JavaFX: Maven und Gradle

Ein schneller, kurzer Überblick über Maven und Gradle. Dies kann keine Einführung in der Tiefe sein – dazu wäre jeweils eine ganze Blog-Serie notwendig und da ist es sinnvoller, die offizielle Dokumentation zu verwenden!

## Teile der JavaFX Serie (aktualisiert)

* [JavaFX 01: Übersicht / Planung](https://blog.kneitzel.de/2020/06/29/javafx-uebersicht-planung/) (YouTube)
* [JavaFX 02: Installation](https://blog.kneitzel.de/2020/06/24/javafx-uebersicht/) (YouTube)
* [JavaFX 03: Maven & Gradle](https://blog.kneitzel.de/2020/11/02/javafx-maven-und-gradle/) (YouTube)
* [JavaFX 04: IntelliJ](https://blog.kneitzel.de/2020/11/02/javafx-intellij/) (YouTube)

### Teile der JavaFX Serie (alt / geplant)

* [JavaFX 05: Scene Builder](https://blog.kneitzel.de/2020/06/26/javafx-scene-builder/)

* [JavaFX --: Erläuterung der Applikation](https://blog.kneitzel.de/2020/06/26/javafx-erlaeuterungen-der-applikation/)
* [JavaFX 06: Model / View / Controller](https://blog.kneitzel.de/2020/06/26/javafx-model-view-controller/)
* [JavaFX 07: MVVM](https://blog.kneitzel.de/2020/06/29/javafx-mvvm/)
* JavaFX: Komplette MVVM Applikation (geplant)
* JavaFX: Layouts (geplant)
* JavaFX: Formulare einfach erstellen (geplant)
* JavaFX: Multi Plattform (geplant)

## Code der JavaFX Serie

Der Source Code sowie alle Dokumente finden sich auf GitHub: <https://github.com/kneitzel/blog-javafx-series>

## Was sind Maven und Gradle?

Bei Maven und Gradle handelt es sich um Tools, mit denen ein Projekt verwaltet und gebaut werden kann. Ein wichtiges Feature ist dabei die Verwaltung von Abhängigkeiten, die vom Tool automatisch in der gewünschten Version von einem Repository heruntergeladen und bereitgestellt werden.

Dies verringert die Aufwände bezüglich Einrichtung eines Arbeitsplatzes enorm, da die Anzahl der Abhängigkeiten, die installiert werden müssen, minimiert werden.

### Vorteile

Der große Vorteil dieser Build Tools ist, dass diese von den gängigen Entwicklungsumgebungen (IntelliJ, Eclipse, Netbeans) unterstützt werden. Damit ist ein Projekt unabhängig von einer Entwicklungsumgebung und jeder Entwickler kann mit der Umgebung arbeiten, die er bevorzugt.

## Maven

### Übersicht

Maven entstand im Jahr 2003 (Version 1.x) / 2004 (Version 1.0) und kann somit auf eine recht lange Entwicklungszeit zurückblicken.

Das Tool zeichnet sich dadurch aus, dass es ein festen Lebenszyklus vorgibt, der feste Phasen umfasst in dem viele Dinge standardmäßig verankert sind, so dass oft nur wenig Anpassungen notwendig sind.

Folgende Phasen sind in Maven vorgesehen:

#### archetype

Dient u.a. der Erstellung von Templates, Abhängigkeiten werden aufgelöst und ggf. herunter geladen.

##### validate

Validierung des Projektes. Ist die Struktur des Projekts ok?

###### compile

Übersetzen des Codes.

###### test

Ausführung der automatischen Tests mit Hilfe geeigneter Testframeworks.

###### package

Die Ergebnisse werden zusammengepackt. Häufig handelt es sich um eine jar Datei.

###### integration-test

Das erstellte Paket wird an einen anderen Ort kopiert und dort getestet. Dies kann z.B. ein Anwendungsserver sein.

###### verify

Das erstellte Softwarepaket wird überprüft, um sicher zu stellen, dass die Struktur korrekt ist.

###### install

Das Softwarepaket wird im lokalen Maven Repository bereitgestellt.

###### deploy

Das Softwarepaket wird im entfernten Maven Repository bereitgestellt.

Dies kann durch Plugins und Maven Archetypen erweitert und verändert werden, so dass z.B. Docker Container gebaut und verteilt werden können.

### Wrapper

Zu Maven gibt es einen Wrapper, der es ermöglicht, dass Maven nicht mehr systemweit auf einem System installiert werden muss. Da unterschiedliche Projekte ggf. unterschiedliche, nicht zueinander kompatible, Versionen von Maven nutzen wollen, ist eine systemweite Installation nicht zielführend.

Stattdessen gibt es ein Script welches die notwendige Version von Maven herunter lädt und innerhalb des Projektverzeichnisses entpackt.

|  |  |
| --- | --- |
| Erzeugung des Wrappers | Es ist möglich, mittels maven ein Wrapper für ein Projekt zu erzeugen. Dazu geht man in das Verzeichnis des Projekts und ruft auf:  „mvn -N io.takari:maven:wrapper“. |

### Installation

Auch wenn eine Installation nicht notwendig ist, ist dies natürlich jederzeit möglich. Das Tool kann einfach als ZIP File heruntergeladen werden. Eine komplexe Installation ist nicht notwendig. Es reicht, das ZIP an einer beliebigen Stelle zu entpacken.

### Dateien und Verzeichnisse

###### pom.xml

Das Projekt wird in einer pom.xml (POM = Project Object Model) beschrieben, welches sich in dem Projektverzeichnis befindet.

###### src

Das src Verzeichnis enthält alle Sourcen der Projekts. Diese sind in der Regel aufgeteilt in mehrere Unterordner: src/main für das eigentliche Projekt und src/test für die automatisierten Tests, die nicht mit ausgeliefert werden. Java Quellcode kommt in ein weiteres Unterverzeichnis java und die Ressourcen in das weitere Unterverzeichnis resources.

###### target

Alle Dateien, die erzeugt werden, werden in dem Verzeichnis target erzeugt. Je nach Art der erzeugten Datei, werden diese in einem entsprechenden Unterverzeichnis angelegt, z.B. target/classes für die class Dateien.

###### mvnw / mvnw.cmd / .mvn

Scripte und Verzeichnis des Wrappers. Das Skript mvnw kann statt mvn bei Maven Aufrufen verwendet werden. Die Dateien des Wrappers incl. Konfiguration finden sich in .mvn/wrapper.

### Beispiel einer pom.xml

|  |
| --- |
| **03 helloword – maven/pom.xml** |
| <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>de.kneitzel</groupId>  <artifactId>hellofx</artifactId>  <packaging>jar</packaging>  <version>1.0-SNAPSHOT</version>  <name>hellofx</name>  <url>https://blog.kneitzel.de</url>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <maven.compiler.source>11</maven.compiler.source>  <maven.compiler.target>11</maven.compiler.target>  </properties>  <dependencies>  <dependency>  <groupId>org.openjfx</groupId>  <artifactId>javafx-controls</artifactId>  <version>11.0.2</version>  </dependency>  <dependency>  <groupId>org.openjfx</groupId>  <artifactId>javafx-fxml</artifactId>  <version>11.0.2</version>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.openjfx</groupId>  <artifactId>javafx-maven-plugin</artifactId>  <version>0.0.3</version>  <configuration>  <mainClass>helloworld.HelloWorld</mainClass>  </configuration>  </plugin>  </plugins>  </build>  </project> |

Diese Datei kann am Anfang immer so übernommen werden. Folgende Werte sind dabei anzupassen:

###### groupId / artifactId / version / name / url

Die ersten drei Werte bilden die id des Projekts, welches eindeutig sein sollte. Die übrigen sollten aber auch angepasst werden.

###### maven.compiler.source / maven.compiler.target

Dies kennzeichnet die Java Version, die wir nutzen wollen. In dem Beispiel ist dies Java 11.

###### dependencies

Die Abhängigkeiten sind hier einzutragen. Die vorhandenen Libraries auf maven central kann man sich auf <https://mvnrepository.com/> ansehen.

###### build/plugins

Für JavaFX Applikationen wird das javafx-maven-plugin benötigt. Dieses muss eingebunden werden. Weiterhin wird die Klasse mit der main Methode angegeben, damit das plugin die Applikation auch starten kann.

### Ausprobieren

Wir können maven einfach direkt mit dem Beispielprojekt ausprobieren. Dazu wechseln wir in das Verzeichnis „03 helloworld – maven“ und geben da einen der folgenden Befehle ein:

* mvnw clean  
  Dies bereinigt das Projekt.
* mvnw package  
  Dies übersetzt das Projekt und baut ein jar File. Dies kann in dem Verzeichnis Target in den diversen Verzeichnissen etwas nachvollzogen werden.
* mvnw javafx:run  
  Dies baut das Projekt und führt die Anwendung danach aus.

## Gradle

### Übersicht

Gradle ist etwas jünger und entstand 2007. Es hat diverse Dinge von Maven übernommen wie z.B. die Verzeichnisstruktur und das automatische Laden von Abhängigkeiten aus Maven Repositories.

So wie Maven auch, gibt es weitgehende Standards, so dass die Projekt-Konfiguration sehr klein ausfallen kann. Für ein Java Projekt könnte eine Zeile „apply plugin: 'java'“ ausreichend sein.

Statt einer XML Datei hat Gradle eine eigene Domänenspezifische Sprache für die Konfiguration.

### Wrapper

So wie bei Maven existiert auch für Gradle ein Wrapper.

|  |  |
| --- | --- |
| Erzeugung des Wrappers | Es ist möglich, mittels gradle ein Wrapper für ein Projekt zu erzeugen. Dazu geht man in das Verzeichnis des Projekts und ruft auf: „gradle wrapper“. |

### Installation

Auch wenn eine Installation nicht notwendig ist, ist dies natürlich jederzeit möglich. Das Tool kann einfach als ZIP File heruntergeladen werden. Eine komplexe Installation ist nicht notwendig. Es reicht, das ZIP an einer beliebigen Stelle zu entpacken.

### Dateien und Verzeichnisse

###### build.gradle / settings.gradle / gradle.properties

Das Projekt wird in einer build.gradle beschrieben, welches sich in dem Projektverzeichnis befindet. Die anderen Dateien sind optional.

###### src

Das src Verzeichnis enthält alle Sourcen der Projekts. Diese sind in der Regel aufgeteilt in mehrere Unterordner: src/main für das eigentliche Projekt und src/test für die automatisierten Tests, die nicht mit ausgeliefert werden. Java Quellcode kommt in ein weiteres Unterverzeichnis java und die Ressourcen in das weitere Unterverzeichnis resources.

###### gradlew / gradlew.bat / gradle

Scripte und Verzeichnis des Wrappers. Das Skript gradlew kann statt gradle bei Gradle Aufrufen verwendet werden. Die Dateien des Wrappers incl. Konfiguration finden sich in gradle/wrapper.

### Beispiel einer build.gradle

|  |
| --- |
| **03 helloword – gradle/build.gradle** |
| plugins {  id 'application'  id 'org.openjfx.javafxplugin' version '0.0.8'  }  javafx {  version = "11.0.2"  modules = [ 'javafx.controls', 'javafx.fxml' ]  }  group 'de.kneitzel'  version '1.0-SNAPSHOT'  mainClassName = 'helloworld.HelloWorld'  repositories {  mavenCentral()  }  dependencies {  compile group: 'org.openjfx', name: 'javafx-controls', version: '11.0.2'  compile group: 'org.openjfx', name: 'javafx-fxml', version: '11.0.2'  } |

Ähnlich wie in Maven findet sich die id der Projekts (group/version – name ist in settings.gradle) und es wird ein javafx Plugin eingebunden.

Die Klasse mit der main Methode wird benannt und die Dependencies aufgelistet.

Zusätzlich ist es aber noch notwendig, die javafx Version sowie die verwendeten Module anzugeben.

In diesem Dokument sind keine Plattformen bei den Abhängigkeiten angegeben worden. Im git finden sich in der Datei die openjfx Abhängigkeiten für Windows, Mac und Linux. Dies kann problematisch sein, sobald weitere Plattformspezifischen Aufgaben ausgeführt werden sollen. So aber kein JLink und ähnliches ausgeführt werden sollen, kann man so auf einer Plattform alle Dateien bekommen, die für die großen Plattformen benötigt werden.

Die Applikation kann gebaut werden mit

gradlew assemble

und das Ergebnis findet sich dann in build/distributions als tar und zip Datei. Dieses Archiv kann auf den drei Plattformen entpackt werden. Im entpackten Verzeichnis befinden sich Startscripte, die unter Windows, Mac und Linux funktionieren. Voraussetzung ist aber eine Vorhandene Java Installation in einer kompatiblen Version.

### JLink und Gradle

Mittels JLink kann man sich ein image bauen, welches neben der eigentlichen Applikation auch eine abgespeckte JRE enthält mit allen benötigten Modulen.

Dabei kann aber nur eine Plattform gebaut werden und das ist in der Regel genau die Plattform, auf der man arbeitet. Eine Cross-Plattform Erstellung des Images ist möglich, aber das wird hier nicht behandelt.

|  |
| --- |
| **03 helloword – gradle with jlink/build.gradle** |
| plugins {  id 'application'  id 'org.openjfx.javafxplugin' version '0.0.9'  id 'org.beryx.jlink' version '2.22.0'  }  javafx {  version = "11"  modules = [ 'javafx.controls', 'javafx.fxml' ]  }  group 'de.kneitzel'  version '1.0-SNAPSHOT'  tasks.withType(JavaCompile) { options.encoding = 'UTF-8' }  modularity.disableEffectiveArgumentsAdjustment()  sourceCompatibility = 1.11  repositories {  mavenCentral()  }  jlink {  launcher {  name = 'helloworld'  }  }  mainClassName = 'HelloWorld.main/helloworld.HelloWorld'  dependencies {  compile group: 'org.openjfx', name: 'javafx-controls', version: '11.0.2'  compile group: 'org.openjfx', name: 'javafx-fxml', version: '11.0.2'  } |

Das Image wird gebaut über den Befehl

gradlew jlink

und das Resultat findet sich dann im Verzeichnis build/image welches alle benötigten Dateien enthält.

Die eigene Applikation kann mit den Start Scripten in image/bin gestartet werden. In unserem Beispiel ist dies helloworld.

## Links

* Maven <https://maven.apache.org/>
* Gradle <https://gradle.org/>
* Maven Repository <https://mvnrepository.com/>

## Änderungsnachweis

* 2020-10-30 Erste Veröffentlichung