1 Einleitung

1.1 Warum Spring Boot?

Das Spring Framework und Spring Boot sind aus dem Java-Ökosystem nicht mehr wegzudenken. Ohne Zweifel führt Spring Boot das Feld der Java-basierten Application-Frameworks an. In einer Umfrage von Jetbrains aus dem Jahr 2022 gaben 67 % der befragten Java-Entwickler an, Spring Boot zu nutzen, und 41 % nutzten Spring MVC (beide Technologien werden in diesem Buch behandelt)[1]. Das Framework auf Platz 3 kommt nur auf einen einstelligen Prozentwert, und 23 % der Befragten gaben an, gar kein Web-Framework zu nutzen, was die Prominenz von Spring nur verdeutlicht.

Softwareunternehmen wie Netflix und Atlassian setzen auf Spring Boot, um die Softwareentwicklung über Hunderte von Teams hinweg zu standardisieren!

Die führende Rolle von Spring Boot kommt nicht von ungefähr. Spring Boot basiert auf dem Spring Framework, das seit Anfang der 2000er Jahre kontinuierlich und sorgsam weiterentwickelt wird. Die Idee, ein auf Spring basierendes »Boot«-Framework zu entwickeln, das es uns vereinfacht, eine Spring-Anwendung zu konfigurieren, zu integrieren und zu betreiben, war nur eine Frage der Zeit.

Spring Boot macht uns als Entwickler produktiver, weil es uns viel Arbeit abnimmt. Es setzt dabei stark auf Konventionen, sodass wir nicht jede Kleinigkeit selbst konfigurieren müssen und sofort loslegen können. Diese große Stärke von Spring Boot wird aber teilweise auch als Hindernis empfunden, wenn man mit diesen Konventionen nicht vertraut ist oder nicht weiß, wo man sie nachschlagen kann.

Genau dieses Hindernis möchten wir mit diesem Buch aus dem Weg räumen.

---

1. *https://www.jetbrains.com/lp/devecosystem-2022/java/#what-web-frameworks-do-you-use-if-any-*

1.2 Für wen ist dieses Buch?

Dieses Buch richtet sich sowohl an Java-Entwickler, die noch keine Erfahrung mit Spring oder Spring Boot haben, als auch an solche, die bereits Spring Boot 2 gearbeitet haben und einige Neuerungen in Spring Boot 3 kennenlernen oder ihr allgemeines Spring-Boot-Wissen auffrischen möchten.

Es bietet sich an, grundlegende Java-Kenntnisse (oder allgemeine Kenntnisse in objektorientierter Programmierung) mitzubringen, andernfalls sind die Konzepte und Codebeispiele in diesem Buch schwer nachzuvollziehen.

Da Spring Boot eine Fülle von Integrationen mit anderen Systemen anbietet, von denen einige in diesem Buch behandelt werden, eignet sich das Buch ebenfalls als Nachschlagewerk für Architekten, die Spring und Spring Boot für den Einsatz in einem Projekt evaluieren möchten.

Auch wenn dieses Buch die wichtigsten Features und Konzepte erläutert, erhebt es keinen Anspruch auf Vollständigkeit. Das Spring-Ökosystem ist mittlerweile so umfangreich, dass ein Buch mit einer vollständigen Abdeckung unpraktikabel wäre und vermutlich nur als Monitorstütze verwendet werden würde. Daher konzentrieren wir uns in diesem Buch auf die wichtigen Aspekte und verweisen an vielen Stellen für bestimmte Details auf die sehr gute Referenzdokumentation der jeweiligen Spring-Projekte. Es geht nicht darum, jeden Konfigurationsparameter auswendig zu lernen, sondern darum, die Konzepte zu verstehen, um sie dann bei Bedarf anwenden (und nachschlagen) zu können.

1.3 Aufbau des Buchs

Dieses Buch ist in drei Teile aufgeteilt.

In Teil I werden die **Grundlagen** von Spring und Spring Boot beschrieben. Dieser Teil ist insbesondere für Leser geeignet, die sich noch nicht viel mit Spring und Spring Boot beschäftigt haben. Leser, die bereits Erfahrung mit beiden Frameworks gesammelt haben, mögen Teil I gegebenenfalls ganz oder teilweise überspringen. Die Konzepte aus Teil I sind für das Verständnis vieler Kapitel aus den Teilen II und III erforderlich.

Teil II ist gedacht als Nachschlagewerk für bestimmte **Anwendungsfälle**. Jedes Kapitel in Teil II betrachtet einen konkreten Anwendungsfall (zum Beispiel die Entwicklung einer REST-API), mit dem man jederzeit in einem Softwareprojekt konfrontiert sein kann. Die Kapitel geben jeweils einen Überblick darüber, wie man den entsprechenden Anwendungsfall mit Spring Boot lösen kann.

Die Kapitel in Teil III sind **Referenzkapitel**, die im Detail auf Spring-Boot-Features eingehen. Hier lernt der Leser alles, was notwendig ist, um diese Features in einer Anwendung zu nutzen.

Mit Ausnahme von Teil I bauen die Kapitel nicht aufeinander auf, sodass die Kapitel größtenteils unabhängig voneinander gelesen werden können. Wir laden dazu ein, der Neugier zu folgen und diejenigen Kapitel zuerst zu lesen, die gerade das Interesse wecken oder im aktuellen Projekt von Bedeutung sind. Sollte ein Kapitel doch mal ein Konzept aus einem anderen Kapitel erfordern, findet sich ein Verweis im Text.

1.4 Codebeispiele

Ein Codebeispiel sagt mehr als tausend Worte. Das haben wir uns mit diesem Buch zu Herzen genommen. Die Konzepte von Spring und Spring Boot werden mit weit über 300 Codebeispielen veranschaulicht, um sie in Aktion zu sehen.

Noch viel mehr als ein Codebeispiel sagt jedoch eine lauffähige Anwendung. Deshalb haben wir für viele der Kapitel jeweils eine Spring-Boot-Anwendung entwickelt und in einem GitHub-Repository zur Verfügung gestellt. Wir empfehlen, dieses Repository zu klonen und mit den darin enthaltenen Spring-Boot-Anwendungen beim Lesen der Kapitel herumzuspielen. Am besten lernt man doch beim Spielen mit Code!

Das Repository ist unter der URL *https://github.com/dxfrontiers/spring-boot-3-buch* frei verfügbar.

Damit alle Codebeispiele aus diesem Buch und dem Repository lauffähig sind, empfehlen wir die Installation von JDK 21 und die Nutzung der aktuellen Versionen von Spring Boot 3.x und Spring 6.x. Die meisten Beispiele werden auch mit etwas älteren Versionen funktionieren, aber einige Features sind erst mit späteren Versionen verfügbar.

2 Hallo, Spring Boot

Im ersten Kapitel bauen wir direkt eine lauffähige »Hello World«-Anwendung, um mit einem Erfolgserlebnis zu starten.

2.1 Einleitung

Dieses Buch ist ein Praxisbuch. In fast allen Kapiteln werden wir anhand von Codebeispielen die verschiedenen Aspekte einer Spring-Boot-Anwendung diskutieren. In den meisten Kapiteln sind diese Codebeispiele einer Beispielanwendung entnommen, deren Code auf GitHub öffentlich verfügbar ist. Wir laden dazu ein, den Beispielcode beim Lesen des Buches in der lokalen IDE durchzustöbern.

In diesem Sinne beginnen wir dieses Buch auch mit einem praktischen »Hello World«-Beispiel. Leser, die bereits Erfahrung mit Spring Boot haben, können dieses Kapitel guten Gewissens überspringen. Wenn nicht, laden wir dazu ein, den Beispielen in diesem Kapitel zu folgen und eine erste lauffähige Spring-Boot-Anwendung zu bauen.

In diesem Kapitel werden wir nicht jedes kleine Detail erläutern. Aber keine Angst, wir verweisen stets auf spätere Kapitel, in denen die Details vertieft werden.

2.2 JDK installieren

Um eine Spring-Boot-Anwendung zu entwickeln, müssen wir ein JDK (Java Development Kit) auf unserem Rechner installieren. Spring Boot 3 benötigt ein JDK der Version 17 oder höher. Wenn Sie bereits ein solches JDK installiert haben, können Sie mit dem nächsten Abschnitt fortfahren.

Wir empfehlen zur Verwaltung des JDK einen Runtime Manager wie asdf[1] , jEnv[2] oder SDKMan[3]. Diese Tools machen es einfach, zwischen verschiedenen Versionen eines JDK hin- und herzuwechseln. SDKMan und asdf automatisieren darüber hinaus auch das Herunterladen von verschiedenen JDK-Versionen. Wir können mit einem einzigen Kommandozeilenbefehl ein bestimmtes JDK herunterladen oder bestimmen, welches der bereits heruntergeladenen JDKs wir gerade benutzen möchten.

Zur Installation der Runtime Manager verweisen wir auf die Websites der Tools. Im Folgenden zeigen wir, wie wir mit asdf ein JDK installieren, das wir für alle Beispiele in diesem Buch benutzen können.

Vorausgesetzt, asdf ist erfolgreich installiert, müssen wir zunächst das Java-Plugin installieren (asdf unterstützt nämlich auch Runtimes anderer Programmiersprachen):

```
> asdf plugin add java
```

Dann können wir zum Beispiel mit dem folgenden Befehl ein JDK installieren:

```
> asdf install java latest:corretto
```

Hier installieren wir die aktuelle Version des Amazon-Corretto-JDK. Wir können uns auch eine Liste aller anderen JDKs anzeigen lassen und dann eines von denen installieren:

```
> asdf list all java | grep \\-21\\.
```

Diese Liste ist sehr lang, sodass wir sie hier mithilfe von `grep` auf die JDKs der Version 21 filtern.

Nun, da wir ein JDK installiert haben (oder mehrere), müssen wir unsere Umgebung noch so konfigurieren, dass dieses JDK auch überall genutzt wird:

```
> asdf local java latest:corretto
```

Der Befehl `java --version` sollte nun die erwartete Version des JDK ausgeben.

2.3 Kickstart mit dem Spring Initializr

Nun können wir unser Beispielprojekt erstellen. Das geht am einfachsten mit dem Spring Initializr, der unter *https://start.spring.io/* verfügbar ist. Hier können wir einige Parameter, wie zum Beispiel die Java-

1. *https://github.com/asdf-vm/asdf*
2. *https://www.jenv.be/*
3. *https://sdkman.io/*

Version, auswählen und uns eine ZIP-Datei mit einem Projektskelett erstellen lassen. Wir werden den Spring Initializr im Rest dieses Buches noch häufig verwenden, um Beispielprojekte für unterschiedliche Anwendungsfälle zu generieren.

Für unser »Hello World«-Beispielprojekt wählen wir die folgenden Parameter:

- **Project**: Gradle-Kotlin
- **Language**: Java
- **Spring Boot**: aktuelle 3.x-Version (Standardeinstellung)
- **Dependencies**: Spring Web

Alle anderen Einstellungen belassen wir mit ihren Standardwerten. Über den Button »Add Dependencies« können wir eine Auswahl bestimmter Integrationen und Features auswählen, die unserem Projekt dann als Dependencies hinzugefügt werden. Wir wählen zu Beginn nur die »Spring Web«-Dependency, um eine einfache Webanwendung zu entwickeln.

Mit einem Klick auf den »Generate«-Button lassen wir uns das Projekt erstellen und laden die ZIP-Datei herunter, die wir auf unserer Festplatte entpacken. Dann laden wir den Ordner als Projekt in unsere IDE und schauen uns im Projekt um.

2.4 Projektstruktur

Die zentrale Datei im Projektordner ist die Datei *build.gradle.kts*, unser Gradle-Build-Skript. Es ist in der Kotlin-DSL geschrieben, weil wir bei Erstellung des Projekts »Gradle-Kotlin« ausgewählt haben. Wir können auch »Gradle-Groovy« wählen, um das Skript in Groovy zu erstellen, oder »Maven«, um Maven als Build-Tool zu nutzen und eine *pom.xml*-Datei zu erzeugen.

Im Build-Skript finden wir eine Dependency zum Modul *spring-boot-starter-web*, das uns alle Features mitbringt, um eine Webanwendung zu entwickeln. Wir finden außerdem eine Dependency zum Modul *spring-boot-starter-test*, das uns einige Testwerkzeuge zur Verfügung stellt, die wir später im Testing-Kapitel noch genauer betrachten.

Das Build-Skript konfiguriert darüber hinaus die Plugins *org.springframework.boot* und *io.spring.dependency-management*, über die wir in Kapitel 6 noch mehr lernen.

Wir finden des Weiteren die Dateien `gradlew` und *gradlew.bat* und den Ordner *gradle*, die alle zum Gradle Wrapper[4] gehören. Die *gradlew-*

4. *https://docs.gradle.org/current/userguide/gradle_wrapper.html*

Skripte können wir nutzen, um Gradle aufzurufen, ohne Gradle lokal installieren zu müssen. Sie prüfen, ob eine Gradle-Installation verfügbar ist, und laden sie, falls erforderlich, herunter. So können wir unser Projekt auf unterschiedlichen Rechnern laufen lassen, ohne auf jedem Rechner Gradle installieren zu müssen. Wir werden unser Projekt später mit `./gradlew build` bauen.

Im *src*-Ordner finden wir genau drei Dateien:

- ***DemoApplication.java***: Diese Klasse ist der Einstiegspunkt zu unserer Spring-Boot-Anwendung. Sie ist mit der Annotation `@SpringBootApplication` versehen, die wir in Kapitel 4 genauer untersuchen. Wird die `main()`-Methode dieser Klasse aufgerufen, startet unsere Anwendung.
- ***application.properties***: Dies ist die zentrale Konfigurationsdatei für unsere Anwendung. Diese Datei ist leer, weil unsere Anwendung aktuell mit der Standardkonfiguration von Spring Boot zufrieden ist. Wir schauen uns die Konfiguration einer Spring-Boot-Anwendung im Detail in Kapitel 5 an.
- ***DemoApplicationTests.java***: Diese Klasse ist mit der Annotation `@SpringBootTest` versehen. Diese sorgt dafür, dass unsere Anwendung für jede `@Test`-Methode gestartet wird. Die leere Test-Methode `contextLoads()` hat augenscheinlich keinen Effekt, da unsere Anwendung aber für diesen Test gestartet wird, prüft dieser leere Test implizit, dass unsere Anwendung fehlerfrei starten kann. Schleicht sich zum Beispiel ein Konfigurationsfehler ein, schlägt dieser Test fehl. Er gibt uns also eine gewisse Sicherheit, obwohl die Test-Methode leer ist. Wir beschäftigen uns mit den Details des Testing mit Spring Boot in Kapitel 7.

Das sind auch schon alle Dateien unserer Anwendung. Mehr braucht unsere Anwendung nicht, um zu starten.

2.5 Die Anwendung bauen

Was macht man als Erstes, wenn man ein neues Projekt lokal untersucht? Richtig, man führt den Build-Prozess aus, um Vertrauen in das Projekt zu gewinnen. Erst wenn der Build-Prozess problemlos läuft, sollten wir damit anfangen, Änderungen am Projekt vorzunehmen.

In unserem Fall nutzen wir Gradle als Build-Werkzeug, und der Build-Befehl lautet wie folgt:

```
> ./gradlew build
```

Mit diesem Befehl starten wir den Gradle Wrapper, der bei Bedarf Gradle herunterlädt, den Code kompiliert, die Tests ausführt und ein Artefakt erzeugt, das unsere Anwendung beinhaltet. Der Build-Prozess sollte mit einem befriedigenden `BUILD SUCCESSFUL` quittiert werden, wenn alles wie erwartet funktioniert.

Gradle legt während des Builds den Ordner *build* an, der alle Ergebnisse des Build-Prozesses beinhaltet. Im Ordner *build/libs* finden wir eine Datei *demo-0.0.1-SNAPSHOT.jar* (der Name kann variieren, je nachdem, was man bei der Initialisierung des Projekts auf start.spring.io eingegeben hat). Diese Datei ist ein sogenanntes »Fat JAR«, also eine JAR-Datei, die alles beinhaltet, was unsere Anwendung braucht. Wir diskutieren die Build-Prozesse und Fat JARs noch im Detail in Kapitel 6.

Nachdem wir sichergestellt haben, dass unsere Anwendung kompiliert, können wir nun etwas mit ihr herumspielen.

2.6 Die Anwendung starten

Wir können unsere Anwendung auf verschiedene Wege starten, um sie auszuprobieren.

Nachdem wir die Anwendung mit `./gradlew build` gebaut haben, können wir die erzeugte JAR-Datei direkt mit dem Befehl `java -jar <DATEINAME>` starten. Diese Art, die Anwendung zu starten, ist einem echten Produktivszenario am ähnlichsten. Die JAR-Datei würde dabei auf eine Zielmaschine kopiert (am besten verpackt in einem Docker-Image, mehr dazu in Kapitel 30) und dort dann einfach gestartet.

In der täglichen Arbeit an unserer Anwendung wäre das allerdings unpraktisch, da wir zuvor immer den Build ausführen und auf deren Fertigstellung warten müssten.

Wir können unsere Anwendung auch direkt aus unserer IDE heraus starten. Wenn wir zum Beispiel in IntelliJ zu unserer `Application`-Klasse navigieren, sehen wir dort einen grünen »Play«-Button. Dieser startet die Anwendung, und wir können die Ausgabe direkt in der IDE-Konsole verfolgen.

Ein dritter Weg, die Anwendung zu starten, ist es, das Spring-Boot-Plugin von Gradle zu nutzen. Wir können einfach den Kommandozeilenbefehl `./gradlew bootRun` ausführen, und die Anwendung wird gestartet. Der Vorteil dieses Vorgehens ist es, dass wir etwaige Konfigurationsparameter unserer Anwendung in der *build.gradle.kts*-Datei hinterlegen können, sodass alle Entwickler die Anwendung in derselben Konfiguration starten können, ohne immer manuell die Parameter eingeben zu müssen. Aktuell hat unsere Anwendung aber noch keinerlei Konfigurationspara-

meter, deshalb ist es egal, wie wir sie starten. Das Kapitel 5 befasst sich noch im Detail mit der Konfiguration einer Spring-Boot-Anwendung.

Egal, wie wir die Anwendung starten, das Ergebnis ist aktuell noch nicht sehr beeindruckend. Das Einzige, was wir sehen, sind einige Log-Ausgaben in der Konsole, die hoffentlich mit einer Meldung wie dieser hier enden:

```
Started DemoApplication in 1.352 seconds
```

Die Anwendung läuft also. Sie tut aber noch nichts. Das möchten wir nun ändern.

2.7 Einen REST-Controller bauen

Das einfachste Element, das wir unserer Anwendung hinzufügen können, um ihr etwas Funktionalität zu geben, ist ein REST-Controller. Da wir das Modul *spring-boot-starter-web* unserem Projekt schon als Dependency hinzugefügt haben, haben wir bereits das Spring Web MVC Framework im Classpath, das alles mitbringt, was wir dafür brauchen.

Das MVC in »Spring Web MVC« steht für »Model, View, Controller«, ein beliebtes Pattern für Anwendungen, die eine Benutzeroberfläche anbieten. Der Controller nimmt Anfragen entgegen, erzeugt ein Model (oder lädt es aus einer Datenbank) und erzeugt eine View, die das Model für den Benutzer aufbereitet und zur Anzeige bringt. Diese View wird dem Benutzer dann angezeigt, in unserem Fall im Browser.

Ein einfacher »Hello World«-Controller sieht so aus:

Listing 2–1
Ein einfacher Web-Controller

```
@RestController
public class HelloController {

    @GetMapping("/hello/{name}")
    public String hello(
      @PathVariable("name") String name
    ) {
        return String.format("Hello %s", name);
    }

}
```

Wir können diese Klasse einfach neben der `DemoApplication`-Klasse ablegen und Spring Boot greift sie automatisch auf.

Mit der Annotation `@RestController` markieren wir die Klasse als REST-Controller. Der Controller bietet also eine REST-Schnittstelle an, die Webanfragen entgegennimmt und ein Textformat als Antwort zurückgibt (üblicherweise JSON). Die »View« aus MVC besteht in diesem Fall also einfach aus Text und nicht aus einer Benutzeroberfläche.

Wie wir eine HTML-Benutzeroberfläche mit Spring MVC entwickeln, lernen wir in Kapitel 13.

Mit der Annotation `@GetMapping` an der `hello()`-Methode teilen wir Spring MVC mit, dass Anfragen über ein HTTP GET an den Pfad */hello/{name}* an diese Methode geroutet werden sollen. Spring Boot startet automatisch einen Webserver, wenn wir Spring MVC benutzen, der HTTP-Anfragen entgegennimmt und an das Framework zur weiteren Verarbeitung übergibt. In der Parameterliste der `hello()`-Methode nutzen wir die Annotation `@PathVariable("name")`, um den Wert der Variable `{name}` im Pfad in den Methodenparameter `name` zu injizieren.

In der Methode nutzen wir die `name`-Variable dann, um einen String zusammenzubauen und an den Aufrufer zurückzugeben. Diesen String können wir als das Model aus MVC betrachten, das in diesem Fall auch gleichzeitig die View ist, weil wir es ungefiltert an den Aufrufer zurückgeben.

Wir können die Anwendung nun starten und im Browser zum Beispiel *http://localhost:8080/hello/bob* aufrufen; wir sollten dann den String »Hello Bob« im Browser sehen. Spring Boot startet den Webserver standardmäßig auf Port 8080, das ist aber konfigurierbar.

Damit haben wir bereits eine erste funktionsfähige Spring-Boot-Anwendung gebaut! Die Annotationen, die wir benutzt haben, mögen vielleicht noch etwas magisch erscheinen, aber keine Angst: Die Annotationen müssen nicht auswendig gelernt werden. Im Rest des Buches werden wir diese Magie entzaubern und Ihnen Ressourcen an die Hand geben, die es Ihnen erleichtern, die richtigen Annotationen für Ihren Anwendungsfall zu finden. Für die Arbeit mit REST-Controllern tun wir das konkret in Kapitel 9.

3 Spring-Grundlagen

Spring Boot basiert auf dem Spring Framework. Um Spring Boot zu verstehen, müssen wir die Grundlagen des Spring Frameworks verstehen, die wir in diesem Kapitel diskutieren.

3.1 Dependency Injection und Inversion of Control

Spring ist im Kern ein Dependency Injection Framework. Es bietet mittlerweile zwar eine Menge anderer Features, die das Entwicklerleben vereinfachen, aber diese bauen meist auf dem Dependency Injection Framework auf.

Dependency Injection wird oft mit Inversion of Control gleichgesetzt. Wir möchten die beiden Begriffe hier kurz erläuternd in das Spring Framework einordnen und diskutieren, wie Dependency Injection mit dem Spring Framework funktioniert.

3.1.1 Inversion of Control

Das Konzept von Inversion of Control (Kontrollumkehr) ist es, die Kontrolle über den Aufruf von Programmcode an ein Framework abzugeben. Dies kann zum Beispiel in Form einer Funktion geschehen, die wir selbst programmieren, dann aber an ein Framework übergeben, das sie anschließend zum richtigen Zeitpunkt aufruft. Diese Funktion nennen wir »Callback«.

Ein Beispiel für einen Callback ist eine Funktion, die in einer Serveranwendung ausgeführt werden soll, wenn eine bestimmte URL aufgerufen wird. Wir programmieren die Funktion, aber wir rufen sie nicht selbst auf. Stattdessen übergeben wir die Funktion an ein Framework, das auf HTTP-Anfragen auf einem bestimmten Port horcht, die Anfrage analysiert und nach bestimmten Parametern dann an eine der registrierten Callback-Funktionen weiterleitet. Das Spring-Web-MVC-Projekt basiert auf genau diesem Mechanismus. Wir werden uns später noch mit Web MVC auseinandersetzen.

3.1.2 Dependency Injection

Dependency Injection ist eine konkrete Ausprägung von Inversion of Control. Wie der Name andeutet, geht es bei Dependency Injection um Abhängigkeiten. Eine Klasse A ist abhängig von einer anderen Klasse B, wenn die Klasse A eine Methode von B aufruft. In Programmcode wird diese Abhängigkeit häufig in der Form eines Attributs in A vom Typ B ausgedrückt:

Listing 3–1
Der Service erzeugt sein eigenes UserDatabase-Objekt.

```
class GreetingService {
  UserDatabase userDatabase = new UserDatabase();

  String greet(Integer userId) {
    User user = userDatabase.findUser(userId);
    return "Hello " + user.getName();
  }
}
```

In diesem Beispiel benötigt der `GreetingService` ein Objekt vom Typ `UserDatabase`, um seine Arbeit zu machen. Wenn wir ein Objekt vom Typ `GreetingService` instanziieren, instanziiert es automatisch ein Objekt vom Typ `UserDatabase`.

Die Klasse `GreetingService` ist also selbst dafür verantwortlich, die Abhängigkeit zu `UserDatabase` aufzulösen. Dies ist aus mehreren Gründen problematisch.

Zunächst erzeugt diese Lösung eine sehr starke Kopplung der beiden Klassen. Der `GreetingService` muss wissen, wie ein `UserDatabase`-Objekt erzeugt wird. Was wäre, wenn die Erzeugung eines `UserDatabase`-Objekts nicht so einfach ist? Um eine Datenbankverbindung zu öffnen, werden üblicherweise einige Parameter benötigt:

Listing 3–2
Der Service braucht Parameter, um selbst ein UserDatabase-Objekt zu erzeugen.

```
class GreetingService {

  UserDatabase userDatabase;

  public GreetingService(
    String dbUrl,
    Integer dbPort){
      this.userDatabase = new UserDatabase(
       dbUrl,
       dbPort);
  }

  String greet(Integer userId){
    User user = userDatabase.findUser(userId);
    return "Hello " + user.getName();
  }
}
```

Der GreetingService erzeugt immer noch seine eigene Instanz vom Typ UserDatabase, aber er muss plötzlich wissen, welche Parameter eine Datenbankverbindung benötigt. Die Kopplung zwischen GreetingService und UserDatabase ist soeben noch stärker geworden. Wir möchten diese Details im GreetingService gar nicht sehen!

Was, wenn andere Klassen in unserer Anwendung auch ein UserDatabase-Objekt benötigen? Wir möchten nicht, dass jede Klasse wissen muss, wie man ein UserDatabase-Objekt erzeugt!

Durch die starke Kopplung werden Details der Klasse UserDatabase über die ganze Codebase verteilt. Eine Änderung an UserDatabase würde also viele Änderungen an anderen Stellen im Code nach sich ziehen.

Dies erschwert nicht nur die Entwicklung des Anwendungscodes, sondern auch das Schreiben von Tests. Wenn wir die Klasse GreetingService testen möchten, brauchen wir in diesem Beispiel die URL und den Port einer echten Datenbank. Wenn wir ungültige Verbindungsparameter übergeben, funktioniert die Methode greet() nicht mehr!

Um die starke Kopplung zwischen den Klassen aufzuheben, ändern wir den Code, sodass wir die Abhängigkeit in den Konstruktor »injizieren« können:

Listing 3–3
Der Service bekommt das UserDatabase-Objekt im Konstruktor »injiziert«.

```
class GreetingService {

  final UserDatabase userDatabase;

  public GreetingService(UserDatabase userDatabase) {
    this.userDatabase = userDatabase;
  }

  String greet(Integer userId){
    User user = userDatabase.findUser(userId);
    return "Hello " + user.getName();
  }
}
```

Es gibt immer noch eine Kopplung zwischen GreetingService und UserDatabase, aber diese ist viel loser als vorher, denn GreetingService muss nun nicht mehr wissen, wie ein UserDatabase-Objekt erzeugt wird. Die Kopplung ist auf das erforderliche Minimum reduziert. Dieses Pattern wird »Constructor Injection« genannt, da wir die Abhängigkeiten einer Klasse in Form von Konstruktorparametern übergeben.

In einem Test können wir nun ein Mock-Objekt vom Typ UserDatabase erzeugen (zum Beispiel mit einer Mock-Bibliothek wie Mockito; mehr dazu in Kap. 7) und dieses an den GreetingService übergeben. Da wir das Verhalten des Mocks kontrollieren, brauchen wir keine Verbindung zu einer echten Datenbank mehr, um die Klasse GreetingService zu testen.

Im Anwendungscode instanziieren wir die Klasse `UserDatabase` nur einmal und übergeben diese Instanz an alle Klassen, die sie benötigen. Anders ausgedrückt »injizieren« wir die Abhängigkeit zu `UserDatabase` in die Konstruktoren der anderen Klassen.

Dieses »Injizieren« von Abhängigkeiten kann in einer echten Anwendung mit mehreren hundert Klassen sehr umständlich werden, da wir alle Klassen in der richtigen Reihenfolge instanziieren und die Abhängigkeiten zwischen ihnen explizit ausprogrammieren müssen. Die Folge ist sehr viel »Boilerplate«-Code, der sich häufig ändert und uns von der eigentlichen Entwicklung abhält.

Genau hier kommt Dependency Injection ins Spiel. Ein Dependency Injection Framework wie Spring übernimmt die Aufgabe, den Großteil der Klassen unserer Anwendung zu instanziieren, sodass wir uns nicht mehr darum kümmern müssen. Hier wird deutlich, dass Dependency Injection eine Ausprägung von Inversion of Control ist, denn wir geben die Kontrolle über die Instanziierung unserer Objekte an das Dependency Injection Framework ab.

Die Aufgabenteilung zwischen Spring und uns als Entwicklern sieht in etwa so aus:

- Wir programmieren die Klassen `GreetingService` und `UserDatabase`.
- Wir drücken die Abhängigkeit zwischen den beiden Klassen durch einen Parameter vom Typ `UserDatabase` im Konstruktor von `GreetingService` aus.
- Wir teilen Spring mit, dass es die Kontrolle über die Klassen `GreetingService` und `UserDatabase` übernehmen soll.
- Spring instanziiert die Klassen in der richtigen Reihenfolge, um Abhängigkeiten aufzulösen, und erzeugt ein Objektnetzwerk mit einem Objekt für jede übergebene Klasse.
- Wenn wir ein Objekt vom Typ `GreetingService` oder `UserDatabase` benötigen, fragen wir Spring nach diesem Objekt.

In einer echten Anwendung verwaltet Spring natürlich nicht nur zwei Objekte, sondern ein komplexes Netzwerk von hunderten oder tausenden Objekten. Dieses Netzwerk wird in Spring »Application Context« genannt, da es den Kern unserer Anwendung ausmacht.

Wie der Application Context funktioniert, schauen wir uns im nächsten Abschnitt an.

3.2 Der Spring Application Context

Der Application Context ist das Herz einer Anwendung, die auf dem Spring Framework basiert. Er beinhaltet all die Objekte, deren Kontrolle wir an Spring übergeben haben. Aus diesem Grund wird er manchmal auch als »IoC Container« bezeichnet (IoC = »Inversion of Control«).

Die Objekte im Application Context werden »Beans« genannt. Wenn man nicht aus der Java-Welt kommt, ist der Begriff »Bean« (Bohne) vermutlich eher verwirrend. Spring ist ein Framework für die Programmiersprache Java. Java ist der Name einer Insel in Indonesien, auf der Kaffee angebaut wird. Die Kaffee-Art wird ebenfalls »Java« genannt. Und Kaffee wird aus Kaffeebohnen hergestellt. In der Java Community hat man sich deshalb dafür entschieden, bestimmte Objekte in Java (der Programmiersprache) »Beans« zu nennen. Ziemlich weit hergeholt, aber der Begriff hat sich eingebürgert.

Der Application Context ist also im Grunde nur ein Netzwerk von Java-Objekten, genannt »Beans«. Spring instanziiert diese Beans für uns und löst die Abhängigkeiten zwischen den Beans über Constructor Injection auf.

Woher weiß Spring aber, welche Beans es für uns erzeugen und in seinem Application Context verwalten soll? Hier kommt der Begriff »Konfiguration« ins Spiel.

Eine Konfiguration im Kontext von Spring ist eine Definition der Beans, die wir für unsere Anwendung benötigen. Im einfachsten Fall ist dies eine Liste von Klassen. Spring nimmt diese Klassen, instanziiert sie und nimmt die entstandenen Objekte (Beans) in den Application Context auf.

Wenn die Instanziierung der Klassen nicht möglich ist (zum Beispiel, wenn ein Bean-Konstruktor eine andere Bean erwartet, die aber nicht Teil der Konfiguration ist), bricht Spring die Erstellung des Application Context mit einer Exception ab.

Dies ist einer der Vorteile, den Spring bietet: Eine fehlerhafte Konfiguration führt in den meisten Fällen dazu, dass die Anwendung gar nicht erst startet und somit keinen Schaden zur Laufzeit anrichten kann.

Es gibt mehrere Wege, eine Spring-Konfiguration zu erstellen. In den meisten Anwendungsfällen ist es komfortabel und praktisch, die Konfiguration in Java zu programmieren. In Anwendungsfällen, wo der Quellcode komplett frei von Abhängigkeiten zum Spring Framework sein soll, kann aber auch eine Konfiguration mit XML sinnvoll sein.

3.3 Application Context mit XML konfigurieren

Zu den Anfängen von Spring musste der Application Context mit XML konfiguriert werden. Die Konfiguration mit XML ermöglicht es, die Konfiguration und den Code komplett voneinander zu trennen. Der Code muss nicht wissen, dass er von Spring verwaltet wird.

Eine Beispielkonfiguration in XML sieht so aus:

Listing 3–4
Beans mit XML definieren

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <bean id="userDatabase"
    class="de.springboot3.xml.UserDatabase"/>
  <bean id="greetingService"
    class="de.springboot3.xml.GreetingService">
      <constructor-arg ref="userDatabase"/>
  </bean>
</beans>
```

In dieser Konfiguration werden die Beans `userDatabase` und `greetingService` definiert. Jede Bean-Deklaration ist eine Anleitung für Spring, wie diese Bean instanziiert wird.

Die Klasse `UserDatabase` hat einen Default-Konstruktor ohne Parameter, weshalb es ausreicht, Spring den Namen der Klasse zu nennen. Die Klasse `GreetingService` hat einen Konstruktorparameter vom Typ UserDatabase, weshalb wir mit dem `constructor-ref`-Element auf die zuvor deklarierte `userDatabase`-Bean verweisen.

Mit dieser XML-Deklaration können wir nun einen Application Context erzeugen:

Listing 3–5
Einen Application Context aus einer XML-Datei erzeugen

```
public class XmlConfigMain {

  public static void main(String[] args) {
    ApplicationContext applicationContext =
      new ClassPathXmlApplicationContext(
        "application-context.xml");

    GreetingService greetingService =
      applicationContext.getBean(
        GreetingService.class);

    System.out.println(greetingService.greet(1));
   }
}
```

Wir übergeben dem Konstruktor von `ClassPathXmlApplicationContext` die XML-Konfiguration, und Spring erzeugt daraus ein `ApplicationContext`-Objekt für uns.

Dieser `ApplicationContext` ist nun unser IoC-Container, und wir können ihn zum Beispiel nach einer Bean vom Typ `GreetingService` fragen.

Die Konfiguration per XML sieht in diesem Beispiel noch recht überschaubar aus, kann aber in größeren Anwendungen sehr umfangreich werden. Für uns als Java-Entwickler wäre es doch angenehm, wenn wir eine solch umfangreiche Konfiguration in Java selbst verwalten und die Vorteile des Java-Compilers und der IDE für uns nutzen könnten.

3.4 Java-Konfiguration im Detail

Spätestens seit dem Erfolg von Spring Boot wird die XML-Konfiguration meist nur noch in Sonderfällen und Legacy-Anwendungen genutzt, und die Konfiguration mit Java ist zum Standard geworden. Deshalb werden wir uns diesen Weg, auch »Java-Config« genannt, genauer anschauen.

3.4.1 @Configuration und @Bean

Der Kern einer Java-Konfiguration ist eine Java-Klasse, die mit der Spring-Annotation `@Configuration` annotiert ist:

Listing 3–6
Eine einfache @Configuration-Klasse

```
@Configuration
public class GreetingConfiguration {

  @Bean
  UserDatabase userDatabase() {
    return new UserDatabase();
  }

  @Bean
  GreetingService greetingService(
    UserDatabase userDatabase) {
    return new GreetingService(userDatabase);
  }
}
```

Diese Konfiguration ist äquivalent zu der XML-Konfiguration aus dem letzten Abschnitt. Mit der Annotation `@Configuration` teilen wir Spring mit, dass diese Klasse einen Teil des Application Context beisteuert. Ohne diese Annotation wird Spring nicht aktiv.

Eine Konfigurationsklasse kann dann Factory-Methoden wie hier `userDatabase()` und `greetingService()` deklarieren, die jeweils ein Objekt erzeugen. Mit der Annotation `@Bean` markieren wir solche Factory-Methoden. Spring findet diese Methoden und ruft sie auf, um einen Application Context zu erzeugen.

Abhängigkeiten zwischen Beans, wie hier die Abhängigkeit von `GreetingService` zu `UserDatabase`, werden über Parameter der Factory-

Methoden aufgelöst. In unserem Fall wird Spring zuerst die Methode `userDatabase()` aufrufen, um eine `UserDatabase`-Bean zu erzeugen, und diese dann in die Methode `greetingService()` übergeben, um eine `GreetingService`-Bean zu erzeugen.

Mithilfe der Klasse `AnnotationConfigApplicationContext` können wir dann einen `ApplicationContext` erzeugen:

Listing 3–7
Einen Application Context aus einer Java-Config erzeugen

```
public class JavaConfigMain {

  public static void main(String[] args) {
    ApplicationContext applicationContext =
      new AnnotationConfigApplicationContext(
        GreetingConfiguration.class);

    GreetingService greetingService =
      applicationContext.getBean(
        GreetingService.class);

    System.out.println(greetingService.greet(1));
  }
}
```

Der Konstruktor von `AnnotationConfigApplicationContext` erlaubt uns auch, mehrere Konfigurationsklassen zu übergeben anstatt nur eine einzige. Dies ist hilfreich für größere Anwendungen, denn wir können die Konfiguration von vielen Beans in mehrere Konfigurationsklassen aufteilen, um die Übersicht zu bewahren.

3.4.2 @Component und @ComponentScan

Die Konfiguration Hunderter oder gar Tausender von Beans via Java für eine große Anwendung kann sehr mühselig werden. Um dies zu vereinfachen, bietet Spring die Möglichkeit, im Java-Classpath nach Beans zu »scannen«.

Dieser Scan wird mit der Annotation `@ComponentScan` aktiviert:

Listing 3–8
Scannen nach Beans im Package de.springboot3

```
@Configuration
@ComponentScan("de.springboot3")
public class GreetingScanConfiguration {
}
```

Wie zuvor erstellen wir eine Konfigurationsklasse (annotiert mit `@Configuration`). Hier definieren wir die Beans aber nicht selbst in Form von Factory-Methoden annotiert mit der `@Bean`-Annotation, sondern fügen die neue Annotation `@ComponentScan` hinzu.

Mit dieser Annotation weisen wir Spring an, im Package *de.springboot3* nach Beans zu scannen. Findet der Scan eine Klasse, die mit der Annotation `@Component` versehen ist, wird aus dieser Klasse eine Bean er-

zeugt (das heißt, die Klasse wird instanziiert und dem Spring Application Context hinzugefügt).

Wir müssen also einfach alle Klassen, für die Spring eine Bean erzeugen soll, wie folgt mit der `@Component`-Annotation versehen:

Listing 3–9
Mit @Component annotierte Klassen werden zu Beans.

```
@Component
public class GreetingService {
  // …
}

@Component
public class UserDatabase {
  // ...
}
```

Abhängigkeiten zwischen den Beans werden wie zuvor durch Konstruktorparameter ausgedrückt, und Spring löst diese automatisch auf – dieser Mechanismus wird häufig auch »Autowiring« genannt.

@Bean vs. @Component

Die Annotationen `@Bean` und `@Component` drücken ein ähnliches Konzept aus: Beide markieren ein Bean, das dem Application Context von Spring hinzugefügt werden soll. Diese Ähnlichkeit kann insbesondere zu Anfang verwirren.

Der Hauptunterschied beider Ansätze ist, dass wir mit der `@Bean`-Annotation selbst die Kontrolle über die Instanziierung der Beans haben und mit der `@Component`-Annotation Spring die Instanziierung überlassen. `@Bean` ist nur an Methoden erlaubt und `@Component` nur an Klassen.

Der Java-Compiler hilft uns hier etwas, denn er meckert, wenn wir eine `@Bean`-Annotation an einer Klasse benutzen oder eine `@Component`-Annotation an einer Methode. Wir können sie also nicht verwechseln. Wir können aber sehr wohl Methoden und Klassen annotieren, die von Spring gar nicht erkannt werden!

Spring wertet die `@Bean`-Annotation nur innerhalb einer `@Configuration`-Klasse und die `@Component`-Annotation nur an Klassen aus, die von einem Component-Scan gefunden werden!

3.4.3 @Configuration und @ComponentScan kombinieren

Spring schreibt uns nicht vor, wie wir die Beans unserer Anwendung konfigurieren. Wir können sie per XML oder per Java-Config konfigurieren oder gar beide Varianten miteinander kombinieren. Wir können auch explizite Bean-Definitionen per `@Bean`-Methode und einen Scan per `@ComponentScan` kombinieren:

Listing 3–10
Mischen von Component-Scan und Java-Config

```
@Configuration
@ComponentScan("de.springboot3.java.mixed")
class MixedConfiguration {
  @Bean
  GreetingService greetingService(
    UserDatabase userDatabase) {
        return new GreetingService(userDatabase);
    }
}

// no @Component annotation!
class GreetingService {…}

@Component
class UserDatabase {...}
```

In dieser Konfiguration erzeugt Spring eine Bean vom Typ `UserDatabase`, da die Klasse mit `@Component` annotiert ist und ein `@ComponentScan` konfiguriert ist. Die Bean vom Typ `GreetingService` hingegen wird durch die explizite `@Bean`-annotierte Factory-Methode definiert.

Modulare Konfiguration

Die Konfiguration einer größeren Anwendung mit hunderten Beans kann schnell unübersichtlich werden.

Die explizite Konfiguration per `@Bean`-Annotation hat den Vorteil, dass die Konfiguration der Beans in einigen wenigen `@Configuration`-Klassen gebündelt ist und so leicht zu begreifen ist.

Die implizite Konfiguration per `@ComponentScan` und `@Component` hat den Vorteil, dass wir nicht jede Bean selbst definieren müssen, ist dafür aber über sehr viele `@Component`-Annotationen und somit über die gesamte Codebase verteilt und schwieriger zu begreifen.

Ein bewährtes Prinzip ist es, die Spring-Konfiguration entlang der Architektur der Anwendung zu gestalten. Jedes Modul der Anwendung sollte in seinem eigenen Package liegen und seine eigene `@Configuration`-Klasse besitzen. In dieser `@Configuration`-Klasse können wir dann entweder einen `@ComponentScan` für das Modulpackage konfigurieren oder eine explizite Konfiguration per `@Bean`-Methoden. Um die Module zu einer Gesamtanwendung zusammenzuführen, erstellen wir eine übergeordnete `@Configuration`-Klasse, die einen `@ComponentScan` für das Hauptpackage definiert. Dieser Scan wird alle `@Configuration`-Klassen in diesem und den untergeordneten Packages aufgreifen.

3.5 Was haben wir vom Spring Container?

Wir wissen nun, dass Spring uns einen IoC-Container bietet, der Objekte (Beans) für uns instanziiert und verwaltet. Das erspart uns die Arbeit, den Lebenszyklus dieser Objekte selbst zu verwalten.

Aber das ist erst der Anfang. Da Spring die Kontrolle über alle Beans hat, kann Spring eine Menge anderer Aufgaben für uns übernehmen. Zum Beispiel kann Spring die Aufrufe von Bean-Methoden abfangen, um zum Beispiel eine Datenbanktransaktion zu starten oder zu committen. Wir können Spring auch als Event-Bus nutzen. Dafür senden wir ein Event an den Spring-Container, und Spring leitet das Event an alle interessierten Beans weiter.

Diesen und vielen anderen Features werden wir im Rest des Buches auf den Grund gehen. Die Grundlage all dieser Features ist das Spring-Programmiermodell, dessen Kern wir in diesem Kapitel bereits kennengelernt haben. Dieses Programmiermodell ist eine Mischung aus Annotationen, Konventionen und Interfaces, die wir zu einer Gesamtanwendung kombinieren können.

Inhaltsübersicht

Inhaltsverzeichnis