //     new paymentRec{category = 1, description = "test01",
    //         amount = 100.00, payeeName = "Someone"},
    //     new paymentRec{category = 2, description = "test02", amount = 200.00},
    //     new paymentRec{category = 3, description = "test03", amount = 300.00}];
    readFromTable();
end

```

```

function cellClicked(myGrid DataGrid in)
    selectedPayment = allPayments_ui.getSelection()[1] as paymentRec;
    selectedPayment_form.publish();
    selectedPayment_category_comboBox.value =
        PaymentLib.getCategoryDesc(selectedPayment.category);
end

function readFromTable()
    call dbService.getAllPayments() returning to updateAll
        onException serviceLib.serviceExceptionHandler;
end

function updateAll(retResult paymentRec[] in)
    allPayments = retResult;
    allPayments_ui.data = allPayments as any[];
end

function addRow(event Event in)
    call dbService.addPayment(new paymentRec) returning to recordAdded
        onException serviceLib.serviceExceptionHandler;
end

function recordAdded(newPayment paymentRec in)
    readFromTable();
end

function deleteRow(event Event in)
    for(i INT from 1 to allPayments.getSize())
        if(allPayments[i].paymentID == selectedPayment.paymentID)
            allPayments.removeElement(i);
            exit for;
        end
    end

    call dbService.deletePayment(selectedPayment) returning to recordRevised
        onException serviceLib.serviceExceptionHandler;
end

function recordRevised(delPayment paymentRec in)
    allPayments_ui.data = allPayments as any[];
end

function sampleData(event Event in)
    call dbService.createDefaultTable() returning to updateAll
        onException serviceLib.serviceExceptionHandler;
end

function selectedPayment_form_Submit(event Event in)
    selectedPayment_category_comboBox.value =
        PaymentLib.getCategoryNum(selectedPayment_category_comboBox.value);

    if (selectedPayment_form.isValid())
        selectedPayment_form.commit();
        selectedPayment_category_comboBox.value =
            PaymentLib.getCategoryDesc(selectedPayment_category_comboBox.value);

        // update allPayments with new version of selectedPayment
        for(i INT from 1 to allPayments.getSize())

            if(allPayments[i].paymentID == selectedPayment.paymentID)
                allPayments[i] = selectedPayment;
                exit for;
            end
        end
    end
end

```

```

        call dbService.editPayment(selectedPayment)
        returning to recordRevised
        onException serviceLib.serviceExceptionHandler;
    end
end

function selectedPayment_form_Publish(event Event in)
    selectedPayment_form.publish();
    selectedPayment_form_Validate();
end

function selectedPayment_form_Validate()
    selectedPayment_form.isValid();
end

function handleValidStateChange_selectedPayment(view Widget in, valid boolean in)

    for (n int from selectedPayment_form.entries.getSize() to 1 decrement by 1)
        entry FormField = selectedPayment_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 clearAllFields(event Event in)
    saveID INT = selectedPayment.paymentID; // retain the key
    selectedPayment = new PaymentRec{};
    selectedPayment.paymentID = saveID;
    selectedPayment_form.publish();
end

function formatCategory(class string, value string, rowData any in)
    value = PaymentLib.getCategoryDesc(value as INT);
end
end

```

### Related tasks

“Lesson 9: Complete the code that supports the user interface” on page 52

Next, you will complete the single-row layout, as well as the code that supports the **Clear** and **Save** buttons.

---

## Appendix. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing  
Legal and Intellectual Property Law  
IBM Japan, Ltd.  
1623-14, Shimotsuruma, Yamato-shi  
Kanagawa 242-8502 Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created

programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software  
IBM Corporation  
5 Technology Park Drive  
Westford, MA 01886  
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

#### COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. enter the year or year, year.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

---

## Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.html>.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.







Printed in USA