



# SpringBoot

Programmieren 4

BIS-20

Christoph Dorner, is201004

Mai 2022

## Inhaltsverzeichnis

1 Spring Boot . . . . .	2
1.1 Vorteile und Eigenschaften von SpringBoot . . . . .	2
1.2 Aufbau von Spring . . . . .	2
1.3 SpringBoot . . . . .	4
1.4 Aufbau einer SpringBoot Applikation . . . . .	5
1.4.1 Einstellungen einer SpringBoot-Applikation . . . . .	5
1.5 Spring-Beans . . . . .	6
1.5.1 Lifecycle von Spring-Beans . . . . .	6
1.6 SpringBoot Auto-Configuration . . . . .	7
1.7 Der CommandLineRunner . . . . .	14
2 RESTful Webservices mit SpringBoot . . . . .	15
2.1 Strukturierung einer SpringBoot Applikation . . . . .	15
2.1.1 Strukturierung nach Layer . . . . .	15
2.1.2 Strukturierung nach Feature . . . . .	17
2.2 REST-Controller . . . . .	18
2.3 Verwendung von Exceptions mit dem REST-Controller . . . . .	20
3 Die H2 Database . . . . .	22
3.1 Speichern von Entitäten in einer H2 unter verwendung von JPA . . . . .	23
3.2 Common Mistakes . . . . .	25
Abbildungsverzeichnis . . . . .	26
Quellcodeverzeichnis . . . . .	27

## 1 Spring Boot

Spring ist ein Java-basiertes Open-Source Framework, mit welchem sich schnell u.A. Micro-Services für REST-Endpoints oder Frontends nach dem MVC-Konzept erstellen lassen. SpringBoot selbst ist ein Modul für das Spring-Framework, welches die Zielsetzung hat, schnell und einfach Spring Applikationen erstellen zu können.

### 1.1 Vorteile und Eigenschaften von SpringBoot

- Einfache Entwicklung von Spring-Applikationen
- Entkopplung der verschiedenen Softwarekomponenten (Loose-Coupling)
- Reduzierung der Entwicklungszeit und Steigerung der Produktivität
- Vermeiden komplexer Konfiguration
- Bietet embedded Server (Tomcat, Jetty, etc.)
- Starter Projects mit Autoconfiguration (Spring-Web, JPA, etc.)

### 1.2 Aufbau von Spring

Spring selbst ist eine vollständige Implementierung der Java Enterprise Edition (JEE oder J2EE) Spezifikation, welche eine grundlegende Empfehlung für den Aufbau von Business-Applikationen vorgibt. Dabei hat Spring den Fokus auf einen schnellen und einfachen Aufbau einer Applikation und entfernt viel des Konfigurationsaufwands und Boilerplate-Codes welcher für EJB (Enterprise Java Beans) benötigt wurde. Die Hauptkomponenten und der Aufbau des Spring Frameworks ist dabei in Abbildung 1 dargestellt.

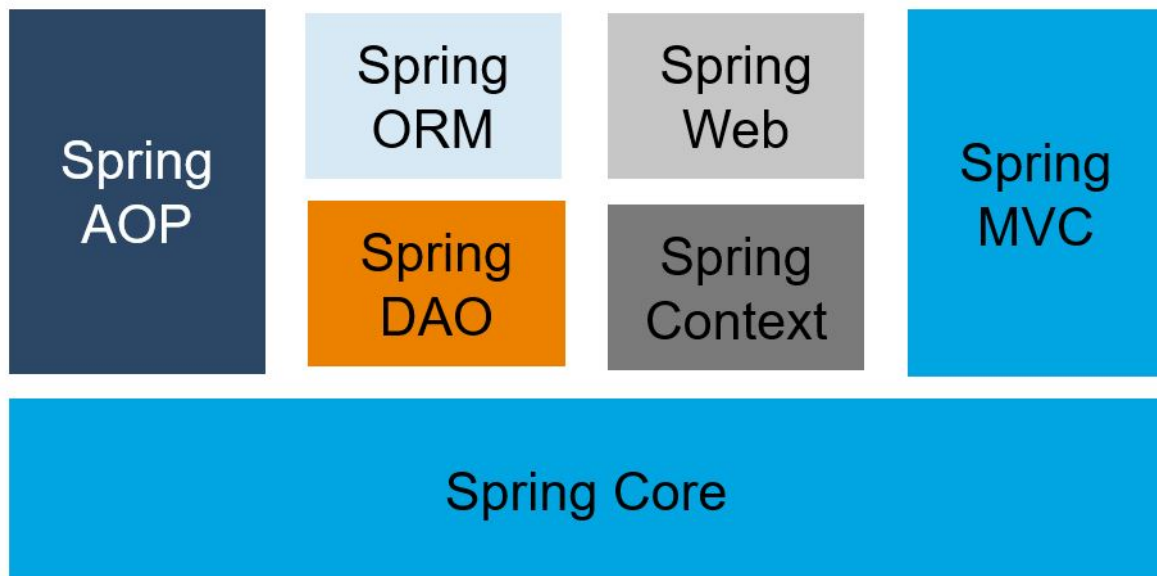


Abbildung 1: Darstellung des Kern des Spring-Frameworks  
**Source:** [blog.doubleslash.de](http://blog.doubleslash.de)

Der **Spring Core** ist die zentrale Komponente des Spring-Frameworks. Dieser stellt eine der Hauptfunktionalitäten des Frameworks bereit: Die **Dependency-Injection** (DI)-Funktionalität. Diese wird mittels einem **Inversion-of-Control** (IoC) Container realisiert, welcher die Dependencies verwaltet.

**Spring-AOP:** Dieses Modul unterstützt die sogenannte aspektorientierte Programmierung. Die Grundidee dahinter ist: verschiedene Aspekte (wie z.B. Logging oder Transaktionsverwaltung) sind Funktionalitäten, die in den verschiedenen Anwendungsklassen immer wieder benötigt werden (Cross-Cutting Concerns). Mit Hilfe der AOP können solche Funktionalitäten in Aspekte ausgelagert werden und damit kann die Kernfunktionalität der Anwendung von solchen Aspekten freigehalten werden.

**Spring-Context** stellt Funktionalitäten für Validierung, Enterprise-Java-Bean-Integration und Scheduling bereit.

Das Modul **Spring-DAO** bietet einen simplen Datenbankzugriff mittels JDBC (Java-Database-Connectivity) und kümmert sich um das Transaktionsmanagement.

Das **Spring-ORM** Modul baut auf dem Spring-DAO Modul auf. Es ermöglicht ORM's (Object-Relational-Mapper; z.B. Hibernate, Eclipselink, JPA, etc.) für den Zugriff auf (Relationale) Datenbanken zu verwenden. Auch der Zugriff auf No-SQL Datenbanken ist damit möglich. Der Unterschied zwischen den Datenbankmodulen ist der, dass Spring-DAO auf Statementbasis arbeitet, und Spring-ORM auf der Basis von ganzen Tabellen die in Java-Klassen übersetzt werden. Dies wird allerdings von den ORM's auch intern über Statements umgesetzt, daher ist dieses Modul auf Spring-DAO aufgebaut.

Das Modul **Spring-Web** stellt die grundlegende Web-Funktionalität zur Verfügung (Also HTTP-Integration, ServletFilter und andere Infrastruktur um weitere HTTP-Technologien zu implementieren).

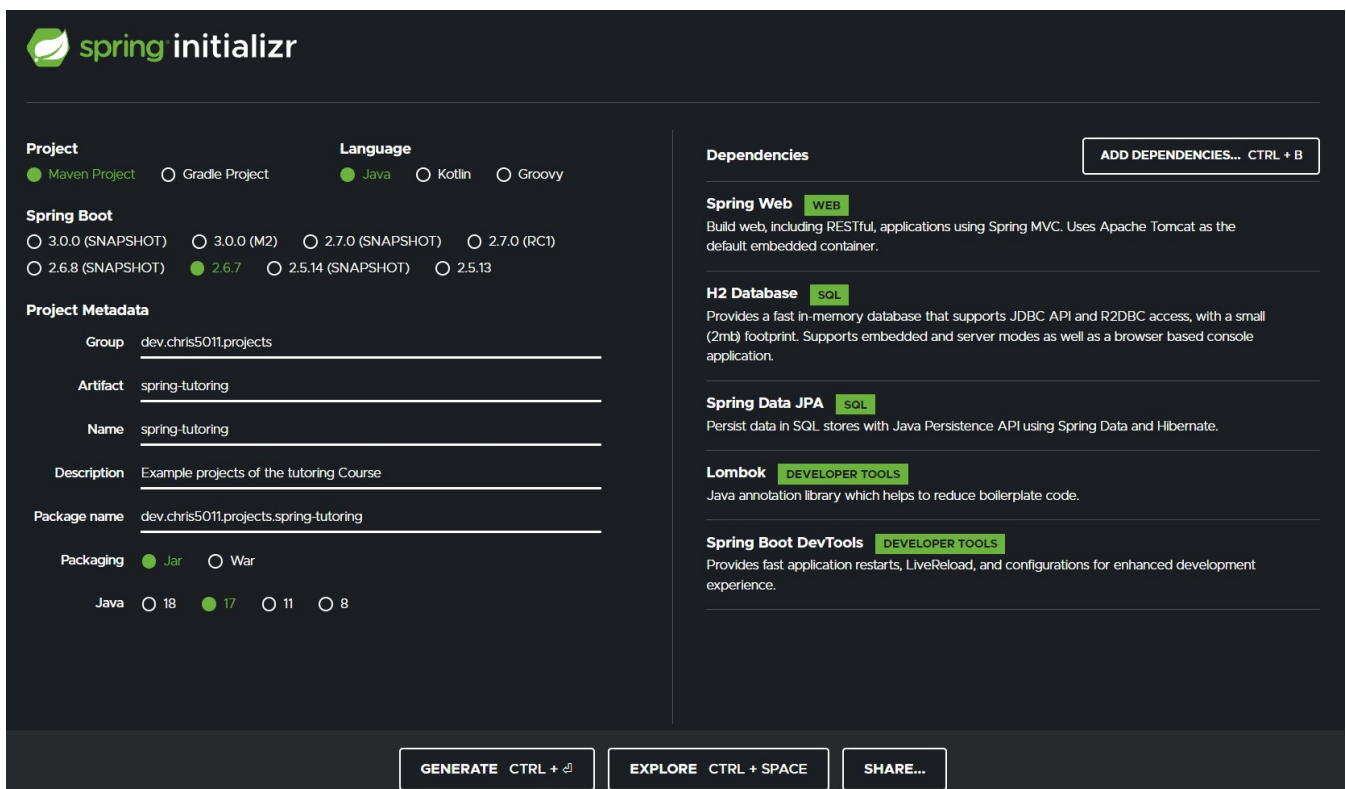
**Spring-MVC** baut dabei auf Spring-Web auf, und bietet erweiterte Funktionalitäten, wie die Möglichkeit der Erstellung von REST-Endpunkten, oder von Frontend-Entwicklung nach dem MVC-Konzept.

Auf dem oben beschriebenen Kern des Spring-Frameworks aufbauend, gibt es mittlerweile noch viele weitere Module wie z.B. Spring-Security, Spring-Cloud, Spring-Data aber auch Spring-Boot. Diese erweitern dabei das Spring-Ökosystem.

### 1.3 SpringBoot

SpringBoot ist das für uns wichtigste Modul, da es die Erstellung einer SpringBoot-Anwendung schnell und einfach gestaltet. Ohne SpringBoot müssten alle Dependencies manuell auf Kompatibilität geprüft und einzeln konfiguriert werden.

Um ein SpringBoot Projekt zu erstellen, kann der SPRING INITIALIZR verwendet werden. Dabei können dort allgemeine Einstellungen zum Projekt getroffen, sowie Dependencies hinzugefügt werden. Dieser ist in Abbildung 2 dargestellt.



The screenshot shows the Spring Initializr web application interface. It is a dark-themed form for configuring a new Spring Boot project. The interface is divided into several sections:

- Project:** Includes radio buttons for **Maven Project** (selected) and **Gradle Project**.
- Language:** Includes radio buttons for **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Includes radio buttons for various versions: 3.0.0 (SNAPSHOT), 3.0.0 (M2), 2.7.0 (SNAPSHOT), 2.7.0 (RC1), 2.6.8 (SNAPSHOT), **2.6.7** (selected), 2.5.14 (SNAPSHOT), and 2.5.13.
- Project Metadata:** Includes text input fields for **Group** (dev.chris5011.projects), **Artifact** (spring-tutoring), **Name** (spring-tutoring), **Description** (Example projects of the tutoring Course), and **Package name** (dev.chris5011.projects.spring-tutoring).
- Packaging:** Includes radio buttons for **Jar** (selected) and **War**.
- Dependencies:** A section on the right with a button **ADD DEPENDENCIES... CTRL + B**. It lists several dependencies with checkboxes:
  - Spring Web** (WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
  - H2 Database** (SQL): Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.
  - Spring Data JPA** (SQL): Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
  - Lombok** (DEVELOPER TOOLS): Java annotation library which helps to reduce boilerplate code.
  - Spring Boot DevTools** (DEVELOPER TOOLS): Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

At the bottom, there are three buttons: **GENERATE CTRL + G**, **EXPLORE CTRL + SPACE**, and **SHARE...**

Abbildung 2: Konfigurieren eines SpringBoot Projekts mit dem Spring Initializr

## 1.4 Aufbau einer SpringBoot Applikation

Der Einstiegspunkt für eine SpringBoot Applikation ist die Klasse die mit `@SpringBootApplication` annotiert wurde. Die einfachste Form einer SpringBoot Einstiegsklasse ist in Listing 1 dargestellt.

```
package dev.chris5011.projects.springtutoring;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringTutoringApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringTutoringApplication.class, args);
    }
}
```

Code 1: Entry-Point einer Spring-Boot Applikation

### 1.4.1 Einstellungen einer SpringBoot-Applikation

Eine SpringBoot Applikation kann mittels der Datei `application.properties`, welche direkt im `resources`-Verzeichnis liegen muss, konfiguriert werden. Dabei sind allerdings Meta-Einstellungen wie der ServerPort, oder Logging- bzw. Datenbankeinstellungen gemeint, und nicht die Spring-Autoconfiguration. Abbildung 3 zeigt die Platzierung der genannten Datei im Projektverzeichnis.

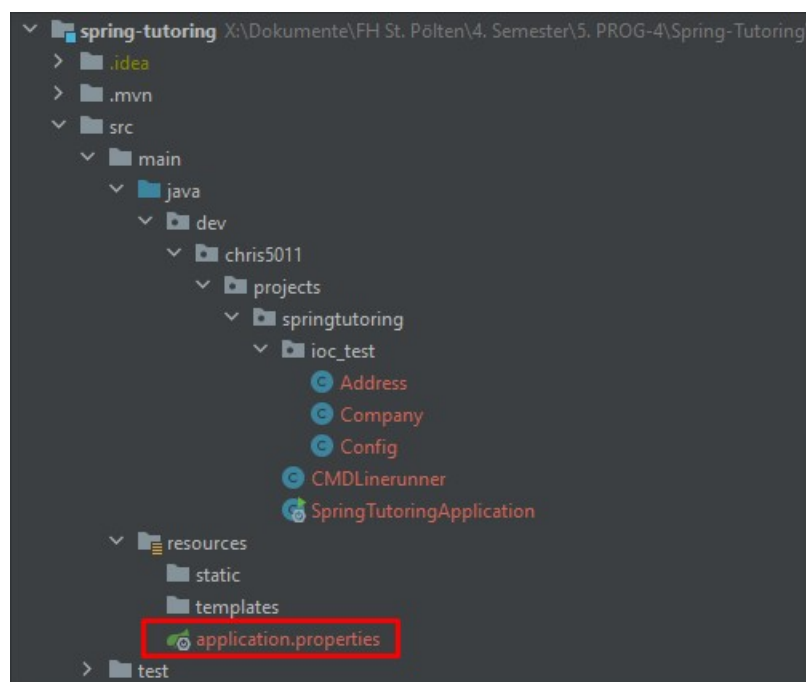


Abbildung 3: Platzierung der application.properties

## 1.5 Spring-Beans

Eine Spring-Bean ist eine Instanz einer Klasse die mit `@Component` o.Ä. Annotiert wurde oder über eine `BeanFactory` (Methode mit `@Bean` annotiert) erstellt wurde. Beans werden vom IoC-Container verwaltet. Dabei wird mit `@Component` die grundlegendste Form einer Bean erstellt. Die anderen Annotationen die Beans erstellen, erzeugen spezialisierte Formen von Beans. (z.B. `@Repository` oder `@Service`)

### 1.5.1 Lifecycle von Spring-Beans

Da Beans vom IoC-Container erzeugt werden, macht es nicht viel Sinn, im Konstruktor Initialisierungen vorzunehmen. Um Beans in ihrem Lebenszyklus trotzdem zu beeinflussen, können die Annotationen `@PostConstruct` und `@PreDestroy` verwendet werden um Methoden im Bean zu annotieren. Dabei wird die mit `@PostConstruct` annotierte Methode nach der Erzeugung und abgeschlossenen Dependency-Injections aufgerufen.

Die Methode welche mit `@PreDestroy` annotiert wurde wird aufgerufen bevor das Bean vom IoC-Container zerstört wird. Abbildung 4 visualisiert den gesamten Prozess beginnend vom Start des Containers bis zum Aufruf der Methode welche mit `@PreDestroy` annotiert ist.

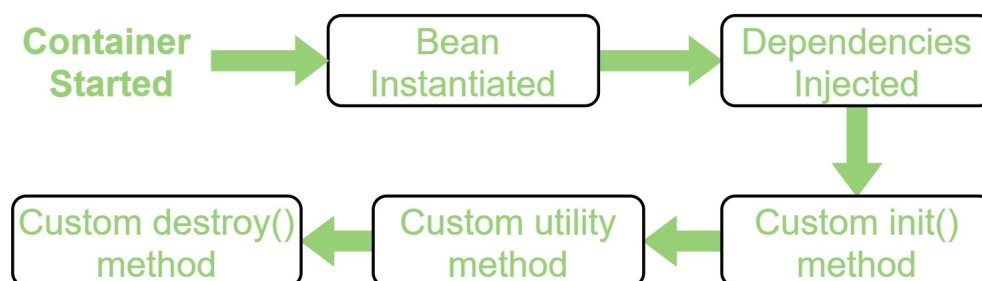


Abbildung 4: Lifecycle eines Spring-Beans

**Source:** [www.geeksforgeeks.org](http://www.geeksforgeeks.org)

Das unten stehende Listing 2 zeigt eine schematische Implementierung der Methoden mit den oben genannten Annotationen.

```
@Component
public class SpringBeanLifecycleDemo {
    private DemoBean demoBean;

    @Autowired
    public SpringBeanLifecycleDemo(DemoBean demoBean) {
        this.demoBean = demoBean;
    }

    @PostConstruct
    public void init(){
        /* Hier kann dieses Spring-Bean initialisiert werden.
         * Dabei kann hier auch schon auf das demoBean zugegriffen werden. */
    }

    @PreDestroy
    public void destroy(){
        /* Hier koennen cleanup-arbeiten durchgefuehrt werden */
    }
}
```

Code 2: Schematische Implementierung des Lebenszyklus einer Spring-Bean

## 1.6 SpringBoot Auto-Configuration

Spring basiert sehr stark auf Konfiguration per Annotationen (bzw. Konfigurierbar auch über externes XML). Der Grundlegende Ablauf beim Starten einer Spring-Boot Applikation ist der Folgende:

1. Der Auto-Konfigurator von Spring-Boot scannt alle Klassen im Classpath, bzw. die Entsprechend annotiert wurden.
2. Es werden Instanzen von den annotierten Klassen erstellt und die jeweiligen Abhängigkeiten aufgelöst.
3. Die zuvor erstellten Klassen werden, wo sie benötigt werden automatisch per Dependency-Injection injected.

Dabei sind die wichtigsten Annotationen im Nachfolgenden aufgelistet:

- **@Bean:** Kann auf Methoden angewandt werden, um diese als Bean-Factory zu markieren. Das heißt, sie wird aufgerufen um ein Spring-Bean zu erzeugen, welches per DI injected werden kann.
- **@Configuration:** Diese Annotation wird verwendet um eine Klasse die Methoden besitzt, welche als Quelle Spring-Beans (Methoden mit @Bean annotiert) dienen zu markieren.
- **@ComponentScan:** Wird verwendet, um Pakete zu markieren, die im Auto-Config Zyklus von Spring erfasst werden sollen.



- **@Component**: Macht aus der Klasse zum Zeitpunkt des Auto-Scans ein Spring-Bean. Die Instanzen dieser Klassen werden mit Hilfe vom IoC-Container verwaltet. Dies ist die allgemeinste Spring-Annotation.
- **@Service**: Quasi das gleiche wie @Component, nur wird diese verwendet um Spring zu signalisieren, dass die annotierte Klasse eine Klasse mit Business-Logic ist, und Spring damit nicht so streng wie mit einem Component umgehen muss.
- **@Autowired**: Dies Annotation wird verwendet um die verschiedenen Spring-Beans einer Applikation miteinander zu verknüpfen. Dabei muss die Klasse in der diese Annotation verwendet wird ein Spring-Bean sein.

Es wird dabei zwischen Konstruktor- Setter bzw. Field-Injection unterschieden. Die folgenden Listings zeigen die unterschiedlichen Verwendungen anhand einer Klasse `Car`, welche die Bean `Engine` injiziert bekommt:

#### Constructor-Injection:

```
@Component
public class Car {
    private Engine engine;

    @Autowired
    public Car(Engine engine) {
        this.engine = engine;
    }
    //Getter, Setter and other Boilerplate Code
}
```

Code 3: Spring Constructor Injection

Sollte in einer Klasse ein Bean injected werden, aber es ist die Annotation @Autowired nicht vorhanden, aber ein Konstruktor mit dem entsprechenden Bean als Parameter, so wird die Injection trotzdem gelingen. Das bedeutet, in obigem Beispiel hätte das @Autowired weggelassen werden können.

#### Setter-Injection:

```
@Component
public class Car {
    private Engine engine;

    @Autowired
    public void setEngine(Engine engine) {
        this.engine = engine;
    }
    //Getter and other Boilerplate Code
}
```

Code 4: Spring Setter Injection

**Field-Injection:**

```

@Component
public class Car {

    @Autowired
    private Engine engine;

    //Getter and other Boilerplate Code
}

```

Code 5: Spring Field Injection

Sollte die Bean `Engine` in diesem Fall ein Interface sein und mehrere konkrete Implementierungen haben, so kann es sein, dass SpringBoot nicht entscheiden kann, welche Bean Injected werden soll und eine `NoUniqueBeanDefinitionException` wirft. Um dies zu Umgehen, können die nächsten zwei Annotationen (`@Qualifier` und `@Primary`), bzw. die unten gezeigt Methode verwendet werden um Klarheit zu schaffen. Dabei ist im Folgenden eine Ausgangssituation gegeben:

```

public interface Engine { ... }

@Component
public class GasolineEngine implements Engine { ... }

@Component
public class ElectricEngine implements Engine { ... }

```

Code 6: Ausgangssituation

Wird die vorherige Klasse `Car` betrachtet, so kann festgestellt werden, dass die IDE schon warnt, dass zwei Kandidaten für die Injection vorhanden wären (Abbildung 5).

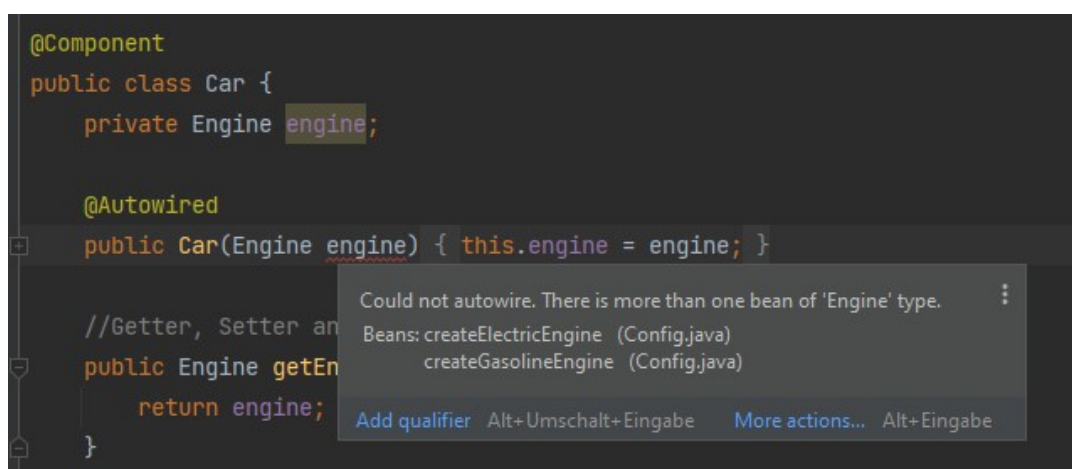


Abbildung 5: Mehrere Injection-Kandidaten verfügbar

Eine (**nicht empfohlene**) Möglichkeit dies zu beheben, ist die Umbenennung des Namens des Parameters der injected werden soll, wie Listing 7 zeigt.

```
@Component
public class Car {
    private Engine engine;

    @Autowired
    public Car(Engine electricEngine) {
        this.engine = electricEngine;
    }
    //Getter, Setter and other Boilerplate Code
}
```

Code 7: Umbenannter Konstruktorparameter

- **@Qualifier:** Diese Annotation wird verwendet um eindeutige Namen von Beans für die Dependency-Injection zu setzen, sowie dem dem Auto-Konfigurator mitzuteilen welche Bean injected werden soll. **Dabei kann diese Annotation nur für Methoden- und Field-Injection verwendet werden,** nicht aber für Konstruktor-Injection. Listing 8 zeigt dabei die Benennung einer Bean mittels dieser Annotation.

```
@Component
@Qualifier("electric")
public class ElectricEngine implements Engine { ... }
```

Code 8: Benennen einer Bean mit @Qualifier

Listing 9 demonstriert, dass auch die @Component-Annotation direkt für die Benennung einer Bean verwendet werden kann:

```
@Component("gasoline")
public class GasolineEngine implements Engine { ... }
```

Code 9: Benennen einer Bean mit @Component

Um nun in der Klasse Car festzulegen, welche Bean injected werden soll, wird auch die @Qualifier Annotation verwendet, wie in Listing 10 zu sehen ist.

```
@Component
public class Car {
    @Autowired
    @Qualifier("gasoline")
    private Engine engine;

    //Getter, Setter and other Boilerplate Code
}
```

Code 10: Markieren der zu verwendenden Bean

- **@Primary:** Diese Annotation kann verwendet werden, um eine bevorzugte Bean zum injecten festzulegen, sollten mehrere Implementierungen eines Interfaces vorhanden sein. Dabei wird die bevorzugte Klasse mit @Primary annotiert und somit wird die Unklarheit für SpringBoot beseitigt. Listing 11 zeigt wie eine solche Klasse annotiert wird und Abbildung 6, dass die Klasse car mit einem normalen @Autowired verwendet werden kann.

```
@Component
@Primary
public class DieselEngine implements Engine{ ... }
```

Code 11: Verwenden der @Primary Annotation anhand einer neuen Engine-Implementierung

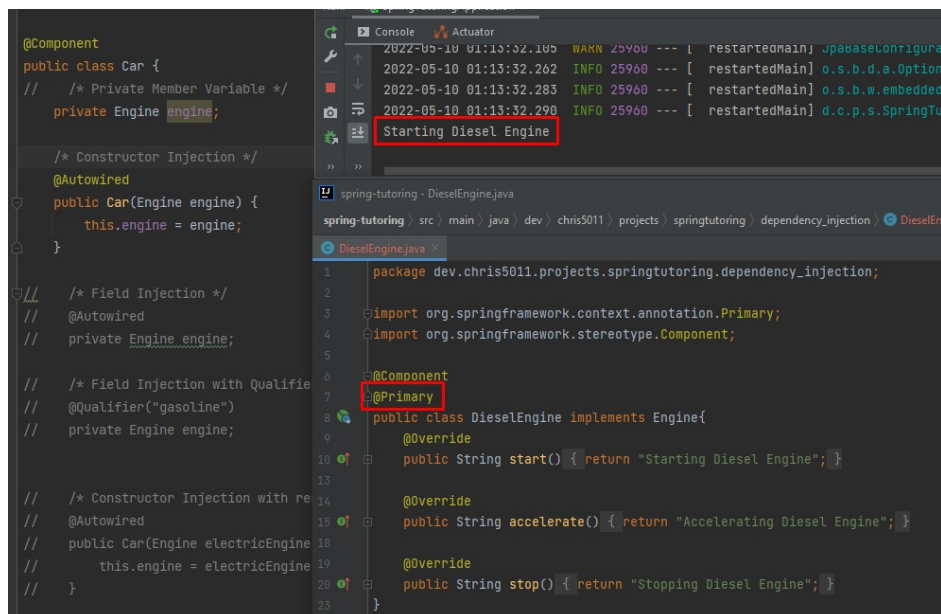


Abbildung 6: Verwendung der @Primary Annotation

- **@Value:** Kann dazu verwendet werden, um externe Ressourcen (Files, etc.) direkt in den Context der SpringBoot Applikation aufzulösen und zu laden. In Listing 12 wird dies durch das laden einer Textdatei und ausgeben des Contents in Abbildung 7 demonstriert.

```

@Component
public class TestValueAnnotation {

    @Value("src/main/resources/test.txt")
    private File textFile;

    public void run(){
        try(BufferedReader br = new BufferedReader(new FileReader(textFile))){
            System.out.println(br.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Code 12: Verwendung der @Value Annotation

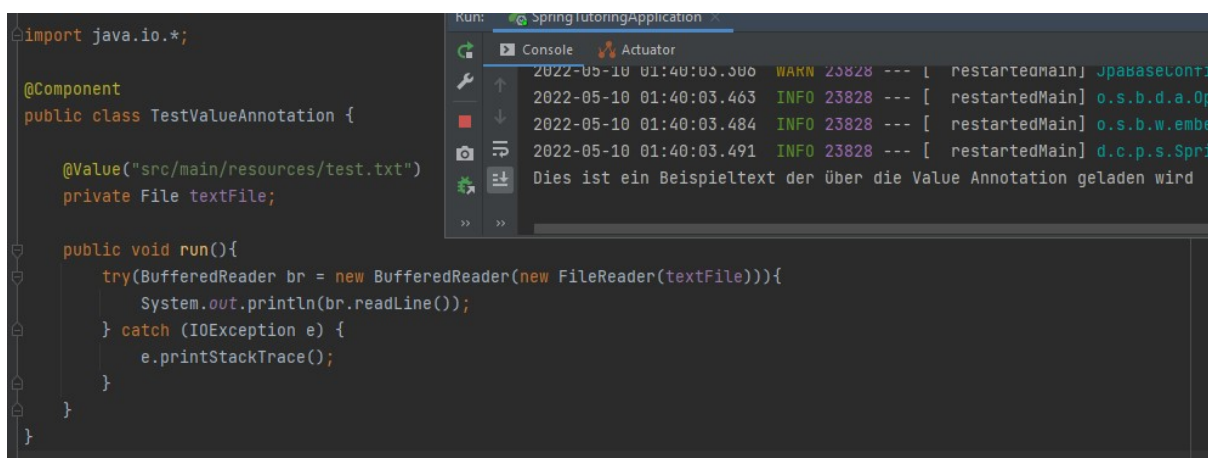


Abbildung 7: Verwendung der Value Annotation

- **@SpringBootTest:** Wird dazu verwendet, um Testklassen zu markieren in welchen die Dependency-Injection in Testfällen getestet werden kann.
- **@SpringBootApplication:** Diese Annotation wird dazu verwendet, um den Eintrittspunkt in eine Spring-Boot Applikation zu definieren. Dabei ist diese Annotation nur eine Zusammenfassung der Annotationen @EnableAutoConfiguration, @ComponentScan und @Configuration. Dabei kann die @SpringBootApplication Annotation auch durch die oben genannten ersetzt werden, wobei keine davon wirklich verpflichtend ist. Das bedeutet, soll die Applikation z.B. keinen Component-Scan durchführen kann die gleichnamige Annotation weggelassen werden.
- **@EnableAutoConfiguration:** Diese Annotation wird verwendet um die Auto-Configuration der SpringBoot Applikation durchzuführen, in dem der Scope der Konfiguration durch den Class-Path angenommen wird.

- **@Scope:** Diese Annotation wird verwendet um den Scope, also eine definierte Lebenszeit einer Bean festzulegen. Dabei gibt es die Folgenden Scopes:

**singleton:** Eine Instanz pro Spring-Context. (Default)

**prototype:** Bei jeder Injection wird ein neues Bean erzeugt.

**request:** Ein Bean wird pro HTTP-Request erstellt.

**session:** Ein Bean wird pro HTTP-Session erstellt.

Dabei kann der Scope wie in den Listings 13 und 14 anhand der Klasse `DieselEngine` gezeigt definiert werden.

```
@Component
@Scope("prototype")
public class DieselEngine implements Engine{ ... }
```

Code 13: Setzen des Scopes der Klasse `DieselEngine` per String

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class DieselEngine implements Engine{ ... }
```

Code 14: Setzen des Scopes der Klasse `DieselEngine` per Enum-Type

Dabei ist die zweite Methode zwar nicht so schön, jedoch ist sie sicherer, da wirklich der Java-Typ, und nicht nur der Name als String angegeben wird.

Der Unterschied zwischen dem Designpattern der GoF eines Singletons und dem Singleton-Scope von Spring ist, dass beim GoF-Pattern nur eine Instanz des Objekts pro JVM (Java Virtual Machine) existiert und beim Spring-Scopes es nur eine Instanz pro ApplicationContext gibt. Dabei können in einer JVM mehrere ApplicationContexte laufen, was bedeutet, dass es auch mehrere Instanzen eines Beans mit Scope Singleton geben kann.

Für eine umfangreichere Beschreibung der wichtigsten Annotationen, sowie ein Cheat-Sheet dafür ist auf der Seite **[www.jrebel.com](http://www.jrebel.com)** zu finden. Das dort verlinkte Cheat-Sheet bietet auch einen guten Überblick über den Scope, auf welchen die Annotationen angewandt werden können.

**Anmerkung:** Die Beschreibungen der Annotationen spezifisch für Spring-Web und Spring-WebMVC sind im Kapitel 2 - RESTful Webservices mit SpringBoot zu finden.

## 1.7 Der CommandLineRunner

SpringBoot bietet ein Feature, damit etwaiger Code beim Start der SpringBoot-Applikation ausgeführt wird. Dazu dient das Interface `CommandLineRunner`. Wird dies von einer Klasse implementiert, muss die Methode `public void run(String... args){ ... }` implementiert werden. Alles was in dieser Methode programmiert wurde, wird dann auch beim Start der SpringBoot-Applikation ausgeführt. Wichtig ist, dass die Klasse mit `@Component` annotiert ist, sonst findet der SpringBoot-Autokonfigurator diese nicht. Im Listing 15 befindet sich eine Beispielklasse die `CommandLineRunner` implementiert.

```
@Component
public class PrepareDatabase implements CommandLineRunner {

    private final CustomerRepository customerRepository;

    @Autowired
    public PrepareDatabase(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

    @Override
    public void run(String... args) {
        //Do Stuff
    }
}
```

Code 15: Beispiel eines CommandLineRunners



## 2 RESTful Webservices mit SpringBoot

Um RESTful Webservices mit SpringBoot zu erstellen, wird die Dependency `spring-boot-starter-web` benötigt.

### 2.1 Strukturierung einer SpringBoot Applikation

Die im Folgenden gezeigten Strukturierungen sind nicht verpflichtend (und sind oft auch unnötig für kleine Applikationen), jedoch ist es Best-Practice, eine dieser Strukturen so gut es geht einzuhalten.

#### 2.1.1 Strukturierung nach Layer

Wird nach dieser Struktur vorgegangen, wird das Konzept von J2EE stringenter verfolgt, indem die Applikation in verschiedene Schichten eingeteilt wird. Dabei sind diese Schichten (in vollem Umfang) in Abbildung 8 dargestellt.

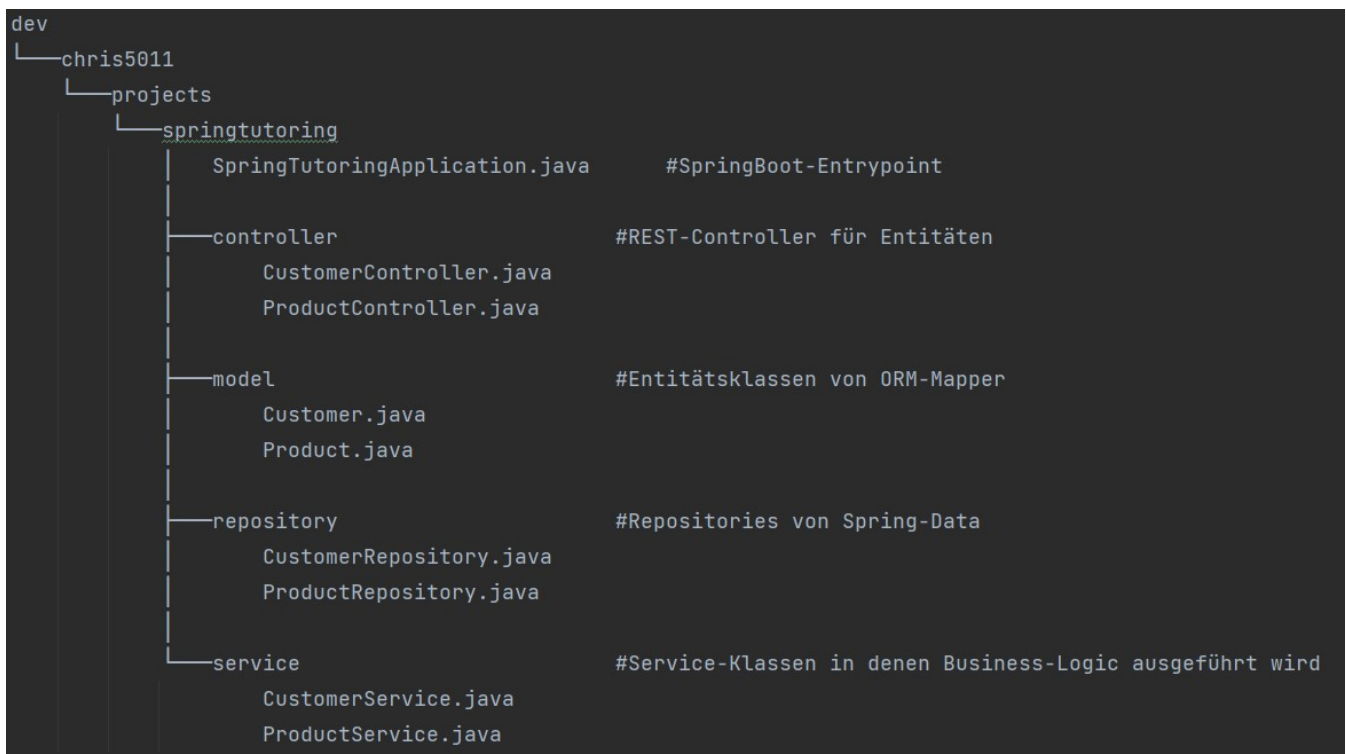


Abbildung 8: Strukturierung der Applikation nach dem Layer-Prinzip



Bei dieser Strukturierung befindet sich die Startklasse der SpringBoot-Anwendung im Hauptpaket der Applikation. In diesem Paket sollte sich sonst nichts befinden.

Unterpakete sollten zumindest für die Web-Controller, die Modellklassen und für die (Datenbank-) Repositories existieren. Wird die Applikation größer, so sollte auch ein Paket mit Service-Klassen erstellt werden, in welchem die Business-Logic auf die Daten aus den Repositories anwenden. In Abbildung 9 ist der Data-Flow in einer Anwendung die nach dem Layer-Prinzip gebaut ist dargestellt.

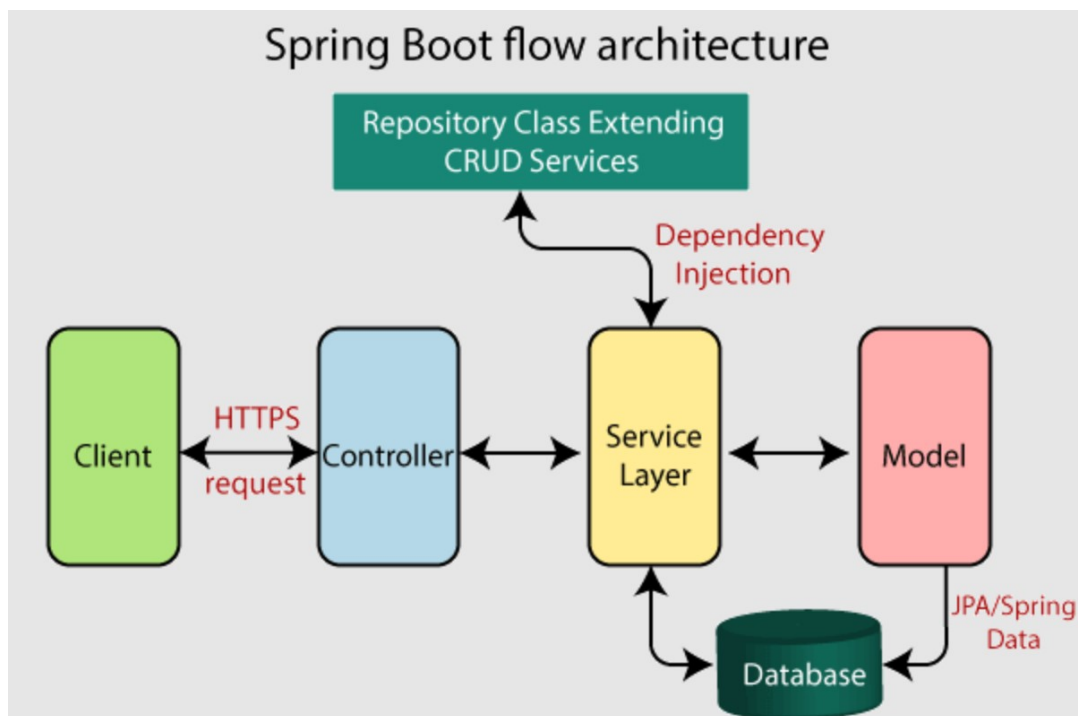


Abbildung 9: Dataflow in einer nach layer strukturierten Applikation

Source: [www.javatpoint.com](http://www.javatpoint.com)

### 2.1.2 Strukturierung nach Feature

Dieser Ansatz ist der Modernere der zwei. Dabei werden die Klassen in Pakete zu einzelnen Features zusammengefasst. Abbildung 10 verdeutlicht diesen Ansatz.

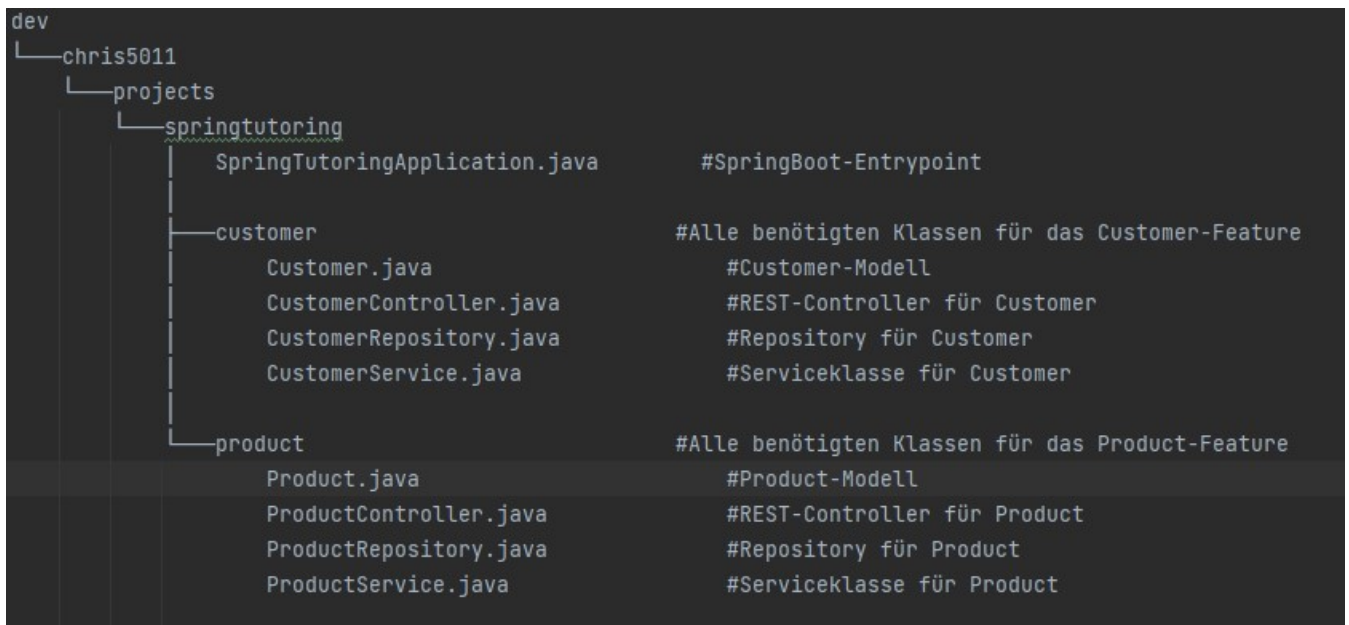


Abbildung 10: Strukturierung der Applikation nach dem Feature-Ansatz

Dieser Ansatz bringt die folgenden Vorteile:

- Features sind logisch in den Klassen aufgeteilt und dadurch leichter zu finden und zu ändern.
- Vereinfachung von Testen und Refactoring.
- Wird ein Paket gelöscht, so kann ein Feature aus der Applikation entfernt werden, ohne großem Refactoring Aufwand.
- Features können einfach einzeln released werden.

## 2.2 REST-Controller

Um ein REST-Webservice bzw. einen REST-Endpoint zu erstellen wird ein REST-Controller benötigt. Dieser kümmert sich um die eingehenden HTTP-Anfragen und leitet diese an die in ihm definierten Methoden je nach Request weiter. Um eine Klasse als REST-Controller zu definieren wird die Annotation `@RestController` benötigt. Diese ist eine spezielle Ausprägung von `@Controller`. In Listing 16 wird ein einfaches Beispiel eines REST-Controllers gezeigt.

```
@RestController
@RequestMapping("/hello-world")
public class ExampleRestController {

    @GetMapping("/hello")
    public String hello(){
        return "Hello BIS-20!";
    }

    /* Das obige Beispiel hätte auch wie folgt geschrieben werden können: */
    @RequestMapping(path = "/hello2", method = RequestMethod.GET)
    public String hello2(){
        return "Hello BIS-20, again!";
    }

    /* Hier wird der name über die Resource-Location in der URL (/URI) erfasst */
    /* Dabei wäre es in diesem Beispiel nicht notwendig, den namen des Parameters
       anzugeben, da sie im Mapping und in Java genau gleich heißen */
    @GetMapping("/hello/{name}")
    public String helloName(@PathVariable(name = "name") String name){
        return "Hello " + name;
    }
}
```

Code 16: REST-Controller Beispiel

Eine Übersicht der oben verwendeten Annotationen sowie weitere die speziell für (REST-) Controller verwendet werden können sind im Nachstehenden beschrieben:

- **@Controller:** Wird verwendet, um Klassen als generellen Web-Controller zu markieren. Dabei ist dieser dafür ausgelegt, generelle Web-Requests zu verarbeiten und Methoden HTTP-Content liefern sollen müssen explizit mit `@ResponseBody` annotiert werden.
- **@ResponseBody:** Wird verwendet um den Return-Typ von Methoden in einen HTTP-Body zu verwandeln. Dabei ist diese Funktionalität sehr gut geeignet, um normale Java Objekte in eine JSON-Darstellung z.B. im Browser zu verwandeln.
- **@RestController:** Diese Annotation ist eine Spezialisierung der `@Controller` Annotation und kapselt die beiden Annotationen: `@Controller` und `@ResponseBody`. Damit wird die Implementierung eines REST-Endpoints stark vereinfacht, da jede Methode in einer Klasse die mit `@RestController` annotiert wurde automatisch als eine REST-Methode konfiguriert wird (d.h. sie sind implizit mit `@ResponseBody` annotiert).
- **@RequestMapping:** Diese Annotation wird verwendet, um eine Methode auf eingehende HTTP-Requests zu binden. Das Standardattribut `value` gibt die zugehörige Route an. Das Attribut `method`

gibt bei einer Java-Methode die HTTP-Methode an, mit der diese REST-Methode aufgerufen werden kann. Diese Annotation ist auf Klassen- sowie auf Methodenebene anwendbar wie auch in obigen Beispiel gezeigt wird.

Spezialisierungen dieser Annotation sind:

**@GetMapping**: Konfiguriert eine Methode als **GET** REST-Endpoint.

**@PostMapping**: Konfiguriert eine Methode als **POST** REST-Endpoint.

**@PutMapping**: Konfiguriert eine Methode als **PUT** REST-Endpoint.

**@DeleteMapping**: Konfiguriert eine Methode als **DELETE** REST-Endpoint.

- **@PathVariable**: Wird verwendet, um einen teil der URI in Parameter für Java-Methoden zu verwandeln. Dafür muss ein Platzhalter im Pfad der in **@RequestMapping** definiert ist, vorhanden sein. Im unten stehenden Listing 17 ist eine Schnittstelle einer solchen Methode gezeigt:

```
@RequestMapping(path="/hello/{name}", method = RequestMethod.GET)
public String helloName(@PathVariable(name = "name") String name){...}
```

Code 17: Beispiel einer Methodenschnittstelle mit @PathVariable

- **@RequestParam**: Damit kann ein Parameter der REST-Methoden annotiert werden. Hier wird ein HTTP-Requestparameter übernommen. Wichtige Attribute sind **name** (Standard) und **required**.

Im begleitenden Github-Repository ([Link](#)) ist ein REST-Controller nach Best-Practice Ansatz implementiert und zeigt Beispiele wie ein Service in diesem verwendet werden kann. Dieser REST-Controller verwaltet eine Customer-API.

## 2.3 Verwendung von Exceptions mit dem REST-Controller

Wird in einer Methode die im REST-Controller aufgerufen wird, eine Exception geworfen, so wird der komplette StackTrace auf der Weboberfläche, die den Aufruf durchführte angezeigt. Dabei ist der HTTP-Status der HTTP-Response 500 - Internal Server Error. Um dies zu vermeiden, bzw. um genaue Informationen zu den Fehlern zu liefern, kann die Annotation `@ControllerAdvice` verwendet werden. Es ist ratsam einen eigenen `ExceptionHandler` zu erstellen, der sich nur um die Exceptions kümmert. Dabei ist diese Annotation eine eigene Spezifizierung der `@Component` Annotation, was bedeutet das der `ExceptionHandler` auch zusätzlich mit `@RestController` annotiert werden muss.

Ein solcher `ExceptionHandler` beinhaltet eine Catch-All Methode, um Exceptions die nicht speziell behandelt werden abzufangen und eine ordentliche Darstellung dieser zu liefern. Ein solcher `ExceptionHandler` mit einer Catch-All Methode, sowie eine Methode zur Behandlung von `CustomerNotFoundExceptions` wird in Listing 18 gezeigt. Dabei wird über die Annotation `@ExceptionHandler` definiert, um welche Exception sich die darunter befindliche Methode kümmern soll.

```
@ControllerAdvice
@RestController
public class ExceptionController extends ResponseEntityExceptionHandler {

    //Catch-All Methode für alle hier nicht behandelten Exceptions
    @ExceptionHandler(Exception.class)
    public ResponseEntity<ExceptionResponseModel> handleAllExceptions(Exception ex,
        WebRequest req) {
        ExceptionResponseModel eR = new ExceptionResponseModel(ex.getMessage(), req.
            getDescription(false), ex.getClass());
        eR.setTimestamp(LocalDateDateTime.now());
        return new ResponseEntity<>(eR, HttpStatus.INTERNAL_SERVER_ERROR);
    }

    /*
     * Behandelt die CustomerNotFoundExceptions. Sollte diese irgendwo im Programm
     * auftreten, so wird die Fehlermeldung der Exception mit Timestamp in das
     * ExceptionResponseModel geschrieben, welches dann mit dem HTTP-Status
     * "Not-Found" als HTTP-Response gesendet wird.
     */
    @ExceptionHandler(CustomerNotFoundException.class)
    public ResponseEntity<ExceptionResponseModel> handleUserNotFoundException(
        CustomerNotFoundException ex, WebRequest req) {
        ExceptionResponseModel eR = new ExceptionResponseModel(ex.getMessage(), req.
            getDescription(false), ex.getClass());
        eR.setTimestamp(LocalDateDateTime.now());
        return new ResponseEntity<>(eR, HttpStatus.NOT_FOUND);
    }
}
```

Code 18: Beispiel eines ExceptionControllers

In obigen Beispiel wird in jeder Methode ein eigens `ExceptionResponseModel` als HTTP-Response ausgeliefert. Dabei ist dies eine eigene Java-Klasse (zu sehen in Listing 19) und kann somit frei mit den benötigten Werten angepasst werden. In den Zeilen des `return` ist außerdem zu sehen, dass der HTTP-Status festgelegt werden kann.

```
public class ExceptionResponseModel {  
    private String message;  
    private String detail;  
    private LocalDateTime timestamp;  
    private Class exceptionClass;  
  
    //Constructors, Getter and Setter  
}
```

Code 19: Die ExceptionResponseModel-Klasse

### 3 Die H2 Database

Die H2 Datenbank-Engine wird oft für Spring-Applikationen eingesetzt, da diese neben den Features einer normalen Datenbank-Engine auch in-Memory betrieben werden kann, was bedeutet, dass kein Server dafür benötigt wird. Dadurch wird auch kein extra Management-Programm benötigt, da sie eine Browser- Based Console mitbringt. Weiter Vorteile und Eigenschaften sind im Nachfolgenden gelistet.

- Schnell und Open-Source
- Kann im Server- sowie im Embedded-Mode betrieben werden.
- Sehr geringe Größe der Dependency ( 2.5 MB)
- Bietet Volltextsuchen
- Datenbanken können bei Bedarf verschlüsselt werden

Um eine H2-Datenbank in einer Applikation zu betreiben muss die in Listing 20 gezeigte Dependency vorhanden sein. Dabei kann im Scope der Dependency auch angegeben werden, ob die Datenbank zur normalen Runtime, oder z.B. nur für Tests geladen werden soll.

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

Code 20: Dependency der H2-Datenbank

Um die Datenbank nun in einer Applikation einzusetzen, muss diese zuerst noch in der `application.properties` konfiguriert werden. Dabei werden die Einstellungen wie in Listing 21 gezeigt verwendet.

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled = true
```

Code 21: Settings für die H2-Datenbank

Für Debug-Zwecke lohnt es sich auch noch die Einstellung `spring.jpa.show-sql = true` in die `application.properties` zu schreiben. Diese Zeile dient dazu, dass zur Laufzeit alle etwaigen SQL-Statements die von SpringBoot ausgeführt werden auch auf der Konsole angezeigt werden. Damit sind die Einstellungen der H2-Datenbank abgeschlossen. Wird das SpringBoot-Projekt nun gestartet, so ist die Datenbank-Console standardmäßig unter `localhost:8080/h2-console` zu erreichen. Dabei muss der Port der hier angegeben wird, mit dem Server-Port in der `application.properties` übereinstimmen.

### 3.1 Speichern von Entitäten in einer H2 unter verwendung von JPA

Um nun Entitäten in der Datenbank zu speichern, müssen die Modell-Klassen der Objekte angepasst werden. Dabei muss eine Klasse die ein Objekt erstellt, das in einer Datenbank speichert mit `@Entity(name = "<tabellenname>")` annotiert werden. Nun braucht die Klasse eine definierte ID, nach welcher die Datenbank diese Objekte verwalten soll. Dazu wird ein Feld der Klasse mit `@Id` annotiert. Besteht die ID des Objekts aus einem natürlichen Attribut, welches eindeutig ist (z.B. bei einem Kunden der Username), so kann dies als ID verwendet werden. Sollte ein solches Attribut nicht vorhanden sein, sollte eine fortlaufende Nummerierung durchgeführt werden. Um dies nicht händisch machen zu müssen, gibt es die Annotation `@GeneratedValue(strategy = GenerationType.IDENTITY)`. Wird diese Annotation bei einem ID-Feld angebracht, so kümmert sich die Datenbank um die Verwaltung der ID's.

Weiters kann jedes weitere Feld in der Modellklasse, welches in der Datenbank aufscheinen soll, mit `@Column(name = "<spaltenname>")` markiert werden, um den Spalten auch eigene Namen geben zu können.

In Listing 22 ist eine Modellklasse für einen Kunden dargestellt, um das oben erklärte zu verdeutlichen.

```
@Entity(name = "customer")
public class Customer implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "cust_id")
    private int id;

    @Column(name = "cust_firstname")
    private String firstName;

    @Column(name = "cust_lastname")
    private String lastName;

    @Column(name = "cust_birthdate")
    private LocalDate birthdate;

    @Column(name = "cust_country")
    private String country;

    //Constructor, Getter, Setter, etc...
```

Code 22: Beispielhafte Modellklasse

Wird nun die SpringBoot Applikation gestartet, so wird diese Entitätsklasse „Forward-Engineered“, was bedeutet, aus dieser Klasse wird eine Tabelle in der Datenbank gemacht. Diese ist standardmäßig leer.

In z.B. einem CommandLineRunner können nun Instanzen von dieser Klasse erstellt werden, und über das Repository gespeichert werden. Dann scheinen diese erstellten Entitäten auch in der zugehörigen Tabelle der Datenbank auf. Dies ist in Listing 23 gezeigt.



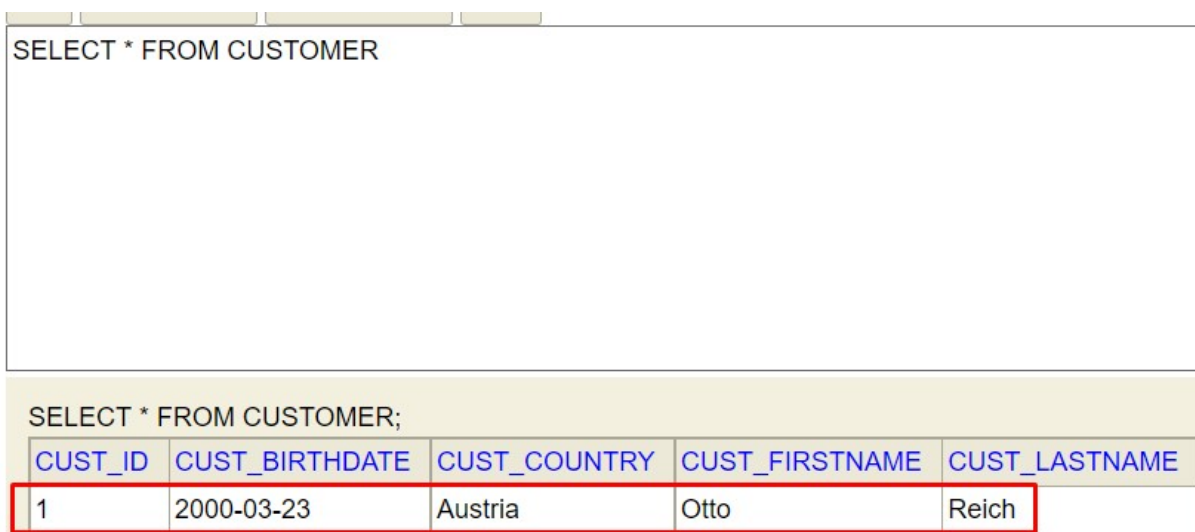
```
public class PrepareDatabase implements CommandLineRunner{
    private final CustomerRepository customerRepository;

    @Autowired
    public PrepareDatabase(CustomerRepository customerRepository, ProductRepository
        productRepository) {
        this.customerRepository = customerRepository;
    }

    @Override
    public void run(String... args) {
        customerRepository.save(new Customer("Otto", "Reich", LocalDate.of(2000, 3, 23), "
            Canada"));
    }
}
```

Code 23: Einfügen eines Customers im CommandLineRunner

Wird die Applikation nun ausgeführt und ein Blick in die Datenbank geworfen, so ist der im CommandLineRunner erstellte Kunde in der Datenbank zu sehen (Abb. 11).

The image shows a screenshot of a database query interface. At the top, the SQL query 'SELECT \* FROM CUSTOMER' is entered. Below the query, the results are displayed in a table. The table has five columns: CUST\_ID, CUST\_BIRTHDATE, CUST\_COUNTRY, CUST\_FIRSTNAME, and CUST\_LASTNAME. A single row of data is shown, with values 1, 2000-03-23, Austria, Otto, and Reich. The first row of the table is highlighted with a red border.

SELECT * FROM CUSTOMER;				
CUST_ID	CUST_BIRTHDATE	CUST_COUNTRY	CUST_FIRSTNAME	CUST_LASTNAME
1	2000-03-23	Austria	Otto	Reich

Abbildung 11: Abfrage der Tabelle Customer

### 3.2 Common Mistakes

Oft passiert es wenn die H2 das erste mal in einem Projekt gestartet wird, dass die vor ausgefüllte JDBC-URL nicht korrekt ist und ein Verbindungsversuch fehlschlägt. Dabei muss die JDBC-URL die im Konsolenfenster der H2 angezeigt wird, mit der in `application.properties` übereinstimmt. (Abb. 12)

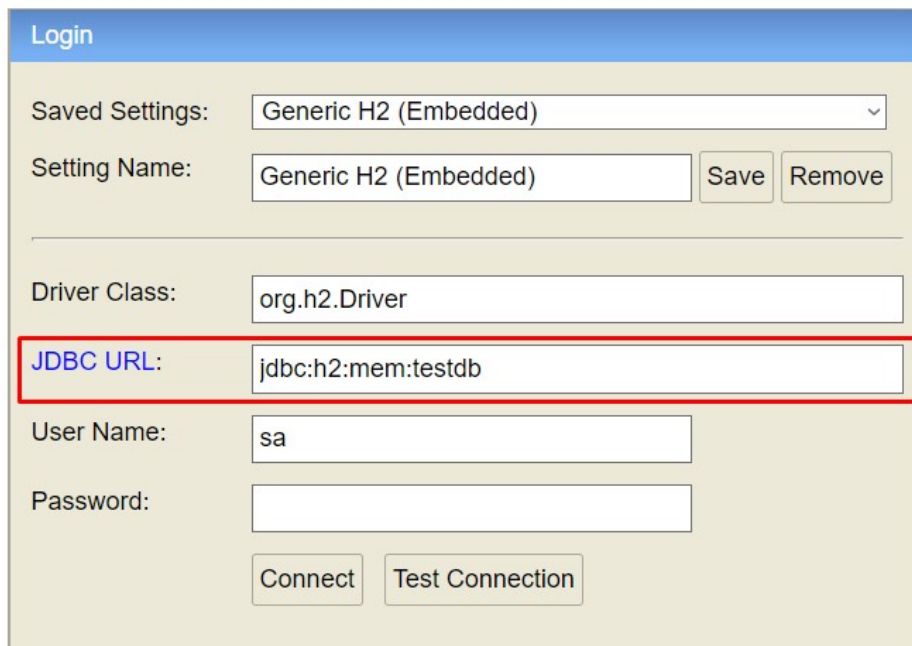
The image shows a screenshot of the H2 database configuration window. The window has a blue title bar with the text 'Login'. Below the title bar, there are several fields and buttons. The 'Saved Settings' field is a dropdown menu showing 'Generic H2 (Embedded)'. The 'Setting Name' field is a text box containing 'Generic H2 (Embedded)', with 'Save' and 'Remove' buttons to its right. Below these, there is a horizontal line. The 'Driver Class' field is a text box containing 'org.h2.Driver'. The 'JDBC URL' field is a text box containing 'jdbc:h2:mem:testdb', which is highlighted with a red rectangular border. Below this, there are 'User Name' and 'Password' fields, both containing 'sa'. At the bottom, there are 'Connect' and 'Test Connection' buttons.

Abbildung 12: Das Feld zum setzen der H2 JDBC-URL

## Abbildungsverzeichnis

1	Darstellung des Kern des Spring-Frameworks . . . . .	3
2	Konfigurieren eines SpringBoot Projekts mit dem Spring Initializr . . . . .	4
3	Platzierung der application.properties . . . . .	5
4	Lifecycle eines Spring-Beans . . . . .	6
5	Mehrere Injection-Kandidaten verfügbar . . . . .	9
6	Verwendung der @Primary Annotation . . . . .	11
7	Verwendung der Value Annotation . . . . .	12
8	Strukturierung der Applikation nach dem Layer-Prinzip . . . . .	15
9	Dataflow in einer nach layer strukturierten Applikation . . . . .	16
10	Strukturierung der Applikation nach dem Feature-Ansatz . . . . .	17
11	Abfrage der Tabelle Customer . . . . .	24
12	Das Feld zum setzen der H2 JDBC-URL . . . . .	25

## Quellcodeverzeichnis

1	Entry-Point einer Spring-Boot Applikation . . . . .	5
2	Schematische Implementierung des Lebenszyklus einer Spring-Bean . . . . .	7
3	Spring Constructor Injection . . . . .	8
4	Spring Setter Injection . . . . .	8
5	Spring Field Injection . . . . .	9
6	Ausgangssituation . . . . .	9
7	Umbenannter Konstruktorparameter . . . . .	10
8	Benennen einer Bean mit @Qualifier . . . . .	10
9	Benennen einer Bean mit @Component . . . . .	10
10	Markieren der zu verwendenden Bean . . . . .	10
11	Verwenden der @Primary Annotation anhand einer neuen Engine-Implementierung . . . . .	11
12	Verwendung der @Value Annotation . . . . .	12
13	Setzen des Scopes der Klasse DieselEngine per String . . . . .	13
14	Setzen des Scopes der Klasse DieselEngine per Enum-Type . . . . .	13
15	Beispiel eines CommandLineRunners . . . . .	14
16	REST-Controller Beispiel . . . . .	18
17	Beispiel einer Methodenschnittstelle mit @PathVariable . . . . .	19
18	Beispiel eines ExceptionControllers . . . . .	20
19	Die ExceptionResponseModel-Klasse . . . . .	21
20	Dependency der H2-Database . . . . .	22
21	Settings für die H2-Datenbank . . . . .	22
22	Beispielhafte Modellklasse . . . . .	23
23	Einfügen eines Customers im CommandLineRunner . . . . .	24