JavaFX Documentation Project

Published 2016-10-24

Table of Contents

Chapter 1. Introduction

The JavaFX Documentation Project aims to pull together useful information for JavaFX developers from all over the web. The project is open source and encourages community participation to ensure that the documentation is as highly polished and useful as possible.

1.1. Contributors

This project would not be possible without the contributors who work hard on the content contained within this documentation. Whenever possible contributors are given attribution within the document when they write a section, but it is also important to gather all names here, at the top of the document, to give the recognition that these contributors deserve.

1.2. Contributing

Contributing to this project is easy - fork the GitHub repo, edit the relevant files, and create a pull request! Once merged, your content will form a part of the documentation and you'll have the unending appreciation of the entire community!

The JavaFX Documentation Project uses AsciiDoc as the syntax of choice for writing the documentation. The AsciiDoc Syntax Quick Reference guide is a great resource for those learning how to write AsciiDoc.

Authors are welcome to include a byline beneath the sections that they have authored. To ensure consistency, the recommended format for the byline is the following:

Contributed by <name> - <website>

Chapter 2. Scenegraph

Placeholder whilst things get built…

Chapter 3. UI Controls

3.1. ListView

3.1.1. ListView Filtering in JavaFX

(Contributed by Carl Walker, October 1, 2016)

This article demonstrates how to filter a ListView in a JavaFX Application. Two lists are managed by the Application. One list contains all of the items in the data model. The second list contains the items currently being viewed. A scrap of comparison logic stored as a filter mediates between the two.

Binding is used heavily to keep the data structures in sync with what the user has selected.

This screenshot shows the Application which contains a top row of ToggleButtons which set the filter and a ListView containing the objects.

Figure 1. Screenshot of ListView Filtering App

The complete code — a single .java file — is listed at the end of the article.

Data Structures

The program begins with a domain model Player and an array of Player objects.

```
static class Player {

    private final String team;
    private final String playerName;
    public Player(String team, String playerName) {
        this.team = team;
        this.playerName = playerName;
    }
    public String getTeam() {
        return team;
    }
    public String getPlayerName() {
        return playerName;
    }
    @Override
    public String toString() { return playerName + " (" + team + ")"; }
}
```

The Player class contains a pair of fields, team and playerName. A toString() is provided so that when the object is added to the ListView (presented later), a custom ListCell class is not needed.

The test data for this example is a list of American baseball players.

```
Player[] players = {new Player("BOS", "David Ortiz"),
                    new Player("BOS", "Jackie Bradley Jr."),
                    new Player("BOS", "Xander Bogarts"),
                    new Player("BOS", "Mookie Betts"),
                    new Player("HOU", "Jose Altuve"),
                    new Player("HOU", "Will Harris"),
                    new Player("WSH", "Max Scherzer"),
                    new Player("WSH", "Bryce Harper"),
                    new Player("WSH", "Daniel Murphy"),
                    new Player("WSH", "Wilson Ramos") };
```

Model

As mentioned at the start of the article, the ListView filtering is centered around the management of two lists. All the objects are stored in a wrapped ObservableList playersProperty and the objects that are currently viewable are stored in a wrapped FilteredList, viewablePlayersProperty. viewablePlayersProperty is built off of playersProperty so updates made to players that meet the FilteredList criteria will also be made to viewablePlayers.

```
ReadOnlyObjectProperty<ObservableList<Player>> playersProperty =
        new SimpleObjectProperty<>(FXCollections.observableArrayList());

ReadOnlyObjectProperty<FilteredList<Player>> viewablePlayersProperty =
        new SimpleObjectProperty<FilteredList<Player>>(
                new FilteredList<>(playersProperty.get()
                        ));
```

filterProperty() is a convenience to allow callers to bind to the underlying Predicate.

```
ObjectProperty<Predicate<? super Player>> filterProperty =
    viewablePlayersProperty.get().predicateProperty();
```

The UI root is a VBox which contains an HBox of ToggleButtons and a ListView.

```
VBox vbox = new VBox();
vbox.setPadding( new Insets(10));
vbox.setSpacing(4);

HBox hbox = new HBox();
hbox.setSpacing( 2 );

ToggleGroup filterTG = new ToggleGroup();
```

Filtering Action

A handler is attached the ToggleButtons which will modify filterProperty. Each ToggleButton is supplied a Predicate in the userData field. toggleHandler uses this supplied Predicate when setting the filter property. This code sets the special case "Show All" ToggleButton.

```
@SuppressWarnings("unchecked")
EventHandler<ActionEvent> toggleHandler = (event) -> {
        ToggleButton tb = (ToggleButton)event.getSource();
        Predicate<Player> filter = (Predicate<Player>)tb.getUserData();
        filterProperty.set( filter );
    };

ToggleButton tbShowAll = new ToggleButton("Show All");
tbShowAll.setSelected(true);
tbShowAll.setToggleGroup( filterTG );
tbShowAll.setOnAction(toggleHandler);
tbShowAll.setUserData( (Predicate<Player>) (Player p) -> true);
```

The ToggleButtons that filter a specific team are created at runtime based on the Players array. This Stream does the following.

1. Distill the list of Players down to a distinct list of team Strings
2. Create a ToggleButton for each team String
3. Set a Predicate for each ToggleButton to be used as a filter
4. Collect the ToggleButtons for addition into the HBox container

```
List<ToggleButton> tbs = Arrays.asList( players)
        .stream()
        .map( (p) -> p.getTeam() )
        .distinct()
        .map( (team) -> {
            ToggleButton tb = new ToggleButton( team );
            tb.setToggleGroup( filterTG );
            tb.setOnAction( toggleHandler );
            tb.setUserData( (Predicate<Player>) (Player p) -> team.equals(p.getTeam())
);
            return tb;
        })
        .collect(Collectors.toList());

hbox.getChildren().add( tbShowAll );
hbox.getChildren().addAll( tbs );
```

ListView

The next step creates the ListView and binds the ListView to the viewablePlayersProperty. This enables the ListView to receive updates based on the changing filter.

```
ListView<Player> lv = new ListView<>();
lv.itemsProperty().bind( viewablePlayersProperty );
```

The remainder of the program creates a Scene and shows the Stage. onShown loads the data set into the playersProperty and the viewablePlayersProperty lists. Although both lists are in sync in this partcular version of the program, if the stock filter is every different than "no filter", this code would not need to be modified.

```
vbox.getChildren().addAll( hbox, lv );

Scene scene = new Scene(vbox);

primaryStage.setScene( scene );
        primaryStage.setOnShown((evt) -> {
            playersProperty.get().addAll( players );
        });

primaryStage.show();
```

This article used binding to tie a list of viewable Player objects to a ListView. The viewable Players were updated when a ToggleButton is selected. The selection applied a filter to a full set of Players which was maintained separately as a FilteredList (thanks @kleopatra_jx). Binding was used to keep the UI in sync and to allow for a separation of concerns in the design.

Further Reading

To see how such a design would implement basic add and remove functionality, visit the following page https://courses.bekwam.net/public_tutorials/bkcourse_filterlistapp.php.

Complete Code

The code can be tested in a single .java file.

```
public class FilterListApp extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {

        //
        // Test data
        //
        Player[] players = {new Player("BOS", "David Ortiz"),
                            new Player("BOS", "Jackie Bradley Jr."),
                            new Player("BOS", "Xander Bogarts"),
                            new Player("BOS", "Mookie Betts"),
                            new Player("HOU", "Jose Altuve"),
                            new Player("HOU", "Will Harris"),
                            new Player("WSH", "Max Scherzer"),
                            new Player("WSH", "Bryce Harper"),
                            new Player("WSH", "Daniel Murphy"),
                            new Player("WSH", "Wilson Ramos") };

        //
        // Set up the model which is two lists of Players and a filter criteria
        //
        ReadOnlyObjectProperty<ObservableList<Player>> playersProperty =
                new SimpleObjectProperty<>(FXCollections.observableArrayList());

        ReadOnlyObjectProperty<FilteredList<Player>> viewablePlayersProperty =
                new SimpleObjectProperty<FilteredList<Player>>(
                        new FilteredList<>(playersProperty.get()
                                ));

        ObjectProperty<Predicate<? super Player>> filterProperty =
            viewablePlayersProperty.get().predicateProperty();


        //
        // Build the UI
```

```
        //
        VBox vbox = new VBox();
        vbox.setPadding( new Insets(10));
        vbox.setSpacing(4);

        HBox hbox = new HBox();
        hbox.setSpacing( 2 );

        ToggleGroup filterTG = new ToggleGroup();

        //
        // The toggleHandler action wills set the filter based on the TB selected
        //
        @SuppressWarnings("unchecked")
        EventHandler<ActionEvent> toggleHandler = (event) -> {
                ToggleButton tb = (ToggleButton)event.getSource();
                Predicate<Player> filter = (Predicate<Player>)tb.getUserData();
                filterProperty.set( filter );
            };

        ToggleButton tbShowAll = new ToggleButton("Show All");
        tbShowAll.setSelected(true);
        tbShowAll.setToggleGroup( filterTG );
        tbShowAll.setOnAction(toggleHandler);
        tbShowAll.setUserData( (Predicate<Player>) (Player p) -> true);

        //
        // Create a distinct list of teams from the Player objects, then create
        // ToggleButtons
        //
        List<ToggleButton> tbs = Arrays.asList( players)
                .stream()
                .map( (p) -> p.getTeam() )
                .distinct()
                .map( (team) -> {
                    ToggleButton tb = new ToggleButton( team );
                    tb.setToggleGroup( filterTG );
                    tb.setOnAction( toggleHandler );
                    tb.setUserData( (Predicate<Player>) (Player p) ->
team.equals(p.getTeam()) );
                    return tb;
                })
                .collect(Collectors.toList());

        hbox.getChildren().add( tbShowAll );
        hbox.getChildren().addAll( tbs );

        //
        // Create a ListView bound to the viewablePlayers property
        //
        ListView<Player> lv = new ListView<>();
```

```
        lv.itemsProperty().bind( viewablePlayersProperty );

        vbox.getChildren().addAll( hbox, lv );

        Scene scene = new Scene(vbox);

        primaryStage.setScene( scene );
        primaryStage.setOnShown((evt) -> {
            playersProperty.get().addAll( players );
        });

        primaryStage.show();

    }

    public static void main(String args[]) {
        launch(args);
    }

    static class Player {

        private final String team;
        private final String playerName;
        public Player(String team, String playerName) {
            this.team = team;
            this.playerName = playerName;
        }
        public String getTeam() {
            return team;
        }
        public String getPlayerName() {
            return playerName;
        }
        @Override
        public String toString() { return playerName + " (" + team + ")"; }
    }
}
```

Chapter 4. CSS

Placeholder whilst things get built…

Chapter 5. Performance

Placeholder whilst things get built…

Chapter 6. Application Structure

Placeholder whilst things get built…

Chapter 7. Best Practices

Placeholder whilst things get built...

1. Styleable Properties

7.1. 1. Styleable Properties

Author: Gerrit Grunwald

```
/* Member variables for StyleablePropertyFactory
 * and StyleableProperty
 */
private static final StyleablePropertyFactory<MY_CTRL> FACTORY =
    new StyleablePropertyFactory<>(Control.getClassCssMetaData());

private static final CssMetaData<MY_CTRL, Color> COLOR =
    FACTORY.createColorCssMetaData("-color", s -> s.color, Color.RED, false);
private final StyleableProperty<Color> color = new
SimpleStyleableObjectProperty<>(COLOR, this, "color");


// Getter, Setter and Property method
public Color getColor() {
    return this.color.getValue();
}

public void setColor(final Color color) {
    this.color.setValue(COLOR);
}

public ObjectProperty<Color> colorProperty() {
    return (ObjectProperty<Color>) this.color;
}


// Return CSS Metadata
public static List<CssMetaData<? extends Styleable, ?>> getClassCssMetaData() {
    return FACTORY.getCssMetaData();
}

@Override public List<CssMetaData<? extends Styleable, ?>> getControlCssMetaData() {
    return getClassCssMetaData();
}
```

7.2. 2. Tasks

Author: Carl Walker

This article demonstrates how to use a JavaFX Task to keep the UI responsive. It is imperative that any operation taking more than a few hundred milliseconds be executed on a separate Thread to avoid locking up the UI. A Task wraps up the sequence of steps in a long-running operation and provides callbacks for the possible outcomes.

The **Task** class also keeps the user aware of the operation through properties which can be bound to UI controls like ProgressBars and Labels. The binding dynamically updates the UI. These properties include

1. **runningProperty** - Whether or not the Task is running
2. **progressProperty** - The percent complete of an operation
3. **messageProperty** - Text describing a step in the operation

7.3. Demonstration

The following screenshots show the operation of an HTML retrieval application.

Entering a URL and pressing "Go" will start a JavaFX Task. When running, the Task will make an HBox visible that contains a ProgressBar and a Label. The ProgressBar and Label are updated throughout the operation.

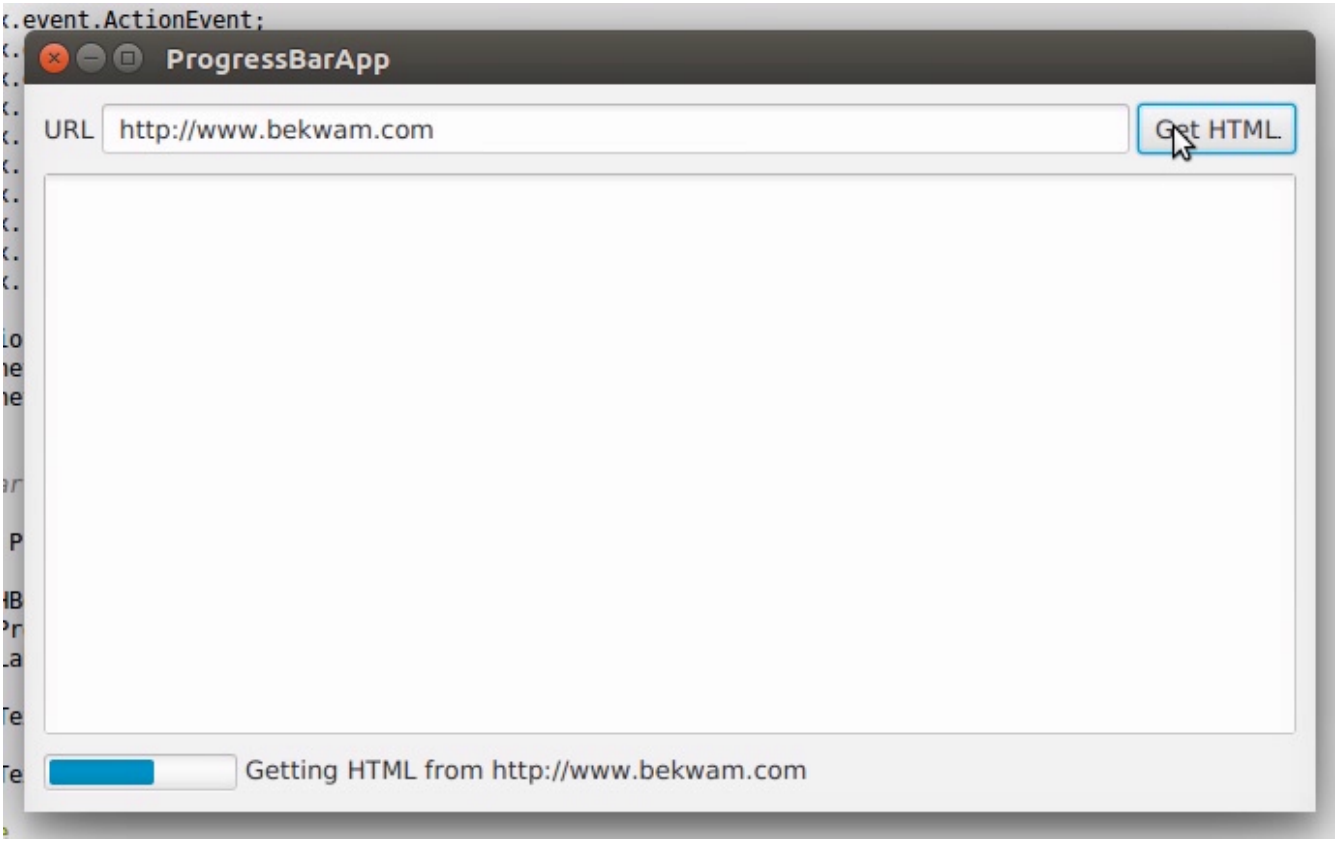

Figure 2. Screenshot of App Showing ProgressBar and Label

When the retrieval is finished, a succeeded() callback is invoked and the UI is updated. Note that the succeeded() callback takes place on the FX Thread, so it is safe to manipuplate controls.

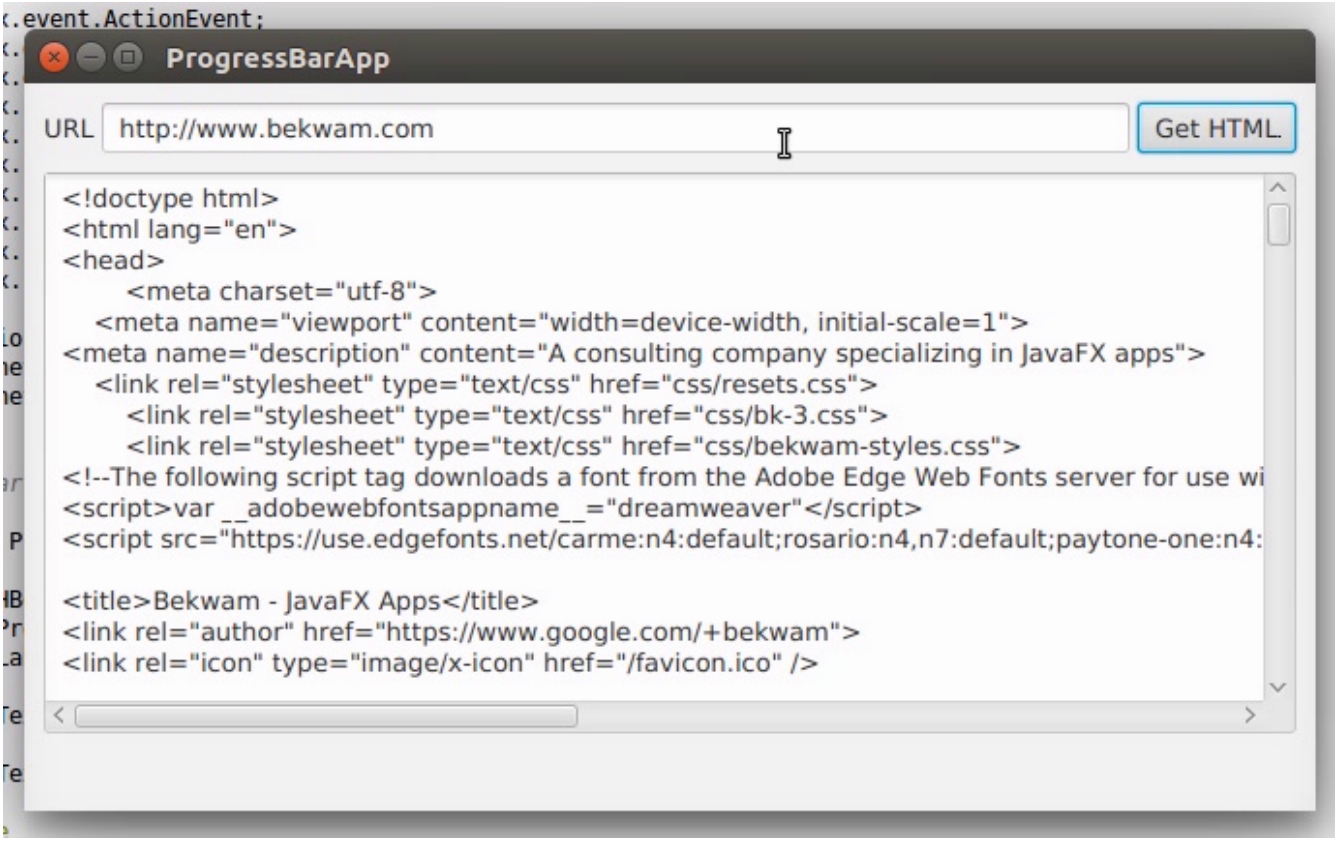

Figure 3. Screenshot of App Showing Successful Retrieval

If there was an error retrieving the HTML, a failed() callback is invoked and an error Alert is shown. failed() also takes place on the FX Thread. This screenshot shows invalid input. An "h" is used in the URL instead of the correct "http".

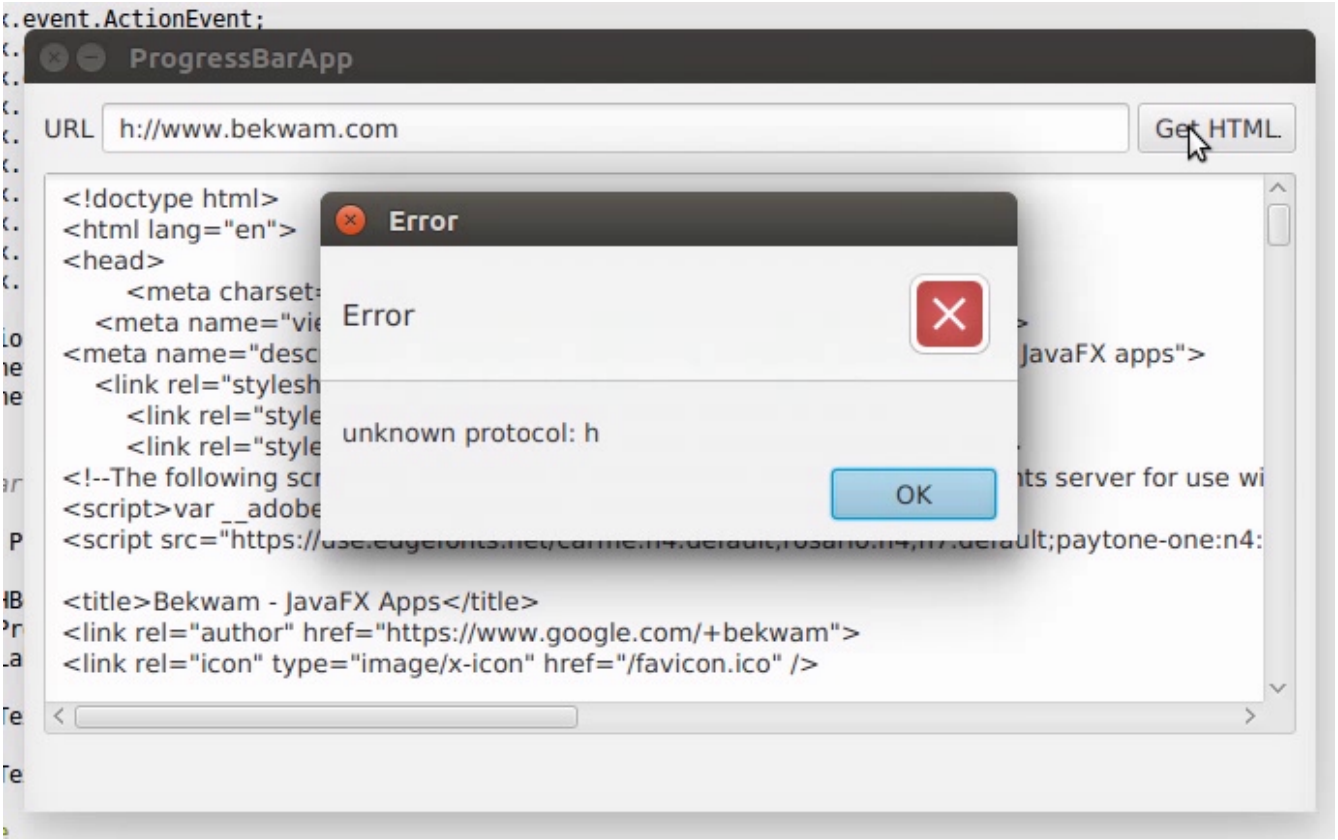

Figure 4. Screenshot of App Showing Failure

7.4. Code

An event handler is placed on the Get HTML Button which creates the Task. The entry point of the Task is the call() method which starts by calling updateMessage() and updateProgress(). These methods are executed on the FX Thread and will result in updates to any bound properties.

The program proceeds by issuing an HTTP GET using standard java.net classes. A String "retval" is built up from the retrieved characters. The message and progress properties are updated with more calls to updateMessage() and updateProgress(). The call() method ends with a return of the String containing the HTML text.

On a successful operation, the succeeded() callback is invoked. getValue() is a Task method that will return the value accrued in the Task (recall "retval"). The type of the value is what is provided in the generic argument, in this case "String". This could be a complex type like a domain object or a Collection. The succeeded() operation runs on the FX Thread, so the getValue() String is directly set on the TextArea.

If the operation failed, an Exception is thrown. The Exception is caught by the Task and converted to a failed() call. failed() is also FX Thread-safe and it displays an Alert.

```
String url = tfURL.getText();

Task<String> task = new Task<String>() {

    @Override
    protected String call() throws Exception {

        updateMessage("Getting HTML from " + url );
        updateProgress( 0.5d, 1.0d );

        HttpURLConnection c = null;
        InputStream is = null;
        String retval = "";

        try {

            c = (HttpURLConnection) new URL(url).openConnection();

            updateProgress( 0.6d, 1.0d );
            is = c.getInputStream();
            int ch;
            while( (ch=is.read()) != -1 ) {
                retval += (char)ch;
            }

        } finally {
            if( is != null ) {
                is.close();
            }
            if( c != null ) {
```

```
                c.disconnect();
            }
        }

        updateMessage("HTML retrieved");
        updateProgress( 1.0d, 1.0d );

        return retval;
    }

    @Override
    protected void succeeded() {
        contents.setText( getValue() );
    }

    @Override
    protected void failed() {
        Alert alert = new Alert(Alert.AlertType.ERROR, getException().getMessage() );
        alert.showAndWait();
    }
};
```

Notice that the Task does not update the ProgressBar and status Label directly. Instead, the Task makes safe calls to updateMessage() and updateProgress(). To update the UI, JavaFX binding is used in the following statements.

```
bottomControls.visibleProperty().bind( task.runningProperty() );
pb.progressProperty().bind( task.progressProperty() );
messageLabel.textProperty().bind( task.messageProperty() );
```

Task.runningProperty is a boolean that can be bound to the bottomControls HBox visibleProperty. Task.progressProperty is a double that can be bound to the ProgressBar progressProperty. Task.messageProperty is a String that can be bound to the status Label textProperty.

To run the Task, create a Thread providing the Task as a constructor argument and invoke start().

```
new Thread(task).start();
```

For any long-running operation — File IO, the Network — use a JavaFX Task to keep your application responsive. The JavaFX Task gives your application a consistent way of handling asynchronous operations and exposes several properties that can be used to eliminate boilerplate and programming logic.

7.5. Complete Code

The code can be tested in a single .java file.

```
```

```
public class ProgressBarApp extends Application {

    private HBox bottomControls;
    private ProgressBar pb;
    private Label messageLabel;

    private TextField tfURL;

    private TextArea contents;

    @Override
    public void start(Stage primaryStage) throws Exception {

        Parent p = createMainView();

        Scene scene = new Scene(p);

        primaryStage.setTitle("ProgressBarApp");
        primaryStage.setWidth( 667 );
        primaryStage.setHeight( 376 );
        primaryStage.setScene( scene );
        primaryStage.show();
    }

    private Parent createMainView() {

        VBox vbox = new VBox();
        vbox.setPadding( new Insets(10) );
        vbox.setSpacing( 10 );

        HBox topControls = new HBox();
        topControls.setAlignment(Pos.CENTER_LEFT);
        topControls.setSpacing( 4 );

        Label label = new Label("URL");
        tfURL = new TextField();
        HBox.setHgrow( tfURL, Priority.ALWAYS );
        Button btnGetHTML = new Button("Get HTML");
        btnGetHTML.setOnAction( this::getHTML );
        topControls.getChildren().addAll(label, tfURL, btnGetHTML);

        contents = new TextArea();
        VBox.setVgrow( contents, Priority.ALWAYS );

        bottomControls = new HBox();
        bottomControls.setVisible(false);
        bottomControls.setSpacing( 4 );
        HBox.setMargin( bottomControls, new Insets(4));

        pb = new ProgressBar();
        messageLabel = new Label("");
```

```
        bottomControls.getChildren().addAll(pb, messageLabel);

        vbox.getChildren().addAll(topControls, contents, bottomControls);

        return vbox;
    }

    public void getHTML(ActionEvent evt) {

        String url = tfURL.getText();

        Task<String> task = new Task<String>() {

            @Override
            protected String call() throws Exception {

                updateMessage("Getting HTML from " + url );
                updateProgress( 0.5d, 1.0d );

                HttpURLConnection c = null;
                InputStream is = null;
                String retval = "";

                try {

                    c = (HttpURLConnection) new URL(url).openConnection();

                    updateProgress( 0.6d, 1.0d );
                    is = c.getInputStream();
                    int ch;
                    while( (ch=is.read()) != -1 ) {
                        retval += (char)ch;
                    }

                } finally {
                    if( is != null ) {
                        is.close();
                    }
                    if( c != null ) {
                        c.disconnect();
                    }
                }

                updateMessage("HTML retrieved");
                updateProgress( 1.0d, 1.0d );

                return retval;
            }

            @Override
            protected void succeeded() {
```

```
                contents.setText( getValue() );
            }

            @Override
            protected void failed() {
                Alert alert = new Alert(Alert.AlertType.ERROR,
getException().getMessage() );
                alert.showAndWait();
            }
        };

        bottomControls.visibleProperty().bind( task.runningProperty() );
        pb.progressProperty().bind( task.progressProperty() );
        messageLabel.textProperty().bind( task.messageProperty() );

        new Thread(task).start();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Chapter 8. Contributing

Placeholder whilst things get built…

Chapter 9. License