JavaFX

> JavaFX is an open source, next generation client application platform for desktop, mobile and embedded systems based on JavaSE. It is a collaborative effort by many individuals and companies with the goal of producing a modern, efficient, and fully featured toolkit for developing rich client applications.

JavaFX is used on JabRef for the user interface.

Resources

- JavaFX Documentation project: Collected information on JavaFX in a central place
- curated list of awesome JavaFX frameworks, libraries, books and etc...
- FXTutorials A wide range of practical tutorials focusing on Java, JavaFX and FXGL
- ControlsFX amazing collection of controls
- CSS Reference
- mvvm framework
- Validation framework
- additional bindings or EasyBind
- Undo manager
- Docking manager or DockFX
- Kubed: data visualization (inspired by d3)
- Foojay Java and JavaFX tutorials

Resources of historical interest

- FXExperience JavaFX Links of the week

Architecture: Model - View - (Controller) - ViewModel (MV(C)VM)

The goal of the MVVM architecture is to separate the state/behavior from the appearance of the ui. This is archived by dividing JabRef into different layers, each having a clear responsibility.

- The *Model* contains the business logic and data structures. These aspects are again encapsulated in the *logic* and *model* package, respectively.
- The *View* controls the appearance and structure of the UI. It is usually defined in a *FXML* file.
- *View model* converts the data from logic and model in a form that is easily usable in the gui. Thus it controls the state of the View. Moreover, the ViewModel contains all the logic needed to change the current state of the UI or perform an action. These actions are usually passed down to the *logic* package, after some data validation. The important aspect is that the ViewModel

contains all the ui-related logic but does *not* have direct access to the controls defined in the View. Hence, the ViewModel can easily be tested by unit tests.
- The *Controller* initializes the view model and binds it to the view. In an ideal world all the binding would already be done directly in the FXML. But JavaFX's binding expressions are not yet powerful enough to accomplish this. It is important to keep in mind that the Controller should be as minimalistic as possible. Especially one should resist the temptation to validate inputs in the controller. The ViewModel should handle data validation! It is often convenient to load the FXML file directly from the controller.

The only class which access model and logic classes is the ViewModel. Controller and View have only access the ViewModel and never the backend. The ViewModel does not know the Controller or View.

More details about the MVVM pattern can be found in an article by Microsoft and in an article focusing on the implementation with JavaFX.

Example

VIEWMODEL

- The ViewModel should derive from `AbstractViewModel`

```java
public class MyDialogViewModel extends AbstractViewModel {
}
```

- Add a (readonly) property as a private field and generate the getters according to the JavaFX bean conventions:

```java
private final ReadOnlyStringWrapper heading = new ReadOnlyStringWrapper();

public ReadOnlyStringProperty headingProperty() {
    return heading.getReadOnlyProperty();
}

public String getHeading() {
    return heading.get();
}
```

- Create constructor which initializes the fields to their default values. Write tests to ensure that everything works as expected!

```java
public MyDialogViewModel(Dependency dependency) {
    this.dependency = Objects.requireNonNull(dependency);
    heading.set("Hello " + dependency.getUserName());
}
```

- Add methods which allow interaction. Again, don't forget to write tests!

```
public void shutdown() {
    heading.set("Goodbye!");
}
```

VIEW - CONTROLLER

- The “code-behind” part of the view, which binds the `View` to the `ViewModel`.
- The usual convention is that the controller ends on the suffix `*View`. Dialogs should derive from `BaseDialog`.

```
public class AboutDialogView extends BaseDialog<Void>
```

- You get access to nodes in the FXML file by declaring them with the `@FXML` annotation.

```
@FXML protected Button helloButton;
@FXML protected ImageView iconImage;
```

- Dependencies can easily be injected into the controller using the `@Inject` annotation.

```
@Inject private DialogService dialogService;
```

- It is convenient to load the FXML-view directly from the controller class.

  The FXML file is loaded using `ViewLoader` based on the name of the class passed to `view`. To make this convention-over-configuration approach work, both the FXML file and the View class should have the same name and should be located in the same package.

  Note that fields annotated with `@FXML` or `@Inject` only become accessible after `ViewLoader.load()` is called.

  a `View` class that loads the FXML file.

```
private Dependency dependency;

public AboutDialogView(Dependency dependency) {
        this.dependency = dependency;

        this.setTitle(Localization.lang("About JabRef"));

        ViewLoader.view(this)
                .load()
                .setAsDialogPane(this);
}
```

- Dialogs should use setResultConverter to convert the data entered in the dialog to the desired result. This conversion should be done by the view model and not the controller.

```
setResultConverter(button -> {
    if (button == ButtonType.OK) {
        return viewModel.getData();
    }
    return null;
});
```

- The initialize method may use data-binding to connect the ui-controls and the `ViewModel`. However, it is recommended to do as much binding as possible directly in the FXML-file.

```
@FXML
private void initialize() {
    viewModel = new AboutDialogViewModel(dialogService, dependency, ...);

    helloLabel.textProperty().bind(viewModel.helloMessageProperty());
}
```

- calling the view model:

```
@FXML
private void openJabrefWebsite() {
    viewModel.openJabrefWebsite();
}
```

VIEW - FXML

The view consists a FXML file `MyDialog.fxml` which defines the structure and the layout of the UI. Moreover, the FXML file may be accompanied by a style file that should have the same name as the FXML file but with a `css` ending, e.g., `MyDialog.css`. It is recommended to use a graphical design tools like SceneBuilder to edit the FXML file. The tool Scenic View is very helpful in debugging styling issues.

FXML

The following expressions can be used in FXML attributes, according to the official documentation

Type	Expression	Value point to	Remark
Location	`@image.png`	path relative to the current FXML file	
Resource	`%textToBeTranslated`	key in ResourceBundle	
Attribute variable	`$idOfControl` or `$variable`	named control or variable in controller (may be path in the namespace)	resolved only once at load time

Type	Expression	Value point to	Remark
Expression binding	`${expression}`	expression, for example `textField.text`	changes to source are propagated
Bidirectional expression binding	`#{expression}`	expression	changes are propagated in both directions (not yet implemented in JavaFX, see feature request)
Event handler	`#nameOfEventHandler`	name of the event handler method in the controller	
Constant	`<text><Strings fx:constant="MYSTRING"/></text>`	constant (here `MYSTRING` in the `Strings` class)	

JavaFX Radio Buttons Example

All radio buttons that should be grouped together need to have a ToggleGroup defined in the FXML code Example:

```
<VBox>
            <fx:define>
                <ToggleGroup fx:id="citeToggleGroup"/>
            </fx:define>
            <children>
                <RadioButton fx:id="inPar" minWidth="-Infinity" mnemonicParsing="false"
                             text="%Cite selected entries between parenthesis" toggleGroup="$citeToggleGroup"/>
                <RadioButton fx:id="inText" minWidth="-Infinity" mnemonicParsing="false"
                             text="%Cite selected entries with in-text citation" toggleGroup="$citeToggleGroup"/>
                <Label minWidth="-Infinity" text="%Extra information (e.g. page number)"/>
                <TextField fx:id="pageInfo"/>
            </children>
</VBox>
```

JavaFX Dialogs

All dialogs should be displayed to the user via `DialogService` interface methods. `DialogService` provides methods to display various dialogs (including custom ones) to the user. It also ensures the displayed dialog opens on the correct window via `initOwner()` (for cases where the user has multiple screens). The following code snippet demonstrates how a custom dialog is displayed to the user:

```
dialogService.showCustomDialog(new DocumentViewerView());
```

If an instance of `DialogService` is unavailable within current class/scope in which the dialog needs to be displayed, `DialogService` can be instantiated via the code snippet shown as follows:

```
DialogService dialogService = Injector.instantiateModelOrService(DialogService.class);
```

Properties and Bindings

JabRef makes heavy use of Properties and Bindings. These are wrappers around Observables. A good explanation on the concept can be found here: JavaFX Bindings and Properties

Features missing in JavaFX

- bidirectional binding in FXML, see official feature request