

JavaFX
Mastering FXML
Release 2.2
E20478-09

January 2014

JavaFX/Mastering FXML, Release 2.2

E20478-09

Copyright © 2011, 2014 Oracle and/or its affiliates. All rights reserved.

Primary Author: Irina Fedortsova

Contributing Author:

Contributor: Greg Brown

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

1 Why Use FXML

Introduction to FXML	1-1
Simple Example of FXML	1-2
Benefits of FXML	1-2
FXML and Scene Builder	1-3

2 FXML—What’s New in JavaFX 2.1

FXML Enhancements for JavaFX 2.1	2-1
FXML Loader Incompatibilities with Previous JavaFX Releases	2-2
Some JavaFX 2.0 FXML Escape Sequences Are Deprecated in JavaFX 2.1	2-2
Backslash Is Now an Escape Character	2-2

3 FXML—What’s New in JavaFX 2.2

4 Creating an Address Book with FXML

Set Up the Project	4-1
Create the Basic User Interface	4-2
Add Columns to the Table	4-4
Define the Data Model	4-4
Associate Data with the Table Columns	4-5
Set Sort Order on Startup	4-7
Define Column Widths	4-8
Set Alignment in Table Cells	4-9
Add Rows to the Table	4-11
Where to Go from Here	4-12

5 Creating a Custom Control with FXML

Set Up a Project	5-1
Create the Basic User Interface	5-2
Create a Controller	5-2
Load the FXML Source File and Define Stage and Scene	5-3

6 Deployment of FXML Applications

Part I

About This Tutorial

This document consists of the following pages:

- [Why Use FXML](#)
A basic description of FXML and the benefits of using it to create user interfaces.
- [FXML—What’s New in JavaFX 2.1](#)
A list of FXML enhancements in JavaFX 2.1 and incompatibilities with previous releases.
- [FXML—What’s New in JavaFX 2.2](#)
A list of FXML enhancements in JavaFX 2.2.
- [Creating an Address Book with FXML](#)
A tutorial that shows how to populate a table with data, sort the data at application startup, align the data in the table cells, and add rows to the table.
- [Creating a Custom Control with FXML](#)
A tutorial that shows how to create a custom control using APIs introduced in JavaFX 2.2.
- [Deployment of FXML Applications](#)
A clarification about why some FXML applications need digital signatures. An alternative to signing the application is also presented.

You can also get information on FXML from the following resources:

- [Creating a User Interface with FXML](#)
A beginning tutorial that shows how to create a login application using FXML.
- [Introduction to FXML](#)
A reference document that provides information on the elements that make up the FXML language. The document is included in the `javafx.fxml` package in the API documentation.
- [JavaFX 2 Forum](#)
A place where you can post questions about FXML.

Why Use FXML

This tutorial provides a basic description of FXML and the benefits of using it to create user interfaces.

FXML is an XML-based language that provides the structure for building a user interface separate from the application logic of your code. This separation of the presentation and application logic is attractive to web developers because they can assemble a user interface that leverages Java components without mastering the code for fetching and filling in the data.

The following sections provide more information about FXML, and when you would choose FXML over other methods of creating a user interface:

- [Introduction to FXML](#)
- [Simple Example of FXML](#)
- [Benefits of FXML](#)
- [FXML and Scene Builder](#)

Introduction to FXML

FXML does not have a schema, but it does have a basic predefined structure. What you can express in FXML, and how it applies to constructing a scene graph, depends on the API of the objects you are constructing. Because FXML maps directly to Java, you can use the API documentation to understand what elements and attributes are allowed. In general, most JavaFX classes can be used as elements, and most Bean properties can be used as attributes.

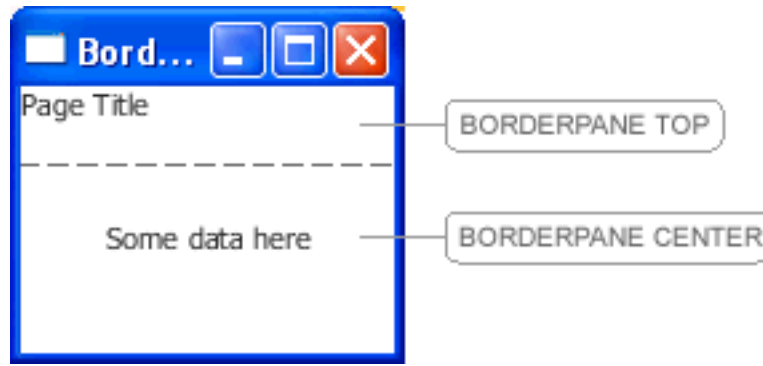
From a Model View Controller (MVC) perspective, the FXML file that contains the description of the user interface is the view. The controller is a Java class, optionally implementing the `Initializable` class, which is declared as the controller for the FXML file. The model consists of domain objects, defined on the Java side, that you connect to the view through the controller. An example of this structure is in the tutorial [Creating an Address Book with FXML](#).

While you can use FXML to create any user interface, FXML is particularly useful for user interfaces that have large, complex scene graphs, forms, data entry, or complex animation. FXML is also well-suited for defining static layouts such as forms, controls, and tables. In addition, you can use FXML to construct dynamic layouts by including scripts.

Simple Example of FXML

The easiest way to show the advantages of FXML is with an example. Take a look at [Figure 1-1](#), which shows a user interface that includes a border pane layout that has a top and center region, each of which contains a label.

Figure 1-1 *Border Pane Simple Example*



First, look at how the user interface is constructed and built directly in the source code, as shown in [Example 1-1](#).

Example 1-1 *Java Code for a User Interface*

```
BorderPane border = new BorderPane();
Label toppanetext = new Label("Page Title");
border.setTop(toppanetext);
Label centerpanetext = new Label("Some data here");
border.setCenter(centerpanetext);
```

Next, look at [Example 1-2](#), which shows the same user interface, but in FXML markup. You can see the hierarchical structure of the user interface, which in turn makes it easier to add components and build upon the user interface.

Example 1-2 *FXML Markup for a User Interface*

```
<BorderPane>
  <top>
    <Label text="Page Title"/>
  </top>
  <center>
    <Label text="Some data here"/>
  </center>
</BorderPane>
```

Benefits of FXML

In addition to providing web developers a familiar approach to designing user interfaces, FXML offers these benefits:

- Because the scene graph is more transparent in FXML, it is easy for a development team to create and maintain a testable user interface.
- FXML is not a compiled language; you do not need to recompile the code to see the changes.

- The content of an FXML file can be localized as the file is read. For example, if an FXML file is loaded using the en_US locale, then it produces the string "First Name" for a label based on the following resource string:

```
<Label text="%firstName" />
```

If the locale is changed to fr_FR and the FXML file is reloaded, then the label shows "Prénom."

The same is not true for Java code, because you must manually update the content of every element of your user interface by obtaining a reference to it and calling the appropriate setter (such as `setText()`).

- You can use FXML with any Java Virtual Machine (JVM) language, such as Java, Scala, or Clojure.
- FXML is not limited to the view portion of the MVC interface. You can construct services or tasks or domain objects, and you can use JavaScript or other scripting languages in FXML. For an example of using JavaScript, see [Use a Scripting Language to Handle Events in the FXML tutorial of the Getting Started guide](#).

FXML and Scene Builder

Just as some developers prefer to work directly in the XML code, other developers prefer to use a tool to author their XML. The same is true with FXML.

If you prefer to use a tool, or if you want to create a quick prototype to get feedback, then you might prefer to use JavaFX Scene Builder. Scene Builder is a design tool that generates the FXML source code as you define the user interface for your application. Scene Builder can help you to quickly create a prototype for an interactive application that connects components to the application logic. For more information, see [Getting Started with JavaFX Scene Builder](#).

Because Scene Builder uses XML as a serialization format, the produced FXML code is very clear and you can further edit FXML files, generated by Scene Builder, in any text or XML editor.

NetBeans IDE 7.2 enables you to open FXML files in JavaFX Scene Builder, provided that the latter is installed on your computer. This tighter integration of NetBeans and Scene Builder gives an additional advantage when developing FXML applications.

FXML—What's New in JavaFX 2.1

This page contains the following sections that describe the FXML enhancements in JavaFX 2.1 and incompatibilities with previous releases:

- [FXML Enhancements for JavaFX 2.1](#)
- [FXML Loader Incompatibilities with Previous JavaFX Releases](#)

FXML Enhancements for JavaFX 2.1

The following FXML enhancements have been added in JavaFX 2.1:

- Support for using a leading backslash as an escape character (RT-18680)
JavaFX 2.0 used consecutive operator characters such as \$\$ as escape sequences. JavaFX 2.1 adds support for escape sequences using the backslash character, such as \\$. These escape sequences are more similar to Unified Expression Language (UEL), making them more familiar to developers. The JavaFX 2.0 escape sequences are deprecated as of JavaFX 2.1. See [Some JavaFX 2.0 FXML Escape Sequences Are Deprecated in JavaFX 2.1](#) and [Backslash Is Now an Escape Character](#).
- An implicit variable for the controller to document the namespace
This new feature facilitates bidirectional binding between the controller and the UI. Bidirectional binding was dropped from JavaFX 2.1, but this feature was retained.
- Convenience constructors to the `FXMLLoader` class (RT-16815)
Several new convenience constructors have been added to the `FXMLLoader` class. These constructors mirror the `static load()` methods defined in JavaFX 2.0, but make it easier to access the document's controller from the calling code.
- Customizable controller instantiation (RT-16724, RT-17268)

In JavaFX 2.0, the calling code did not have any control over controller creation. This prevented an application from using a dependency injection system such as Google Guice or the Spring Framework to manage controller initialization. JavaFX 2.1 adds a `Callback` interface to facilitate delegation of controller construction:

```
public interface Callback {  
    public Object getController(Class<?> type);  
}
```

When a controller factory is provided to the `FXMLLoader` object, the loader will delegate controller construction to the factory. An implementation might return a null value to indicate that it does not or cannot create a controller of the given type; in this case, the default controller construction mechanism will be employed

by the loader. Implementations might also "recycle" controllers such that controller instances can be shared by multiple FXML documents. However, developers must be aware of the implications of doing this: primarily, that controller field injection should not be used in this case because it will result in the controller fields containing values from only the most recently loaded document.

- Easier style sheets to work with (RT-18299, RT-15524)

In JavaFX 2.0, applying style sheets in FXML was not very convenient. In JavaFX 2.1, it is much simpler. Style sheets can be specified as an attribute of a root <Scene> element as follows:

```
<Scene stylesheets="/com/foo/stylesheets1.css, /com/foo/stylesheets2.css">
</Scene>
```

Style classes on individual nodes can now be applied as follows:

```
<Label styleClass="heading, firstPage" text="First Page Heading"/>
```

- Caller-specified no-arg controller method as an event handler (RT-18229)

In JavaFX 2.0, controller-based event handlers must adhere to the method signature defined by an event handler. They must accept a single argument of a type that extends the `Event` class and return `void`. In JavaFX 2.1, the argument restriction has been lifted, and it is now possible to write a controller event handler that takes no arguments.

FXML Loader Incompatibilities with Previous JavaFX Releases

The following sections contain compatibility issues that users might encounter if they load a JavaFX 2.0 FXML file with a JavaFX 2.1 FXML loader:

- [Some JavaFX 2.0 FXML Escape Sequences Are Deprecated in JavaFX 2.1](#)
- [Backslash Is Now an Escape Character](#)

Some JavaFX 2.0 FXML Escape Sequences Are Deprecated in JavaFX 2.1

Table 2–1 shows the double-character escape sequences that were used in FXML in JavaFX 2.0, but are **deprecated** in JavaFX 2.1. Instead, use a backslash as the escape character.

Table 2–1 *Deprecated and Current Escape Sequences*

JavaFX 2.0 Escape Sequence	JavaFX 2.1 Escape Sequence
\$\$	\\$
%%	\%
@@	\@

If Scene Builder encounters any of these deprecated escape sequences, then the console displays a warning, but loads the FXML anyway. The next time the file is saved, Scene Builder automatically replaces the deprecated escape characters with the new syntax.

Backslash Is Now an Escape Character

In JavaFX 2.1, the backslash `\` is an escape character in FXML. As a result, JavaFX 2.0 applications with FXML files that contain FXML string attributes starting with a backslash might prevent the FXML from loading, or it might cause the FXML loader to misinterpret the string.

Solution: For any FXML backslash text in a JavaFX 2.0 application, add an additional backslash to escape the character.

Example:

Remove this line of code:

```
<Button text="\"/>
```

Replace it with this line of code:

```
<Button text="\""/>
```

FXML—What's New in JavaFX 2.2

This page contains a list of FXML enhancements added in JavaFX 2.2.

- `<fx:constant>` tag

The `<fx:constant>` tag has been added to FXML to facilitate lookup of class constants. For example, the `NEGATIVE_INFINITY` constant defined by the `java.lang.Double` class can now be referenced as follows:

```
<Double fx:constant="NEGATIVE_INFINITY"/>
```

- Improved access to sub-controllers in FXML

In JavaFX 2.1 and earlier, it was not easy to access subcontrollers from a root controller class. This made it difficult to use a controller to open and populate a dialog window whose contents were defined in an include statement, for example.

JavaFX 2.2 maps nested controller instances directly to member fields in the including document's controller, making it much easier to interact with nested controllers. Consider the following FXML document and controller:

```
<VBox fx:controller="com.foo.MainController">
...
  <fx:include fx:id="dialog" source="dialog.fxml"/>
...
</VBox>
```

```
public class MainController extends Controller {
    @FXML private Window dialog;
    @FXML private DialogController dialogController;
    ...
}
```

When the controller's `initialize()` method is called, the `dialog` field will contain the root element loaded from the `dialog.fxml` file, and the `dialogController` object will contain the include statement's controller. The main controller can then invoke methods on the include statement's controller, to populate and show the dialog, for example.

- Support for controller initialization via reflection

In JavaFX 2.1 and earlier, controller classes were required to implement the `Initializable` interface to be notified when the contents of the associated FXML document had been completely loaded. In JavaFX 2.2, this is no longer necessary. An instance of the `FXMLLoader` class simply looks for the `initialize()` method on the controller and calls it, if available. Note that, similar to other FXML callback

methods such as event handlers, this method must be annotated with the `@FXML` annotation if it is not public.

It is recommended that developers use this approach for new development. The `Initializable` interface has not been deprecated, but might be in a future release.

- Simplified creation of FXML-based custom controls

In previous releases, it was fairly cumbersome to create custom controls whose internal structure was defined in FXML. JavaFX 2.2 includes some subtle but powerful enhancements that significantly simplify this process. The new `setRoot()` and `setController()` methods enable the calling code to inject document root and controller values, respectively, into the document namespace, rather than delegate creation of these objects to `FXMLLoader`. This enables a developer to create reusable controls that are internally implemented using markup, but (from an API perspective) appear identical to controls implemented programmatically.

For example, the following code markup defines the structure of a simple custom control containing a `TextField` and a `Button` instance. The root container is defined as an instance of the `javafx.scene.layout.VBox` class:

```
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<fx:root type="javafx.scene.layout.VBox" xmlns:fx="http://javafx.com/fxml">
    <TextField fx:id="textField"/>
    <Button text="Click Me" onAction="#doSomething"/>
</fx:root>
```

The `<fx:root>` tag, which was added for JavaFX 2.2, specifies that the element's value will be obtained by calling the `getRoot()` method of the `FXMLLoader` class. Prior to calling the `load()` method, the calling code must specify this value by calling the `setRoot()` method. The calling code can also provide a value for the document's controller by calling the `setController()` method.

For more information, see [Creating a Custom Control with FXML](#).

Creating an Address Book with FXML

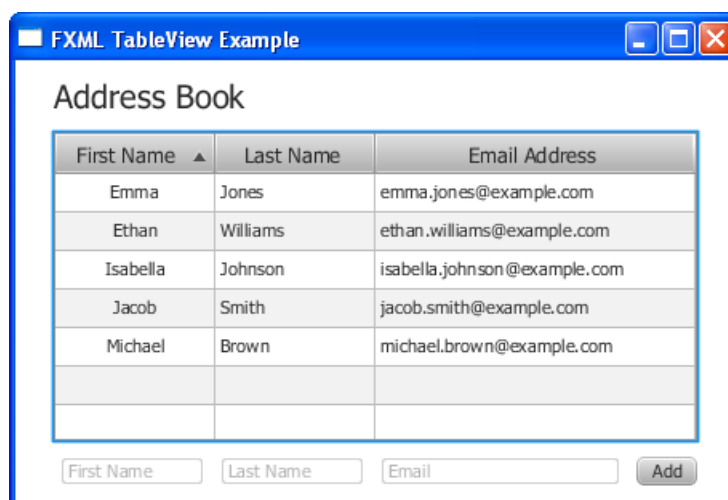
In this tutorial, you create an Address Book application that includes a table of names and email addresses, as shown in [Figure 4-1](#). The tutorial shows how to populate a table with data, sort the data at application startup, align the data in the table cells, and add rows to the table.

Some amount of knowledge of FXML and application development is assumed for this tutorial. Before you start, you should have completed the FXML tutorial in the Getting Started series, because it teaches the basics of FXML development. Specifically, for the Address Book tutorial, you should know:

- The basic structure of an FXML project (.java, .FXML, and controller files)
- How to create and run a JavaFX FXML project in NetBeans IDE
- The basics of layout and user interface components

Before you begin this tutorial, ensure that the version of NetBeans IDE that you are using supports your version of JavaFX 2. See the System Requirements for details.

Figure 4-1 Address Book Application



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** category, choose **JavaFX FXML Application**. Click **Next**.
3. Name the project **FXMLTableView** and click **Finish**.

NetBeans IDE opens an FXML project that includes the code for a basic Hello World application. The application includes three files: `FXMLTableView.java`, `Sample.fxml`, and `SampleController.java`.
4. Rename `SampleController.java` to `FXMLTableViewController.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 **FXMLTableViewController**, and then click **Refactor**.
5. Rename `Sample.fxml` to `fxml_tableview.fxml`.
 - a. Right-click **Sample.fxml** and choose **Rename**.
 - b. Enter **fxml_tableview** and click **OK**.
6. Open `FXMLTableView.java` and edit the `FXMLTableView` class to look like [Example 4-1](#).

Example 4-1 FXMLTableView.java

```
public class FXMLTableView extends Application {  
  
    @Override  
    public void start(Stage primaryStage) throws Exception {  
        primaryStage.setTitle("FXML TableView Example");  
        Pane myPane = (Pane)FXMLLoader.load(getClass().getResource  
("fxml_tableview.fxml"));  
        Scene myScene = new Scene(myPane);  
        primaryStage.setScene(myScene);  
        primaryStage.show();  
    }  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

Note that the Java file does not contain the code for the scene. In the next section of the tutorial, [Create the Basic User Interface](#), you will add the code for the scene in the FXML file.

7. Press **Ctrl** (or **Cmd**) + **Shift** + **I** to correct the import statements.

Create the Basic User Interface

Define the user interface by creating a `GridPane` layout container as the root node of the scene. Then, add a `Label` and a `TableView` component as child nodes of the `GridPane` layout container.

1. Open the `fxml_tableview.fxml` file.
2. Delete the `<AnchorPane>` markup that NetBeans IDE automatically generated.
3. Add a `GridPane` layout container as the root node of the scene as shown in [Example 4-2](#).

Example 4-2 GridPane

```
<GridPane alignment="CENTER" hgap="10.0" vgap="10.0"
  xmlns:fx="http://javafx.com/fxml"
  fx:controller="fxmtableview.FXMLTableViewController">
  <padding>
    <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
  </padding>
</GridPane>
```

You can ignore the error "File not found in the specified address: http://javafx.com/fxml" that might appear in the output window.

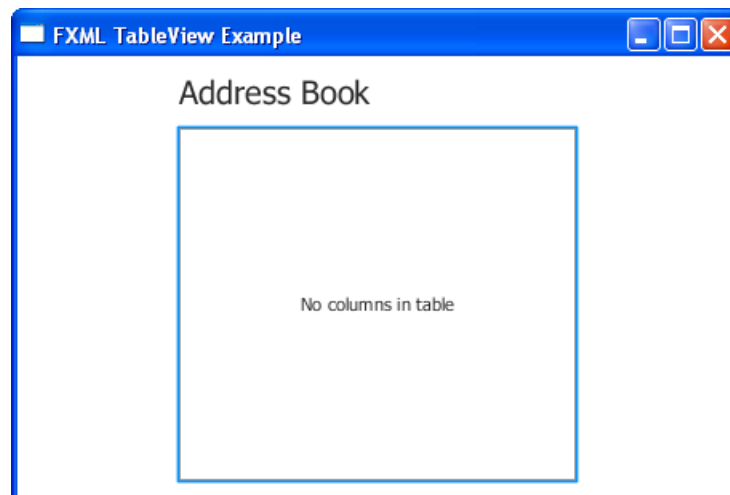
4. Add a Label and a TableView component to the GridPane layout container. The code is in [Example 4-3](#).

Example 4-3 Label and TableView

```
<GridPane alignment="CENTER" hgap="10.0" vgap="10.0"
  xmlns:fx="http://javafx.com/fxml"
  fx:controller="fxmtableview.FXMLTableViewController">
  <padding>
    <Insets bottom="10.0" left="10.0" right="10.0" top="10.0"/>
  </padding>
  <Label style="-fx-font: NORMAL 20 Tahoma;" text="Address Book"
    GridPane.columnIndex="0" GridPane.rowIndex="0">
  </Label>
  <TableView fx:id="tableView" GridPane.columnIndex="0"
    GridPane.rowIndex="1">
  </TableView>
</GridPane>
```

5. Add the import statement for the Insets class.


```
<?import javafx.geometry.Insets?>
```
6. Run the program. You will see the label Address Book and a table with the text "No columns in table," which is the default caption defined by the TableView implementation, as shown in [Figure 4-2](#).

Figure 4-2 Table with No Columns

Add Columns to the Table

Use the `TableColumn` class to add three columns for displaying the data: First Name, Last Name, and Email Address. The code is in [Example 4-4](#).

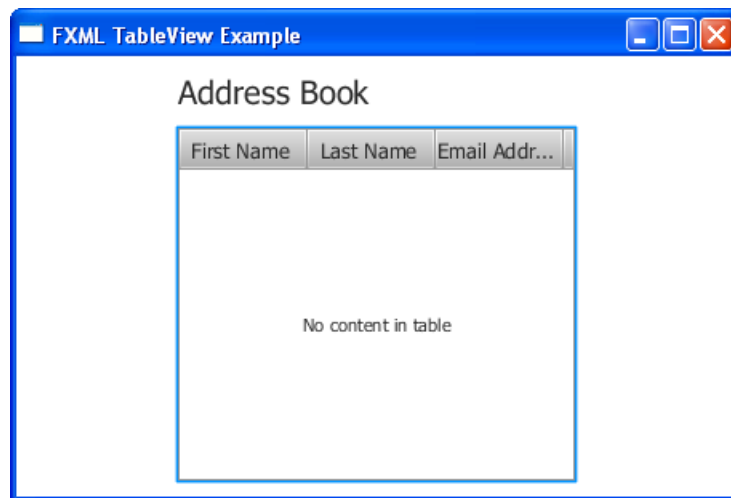
Example 4-4 Table Columns

```
<TableView fx:id="tableView" GridPane.columnIndex="0" GridPane.rowIndex="1">
  <columns>
    <TableColumn text="First Name">
    </TableColumn>
    <TableColumn text="Last Name">
    </TableColumn>
    <TableColumn text="Email Address">
    </TableColumn>
  </columns>
</TableView>
```

Tip: For more information on the `TableColumn` class or any other JavaFX class discussed in this tutorial, see the API documentation.

[Figure 4-3](#) shows the table with the columns for First Name, Last Name, and Email Address.

Figure 4-3 Address Book with Three Columns



Define the Data Model

When you create a table in a JavaFX application, it is a best practice to implement a class that defines the data model and provides methods and fields to further work with the table. Create a `Person` class to define the data for the address book.

1. In NetBeans IDE, right-click the `fxmltableview` folder under Source Packages, and choose **New** then **Java Class**.
2. Name the class **Person** and then click **Finish**.
3. Implement a `Person` class to define the data, as shown in [Example 4-5](#).

Example 4-5 Person Class

```

package fxmhtableview;

import javafx.beans.property.SimpleStringProperty;

public class Person {
    private final SimpleStringProperty firstName = new SimpleStringProperty("");
    private final SimpleStringProperty lastName = new SimpleStringProperty("");
    private final SimpleStringProperty email = new SimpleStringProperty("");

    public Person() {
        this("", "", "");
    }

    public Person(String firstName, String lastName, String email) {
        setFirstName(firstName);
        setLastName(lastName);
        setEmail(email);
    }

    public String getFirstName() {
        return firstName.get();
    }

    public void setFirstName(String fName) {
        firstName.set(fName);
    }

    public String getLastName() {
        return lastName.get();
    }

    public void setLastName(String fName) {
        lastName.set(fName);
    }

    public String getEmail() {
        return email.get();
    }

    public void setEmail(String fName) {
        email.set(fName);
    }
}

```

Associate Data with the Table Columns

The next tasks are to define rows for the data and associate the data with the table columns. You add this code to the FXML file.

1. In the `fxml_tableview.fxml` file, create an `ObservableList` array and define as many data rows as you would like to show in your table. Sample code is in [Example 4-6](#). Add the code between the `</columns>` and `</TableView>` markup.

Example 4-6 ObservableList Array

```

</columns>
<items>

```

```

<FXCollections fx:factory="observableArrayList">
  <Person firstName="Jacob" lastName="Smith"
    email="jacob.smith@example.com"/>
  <Person firstName="Isabella" lastName="Johnson"
    email="isabella.johnson@example.com"/>
  <Person firstName="Ethan" lastName="Williams"
    email="ethan.williams@example.com"/>
  <Person firstName="Emma" lastName="Jones"
    email="emma.jones@example.com"/>
  <Person firstName="Michael" lastName="Brown"
    email="michael.brown@example.com"/>
</FXCollections>
</items>
</TableView>

```

- Specify a cell factory for each column to associate the data with the column, as shown in [Example 4-7](#).

Example 4-7 Cell Factories

```

<columns>
  <TableColumn text="First Name">
    <cellValueFactory><PropertyValueFactory property="firstName" />
  </cellValueFactory>
</TableColumn>
  <TableColumn text="Last Name">
    <cellValueFactory><PropertyValueFactory property="lastName" />
  </cellValueFactory>
</TableColumn>
  <TableColumn text="Email Address">
    <cellValueFactory><PropertyValueFactory property="email" />
  </cellValueFactory>
</TableColumn>
</columns>

```

Cell factories are implemented by using the `PropertyValueFactory` class, which uses the `firstName`, `lastName`, and `email` properties of the table columns as references to the corresponding methods of the `Person` class.

- Import the required packages, as shown in [Example 4-8](#):

Example 4-8 Import Statements

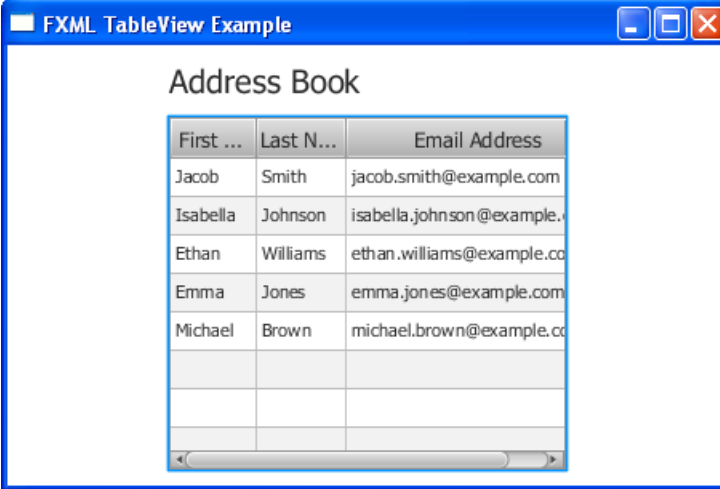
```

<?import javafx.scene.control.cell.*?>
<?import javafx.collections.*?>
<?import fxmtableview.*?>

```

Running the application at this point shows the table populated with data, as shown in [Figure 4-4](#).

Figure 4–4 Table with Data



First ...	Last N...	Email Address
Jacob	Smith	jacob.smith@example.com
Isabella	Johnson	isabella.johnson@example.com
Ethan	Williams	ethan.williams@example.com
Emma	Jones	emma.jones@example.com
Michael	Brown	michael.brown@example.com

Here are some built-in features of the `TableView` class for you to try:

- Resize a column width by dragging the column divider in the table header to the left or right.
- Move a column by dragging the column header.
- Alter the sort order of data by clicking a column header. The first click enables an ascending sort order, the second click enables a descending sort order, and the third click disables sorting. By default, no sorting is applied.

Set Sort Order on Startup

In this task, you set the sort order so that the entries in the First Name column appear in ascending alphabetical order on application startup. You do this by creating an ID for the table column and then setting up a reference to it.

1. Add an ID to the First Name column:

```
<TableColumn fx:id="firstNameColumn" text="First Name">
```

2. Specify the sort order by adding the code in [Example 4–9](#) between the `</items>` and `</TableView>` markup.

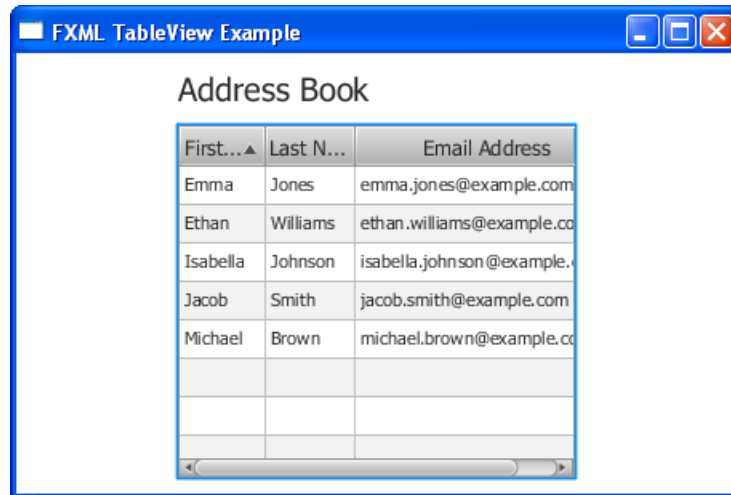
Example 4–9 Sort Order

```

</items>
<sortOrder>
  <fx:reference source="firstNameColumn"/>
</sortOrder>
</TableView>

```

You can see the results in [Figure 4–5](#).

Figure 4–5 Table with First Column Data Sorted at Startup

Define Column Widths

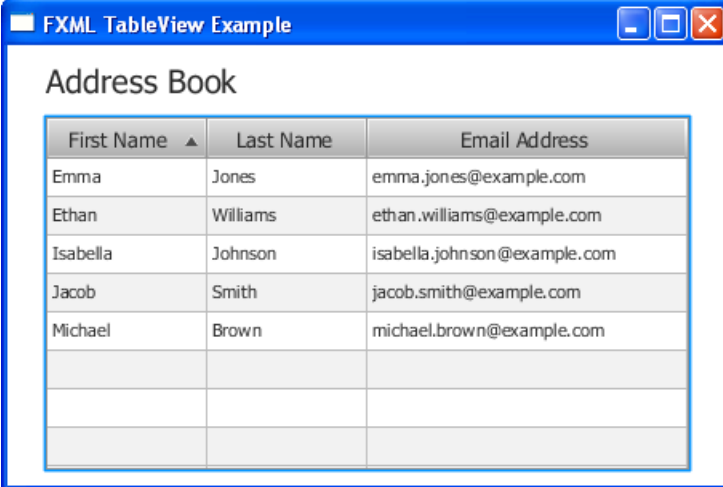
Add the `prefWidth` property to increase the column widths, as shown in [Example 4–10](#).

Example 4–10 Column Widths

```
<TableColumn fx:id="firstnameColumn" text="First Name" prefWidth="100">
  <cellValueFactory><PropertyValueFactory property="firstName" />
</cellValueFactory>
</TableColumn>
<TableColumn text="Last Name" prefWidth="100">
  <cellValueFactory><PropertyValueFactory property="lastName" />
</cellValueFactory>
</TableColumn>
<TableColumn text="Email Address" prefWidth="200">
  <cellValueFactory><PropertyValueFactory property="email" />
</cellValueFactory>
</TableColumn>
```

The result is in [Figure 4–6](#). The column widths have been increased so that all data is visible in each table row.

Figure 4–6 Table with Column Widths Set



First Name ▲	Last Name	Email Address
Emma	Jones	emma.jones@example.com
Ethan	Williams	ethan.williams@example.com
Isabella	Johnson	isabella.johnson@example.com
Jacob	Smith	jacob.smith@example.com
Michael	Brown	michael.brown@example.com

Set Alignment in Table Cells

Another customization is to set the alignment of the data in the table cells. You implement the logic in a new class named `FormattedTableCellFactory` and then set the alignment in the `<TableColumn>` markup in the FXML code.

1. In NetBeans IDE, right-click the `fxmltableview` folder under Source Packages, and choose **New** then **Java Class**.
2. Name the class **FormattedTableCellFactory** and then click **Finish**.
3. Modify the `FormattedTableCellFactory` class by implementing the `Callback` class and creating instances of the `TextAlignment` and `Format` classes, as shown in [Example 4–11](#). The `S` parameter is the type of the `TableView` generic type and the `T` parameter is the type of the content of all cells in this table column.

Example 4–11 Callback Class

```
public class FormattedTableCellFactory<S, T>
    implements Callback<TableColumn<S, T>, TableCell<S, T>> {
    private TextAlignment alignment;
    private Format format;

    public TextAlignment getAlignment() {
        return alignment;
    }

    public void setAlignment(TextAlignment alignment) {
        this.alignment = alignment;
    }

    public Format getFormat() {
        return format;
    }

    public void setFormat(Format format) {
        this.format = format;
    }
}
```

4. Implement the `TableCell` and `TableColumn` classes by appending the code in [Example 4–12](#). This code overrides the `updateItem` method of the `TableCell` class and calls the `setTextAlignment` method on the table cell.

Example 4–12 `TableCell` and `TableColumn` Classes

```
@Override
@SuppressWarnings("unchecked")
public TableCell<S, T> call(TableColumn<S, T> p) {
    TableCell<S, T> cell = new TableCell<S, T>() {
        @Override
        public void updateItem(Object item, boolean empty) {
            if (item == getItem()) {
                return;
            }
            super.updateItem((T) item, empty);
            if (item == null) {
                super.setText(null);
                super.setGraphic(null);
            } else if (format != null) {
                super.setText(format.format(item));
            } else if (item instanceof Node) {
                super.setText(null);
                super.setGraphic((Node) item);
            } else {
                super.setText(item.toString());
                super.setGraphic(null);
            }
        }
    };
    cell.setTextAlignment(alignment);
    switch (alignment) {
        case CENTER:
            cell.setAlignment(Pos.CENTER);
            break;
        case RIGHT:
            cell.setAlignment(Pos.CENTER_RIGHT);
            break;
        default:
            cell.setAlignment(Pos.CENTER_LEFT);
            break;
    }
    return cell;
}
```

5. Correct the import statements.
6. In the `fxml_tableview.fxml` file, add the following code under the `<cellValueFactory>` markup to provide a center alignment for the First Name column, as shown in [Example 4–13](#).

Example 4–13 `Alignment in Data Cell`

```
<TableColumn fx:id="firstNameColumn" text="First Name" prefWidth="100">
    <cellValueFactory><PropertyValueFactory property="firstName" />
</cellValueFactory>
<cellFactory>
    <FormattedTableCellFactory alignment="center">
</FormattedTableCellFactory>
```

```

    </cellFactory>
</TableColumn>

```

You can create an alignment for the remaining columns using left, right, or center values.

Running the application now results in data that is aligned in the center of the First Name column, as shown in [Figure 4-7](#).

Figure 4-7 Data Center-Aligned in First Name Column

First Name ▲	Last Name	Email Address
Emma	Jones	emma.jones@example.com
Ethan	Williams	ethan.williams@example.com
Isabella	Johnson	isabella.johnson@example.com
Jacob	Smith	jacob.smith@example.com
Michael	Brown	michael.brown@example.com

Add Rows to the Table

You can add the ability for users to add a row of data to the table. Add the application logic in the `FXMLTableViewController` class. Then, modify the user interface to include three text fields and a button for entering the data.

1. Open the `FXMLTableViewController.java` file.
2. Edit the `FXMLTableViewController` class so it looks like the code in [Example 4-14](#).

Example 4-14 `FXMLTableViewController.java`

```

public class FXMLTableViewController {
    @FXML private TableView<Person> tableView;
    @FXML private TextField firstNameField;
    @FXML private TextField lastNameField;
    @FXML private TextField emailField;

    @FXML
    protected void addPerson(ActionEvent event) {
        ObservableList<Person> data = tableView.getItems();
        data.add(new Person(firstNameField.getText(),
            lastNameField.getText(),
            emailField.getText()
        ));

        firstNameField.setText("");
        lastNameField.setText("");
        emailField.setText("");
    }
}

```

}

- Correct the import statements, as shown in [Example 4-15](#).

Example 4-15 Import Statements in FXMLTableViewController

```
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
```

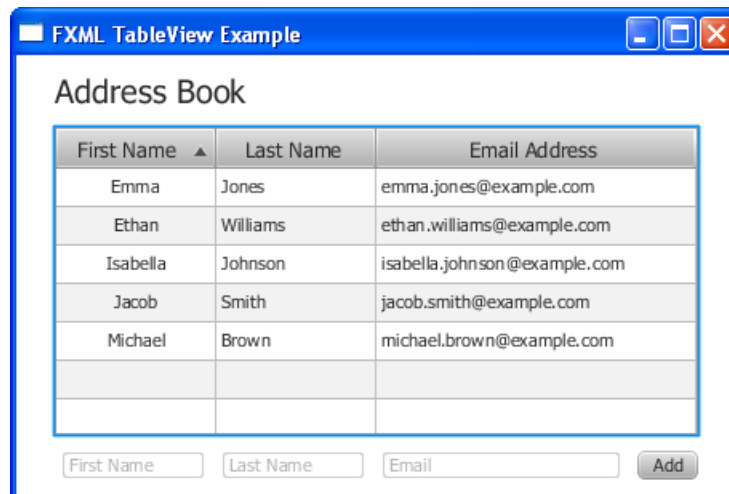
- In the `fxml_tableview.fxml` file, add the following code before the `</GridPane>` markup, as shown in [Example 4-16](#).

Example 4-16 Text Fields and Button for Adding a Row

```
</TableView>
<HBox spacing="10" alignment="bottom_right" GridPane.columnIndex="0"
  GridPane.rowIndex="2">
  <TextField fx:id="firstNameField" promptText="First Name"
    prefWidth="90"/>
  <TextField fx:id="lastNameField" promptText="Last Name"
    prefWidth="90"/>
  <TextField fx:id="emailField" promptText="email"
    prefWidth="150"/>
  <Button text="Add" onAction="#addPerson"/>
</HBox>
</GridPane>
```

Run the application and you will see that the text fields and button appear below the table, as shown in [Figure 4-8](#). Enter data in the text fields and click **Add** to see the application in action.

Figure 4-8 Table with Text Fields and Button for Adding Data



Where to Go from Here

This concludes the Address Book tutorial, but here are some things for you to try next:

- Provide a filter to verify that data was entered in the correct format.
- Customize the table by applying a cascading style sheet to distinguish between empty and non-empty rows. See "Styling UI Controls with CSS" in JavaFX UI Controls for more information.
- Enable editing of data in the table. See Editing Data in the Table in Using JavaFX UI Controls for pointers.
- See [Deployment of FXML Applications](#) for additional deployment options.
- 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
- For an example of an FXML application that uses data from a database, take a look at the Henley Sales Application sample by downloading the JavaFX Samples zip file at <http://www.oracle.com/technetwork/java/javafx/downloads/>

The NetBeans projects and source code for this sample (called DataApp) are included in the samples zip file. See the readme for instructions on how to set up and run the application.

This DataApp sample provides several examples of how to populate a table from a database. In particular, look at the following files:

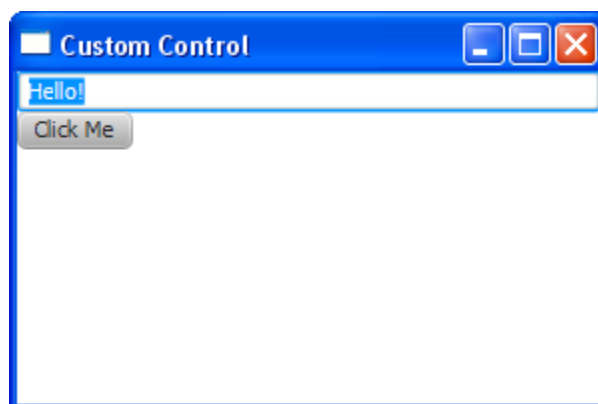
- DataAppClient\src\com\javafx\experiments\dataapp\client\historytab\history-tab.fxml
- DataAppClient\src\com\javafx\experiments\dataapp\client\livetab\live-tab.fxml
- DataAppClient\src\com\javafx\experiments\dataapp\client\productstab\products-tab.fxml

Creating a Custom Control with FXML

In this tutorial, you create an application with a custom control that consists of a text field and a button, as shown in [Figure 5-1](#).

Before you start, ensure that the version of NetBeans IDE that you are using supports JavaFX 2.2. It is assumed that you are familiar with the basic structure of an FXML project (.java, .fxml, and controller files). If you are not familiar with it, then first complete the FXML tutorial in the Getting Started series and then continue with this tutorial.

Figure 5-1 Custom Control Application



Set Up a Project

Open your NetBeans IDE and perform the following steps to set up a JavaFX FXML project:

1. From the **File** menu, choose **New Project**.
2. In the **JavaFX** category, choose **JavaFX FXML Application**. Click **Next**.
3. Name the project **CustomControlExample** and click **Finish**.
4. Rename `SampleController.java` to `CustomControl.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 **CustomControl**, and then click **Refactor**.
5. Rename `Sample.fxml` to `custom_control.fxml`

- a. Right-click **Sample.fxml** and choose **Rename**.
- b. Enter **custom_control** and click **OK**.

Create the Basic User Interface

Define the structure of a simple custom control containing a `TextField` and a `Button` instance. The root container is defined as an instance of the `javafx.scene.layout.VBox` class.

1. Open the `custom_control.fxml` file.
2. Delete the `<AnchorPane>` markup that NetBeans IDE automatically generated.
3. Add code for the root container as shown in [Example 5-1](#).

Example 5-1 Defining the Root Container

```
<fx:root type="javafx.scene.layout.VBox" xmlns:fx="http://javafx.com/fxml">
  <TextField fx:id="textField"/>
  <Button text="Click Me" onAction="#doSomething"/>
</fx:root>
```

4. Remove unused import statements, as shown in [Example 5-2](#).

Example 5-2 Unused Import Statements

```
<?import java.lang.*?>
<?import java.util.*?>
```

Create a Controller

In this example, the `CustomControl` class extends the `VBox` class (the type declared by the `<fx:root>` element), and sets itself as both the root and controller of the FXML document in its constructor. When the document is loaded, the contents of the `CustomControl` instance will be populated with the contents of the document.

1. Open the `CustomControl.java` file and remove the code that NetBeans IDE automatically generated.
2. Add code as shown in [Example 5-3](#).

Example 5-3 The CustomControl Class as Both the Root and Controller of the FXML Document

```
package customcontrolexample;

import java.io.IOException;

import javafx.beans.property.StringProperty;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;

public class CustomControl extends VBox {
    @FXML private TextField textField;

    public CustomControl() {
        FXMLLoader fxmlLoader = new FXMLLoader(getClass().getResource(
```



```

"custom_control.fxml"));
    FXMLLoader.setRoot(this);
    FXMLLoader.setController(this);

    try {
        FXMLLoader.load();
    } catch (IOException exception) {
        throw new RuntimeException(exception);
    }
}

public String getText() {
    return textProperty().get();
}

public void setText(String value) {
    textProperty().set(value);
}

public StringProperty textProperty() {
    return textField.textProperty();
}

@FXML
protected void doSomething() {
    System.out.println("The button was clicked!");
}
}

```

Load the FXML Source File and Define Stage and Scene

The CustomControlExample.java file contains code for setting up the main application class. It defines the stage and scene, and loads the FXML source file. More specific to FXML, this class loads the FXML source file using the CustomControl class.

1. Open the CustomControlExample.java file.
2. Remove the line of code that contains a call to the FXMLLoader class as shown in [Example 5-4](#).

Example 5-4 Removing the Call to the FXMLLoader Class

```
Parent root = FXMLLoader.load(getClass().getResource("Sample.fxml"));
```

3. Create an instance of the CustomControl class and specify the text for the custom control as shown in [Example 5-5](#).

Example 5-5 Instantiating the CustomControl Class

```
CustomControl customControl = new CustomControl();
customControl.setText("Hello!");
```

4. Remove the lines of code that set the stage and scene, and define the stage and scene as shown in [Example 5-6](#).

Example 5-6 Defining the Stage and Scene

```
stage.setScene(new Scene(customControl));
stage.setTitle("Custom Control");
```

```
stage.setWidth(300);  
stage.setHeight(200);  
stage.show();
```

5. Press Ctrl (or Cmd) + Shift + I to correct the import statements.

After you create a custom control, you can use instances of this control in code or in markup, just like any other control as shown in [Example 5-7](#) and [Example 5-8](#).

Example 5-7 Using an Instance of the CustomControl Class in Code

```
HBox hbox = new HBox();  
CustomControl customControl = new CustomControl();  
customControl.setText("Hello World!");  
hbox.getChildren().add(customControl);
```

Example 5-8 Using an Instance of the CustomControl Class in Markup

```
<HBox>  
    <CustomControl text="Hello World!"/>  
</HBox>
```

To download the source code of the Custom Control application, click the [CustomControlExample.zip](#) link.

Deployment of FXML Applications

This page describes why some FXML applications need digital signatures and presents an alternative to signing the application.

For information about how to begin deploying simple applications, see "Basic Deployment" in the JavaFX Getting Started tutorials at http://docs.oracle.com/javafx/2/get_started/basic_deployment.htm

For detailed information about application packaging and deployment, see *Deploying JavaFX Applications* at <http://docs.oracle.com/javafx/2/deployment/jfxpub-deployment.htm>

For web deployment, applications must be signed when there are `@FXML` annotations in the controller class. The `FXMLLoader` class uses reflection to set annotated fields. It calls the `setAccessible()` method on controller fields that are protected or private so it can inject the value from the FXML markup. The `setAccessible()` method is a privileged operation, and that privilege is not enabled by default for web applications.

If you use NetBeans IDE, your application is packaged automatically as a JavaFX application, and by default the JAR file is signed to ensure that it will run on the web. If you plan to run your application only on the desktop, then you can change the project properties so the application is not signed, by clearing the Request Unrestricted Access checkbox in the Deployment screen.

If you deploy using the JavaFX Packager tool or an Ant task, then include the FXML file in the JAR file as an uncompiled file, and sign the application if it contains `@FXML` annotations and you plan to deploy to the web.

The alternative to signing the application is to make your controller fields and handler methods public. While this is usually not considered a good practice, in the case of FXML applications, the controller instance is generally visible only to the FXML Loader that created it, which in effect is similar to creating a private inner class with public members.

The signing requirement for applications that contain `@FXML` annotations is being tracked as an Atlassian Jira issue: <http://javafx-jira.kenai.com/browse/RT-14883>

