# Countodown

Ich habe mich bei Countodown für JavaFX entschieden, da ich mich primär auf Java konzentrieren wollte. JavaFX wird von Oracle/Sun als neuer Standard gehandelt und momentan von vielen Stellen gehyped.

Vor dem ersten Gate-meeting habe ich mir verschiedene Frontend Technologien angeschaut. JavaFX hat mir am meisten zugesagt. Dies liegt daran, dass ich für JavaFX keine neue Sprache lernen musste. JavaFX ist die neuste offizielle Frontend Technologie von Java und sehr zukunftsträchtig, weswegen es für mich von Vorteil ist sie zu beherrschen. Mir persönlich Desktop Applikationen immer besser gefallen wie Web Applikationen.

## Package Struktur.

Um JavaFX effektiv benutzen zu können, habe ich mir ein System überlegt, das prinzipiell auf dem Model-View-Controller Prinzip beruht. Ich nenne mein System Model-View-Controller + Loader. In diesem System ist in jedem Package eine View, diese View heißt genauso wie das Package (nur mit führendem Großbuchstaben). Das Paket beinhaltet außerdem alles was zur View dazugehört also Sprachdateien(.properties), Styling Dateien (.css), den Loader und den Controller. Hierbei gilt "convention over configuration". Dieses Package Prinzip ist eine Abwandlung von Adam Biens Prinzip. Mein MVCLoader beruht auf Adam Biens Klasse FXMLView die man hier einsehen kann.

https://github.com/AdamBien/afterburner.fx/blob/master/src/main/java/com/airhacks/afterburner/views/FXMLView.java

Ein Beispiel:

Das Package 'beispiel' enthält die Elemente:

beispiel

|

|- Beispiel.fxml

|- BeispielLoader.java

|- BeispielController.java

|- Beispiel.css (optional)

|- Beispiel\_en.properties (optional)

|- Beispiel\_xxx.properties (optional xxx durch ein Sprachkürzel ersetzen)

BeispielController implements Initializable

BeispielLoader extends MVCLoader

BeispielLoader überschreibt die abstrakte Methode 'protected abstract Initializable controllerInstance()' Diese Methode wird innerhalb des Loaders einmal aufgerufen und liefert eine Instanz von BeispielController zurück. Das erzeugte Controller Objekt wird dann an den FXMLLoader übergeben. Dieser befüllt dann alle mit @FXML annotierten Felder und ordnet die annotierten Methoden den im fxml definierten Listenern zu. Danach wird die Methode initialize auf [public interface Initializable{ public void initialize(URL location, ResourceBundle resources);}]. In dieser Methode kann man dann alle Initialisierungen vornehmen, für die man bereits gefüllte Felder benötigt. Dies hat den Nachteil dass viele Initialisierungen außerhalb des Konstruktors geschehen und deswegen viele Felder nicht final sein können.

Wenn man eine View öffnen möchte, instanziiert man den entsprechenden Loader, von diesem kann man sich dann View und Controller beziehen.

Wenn man nun dem Controller-Konstruktor Argumente übergeben will, so übergibt man sie dem Loader der sie dann wiederum an den Controller weitergibt.

Die übrige Paketstruktur sieht so aus, dass zusammengehörende Strukturen in Unterpaketen sind.

## MVCLoader

Um den MVCLoader gut erklären zu können, muss ich erste die Klasse LazyInitalizingFinalWrapper erklären. Dieser generische Wrapper hat einen Getter und einen Konstruktor, deswegen FinalWrapper. Diesem Konstruktor übergibt man jedoch nicht das Element, sondern einen Initializer. Dieser Initializer ist ein @FunctionalInterface, ich habe es geschrieben da ich zu diesem Zeitpunkt den Supplier noch nicht kannte. Ich habe den Initializer jedoch nicht durch einen Supplier ersetzt da es weder einen Verständnisvorteil noch einen Code-Vorteil bringt. Der Name Initializer mit der Methode initialize ist in diesem Kontext besser zu verstehen wie Supplier und get. Des Weiteren ist die Dokumentation von Supplier für diesen Fall eher verwirrend als Hilfreich.

https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html

der Initializer liefert eine Instanz des Elements und wird nur ein einziges Mal aufgerufen. Bei jedem Aufruf von LazyInitalizingFinalWrapper.get() wird überprüft, ob das Element schon initialisiert ist, wenn nicht (also beim ersten Aufruf) wird mit Initializer.initialize() das Objekt instanziiert. Deswegen LazyInitalizing.

Der MVCLoader hat 4 von diesen LazyInitalizingFinalWrapper, je einen für den FXMLLoader, die Ressourcen(Sprachdateien), das Parent-Objekt der View und den Controller. Alle vier erhalten als Argument ein Lambda (den Initializer). Diese Lambdas machen nichts anderes als eine private Methode des Loaders zurück zu geben. Ich habe die Erfahrung gemacht, dass es sich rentiert Lambdas so klein wie möglich zu halten. Am besten ist es, wenn man den gesamten Inhalt von Anfang an in eine private Hilfsmethode ausgliedert. Dann hat man deutlich weniger Probleme beim Debuggen, Erweitern und Verändern des Codes. Durch die Idee alles Lazy zu initialisieren, habe ich die Möglichkeit bei der Initialisierung abstrakte Methoden in den Unterklassen aufzurufen, für die es eine vollständig initialisierte Unterklasse bedarf.

Der MVCLoader setzt im FXMLLoader die entsprechenden Werte für den Controller, die FXML-Datei und die Ressourcen. Diese erhält er in dem er vom Namen der erbenden Klasse ausgehend einen SimpleName durch entfernen des Loader-Postfix generiert. Dieser Simplename wird nun zur Ressourcenfindung benutzt. Wenn man an den SimpleName ein ".fxml" anhängt erhält man die FXML-Datei, wenn man ein ".css" anhängt erhält man den die Style-Datei.

## Backend:

Ich habe das Backend mit einem Threadpool initialisiert. Dies habe ich gemacht damit sämtliche blockierende Aufrufe wie z.B. Jira Abfragen nicht das gesamte System lahm legen.

Abfragen an das Backend gehen immer über die Klasse EntrypointBackend. Beim Task laden liefert das Backend eine Queue (mit PoisonPill) zurück, die dann schrittweise von durch den Threadpool befüllt wird. Leider besitzt Java keine BlockingQueue die Schließbar ist. Dies hat zur Folge, dass man im Code unglaublich aufpassen muss, dass es zu keinen Deadlocks kommt. Die Lösung mit der PoisonPill ist zwar suboptimal, war jedoch weniger Arbeit als eine eigene CloseableBlockingQueue zu implementieren.

Zur Synchronisation im Backend habe ich mir die Klasse Occurrence geschrieben die sich an Pythons threading.event orientiert (https://docs.python.org/3/library/threading.html#event-object