SENG202 – Software Engineering Project Workshop

2020

Tutorial 4 – User Interface with JavaFX

1. Introduction

Today's tutorial should be completed individually and can be kept completely local (i.e. there is no need to commit this to any repository).

JavaFX is a set of graphics and media packages that enable developers to design, create, test, debug, and deploy rich client applications that operate consistently across diverse platforms. Before JavaFX, Swing was the most Java GUI library, but nowadays JavaFX is taking the lead and has become the preferred UI library for Java.

The main difference between JavaFX and Swing, is the ability to better separate the view from the model. JavaFX is built around an MVC approach, where the user interface can be easily separated from your logic – using .fxml files, however there still remains the option to write procedural JavaFX code.

Summary of JavaFX Features

- New and improved interface tools (buttons, checkboxes, bar and pie charts, date-pickers, accordion panes, tabbed panes, media player, web rendering controls etc.). You can find the summary of JavaFX controls here (https://docs.oracle.com/javafx/2/ui_controls/overview.htm).
- Cascading Style Sheets (CSS) separate appearance and style from implementation so that developers can concentrate on coding.
- A new 'language' called FXML (XML for JavaFX layouts), which like HTML is used only to define the interface of an application, keeping it completely separate from the code logic.
- 'Scene Builder' is a tool used to build user interfaces in an interactive manner, rather than editing raw markup code.
- An integrated library for graphics (both 2D and 3D) as well as animation tools.
- Canvas API. You can draw directly inside a JavaFX scene area using the Canvas API, which consist of one graphical element (node).
- Mobile platform development tools.
- Multitouch Support. JavaFX provides support for multitouch operations, based on the capabilities of the underlying platform.
- JavaFX is an open source library, therefore it is possible for the wide Java community to be developing tools, plugins and UI controls that go beyond the offerings of Oracle.

Objectives:

- Understand what JavaFX is and how to use it
- Understand basic JavaFX application structure that you can build on
- Learn how to create a simple JavaFX application
- Learn how to use SceneBuilder
- Complete the JavaFX quiz on the Quiz Server. The quiz will open in the second hour of the lab.

2. JavaFX Application Layout Overview

In general, a JavaFX application contains one or more stages which correspond to windows. Each stage has a scene attached to it. Each scene can have an object graph of controls, layouts etc. attached to it – this is called the scene graph. An example of a typical hierarchy is shown in Figure 1 below.

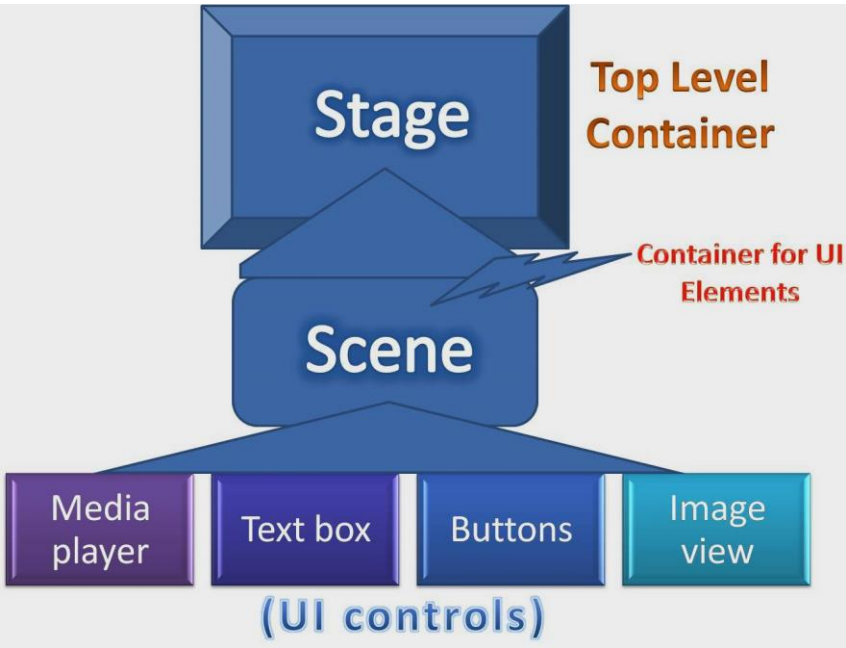

Figure 1. JavaFX Application Layout Hierarchy

When used in a desktop environment, a JavaFX application can have multiple windows open. Each window has its own stage. Each stage is represented by a Stage object inside a JavaFX application. A JavaFX application has a primary Stage object which is created for you by the JavaFX runtime. A JavaFX application can create additional Stage objects if it needs additional windows open. For instance, dialogs and wizards etc. To display anything on a stage in a JavaFX application, you need a scene. A stage can only show one scene at a time, but it is possible to exchange the scene at runtime. Just like a stage in a theater can be rearranged to show multiple scenes during a play, a stage object in JavaFX can show multiple scenes (one at a time) during the life time of a JavaFX application.

You might wonder why a JavaFX application would ever have more than one scene per stage. Imagine a computer game. A game might have multiple "screens" to show to the user. For instance, an initial menu screen, the main game screen (where the game is played) and a high score screen. Each of these

screens can be represented by a different scene. When the game needs to change from one screen to the next, it simply attaches the corresponding scene to the Stage object of the JavaFX application.

A scene is represented by a Scene object inside a JavaFX application. A JavaFX application must create all Scene objects it needs. All visual components (controls, layouts etc.) must be attached to a scene to be displayed, and that scene must be attached to a stage for the whole scene to be visible. The total object graph of all the controls, layouts etc. attached to a scene is called the scene graph.

A single element in a scene graph is called a **Node**. Each node has an ID, style class, and bounding volume. With the exception of the root node of a scene graph, each node in a scene graph has a single parent and zero or more children. Nearly all GUI components in JavaFX inherit from the Node class.

Unlike in Swing and Abstract Window Toolkit (AWT), the JavaFX scene graph also includes the graphics primitives, such as rectangles and text, in addition to having controls, layout containers, images and media.

3. Creating a Simple JavaFX Application

First, make sure to enable Internet Enabler if you're on the lab computers.

We're going to use the JavaFX-project-template.zip provided, but you could easily recreate the same structure manually starting with a Maven project from IntelliJ or Eclipse. Unzip the project and then open it in your IDE. IntelliJ and Eclipse should both auto detect the Maven configuration and download the requirements. If you see a prompt in the bottom right, select "Enable auto-import" which will ensure your POM stays up to date when you add dependencies.

You will also need to change a configuration in IntelliJ to detect the changes. Navigate to the Maven settings either by [Ctrl + Shift + A -> Maven Settings] ***or*** [File -> Settings -> Build, Execution, Deployment -> Build Tools -> Maven]. **Check** "Always update snapshots".

Your project will now have five important files added - have a look at what has been put in place for you. Do not despair if you do not understand much at this point there is a long explanation coming. Read on[1].

Here are the important things to know about the basic structure of a JavaFX application:

1. The main class for a JavaFX application extends the javafx.application.Application class. The start() method is the main entry point for all JavaFX applications, and loads the first window. **- see: SampleApplication.java**
2. Controller files contain the code needed to run when the user interacts with the program. As a simple example, when the user clicks the button, a handler for the button click event triggers a counter to increase in this case.[2] **- see: SampleController.java**

---

[1] Throughout this lab we will refer to IntelliJ IDE but you are welcome to work on equivalent tasks in Eclipse if that is your preferred IDE

3. FXML documents are where most of the interface information will go. If you are familiar with XML, it will look somewhat familiar. You can actually write your own code in here to modify the interface. More on this later. **- see: sample.fxml**
4. **pom.xml** – This is where our dependencies go, as well as the instructions for how to build our project. As you can see, we've already added some dependencies and build instructions for JavaFX.
5. **Main.java** - This is where you will be launching the application from. It simply links to the SampleApplication.java and launches it but is required as launching the SampleApplication.java directly is not possible due to the way the *extends Application* works.

Go ahead and run this JavaFX application **using the Main class**. You should see a small window with the title "Hello World", with a button and a counter as below in Figure 2**.** The design for this can be seen in sample.fxml and the counter logic in SampleController.java. If you do not see this window, or are having issues building, please ask for assistance from one of the tutors.

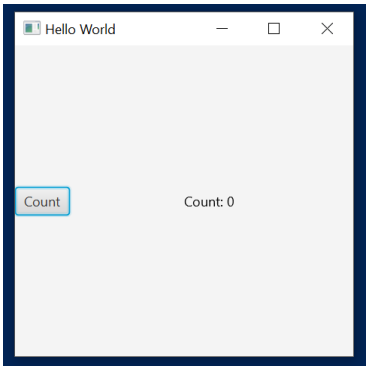

Figure 2. Hello World counter application

We are going to start by creating a simple "Login" application. This app will have a main form with Username and Password text boxes, and a button to submit credentials. To make the example simple enough to cover basics, the button will only display a message rather than actually verifying the credentials.

Forms and GridPane Layout

It is important to note that you could lay out a UI by simply setting the position and size properties for each UI element. However, this doesn't scale well when dealing with different window sizes and resolutions.

---

[2] You can read more about events and how to process them here : https://docs.oracle.com/javafx/2/events/processing.htm

The better (and easier) option is to make use of layout panes. As a window is resized, the layout pane automatically repositions and resizes the nodes that it contains according to the properties for the nodes.[3]

For this exercise we will use GridPane layout because it enables us to create a flexible grid of rows and columns in which to lay out controls. We can place controls in any cell in the grid, and we can make controls span cells where needed. The chosen layout will represent the root node. All the controls (e.g. text boxes, buttons) will be child nodes of this root node.

Creating a form is a common activity when developing an application. This tutorial teaches you the basics of screen layout, how to add controls to a layout pane, and how to create input events.

In the application that we created previously, comment out the existing start() method (we will need this again later), and add the following start(...) method (in SampleApplication.java) to set up the Stage[4]:

```
@Override
public void start(Stage primaryStage) {
    primaryStage.setTitle("Login");
    primaryStage.show();
}
```

GridPane Layout

Add the following code to create GridPane layout. This code should be called ***before*** the stage is shown. Either place this code in the start() method, or create an additional method such as "createLayout()".

When you add the follow code you will need to include the necessary imports (you can import classes in IntelliJ by clicking on the red class and then pressing alt + enter) and try to understand what we have just done. Read on and check if your assumptions were correct. Talk to a tutor if unsure.

Remember to use javafx.* imports, not AWT or Swing imports, as it is common to have overlap.

```
GridPane gridPane = new GridPane();
gridPane.setAlignment(Pos.CENTER);
gridPane.setHgap(10);
gridPane.setVgap(10);
gridPane.setPadding(new Insets(25, 25, 25, 25));

Scene scene = new Scene(gridPane, 400, 250);
primaryStage.setScene(scene);
```

Remember to use javafx.* imports, not AWT or Swing imports, as it is common to have overlap.

---

[3] To read more about layout panes go to: http://docs.oracle.com/javafx/2/layout/builtin_layouts.htm

[4] For the purpose of this exercise we are not going to use the FXML files and css file. More about that later.

We have created a GridPane object and assigned it to the variable named gridPane. 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. The insets are in the order of top, right, bottom, and left. In this example, there are 25 pixels of padding on all sides. The scene is created with the grid pane as the root node, which is a common practice when working with layout containers. Thus, as the window is resized, the nodes within the grid pane are resized according to their layout constraints. In this example, the grid pane remains in the center when you grow or shrink a window. The padding properties ensure there is padding around the grid pane when you make the window smaller. This code sets the scene width and height to 400 by 250. If you do not set the scene dimensions, the scene defaults to the minimum size needed to display its contents.

Go ahead and run your application, you should see a window similar to the sample application, and as shown in Figure 3 below.

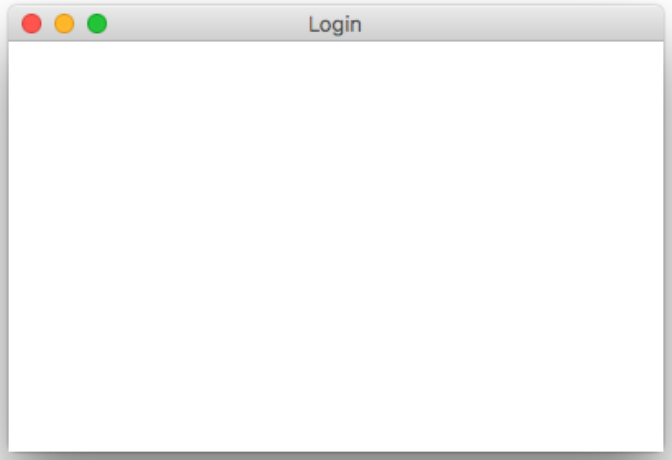

Figure 3. JavaFX Empty Login Window

Adding Text, Labels, and TextFields to your JavaFX Scene

Our form requires username and password fields for gathering information from the user. The code for creating these controls is below. Add this code after the line that sets the grid padding property:

```
Text sceneTitle = new Text("Welcome!");
sceneTitle.setFont(Font.font("Tahoma", FontWeight.NORMAL, 20));
gridPane.add(sceneTitle, 0, 0, 2, 1); // Col 0, Row 0, ColSpan 2, RowSpan 1

Label username = new Label("Username:");
gridPane.add(username, 0, 1); // Col 0, Row 1

TextField usernameTextField = new TextField();
gridPane.add(usernameTextField, 1, 1); // Col 1, Row 1
```

```
Label password = new Label("Password:");
gridPane.add(password, 0, 2); // Col 0, Row 2

PasswordField passwordField = new PasswordField();
gridPane.add(passwordField, 1, 2); // Col 1, Row 2
```

Again, add the necessary imports for controls used and inspect the code. Can you predict how the screen will look like? Run the application and see if you were right.
Read the following explanation if you still unsure.

The first line creates a Text object that cannot be edited, sets the text to Welcome, and assigns it to a variable named sceneTitle. The next line uses the *setFont()* method to set the font family, weight, and size of the sceneTitle variable. Using an inline style is appropriate where the style is bound to a variable, but a better technique for styling the elements of your user interface is by using a stylesheet (CSS). We will cover this in the next exercise.

The *GridPane.add()* method adds the sceneTitle variable to the layout grid. The numbering for columns and rows in the grid starts at zero, and sceneTitle is added in column 0, row 0. The last two arguments of the *grid.add()* method set the column span to 2 and the row span to 1.

The next lines create a Label object with text Username at column 0, row 1 and a TextField object that can be edited. The text field is added to the grid pane at column 1, row 1. A password field and label are created and added to the grid pane in a similar fashion.

Run the application and confirm it looks similar to that of Figure 4.

Note: When working with a Grid pane, you can display the grid lines, which is useful for debugging purposes. In this case, you can add *gridPane.setGridLinesVisible(true)* after the line that adds the password field. When you run the application, you'll see the lines for the grid columns and rows as well as the gap properties.

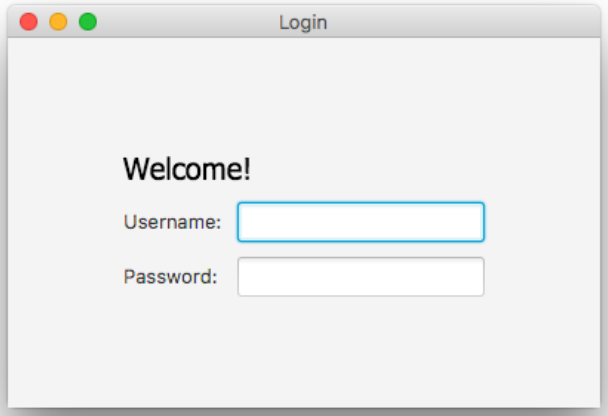

Figure 4. Login Form

Adding 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 once the button is pressed.

Add this code before the code for the scene:

```
Button button = new Button("Sign in");
HBox hboxButton = new HBox(10);
hboxButton.setAlignment(Pos.BOTTOM_RIGHT);
hboxButton.getChildren().add(button);
gridPane.add(hboxButton, 1, 4);
```

The first line creates a button named *button* with the label "Sign in", and the second line creates an HBox layout pane named *hboxButton* with spacing of 10 pixels. An HBox layout is (as the name might suggest), a layout where nodes will appear next to each other horizontally.

The HBox sets an alignment for the button that is different from the alignment applied to the other controls in the grid pane. The alignment property has a value of Pos.BOTTOM_RIGHT, which positions a node at the bottom of the space vertically and at the right edge of the space horizontally. The button is added as a child of the HBox pane, and the HBox pane is added to the grid in column 1, row 4.

At this point the Button won't be verifying the user name and password as this is only a layout exercise. We will only display a message once the Button is clicked.

Now, add a Text control for displaying the message:

```
Text actionTarget = new Text();
gridPane.add(actionTarget, 1, 6);
```

Add Code to Handle an Event

Finally, make the button display the text message when the user presses it. Add the following code before the code for the scene. Add necessary imports.

```
button.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
        actionTarget.setFill(Color.FIREBRICK);
        actionTarget.setText("Sign in button pressed.");
    }
});
```

Run the Application

Right-click the Login project node in the Projects window, choose Run, and then click the Sign in button. You should get “Sign in button pressed” message displayed.

4. Creating the UI with SceneBuilder

Now that you have had a chance to learn the fundamentals of JavaFX, quite likely you will have noticed the difficulties and frustration in building a GUI in this way.

Rather than write our UI code in raw Java, we use the application "SceneBuilder", which allows us to design a user interface in an interactive manner – think drag-and-drop.

You will remember earlier, when we created a JavaFX project we had initial files, including a sample.fxml file. SceneBuilder creates/modifies FXML files, allowing you to keep the logic of your application separate from the user interface.

The lab computers will already have SceneBuilder installed, if you are working on a personal machine, then head on over to the SceneBuilder website and download the appropriate version for your operating system.

Open up SceneBuilder and you should be greeted with a screen similar to that shown in Figure 5.

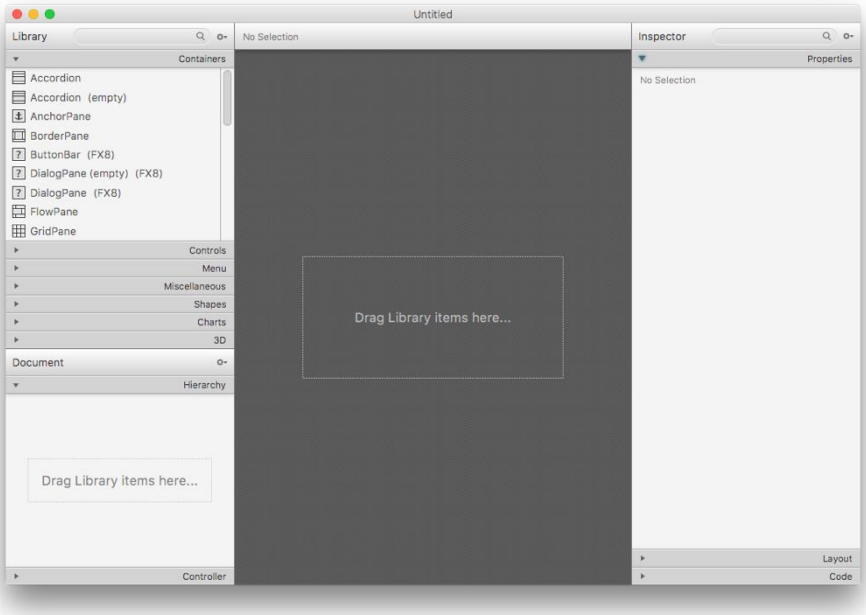

Figure 5. SceneBuilder upon first launch.

There are four main sections of the SceneBuilder, Library, Document and Inspector and View.

The **Library** section is where you can choose a particular GUI component. These are separated into categories appropriately.

The **Document** section shows the Scene graph of the current document and the hierarchy of the components (nodes).

The **Inspector** section is for editing the properties of a particular GUI component. When a node is selected (in either the Document section, or the View section), the Inspector will update to show the editable attributes of this node. Nodes have many different attributes and settings, such as the "alignment" attribute we specified in the code previously for an HBox and GridPane.

The **View** section is (as the name suggests) where you can view your interface in an interactive manner. You can drag components from the Library section here.

Login Application

The next task is to recreate the same interface as before, using SceneBuilder. We will use exactly the same components as before. It's probably a good idea to save immediately and frequently as there is no Local History like we are accustomed to in IntelliJ.

To start with, select **GridPane** from the Library section, and drag it into the View section. By default, this will create a grid pane with two columns, and three rows – don't worry about this for now.

Go ahead and make the following changes to the GridPane we just created, by finding and editing the appropriate attribute within the **Inspector** section:

1. Alignment: CENTER
2. Hgap: 10 (pixels) - Located within the "layout" tab
3. Vgap: 10 (pixels) - Located within the "layout" tab
4. Padding: 25, 25, 25, 25 (Top, Right, Bottom, Left) - The ">" arrow next to the first cell will set all of the other cells to match its value.

Next we will move on to adding the remaining components like we did previously.

Start with a new Text component. Drag this into the cell at column 0, row 0. Change the text to "Welcome!", and adjust the font as necessary, if you can't find the exact font options we defined in the procedural code before don't worry, just make the font size larger. Change the column span to 2 as we defined previously.

Now it is on to the labels. Drag a label into each of the appropriate cells, then drag a TextField and a PasswordField into the view, in the correct position. You should end up with something similar to that of Figure 6 (ignore the arrow for now).

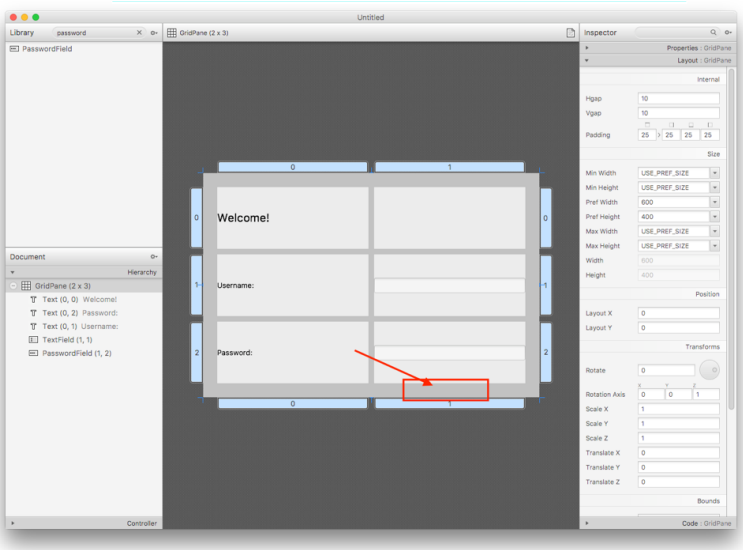

Figure 6. SceneBuilder after adding a Text node, two Labels, a TextField and a PasswordField.

Now comes to adding the Login button. As you will see, there are only three rows, we need two more rows for the button, and some additional text. There are two ways of doing this, either by dragging the new component onto the area shown in Figure 6, or by right clicking the bottom row number and selecting "Add row below".

Add the login button and the remaining Text node to the appropriate cells, and change the text displayed accordingly (Note the text for the Text node should be "Sign in button pressed.").

Now comes to alignment, under the Properties tab in the Inspector section, you will see that changing the Alignment will have no effect. This is because the properties defined under this tab are related to

the node's contents, the alignment setting shown refers to the text position within the button – this is not what we want.

Rather than wrap this button in an HBox as we did previously, we can edit the Layout parameters of this button and specify the layout alignment in both the horizontal and vertical directions. Change the Valignment to BOTTOM, and the Halignment to RIGHT. At this stage you should be left with something similar to that of Figure 7.

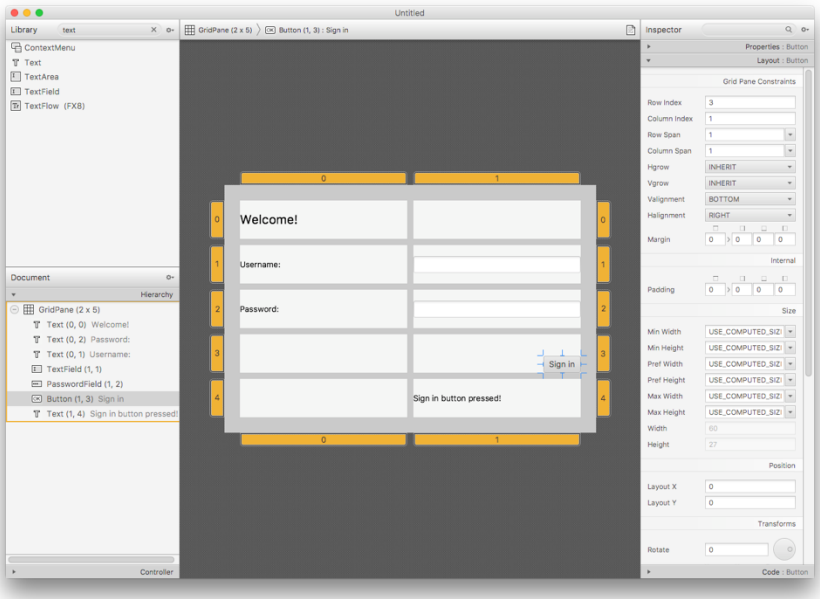

Figure 7. SceneBuilder after aligning the button to the bottom right.

There are now just two more steps until we have finished creating our interface in SceneBuilder. We need to hide the text shown ("Sign in button pressed!") by default – we will later show this when the button is pressed. To do this, uncheck the "Visible" section within the Properties tab in the Inspector.

Finally, you have probably noticed the sizing of everything so far – much bigger than in the previous example. To fix this, we have to adjust the size of the root node – the GridPane. You will find the options to do this in the Inspector, under the Layout tab. Change the following:

1. Pref Width: 400
2. Pref Height: 250

Save this file in the same directory as the sample.fxml file is located – it should be named appropriately (perhaps "login.fxml").

5. Creating the UI with SceneBuilder

You have learnt the concept of MVC (Model-View-Controller) in SENG201. As you have now created the view for the login window, the next step is to create the controller.

Before we implement the controller we must first be able to reference the TextField, PasswordField, Button and the Text objects. Inside each of those components there is a field called "fx:id" inside the Code tab, locate each of these and give them appropriate names (you only need to do this for the components that will be either performing actions or be acted upon and therefore the labels do not need to be assigned an id).

In SceneBuilder navigate to the document section in the bottom left of the window, go to the controller tab and write "seng202.LoginController" inside the "Controller Class" text field. This tells the fxml file where the controller class is located. This maps to the folder structure after src/main/java to your given controller file.

Next we want to define the method (located within the LoginController class) that will be called when the button is clicked on. To do this we are going to click on the Button node inside SceneBuilder and navigate to the code tab (Inspector). Inside the "On Action" text field enter "login".

This on action text field denotes which method will be called when the button is pressed, in this case it will be a method called "login()" that is located within the LoginController class. However, at this point we have not actually implemented the login method (or class) but that is coming very soon. Save the file as "login.fxml" to the same directory as the sample.fxml files is located.

Return to your IDE.

Now that we have defined ids for the nodes of interest, we can access them from within our corresponding Controller class. Create a class called "LoginController" and add the following code to your class, replacing the names with the ids that you have used:

```
@FXML
private TextField usernameTextField;

@FXML
private Text buttonPressedText;
```

The @FXML annotation you see above these fields allows the FXMLLoader to inject the values of these fields into the Controller object instantiated. This allows us to interact with these nodes in the same way that we did previously with the procedural code in Section 3.

We defined an "On Action" method name earlier "login" – this means that when the log in button is pressed, the login() method is called within the LoginController class. We are going to implement this method so that when the user presses the login button, the text displayed will greet them by their username.

Add the following method to your Controller class:

```
public void login() {
    // Ensure the user has entered some text before preceeding to the next step.
    if (!usernameTextFeild.getText().isEmpty()) {
        // Get that text and put it inside a preformatted message.
        String loginText = String.format("Thanks for logging in, %s!", usernameTextFeild.getText());
        // Set the hidden Text node value to be that string.
        buttonPressedText.setText(loginText);
        // Show the Text node.
        buttonPressedText.setVisible(true);
    }
}
```

Remember the code you commented out in the first section in SampleApplication.java? Now we need it back. Remove the code we wrote earlier and uncomment the old part.

So that the application loads the correct fxml file you will need to change the FXML loading line in from "sample.fxml" to "login.fxml" (or whatever you saved the file as).

Go ahead and run your application, hopefully you will see a friendly greeting when you log in, as shown in Figure 8.

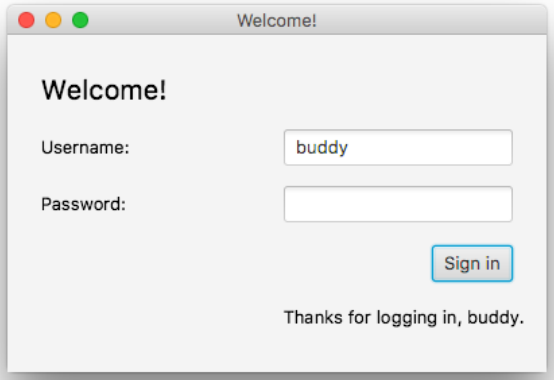

Figure 8. Login application working

6. More advanced GUI

Now we are going to create something that might be useful for your project. Create a new Java class inside your project and name it "MainController", don't worry about adding any code to it yet, that will come shortly. Go into SceneBuilder and create a new file named "main.fxml" and save it to the same directory as the "login.fxml" file.

Next we want to get an BorderPane and add it to the window, set the "prefHeight" to 400 and the "prefWidth" to 600. Next search for a "MenuBar" and drop it within the BorderPane in the top section as shown in Figure 9.

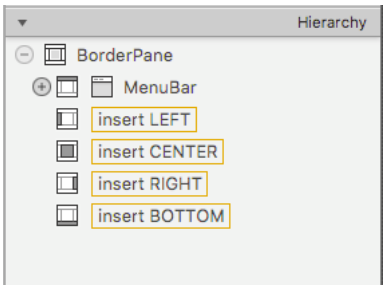

Figure 9. BorderPane

BorderPanes allow for declaring where a certain child node will be located within the window. The result should be a MenuBar across the top of your screen like in Figure 10.

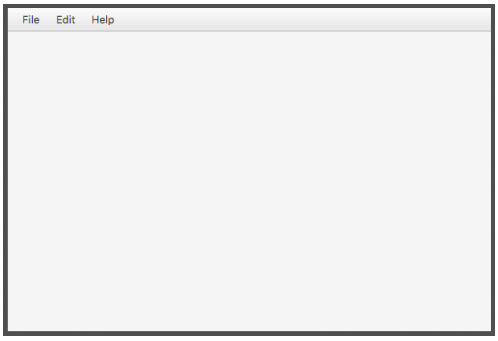

Figure 10. MenuBar in place

Following that we want to add an additional MenuItem to the MenuBar, search for it and drag it into the “Hierarchy” section of the menu as shown in Figure 11, change the text to something appropriate (Login).

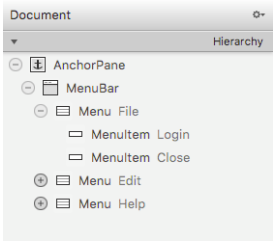

Figure 11. Login MenuItem added to the MenuBar

Set the “onAction” to be “showLogin” inside the code tab like you have done previously in the tutorial. In the Bottom left under the “Controller” tab set the Controller class as “seng202.MainController” and save the file.

Change the value inside the FXMLLoader call in SampleApplication to load “main.fxml” instead of "login.fxml", and the values of the new Scene from (400, 250) to (600, 400), you may also want to change the title of the window to something more appropriate.

Inside the MainController class add the following function:

```
public void showLogin() {
    try {
        Parent root = FXMLLoader.load(getClass().getResource("login.fxml"));
        Stage stage = new Stage();
        // This will cause the login window to always be infront of the main window
        stage.initModality(Modality.APPLICATION_MODAL);
        stage.setResizable(false);
        stage.setTitle("Login Window");
        stage.setScene(new Scene(root, 400, 250));
        stage.show();
    } catch (IOException e) {
        // This is where you would enter the error handling code, for now just print the stacktrace
        e.printStackTrace();
    }
}
```

Now run the application and you should see something similar to figure 12.

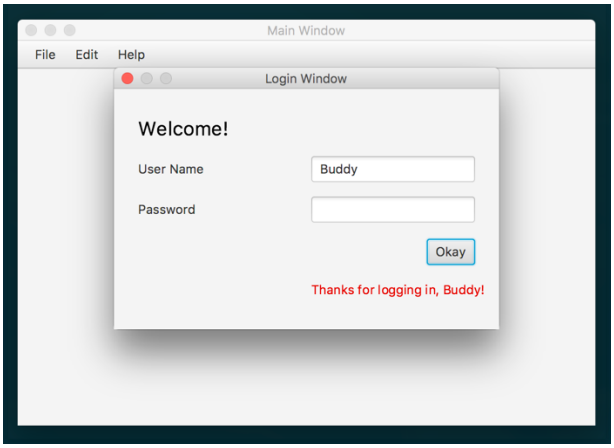

Figure 12. Main window with MenuBar and Login Window

7. Final notes

We expect you to use FXML files to design your GUIs, rather than defining them in code like we did in the beginning. Defining your GUI using FXML allows your GUI design and from to be separated entirely from the implementation of the functionality it has, which follows the MVC principles.

For every FXML file or window you build, there should be an accompanying controller that handles the logic for that window. For example you will have a login.fxml and a LoginController.java, a register.fxml and a RegisterController.java, etc.