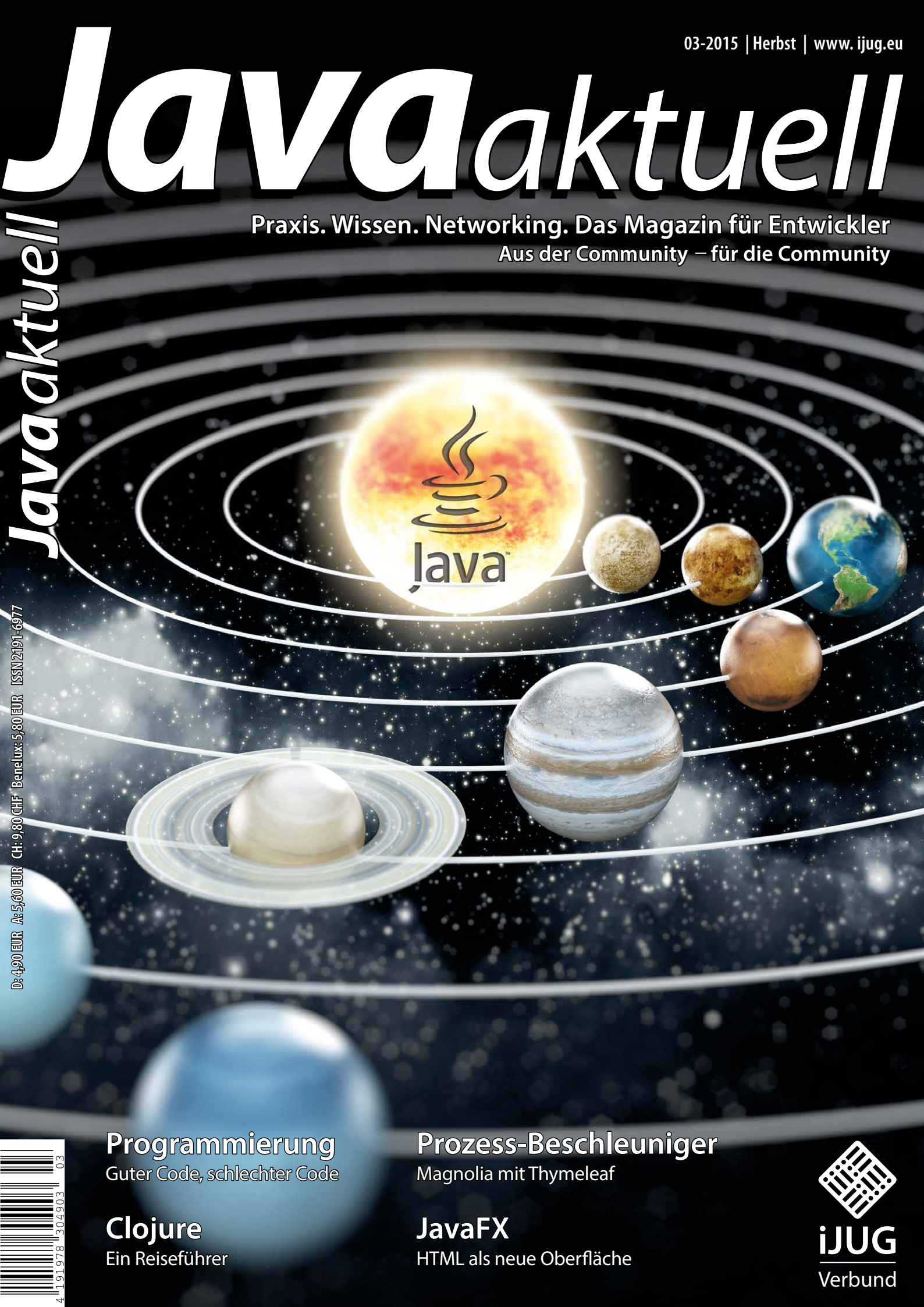
03-2015 | Herbst | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Aus der Community – für die Community

Java aktuell



D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

Programmierung
Guter Code, schlechter Code

Clojure
Ein Reiseführer

Prozess-Beschleuniger
Magnolia mit Thymeleaf

JavaFX
HTML als neue Oberfläche

iJUG Verbund

Kunstprojekt im JavaLand 2015

Seit Java 1.5 erlaubt die Java Virtual Machine die Registrierung sogenannter „Java-Agenten"

Bei Mgnolia arbeiten Web-Entwickler und CMS-Experten mit ein und demselben Quellcode

Ein Heim-Automatisierungs-Projekt

Collections" angelehnt. Unter anderem gibt es Methoden, die eine „read only"-Ansicht einer Collection zurückliefern:

- ObservableList<E> unmodifiableObservableList(ObservableList<E> list)
- ObservableMap<K,V> unmodifiableObservableMap(ObservableMap<K,V> map)
- ObservableSet<E> unmodifiableObservableSet(ObservableSet<E> set)

Hier gilt es aber seit Java 8 ganz genau aufzupassen, denn die zurückgelieferte „observable" Collection ist mit der ursprünglichen Collection nur durch einen „weak"-Listener verbunden. So kann es passieren, das Listener nicht aufgerufen werden, wenn die „read-only"-Collection aus dem Scope fällt *(siehe Listing 14)*.

Da der „CollectionHolder" immer eine neue Instanz der „read only"-Collection zurückliefert und diese im „UnmodifiableCollectionsExample" nicht gespeichert wird, kann die „read only"-Collection des Garbage Collector samt allen Listenern gelöscht werden. Damit wird der Listener nach dem nächsten GC-Durchlauf beim „collectionHolder.add(„foo")" nicht mehr aufgerufen. Dieses Verhalten ist neu seit Java 8, die Dokumentation wurde jedoch leider nicht entsprechend angepasst.

Fazit

Das Konzept von Property und Binding ist sehr mächtig, erfordert allerdings anfangs ein gewisses Umdenken. Die hier beschriebenen Pitfalls sind sicher kein Grund, JavaFX nicht zu verwenden, ganz im Gegenteil – JavaFX ist ein sehr gelungener Nachfolger von Swing.

Jan Zarnikov
jan.zarnikov@irian.at

Jan Zarnikov studierte Informatik an der Technischen Universität Wien. Seit dem Jahr 2010 arbeitet er als Software-Entwickler für Irian Solutions, wo er hauptsächlich mit Java beschäftigt ist. Derzeit ist er als Consultant bei einem JavaFX-Projekt für einen großen Kunden aus Deutschland tätig.

http://ja.ijug.eu/15/3/8

Asynchrone JavaFX-8-Applikationen mit JacpFX

Andy Moncsek, Trivadis AG

JacpFX ist eine JavaFX-basierte Rich Client Platform (RCP) mit Fokus auf einfacher Strukturierung von Applikationen, Kommunikation zwischen Komponenten und Unterstützung von asynchronen Prozessen im User Interface (UI).

UI-Toolkits gehören noch heute zu den wenigen Beispielen in der Software-Entwicklung, die überwiegend „Single-thread" sind, und das aus gutem Grund. „Multi-threading" steigert einerseits die Gefahr von Deadlocks und Race Conditions, andererseits steigt die Komplexität im Toolkit selbst. Das Abstract Window Toolkit (AWT) wurde beispielsweise ursprünglich als „multi-threaded" Java Library entworfen. Angesichts der Komplexität und der ersten Erfahrungen mit den Prototypen ist dieser Ansatz verworfen worden *[1]*. Stattdessen einigte man sich auf das bekannte Event-getriebene Modell im Event Dispatch Thread (EDT). Dieser „single-threaded"-Ansatz führt allerdings traditionell zu den üblichen Problemen der blockierenden UIs bei lang laufenden Prozessen wie zum Beispiel Datenbank-Anfragen.

JavaFX ist in dieser Hinsicht seinem Vorgänger Swing sehr ähnlich. Das Konzept des JavaFX Application Thread entspricht in etwa dem bekannten Event Dispatch Thread (EDT) aus Swing/AWT. Alles läuft auf einem Thread, und wenn dieser beschäftigt ist, blockiert die UI. Natürlich gab und gibt es Lösungen, um lang laufende Prozesse auszulagern. In JavaFX dürfen beispielsweise neue Knoten auch außerhalb des Application Thread erstellt oder manipuliert werden, solange sie nicht im JavaFX SceneGraph eingebunden sind *[2]*.

Mit „SwingWorker" etablierte sich in Swing ein Hilfsmittel, um Prozesse auszulagern und das Ergebnis dem EDT zu übergeben. JavaFX hat diesen Ansatz weiterverfolgt und bietet die Möglichkeit, Prozesse in „Tasks" und „Services" auszulagern. Ein „Task" *(siehe Listing 1)* ermöglicht die einma-

lige Ausführung eines lang laufenden Prozesses im Hintergrund und erlaubt es, dabei Status-Informationen an die GUI weiterzugeben. Ein „Service" hingegen verwendet einen „Task" und kann für wiederkehrende Prozesse eingesetzt werden.

Lang laufende Prozesse sind jedoch nur ein Aspekt, bei dem der Application Thread beachtet werden muss. Callback Events aus anderen Frameworks (etwa bei Verwendung von WebSocket, MQTT oder JMS) müssen Änderungen an der UI ebenfalls über diesen durchführen. Dieser Übergang zum Application Thread wurde in Swing mithilfe der Klasse „SwingUtilities" und der Methoden „invokeAndWait" und „invokeLater" umgesetzt. In JavaFX wird diese Rolle durch die Klasse „javafx.application.Platform" übernommen, allerdings bietet diese nur eine „runLater"-Methode an *(siehe Listing 2)*.

```
Task task = new Task<Void>() {
@Override public Void call() {
static final int max = 1000000;
for (int i=1; i<=max; i++) {
if (isCancelled()) {
break;
}
updateProgress(i, max);
}
return null;
}
};
ProgressBar bar = new ProgressBar();
bar.progressProperty().bind(task.progressProperty());
new Thread(task).start();
```

Listing 1

```
public class MqttGUI extends VBox implements MqttCallback {
private TextField textField;
@Override
public void messageArrived(String s, MqttMessage m) throws Exception {
// schlägt fehl, da nicht in Application Thread !
textField.setText(m.getPayload());
// richtig:
Platform.runLater(()->textField.setText(m.getPayload()));
}
}
```

Listing 2

JacpFX

In einem umfangreichen JavaFX-Projekt wird man sich früher oder später mit beiden Aspekten des JavaFX Application Thread auseinandersetzen müssen. Diese Problematik für Entwickler zu vereinfachen, ist eines der Ziele von JacpFX.

Generell folgt JacpFX einem Komponenten-orientierten Ansatz und unterscheidet dabei zwischen Workbench, Perspective, UI- und Non-UI-Komponenten *(siehe Abbildung 1)*.

Referenzen zwischen der Workbench, ihren Perspektiven und deren Komponenten werden per ID hergestellt. Dabei haben JacpFX-Komponenten nie direkte Abhängigkeiten zu anderen Komponenten und sind somit lose gekoppelt *(siehe Abbildung 2)*.

Das aus dieser Hierarchie resultierende ID-Schema („perspectiveId.componentId") dient gleichzeitig als Adressschema für den Message Bus in JacpFX. Dieser erlaubt es, Komponenten zu benachrichtigen, ohne eine direkte Abhängigkeit zwischen ihnen zu schaffen. Eine eintreffende Nachricht löst dabei folgenden Zwei-Phasen-Lifecycle in der Ziel-Komponente aus *(siehe Abbildung 3)*:

- In der Phase eins („Task") erfolgt die Ausführung der „handle(message)"-Methode der Ziel-Komponente im Worker Thread

```
mvn archetype:generate  -DarchetypeGroupId=org.jacpfx
-DarchetypeArtifactId=JacpFX-simple-quickstart -DarchetypeVersion=2.0.2
```

Listing 3

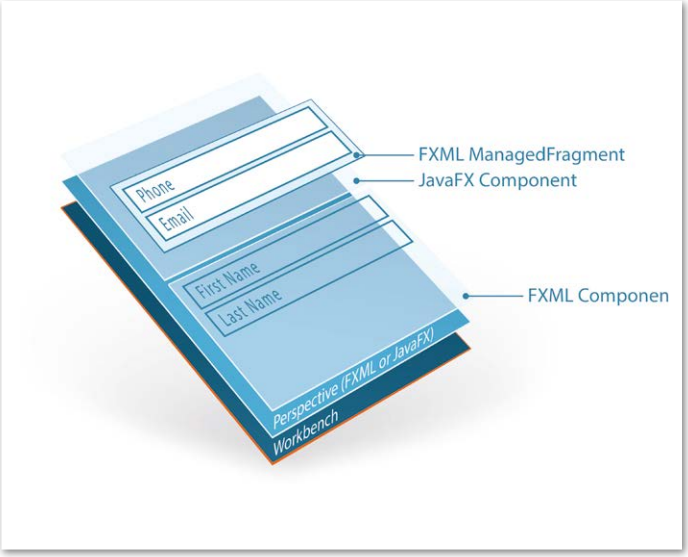


Abbildung 1: JacpFX-Komponenten-Struktur

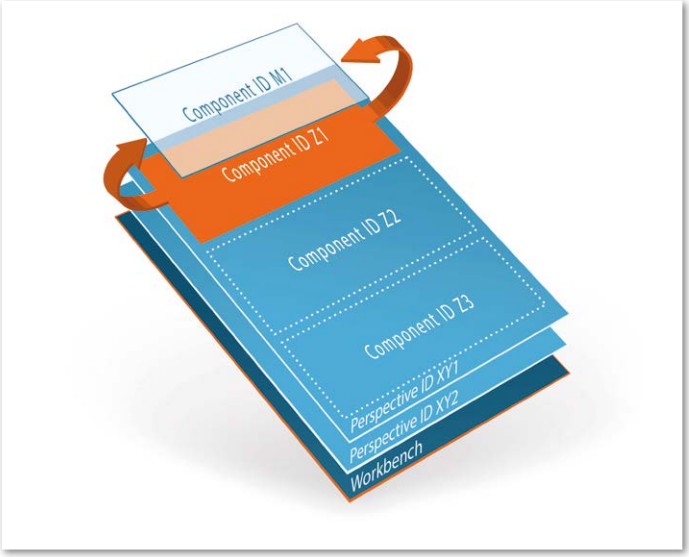


Abbildung 2: Lose Kopplung von JacpFX-Komponenten

- Der Rückgabewert wird anschließend in Phase 2 („State“) der „postHandle(value, message)“-Methode im Application Thread übergeben

Eine Komponenten-Instanz ist in JacpFX immer eindeutig pro Perspektive. Komponenten können jedoch durch mehrere Perspektiven referenziert oder zwischen ihnen verschoben werden. Mit diesem Ansatz soll die Wiederverwendung von Komponenten erleichtert werden, wobei das Adress-Schema eindeutig bleibt.

Die Verarbeitung der eintreffenden Nachrichten erfolgt innerhalb eines Worker Thread und ist immer sequenziell. Da Komponenten auch keine direkten Referenzen untereinander haben, ist eine Synchronisierung von Ressourcen nicht notwendig. Entwickler können den Komponenten-Lifecycle für nebenläufige Prozesse nutzen und müssen sich nicht mit Details der JavaFX-Implementierung und den Fallstricken von Multi Threading auseinandersetzen.

Es folgt ein Beispiel dafür, wie man mit JacpFX eine einfache Chat-Applikation erstellt und darin mit WebSocket-Callback-Events umgeht. Dieser Ansatz kann natürlich auch für klassische Datenbank-Abfragen oder andere lang laufende Prozesse, in denen die GUI nicht blockiert werden soll, übernommen werden *(siehe Abbildung 4)*.

```
public class ApplicationLauncher extends AFXSpringJavaConfigLauncher {
@Override
protected Class<? extends FXWorkbench> getWorkbenchClass() {
return JacpFXWorkbench.class;
}

@Override
protected String[] getBasePackages() {
return new String[]{"package.name"};
}

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

@Override
public void postInit(Stage stage) {
}

@Override
protected Class<?>[] getConfigClasses() {
return new Class<?>[]{BasicConfig.class};
}
}
```

Listing 4

```
@Workbench(id = "id1", name = "workbench",
perspectives = {
BasicConfig.PERSPECTIVE_ONE
})
public class JacpFXWorkbench implements FXWorkbench { … }
```

Listing 5

Erstellen einer JacpFX-Applikation

Zum Erstellen einer JacpFX-Applikation verwenden wir den „JacpFX-simple-quickstart“-Maven-Archetype *(siehe Listing 3)*. Dieser legt eine kleine Beispiel-Applikation an, die im Folgenden zu einer Chat-Applikation angepasst wird. Die wesentlichen Bestandteile der Beispiel-Applikation sind der „ApplicationLauncher“, die „JacpFX-Workbench“, eine FXML-Perspektive, zwei UI-Komponenten und eine Non-UI-Komponente *(siehe Abbildung 5)*.

Der „ApplicationLauncher“ enthält die „main“-Methode und die Angabe, welche Workbench-Klasse verwendet werden soll. Der eingesetzte „ApplicationLauncher“ ist vom Typ „AFXSpringJavaConfigLauncher“ und nutzt Spring 4.x zum Starten aller Komponenten *(siehe Listing 4)*.

Die „JacpFXWorkbench“ dient der Basis-Konfiguration der Applikation. Darin lassen

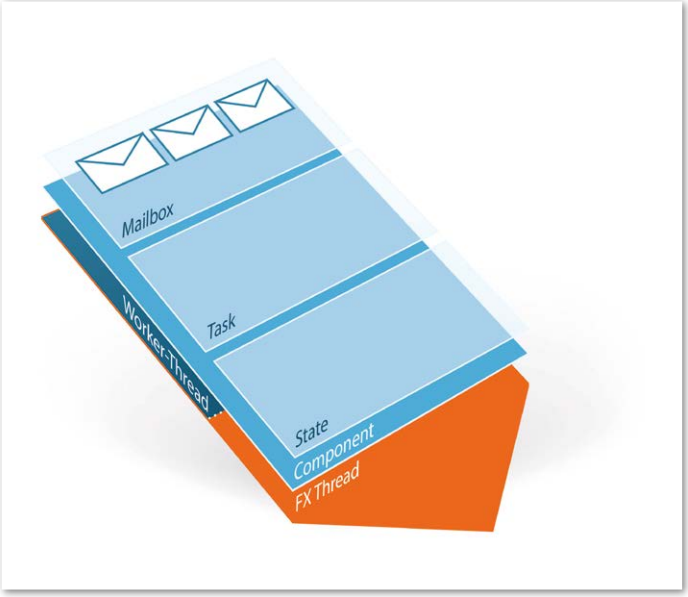


Abbildung 3: JacpFX-Komponenten-Lifecycle

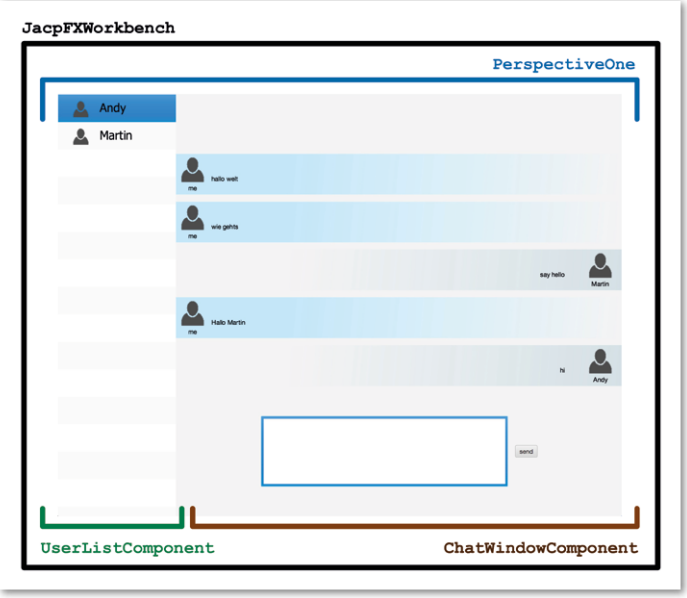


Abbildung 4: JacpFX-Chat-Applikation

sich die Fenstergröße bestimmen, Menüs und Toolbars aktivieren sowie alle Referenzen zu den Perspektiven in unserer Applikation definieren. Für das Beispiel benötigt man nur eine Perspektive und passt die Referenzen in der Workbench entsprechend an *(siehe Listing 5)*.

Die Klasse „PerspectiveOne" wird nun genauer betrachtet *(siehe Listing 6)*. Eine Perspektive ist in JacpFX eine Controller-Klasse mit einer View. Diese definiert das Layout für die aktuelle Ansicht in der Workbench. „PerspectiveOne" verwendet dabei eine FXML-View, alternativ kann die View auch programmatisch mit JavaFX erstellt sein.

In der „@Perspective"-Annotation sind unter anderem Referenzen zu den Komponenten und der verwendeten FXML-View der Perspektive definiert. Eine entscheidende Aufgabe der Perspektive ist die Definition von Platzhaltern, in denen die UI-Komponenten dargestellt werden. Innerhalb der „@PostConstruct"-Methode registriert man dafür im „PerspectiveLayout" zwei JavaFX-„GridPanes", die als UI-Container für die Komponenten zum Einsatz kommen *(siehe Listing 7)*.

Perspektiven dienen ausschließlich der Definition des Layouts und laufen immer im Application Thread, daher sollten hier keine blockierenden Prozesse ausgeführt werden. Nun geht es um die referenzierten JacpFX-Komponenten der Perspektive: Jede Komponente ist einem eigenen Worker Thread zugeordnet, der die eingehenden Nachrichten verarbeitet, die „handle(message)"-Methode ausführt und den Übergang zum Application Thread regelt. Dabei besteht eine UI-Komponente, ähnlich einer Perspektive, aus einer View und einer Controller-Klasse. Die View kann ebenso wie bei Perspektiven als FXML oder in JavaFX vorliegen.

Die Beispiel-Applikation soll auf der linken Seite eine Übersicht aller angemeldeten User und auf der rechten Seite das Chat-Fenster mit den Nachrichten anzeigen *(siehe Abbildung 4)*. Für die Chat-Applikation geht man von zwei WebSocket-Endpunkten aus: „ws://host/users" und „ws://host/chat".

Die „UserListComponent"-Komponente listet alle angemeldeten User im Chat auf. Der Controller dient dabei gleichzeitig als WebSocket Client, der bei eintreffender Nachricht eine Änderung in der GUI vornimmt. Die „ChatWindowComponent"-Komponente hingegen lagert die WebSocket-Implementierung exemplarisch in eine JacpFX-Non-UI-Komponente aus und kommuniziert mit ihr über den JacpFX Message Bus.

Listing 8 stellt einen Ausschnitt der „UserListComponent"-Komponente dar. Die „handle(message)"-Methode wird hier nicht verwendet. Es können aber nach Belieben lang laufende Prozesse ausgeführt werden. Währenddessen wird die Komponenten-View nicht blockiert und die UI ist für den Anwender weiterhin nutzbar. In der „postHandle(value,message)"-Methode fügen wir neue Chat-User ein, die von der WebSocket-@OnMessage-Methode empfangen und an den JacpFX Message Bus weitergeleitet werden *(siehe Listing 9)*.

Für die Chat-Ansicht verwendet man zusätzlich eine Non-UI-Komponente und trennt damit die Service-Implementierung von der UI-Logik. Durch die lose Kopplung aller Komponenten in JacpFX lässt sich so jederzeit die Implementierung schnell auswechseln und beispielsweise durch eine SocketJS-Implementierung austauschen. Aus Sicht der UI-Komponente werden somit Nachrichten mit einer weiteren JacpFX-Komponente aus-

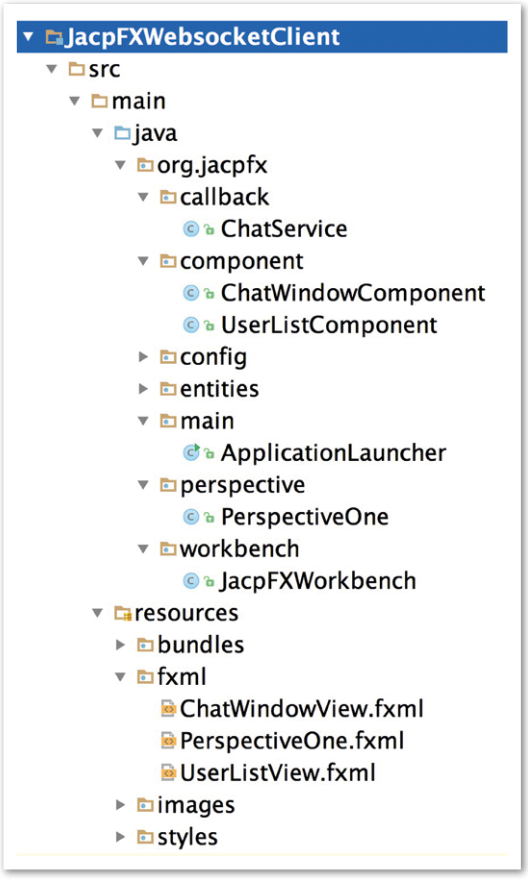


Abbildung 5: JacpFX-WebsocketClient-Struktur

```
@Perspective(id = BasicConfig.PERSPECTIVE_ONE,
name = "contactPerspective",
components = {
BasicConfig.COMPONENT_LEFT,
BasicConfig.COMPONENT_RIGHT,
BasicConfig.STATEFUL_CALLBACK},
viewLocation = "/fxml/PerspectiveOne.fxml",
resourceBundleLocation = "bundles.languageBundle")
public class PerspectiveOne implements FXPerspective {
@FXML
private GridPane left;
@FXML
private GridPane main;
@Resource
public Context context;

@Override
public void handlePerspective(Message<Event, Object> m, PerspectiveLayout l) {
        …
}

@PostConstruct
public void onStartPerspective(final PerspectiveLayout pl, FXComponentLayout l,
ResourceBundle b) {
pl.registerTargetLayoutComponent(BasicConfig.LEFT, left);
pl.registerTargetLayoutComponent(BasicConfig.RIGHT, main);
}

@PreDestroy
public void onTearDownPerspective(final FXComponentLayout layout) {…}
}
```

Listing 6

```
pl.registerTargetLayoutComponent(BasicConfig.LEFT, left);
pl.registerTargetLayoutComponent(BasicConfig.RIGHT, main);
```

Listing 7

getauscht, ohne weitere Details der Service-Implementierung zu kennen.

Listing 10 zeigt den JacpFX-Chat-Service „ChatService". Diese Non-UI-Komponente läuft immer innerhalb eines Worker Thread und kann somit die UI nicht blockieren. Die „handle(message)"-Methode verarbeitet Chat-Nachrichten, die von der UI-Komponente gesendet werden. Diese Nachrichten werden mithilfe der WebSocket-Session an den Chat-Endpunkt geleitet, der sie an den eigentlichen Adressaten schickt.

Der Rückgabewert der „handle(message)"-Methode wird normalerweise an den Sender zurückgeschickt. In diesem Fall verzichtet man auf eine Antwort und gibt stattdessen „Null" zurück, wodurch eine Rückantwort unterdrückt wird. Die @OnMessage-Methode empfängt eingehende Nachrichten vom WebSocket-Endpunkt und leitet sie an die „ChatWindowComponent"-Komponente weiter.

Listing 11 zeigt einen Ausschnitt der „ChatWindowComponent"-Komponente. In der „postHandle(value,message)"-Methode wird die eingehende Nachricht zum Chat-Fenster hinzugefügt. Während der gesamten Kommunikation braucht man den Application Thread nicht zu beachten. Durch das Routing über den JacpFX Message Bus entfällt ein manuelles Eingreifen zum Synchronisieren der Threads.

Fazit

Mit dieser Beispiel-Applikation ist eine asynchrone, Nachrichten-basierte JavaFX-Applikation entstanden, ohne sich mit den Details des JavaFX Application Threads auseinandersetzen zu müssen. Dieser Ansatz kann ebenso verwendet werden, um klassische Datenbank-Anfragen oder andere lang laufende Prozesse asynchron zu verarbeiten, ohne die UI zu blockieren. Der Nutzer kann beispielsweise eine Suchanfrage starten und währenddessen in einer anderen Perspektive weiterarbeiten.

Dem Entwickler hilft JacpFX, die Applikation zu strukturieren, die Kommunikation zwischen Komponenten zu erleichtern und die Handhabung von asynchronen Prozessen innerhalb der Anwendung zu vereinfachen. Aktuell liegt JacpFX in Version 2.0.1 vor, ein Release 2.1 ist für Ende Sommer 2015 geplant.

Weitere Informationen

[1] https://weblogs.java.net/blog/kgh/archive/2004/10/multithreaded_t.html
[2] http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-architecture.htm
[3] http://jacpfx.org
[4] http://jacpfx.org/documentation_main.html
[5] https://github.com/JacpFX/JacpFX-demos/tree/master/JacpFXWebsocketClient

Alle weiteren Listings finden Sie online unter:
www.ijug.eu/go/java_aktuell/201503/listings

```
@DeclarativeView(id = BasicConfig.COMPONENT_LEFT,
name = "SimpleView",
viewLocation = "/fxml/UserListView.fxml",
active = true,
resourceBundleLocation = "bundles.languageBundle",
initialTargetLayoutId = BasicConfig.LEFT)
@ClientEndpoint(decoders = UserDecoder.class)
public class UserListComponent implements FXComponent {
@Resource
private Context context;
@FXML
private ListView userList;

private ObservableList<User> users;
private static final String WEBSOCKET_URL="ws://…";

@Override
public Node handle(final Message<Event, Object> message) {
// runs in worker thread
return null;
}

@Override
public Node postHandle(Node resultNode, Message<Event, Object> m) {
// runs in FX application thread
if(m.isMessageBodyTypeOf(User.class)) {
users.add(m.getTypedMessageBody(User.class));
}
return null;
}

@PostConstruct
public void onPostConstructComponent(FXComponentLayout layout, final Re-
sourceBundle b) {
            …
connectToEndpoint();
}

private void connectToEndpoint() {
…
WebSocketContainer container = ContainerProvider.getWebSocketContainer();
container.connectToServer(this, URI.create(ComponentLeft.WEBSOCKET_URL));

…
     }

@OnMessage
public void onNewUser(User user) {
this.context.send(user);
}
}
```

Listing 8

Andy Moncsek
andy.moncsek@trivadis.com
@AndyAHCP

Andy Moncsek arbeitet als Senior Consultant im Bereich „Application Development" bei der Trivadis AG in Zürich. Neben seinem Fokus auf Java-Enterprise-Applikationen und -Architekturen ist er als Disziplinen-Manager „Java Application Development" tätig. In seiner Freizeit leitet er das Open-Source-Projekt „JacpFX" und ist in vielen Bereichen des Java-Ökosystems unterwegs.

http://ja.ijug.eu/15/3/9