101 Technologien - Teil 1

zwiebelbrot

2019-03-15 10:00

Die Auswahl der Technologien ist für manch Entwickler eine Glaubensangelegenheit. Für andere wieder ist die Suche nach der 'perfekten' Technologie so was wie der heilige Gral der Softwareentwicklung. Diese Blogreihe beschäftigt sich anhand einer Beispielanwendung mit vergleichsweise 'modernen' Technologien, die sich aktuell großer Beliebtheit erfreuen.

"Have you ever seen so many puppies?"

Die Analogie zu dem Klassiker vom Regisseur Wolfgang Reitherman ist nicht so weit hergeholt. Es gibt eine Menge Technologien. Und es werden immer mehr! Front-End-Entwickler können ein Lied von singen, denn gerade in diesem Bereich tummeln sich schier unendlich viele Frameworks. Doch auch im Back-End bewegt sich einiges. In dieser Blogreihe zeige ich dir anhand einer Beispielanwendung fünf spannende, neue Technologien. Für jede Technologie ist ein eigener Artikel geplant. Hier eine kurze Übersicht der Themen und damit verbundenen Technologien, die dich erwarten:

- **1. Artikel:** *Initialisierung des Projektes + Programmiersprache*
- **2. Artikel:** *Benutzerverwaltung*
- **3. Artikel:** *Datenbank*
- **4. Artikel:** *API*
- **5. Artikel:** *Front-End*

Die Beispielanwendung in dieser Blogreihe sollte dir sehr bekannt vorkommen. Es ist eine soziale Plattform, in der Nutzer Kurznachrichten posten und andere Nutzer folgen können. Zunächst müssen grobe Anforderungen an die Beispielanwendungen festgelegt werden. Der Nutzer soll…

- … sich bei dem Dienst anmelden / abmelden.
- … Kurznachrichten posten.
- … eine Timeline mit allen Kurznachrichten von gefolgten Nutzern und einem selber sehen.
- … eine Liste mit allen verfügbaren Nutzern sehen.
- … in der Liste der verfügbaren Nutzern Nutzer folgen / entfolgen.

Lasst die Hunde los!

Ohne ein Fundament steht selbst das stabilste Haus nicht, weswegen die Beispielanwendung auf zwei bereits etablierte Technologien aufbaut, die nicht zu den fünf modernen Technologien zählen: Spring Boot und Gradle. Mit diesen zwei Technologien wird dir eine Menge Arbeit erspart. Du kannst mit denen ohne viel Konfigurationen eine eigenständige Anwendung entwickeln und ausliefern. Mit dem Spring Initializr kannst du ein solches Projekt initialisieren. Die Parameter kannst du wie in der folgenden Abbildung auswählen. Abhängigkeiten fügen wir später händisch hinzu, weswegen du hier noch keine hinzufügen musst. *Group* und *Artifact* kannst du beim Standard belassen.

Wenn du aus Gewohnheit Java ausgewählt hast, keine Sorge: Mir ist das auch passiert. Doch die Beispielanwendung soll in der Programmiersprache Kotlin entwickelt werden. Kotlin ist somit die erste von den fünf modernen Technologien, die in dieser Blogreihe behandelt werden. Bevor du die Besonderheiten von Kotlin kennenlernst, muss die Beispielanwendung noch weiter konfiguriert werden.

Die Beispielanwendung soll aus den folgenden zwei Modulen bestehen:

- **api** - *Beinhaltet die main-Funktion und die Schnittstelle zur Außenwelt*
- **core** - *Beinhaltet die Konfiguration, Services, Entitäten, usw.*

Module kannst du beispielsweise in IntelliJ mittels eines Rechtsklick und dann auf *New -> Module* hinzufügen. Beim Hinzufügen eines Moduls musst du darauf achten, dass du Kotlin anstatt Java auswählst. Außerdem musst du die ArtifactId, also in unserem fall api und core, vergeben. Erstellst du über IntelliJ ein Modul, wird auch automatisch eine build.gradle-Datei erzeugt. Den Inhalt der build.gradle-Dateien kannst du zunächst in beiden Modulen löschen. Außerdem kann es passieren, dass IntelliJ vergisst, die Module der settings.gradle-Datei hinzuzufügen. Sollte dies zutreffen, musst die Module nachträglich einfügen. In jedem Fall sollte die settings.gradle-Datei wie folgt aussehen:

```
rootProject.name = 'demo'
include 'api'
include 'core'
```

Aktuell befindet sich bis auf die build.gradle-Dateien nichts in den Modulen. Um die gewohnte Verzeichnisstruktur *src/…* zu erhalten, kannst du das src-Verzeichnis aus dem Stammverzeichnis kopieren und in beide Module einfügen. Anschließend muss noch in beiden Modulen ein Package mit dem jeweiligen Modulnamen, also api und core, erzeugt werden. Das src-Verzeichnis aus dem Stammverzeichnis kannst du löschen.

Im Stammverzeichnis selbst befindet sich ebenfalls eine build.gradle-Datei. In dieser kannst du die beiden Module konfigurieren. Dafür musst du alle Zeilen bis auf das Closure `buildscript { ... }` in das Closure `subproject { ... }` einfügen. Konfigurationen im Closure `subproject { ... }` gelten für

alle Module im Projekt. Damit die Referenzen zwischen den Modulen und Abhängigkeiten korrekt aufgelöst werden, musst du die Zeile `apply plugin: 'org.springframework.boot'` entfernen und die folgenden Zeilen hinzugefügen. Ansonsten kann Gradle die Abhängigkeiten nicht korrekt auflösen und dir es hagelt *ReferenceUnkown*-Errors.

```
dependencyManagement {
    imports {
        mavenBom("org.springframework.boot:spring-boot-dependencies:${springBootVersion}")
    }
}
```

Als Kotlin-Version kannst du die Version 1.3.21 verwenden. Da Gradle aktuell noch ein wenig wegen der Kotlin-Version durcheinander gerät, musst du zusätzlich im Stammverzeichnis eine gradle.properties-Datei erzeugen und mit folgendem Inhalt füllen:

```
kotlin.version=1.3.21
```

Zum Schluss musst du noch die build.gradle-Datei im Modul api wie folgt anpassen:

```
// U.a. für den Run-Task
apply plugin: 'application'

// Main-Klasse
// Für eine Kotlin-Klasse wird ein "Kt" angehängt
mainClassName = 'com.example.demo.api.DemoApplicationKt'

// Das Modul "core" wird als Abhängigkeit hinzugefügt
dependencies {
    implementation project(':core')
}
```

Die Konfiguration ist damit fertig. Zumindest fast… es fehlen noch die Abhängigkeiten. Doch der spannungshalber erfährst du die Abhängigkeiten, und somit die konkreten Technologien, erst in den jeweiligen Artikeln.

Sitz, Pongo!

Beginnen wir mit einem Vergleich: Google Trends

Da kommt Kotlin aber nicht gut weg. Zugegebenermaßen, der Vergleich hinkt ein wenig. C und Java existieren bereits seit Jahrzenten. Gerade Java hat sich mit der Zeit vermutlich zu der ersten Sprache eines herangehenden Entwicklers hervorgetan. Ein Vergleich mit dem direkten Konkurrenten Groovy sieht dann schon etwas anders aus. Kotlin konnte gerade in den Jahren 2017 und 2018 an Interesse gewinnen.

Kotlin wird von JetBrains entwickelt und ist eine statisch typisierte Programmiersprache. Das heißt, es werden bereits während der Kompilierung die Typen aller Variablen festgelegt. Mit ihr entwickelte Programme werden in Bytecode übersetzt und dieser ist in der *Java Virtual Machine* (JVM) lauffähig. Neben der Möglichkeit der funktionalen Programmierung bietet die Sprache Null-Sicherheit, Typinferenz, Datenklassen und vieles mehr.

Programmieren mit Kotlin

In diesem Blogartikel kann ich dir leider nur einen kurzen Einblick in Kotlin bieten. Das Hauptaugenmerk dieser Blogreihe soll auf das Präsentieren von modernen Technologien liegen. Hast du Interesse an Kotlin, kann ich dir eines der zahlreichen Bücher empfehlen. Oder du schaust dich in der ausführlichen Referenz von Kotlin um. Es gibt viele spannende Möglichkeiten von Kotlin zu entdecken. Und vielleicht erscheinen noch weitere Blogartikel, die sich tiefergehend mit der Materie auseinandersetzen ;). Im Folgenden zeige ich dir anhand kleinerer Beispiele einige der grundlegenden Funktionen, die für die Beispielanwendung benötigt werden. Die Beispiele gehen nicht konkret auf die Beispielanwendung ein, dienen jedoch als Grundkenntnisse für die nächsten Blogartikel.

Semikolon

Studenten, die die Entwicklung mit Java beginnen, verzweifeln regelrecht an ihnen: die geliebten Semikolons. Doch ich muss leider enttäuschen. In Kotlin existieren Semikolons weiterhin. Doch dank der Semikolon-Inferenz, also dem Herleiten von Semikolons anhand definierter Regeln, nimmt uns Kotlin eine potentiell nervige Fehlerquelle ab. Dadurch muss am Ende eines Ausdrucks kein Semikolon stehen. Zeilenumbrüche gewinnen durch die Semikolon-Inferenz jedoch an Bedeutung. Folgendes Listing ist somit syntaktisch korrekter Kotlin-Quellcode.

```
System.out.print("Hello, ") // Es wird kein Semikolon benötigt
System.out.print("World") // Durch den Zeilenumbruch werden die Ausdrücke getrennt
```

Variable

In Kotlin wird, ähnlich zu JavaScript, eine Variable mit den Schlüsselwörtern `val` und `var` angekündigt. Eine veränderbare Variable wird mit dem Schlüsselwort `var` angekündigt. Veränderbare Variablen können beliebig oft neue Werte zugewiesen werden. Eine schreibgeschützte Variable wird hingegen mit dem Schlüsselwort `val` angekündigt. Bei einer schreibgeschützten Variable ist nach einer Zuweisung Schluss.

```
val user1 = User("Sebastian")
var user2 = User("Sebastian")
```

```
user1 = User("Max") // Error
user2 = User("Max") // Funktioniert
```

Aber Vorsicht, schreibgeschützt ist nicht gleich *nicht veränderbar*. Dazu ein kleines Beispiel:

```
class User {
    val birthYear: Int

    val age: Int {
        Calendar.getInstance().get(Calendar.YEAR) - birthYear;
    }
}
```

Die schreibgeschützt Variable `age` wird dynamisch berechnet. Vergeht ein Jahr, so wird sich auch der Inhalt von age ändern. Zwar kann beispielsweise nicht explizit mit `age = 18` ein neuer Wert der Variable zugewiesen werden, das bedeutet im Umkehrschluss aber nicht, dass der Wert der Variable sich nie verändert.

Typinferenz

Neue Programmiersprachen versuchen den Entwicklern möglichst viel Schreibarbeit zu sparen, ohne die Aussagekraft des Quellcodes zu verändern. Eine Möglichkeit kennst du bereits: Die Semikolon-Inferenz. In Kotlin kommt dazu noch die Typinferenz, also das Herleiten von Typen für Variablen anhand definierter Regeln. Wie in dem folgenden Beispiel musst du also nicht den Typ der Variable angeben, wenn dieser klar ersichtlich ist.

```
var username = "Sebastian"
username = 123 // Error
```

So ist die Variable username vom Typ String. Der Compiler bricht mit einem Error beim Zuweisen von 123 ab, da der Integer-Wert 123 nicht dem Typ String entspricht. Denn Kotlin ist weiterhin eine statisch typisierte Programmiersprache, in der alle Typen bei der Kompilierung festgelegt werden.

Interne Sichtbarkeit

Neben den Sichtbarkeitsmodifizierern `private`, `public` und `protected`, die in ähnlicher Form auch beispielsweise in Java vorkommen, existiert in Kotlin auch noch die Sichtbarkeit `internal`. Elemente mit der Sichtbarkeit `internal` sind nur im eigenen Modul sichtbar, wie das folgende Beispiel verdeutlicht:

```
// Folgende Zeile befindet sich im Modul api
val user = User("Sebastian") // Compiler bricht mit Error ab

// Folgende Zeile befindet sich im Modul core
internal data class User(val username: String)
```

```
...
val user = User("Sebastian") // Funktioniert
```

Datenklasse

Klassen, die nur dazu dienen, Daten zu halten, werden Datenklassen genannt. Innerhalb von Kotlin können diese mit `data` markiert werden. Der Compiler generiert für solche Klassen Funktionen wie beispielsweise equals und hashCode. Alle Variablen innerhalb des Konstruktors werden dabei berücksichtigt.

```
// Die Datenklasse User
data class User(
    val username: String
) {
    var password: String = "123"
}

// Nur Variablen innerhalb des Konstruktors werden berücksichtigt
val user1 = User("Sebastian")
val user2 = User("Sebastian")
user1.password = "12345"
user2.password = "98765"
System.out.println(user1 == user2) // Ruft die Methode equals von der Datenklasse User auf,
```

Null-Sicherheit

Jeder kennt sie, jeder liebt sie: Die *null pointer exception*. Tony Hoare nennt es seinen *billion-dollar mistake*, die Erfindung der Null-Referenz. Kotlin möchte damit brechen und bietet daher eine Null-Sicherheit.

```
var username: String = "Sebastian"
username = null // Compiler bricht mit Error ab
```

Der Compiler bricht mit einem Error ab. Damit entfällt eine Fehlerquelle, denn so kann einer Variable nicht *ausversehen* null zugewiesen werden. Dennoch existieren Anwendungsfälle, in denen null erforderlich ist. Beispielsweise wenn du Daten aus einer Datenbank abfragst, die eventuell nicht vorhanden sind. Zur Verdeutlichung ein kurzer Ausflug in die Welt von Java.

```
// Null wird zurückgegeben, da Username nicht gefunden
private String username = userRepository.getUsernameById(1L);

...

if (username.length == 0) { // NullPointerException
    ...
}
```

Der Username wird nicht gefunden und somit wird `null` von der Methode `getUsernameById` zurückgegeben, weswegen beim Abfragen der Länge des Usernamen eine NullPointerException geworfen wird. Java bietet seit der Version 8 hierfür die Klasse Optional. Kotlin löst die Null-Sicherheit auf einen anderen Weg. Damit ein Typ einen null-Wert annehmen kann, wird dieser explizit mit `?` gekennzeichnet.

```
var username: String? = "Sebastian"
username = null // Funktioniert
```

Variablen mit nullable-Typen können nicht ohne Weiteres verwendet werden.

```
System.out.println(username.length) // Compiler bricht mit Error ab
```

Es existieren vier Varianten, um diese Variablen zu verwenden. In der Beispielanwendung werden der sichere Aufruf (Safe Call) und der Elvis-Operator verwendet. Für Entwickler, die ohne NullPointerExceptions nicht leben können, existiert zusätzlich der `!!`-Operator. Dadurch wird jeder Wert in einen null-Typ konvertiert. Ansonsten besteht für nicht veränderbare Variablen die Möglichkeit, mittels einer if-Verzweigung auf den null-Typ zu überprüfen (Funktioniert nicht für veränderbare Variablen, da ansonsten nach der Überprüfung der Variable null zugewiesen werden könnte).

Der **Safe Call** kann mit dem `.?`-Operatoren ausgeführt werden. Dabei wird überprüft, ob der Wert nicht null ist. Wenn dies zutrifft, wird der Wert verwendet, ansonsten wird null zurückgegeben ohne eine *null pointer exception* zu werfen.

```
val username: String? = null
System.out.println(username?.length) // Funktioniert, es wird null ausgegeben
```

Mit dem Elvis-Operator **`?:`** kannst du (ähnlich einer if-Verzweigung) überprüfen, ob der Wert vom null-Typ ist und eine bestimmte Anweisung ausführen, ansonsten eine andere.

```
val username: String? = null
val length = username?.length ?: throw NotFoundException() // Exception wird geworfen

val username: String? = "123"
val length = username?.length ?: throw NotFoundException() // Der Wert 3 wird zugewiesen
```

Fazit

Die vorgestellten Funktionen von Kotlin existieren in ähnlicher Form auch in anderen Programmiersprachen, denn Technologien bauen in der Regel aufeinander auf. Java wurde auch maßgeblich von Smalltalk beeinflusst. Doch neue Technologien ermöglichen es, immer schneller & effizienter sichere Anwendungen zu entwickeln. *Boilerplate code* wird vermieden und die Entwickler können sich auf das Wesentliche konzentrieren.

In diesem ersten Artikel der Blogreihe “101 Technologien” hast du das Projekt für die Beispielanwendung initialisiert. Außerdem hast du die grundlegenden Funktionen von Kotlin kennengelernt, die für die nächsten Blogartikel benötigt werden. Im nächsten Blogartikel richtest du die Benutzerverwaltung für die Beispielanwendung ein. Dazu stelle ich dir eine weitere moderne Technologie vor. Bis demnächst!