Release: JavaFX 2.2.40
Last Updated: September 2013
Download as PDF

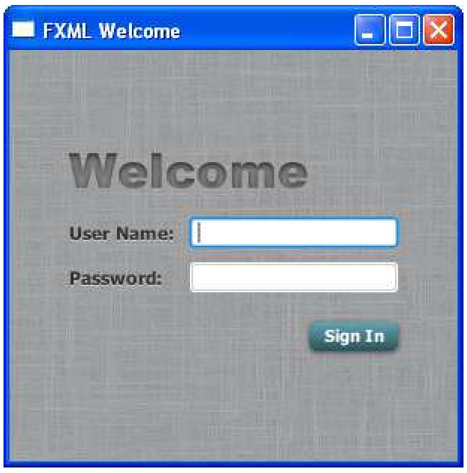
[+] Show/Hide Table of Contents

4 Using FXML to Create a User Interface

This tutorial shows the benefits of using JavaFX FXML, which is an XML-based language that provides the structure for building a user interface separate from the application logic of your code.

If you started this document from the beginning, then you have seen how to create a login application using just JavaFX. Here, you use FXML to create the same login user interface, separating the application design from the application logic, thereby making the code easier to maintain. The login user interface you build in this tutorial is shown in Figure 4-1.

Figure 4-1 Login User Interface



Description of "Figure 4-1 Login User Interface"

This tutorial uses NetBeans IDE. Ensure that the version of NetBeans IDE that you are using supports JavaFX 2.2. See the System Requirements for details.

Set Up the Project

Your first task is to set up a JavaFX FXML project in NetBeans IDE:

1. From the **File** menu, choose **New Project**.
2. In the **JavaFX** application category, choose **JavaFX FXML Application**. Click **Next**.
3. Name the project **FXMLExample** and click **Finish**.

   NetBeans IDE opens an FXML project that includes the code for a basic Hello World application. The application includes three files:

   - **FXMLExample.java.** This file takes care of the standard Java code required for an FXML application.
   - **Sample.fxml.** This is the FXML source file in which you define the user interface.
   - **SampleController.java.** This is the controller file for handling the mouse and keyboard input.

Application Files

View Source Code

- Login.java
- Login.java with CSS
- Login.css
- background.jpg
- FXMLExample.java
- fxml_example.fxml
- FXMLExampleController.java
- ColorfulCircles.java

Download Source Code

- Login.zip
- LoginCSS.zip
- FXMLExample.zip
- ColorfulCircles.zip

Profiles

Gail Chappell
Technical Writer, Oracle

Gail is part of the JavaFX documentation team and enjoys working on cutting-edge, innovative documentation.

Nancy Hildebrandt
Technical Writer, Oracle

Nancy is a technical writer in the JavaFX group. She has a background in content management systems, enterprise server-client software, and XML. She lives on 480 acres in the middle of nowhere with horses, a donkey, dogs, cats, and chickens, and stays connected by satellite.

We Welcome Your Comments

Send us feedback about this document.

If you have questions about JavaFX, please go to the forum.

4. Rename SampleController.java to FXMLExampleController.java so that the name is more meaningful for this application.
   a. In the Projects window, right-click **SampleController.java** and choose **Refactor** then **Rename**.
   b. Enter **FXMLExampleController**, and click **Refactor**.
5. Rename Sample.fxml to fxml_example.fxml.
   a. Right-click **Sample.fxml** and choose **Rename**.
   b. Enter **fxml_example** and click **OK**.

Load the FXML Source File

The first file you edit is the `FXMLExample.java` file. This file includes the code for setting up the application main class and for defining the stage and scene. More specific to FXML, the file uses the `FXMLLoader` class, which is responsible for loading the FXML source file and returning the resulting object graph.

Make the changes shown in bold in Example 4-1.

Example 4-1 FXMLExample.java

```
    @Override
    public void start(Stage stage) throws Exception {
       Parent root = FXMLLoader.load(getClass().getResource("fxml_example.fxml"));

        Scene scene = new Scene(root, 300, 275);

        stage.setTitle("FXML Welcome");
        stage.setScene(scene);
        stage.show();
    }
```

A good practice is to set the height and width of the scene when you create it, in this case 300 by 275; otherwise the scene defaults to the minimum size needed to display its contents.

Modify the Import Statements

Next, edit the `fxml_example.fxml` file. This file specifies the user interface that is displayed when the application starts. The first task is to modify the import statements so your code looks like Example 4-2.

Example 4-2 XML Declaration and Import Statements

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

<?import java.net.*?>
<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.text.*?>
```

As in Java, class names can be fully qualified (including the package name), or they can be imported using the import statement, as shown in Example 4-2. If you prefer, you can use specific import statements that refer to classes.

Create a GridPane Layout

The Hello World application generated by NetBeans uses an `AnchorPane` layout. For the login form, you will use a `GridPane` layout because it enables you to create a flexible grid of rows and columns in which to lay out controls.

Remove the `AnchorPane` layout and its children and replace it with the `GridPane` layout in Example 4-3.

Example 4-3 GridPane Layout

```
<GridPane fx:controller="fxmlexample.FXMLExampleController"
    xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10">
<padding><Insets top="25" right="25" bottom="10" left="25"/></padding>

</GridPane>
```

In this application, the `GridPane` layout is the root element of the FXML document and as such has two attributes. The `fx:controller` attribute is required when you specify controller-based event handlers in your markup. The `xmlns:fx` attribute is always required and specifies the `fx` namespace.

The remainder of the code controls the alignment and spacing of the grid pane. The alignment property changes the default position of the grid from the top left of the scene to the center. The `gap` properties manage the spacing between the rows and columns, while the `padding` property manages the space around the edges of the grid pane.

As the window is resized, the nodes within the grid pane are resized according to their layout constraints. In this example, the grid remains in the center when you grow or shrink the window. The padding properties ensure there is a padding around the grid when you make the window smaller.

Add Text and Password Fields

Looking back at Figure 4-1, you can see that the login form requires the title "Welcome" and text and password fields for gathering information from the user. The code in Example 4-4 is part of the `GridPane` layout and must be placed above the `</GridPane>` statement.

Example 4-4 Text, Label, TextField, and Password Field Controls

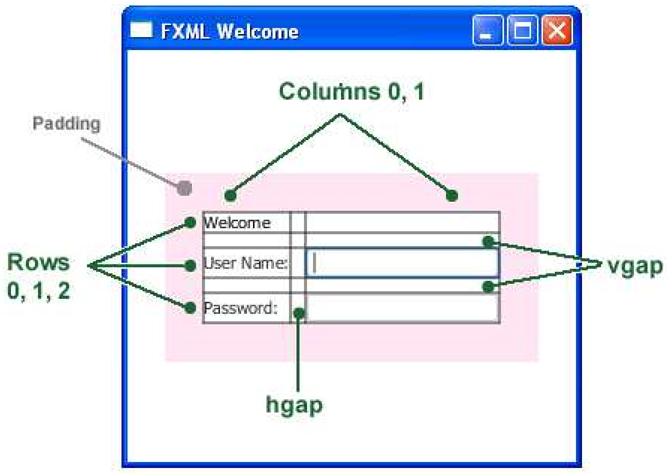
```
    <Text text="Welcome"
        GridPane.columnIndex="0" GridPane.rowIndex="0"
        GridPane.columnSpan="2"/>

    <Label text="User Name:"
        GridPane.columnIndex="0" GridPane.rowIndex="1"/>

    <TextField
        GridPane.columnIndex="1" GridPane.rowIndex="1"/>

    <Label text="Password:"
        GridPane.columnIndex="0" GridPane.rowIndex="2"/>

    <PasswordField fx:id="passwordField"
        GridPane.columnIndex="1" GridPane.rowIndex="2"/>
```

The first line creates a `Text` object and sets its text value to `Welcome`. The `GridPane.columnIndex` and `GridPane.rowIndex` attributes correspond to the placement of the `Text` control in the grid. The numbering for rows and columns in the grid starts at zero, and the location of the `Text` control is set to (0,0), meaning it is in the first column of the first row. The `GridPane.columnSpan` attribute is set to 2, making the Welcome title span two columns in the grid. You will need this extra width later in the tutorial when you add a style sheet to increase the font size of the text to 32 points.

The next lines create a `Label` object with text `User Name` at column 0, row 1 and a `TextField` object to the right of it at column 1, row 1. Another `Label` and `PasswordField` object are created and added to the grid in a similar fashion.

When working with a grid layout, you can display the grid lines, which is useful for debugging purposes. In this case, set the `gridLinesVisible` property to `true` by adding the statement `<gridLinesVisible>true</gridLinesVisible>` right after the `<padding></padding>` statement. Then, when you run the application, you see the lines for the grid columns and rows as well as the gap properties, as shown in Figure 4-2.

Figure 4-2 Login Form with Grid Lines



Description of "Figure 4-2 Login Form with Grid Lines"

Add a Button and Text

The final two controls required for the application are a `Button` control for submitting the data and a `Text` control for displaying a message when the user presses the button. The code is in Example 4-5. Add this code before `</GridPane>`.

Example 4-5 HBox, Button, and Text

```
<HBox spacing="10" alignment="bottom_right"
        GridPane.columnIndex="1" GridPane.rowIndex="4">
        <Button text="Sign In"
        onAction="#handleSubmitButtonAction"/>
</HBox>

<Text fx:id="actiontarget"
       GridPane.columnIndex="1" GridPane.rowIndex="6"/>
```

An `HBox` pane is needed to set an alignment for the button that is different from the default alignment applied to the other controls in the `GridPane` layout. The `alignment` property is set to `bottom_right`, which positions a node at the bottom of the space vertically and at the right edge of the space horizontally. The `HBox` pane is added to the grid in column 1, row 4.

The `HBox` pane has one child, a `Button` with `text` property set to `Sign in` and an `onAction` property set to `handleSubmitButtonAction()`. While FXML is a convenient way to define the structure of an application's user interface, it does not provide a way to implement an application's behavior. You implement the behavior for the `handleSubmitButtonAction()` method in Java code in the next section of this tutorial, Add Code to Handle an Event.

Assigning an `fx:id` value to an element, as shown in the code for the `Text` control, creates a variable in the document's namespace, which you can refer to from elsewhere in the code. While not required, defining a controller field helps clarify how the controller and markup are associated.

Add Code to Handle an Event

Now make the `Text` control display a message when the user presses the button. You do this in the `FXMLExampleController.java` file. Delete the code that NetBeans IDE generated and replace it with the code in Example 4-6.

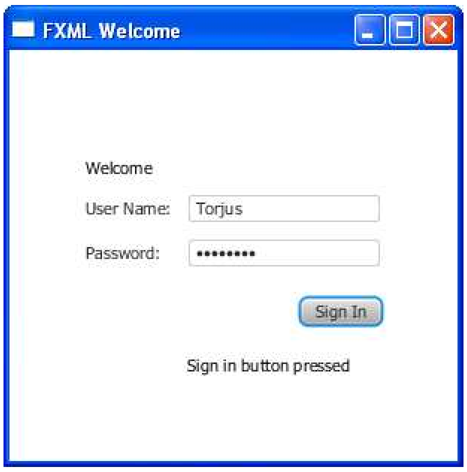
Example 4-6 FXMLExampleController.java

```
package fxmlexample;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.text.Text;

public class FXMLExampleController {
    @FXML private Text actiontarget;

    @FXML protected void handleSubmitButtonAction(ActionEvent event) {
        actiontarget.setText("Sign in button pressed");
    }

}
```

The `@FXML` annotation is used to tag nonpublic controller member fields and handler methods for use by FXML markup. The `handleSubmtButtonAction` method sets the `actiontarget` variable to `Sign in button pressed` when the user presses the button.

You can run the application now to see the complete user interface. Figure 4-3 shows the results when you type text in the two fields and click the Sign in button. If you have any problems, then you can compare your code against the FXMLLogin example.

Figure 4-3 FXML Login Window



Description of "Figure 4-3 FXML Login Window"

Use a Scripting Language to Handle Events

As an alternative to using Java code to create an event handler, you can create the handler with any language that provides a JSR 223-compatible scripting engine. Such languages include JavaScript, Groovy, Jython, and Clojure.

Optionally, you can try using JavaScript now.

1. In the file `fxml_example.fxml`, add the JavaScript declaration after the XML doctype declaration.

   ```
   <?language javascript?>
   ```

2. In the `Button` markup, change the name of the function so the call looks as follows:

   ```
   onAction="handleSubmitButtonAction(event);"
   ```

3. Remove the `fx:controller` attribute from the `GridPane` markup and add the JavaScript function in a `<script>` tag directly under it, as shown in Example 4-7.

   Example 4-7 JavaScript in FXML

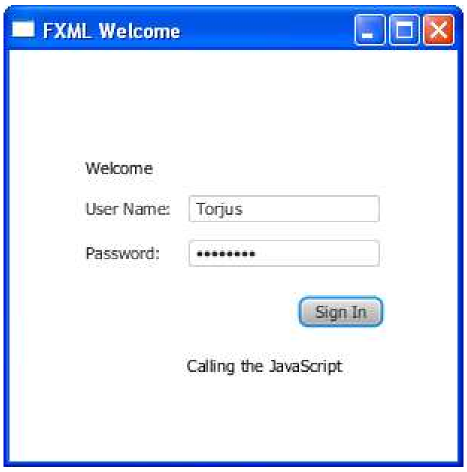
   ```
       <GridPane xmlns:fx="http://javafx.com/fxml"
                 alignment="center" hgap="10" vgap="10">
           <fx:script>
               function handleSubmitButtonAction() {
                   actiontarget.setText("Calling the JavaScript");
               }
           </fx:script>
   ```

   Alternatively, you can put the JavaScript functions in an external file (such as `fxml_example.js`) and include the script like this:

   ```
   <fx:script source="fxml_example.js"/>
   ```

The result is in Figure 4-4.

Figure 4-4 Login Application Using JavaScript



Description of "Figure 4-4 Login Application Using JavaScript"

If you are considering using a scripting language with FXML, then note that an IDE might not support stepping through script code during debugging.

Style the Application with CSS

The final task is to make the login application look attractive by adding a Cascading Style Sheet (CSS).

1. Create a style sheet.
   a. In the Project window, right-click the login folder under Source Packages and choose **New**, then **Other**.
   b. In the New File dialog box, choose **Other**, then **Cascading Style Sheet** and click **Next**.
   c. Enter **Login** and click **Finish**.
   d. Copy the contents of the `Login.css` file attached to this document into your CSS file. For a description of the classes in the CSS file, see Fancy Forms with JavaFX CSS.

2. Download the gray, linen-like image for the background in the `background.jpg` file and add it to the fxmlexample folder.
3. Open the `fxml_example.fxml` file and add a stylesheets element before the end of the markup for the `GridPane` layout as shown in Example 4-8.

   Example 4-8 Style Sheet

   ```
     <stylesheets>
       <URL value="@Login.css" />
     </stylesheets>

   </GridPane>
   ```

   The @ symbol before the name of the style sheet in the URL indicates that the style sheet is in the same directory as the FXML file.
4. To use the root style for the grid pane, add a style class to the markup for the `GridPane` layout as shown in Example 4-9.

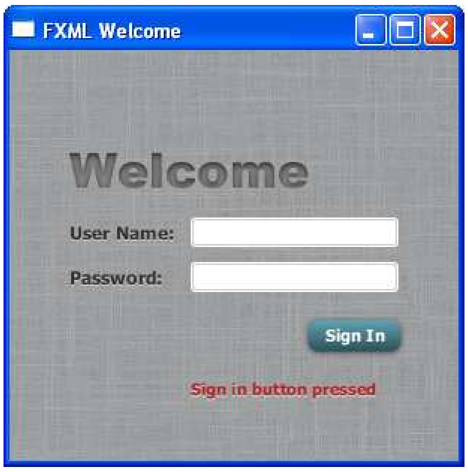
   Example 4-9 Style the GridPane

   ```
   <GridPane fx:controller="fxmlexample.FXMLExampleController"
       xmlns:fx="http://javafx.com/fxml" alignment="center" hgap="10" vgap="10"
       styleClass="root">
   ```
5. Create a `welcome-text` ID for the Welcome `Text` object so it uses the style `#welcome-text` defined in the CSS file, as shown in Example 4-10.

   Example 4-10 Text ID

   ```
   <Text id="welcome-text" text="Welcome"
           GridPane.columnIndex="0" GridPane.rowIndex="0"
           GridPane.columnSpan="2"/>
   ```
6. Run the application. Figure 4-5 shows the stylized application.

   Figure 4-5 Stylized Login Application

   Description of "Figure 4-5 Stylized Login Application"

   For information about how to run your application outside NetBeans IDE, see Deploying Your First JavaFX Application.

Where to Go from Here

Now that you are familiar with FXML, look at Introduction to FXML, which provides more information on the elements that make up the FXML language. The document is included in the javafx.fxml package in the API documentation at `http://docs.oracle.com/javafx/2/api/javafx/fxml/doc-files/introduction_to_fxml.html`.

You can also try out the JavaFX Scene Builder tool by opening the `fxml_example.fxml` file in Scene Builder and making modifications. This tool provides a visual layout environment for designing the UI for JavaFX applications and automatically generates the FXML code for the layout. Note that the FXML file might be reformatted when saved. See Getting Started with JavaFX Scene Builder for more information on this tool. The Skinning with CSS and CSS Analyzer section of the JavaFX Scene Builder User Guide also give you information on how you can skin your FXML layout.

Previous Page   Next Page