



# Eine Einführung in Java Spring Boot

## und die Erstellung von Enterprise Applications

Programmieren und Software Engineering  
HTL-Donaustadt, Abteilung für Informatik

Thomas Schlägl

The screenshot shows a web browser window titled "Spring-Boot-MVC-Thymeleaf-Web" with the URL "localhost:8080/bookmarks/get". The page displays a form for creating a new bookmark and a table listing existing bookmarks.

**Bookmarksverwaltungsprogramm mit Spring, JPA und Thymeleaf**

**Erstelle ein neues Bookmark**

Name:

Adresse:

Schlagwörter:

**Gespeicherte Bookmarks**

| Name           | Adresse                                  | Schlagwörter            | Erstellungszeitpunkt       |
|----------------|--|-------------------------|----------------------------|
| Baedlung       | https://www.baeldung.com                 | Java, Spring            | 2019-11-23T22:17:58.652119 |
| HTL-Donaustadt | http://www.hht-donaustadt.at             | HTL, Informatik         | 2019-11-23T22:18:26.161143 |
| ORF            | http://www.orf.at                        | doku, tv                | 2019-11-23T22:33:37.845498 |
| Instagram CCC  | https://www.instagram.com/codingcontest/ | Coding Contest          | 2019-11-23T22:34:24.826056 |
| Kurier         | http://www.kurier.at                     | sport, fußball, politik | 2019-11-23T22:44:50.170429 |
| Die Presse     | http://www.diepresse.at                  | Politik                 | 2019-11-23T22:48:44.626168 |

Am Weg zur ersten eigenen Webapplikation... Bitte umblättern!

# Inhaltsverzeichnis

|  |           |
|--|-----------|
| <b>1 Einleitung: Was Java Spring überhaupt ist</b>                         | <b>5</b>  |
| 1.1 Was bedeutet "Boot" in Java Spring Boot? . . . . .                     | 5         |
| 1.2 Enterprise Applications . . . . .                                      | 5         |
| 1.3 Was ist ein Application Framework? . . . . .                           | 6         |
| 1.4 Open Source Projekte im Enterprise Bereich . . . . .                   | 7         |
| 1.5 Die Geschichte wie es zu Java Spring kam . . . . .                     | 8         |
| 1.5.1 Java am Client: Applets . . . . .                                    | 8         |
| 1.6 Java am Server: Servlets . . . . .                                     | 9         |
| 1.7 Java Platform, Enterprise Edition (JEE) . . . . .                      | 9         |
| 1.7.1 Enterprise Java Beans (EJB) . . . . .                                | 9         |
| 1.7.2 Java Spring - Die Geburtsstunde . . . . .                            | 10        |
| 1.7.3 POJO - Plain Old Java Objects . . . . .                              | 11        |
| <b>2 Das Prinzip einer Webapplikation</b>                                  | <b>12</b> |
| 2.1 Arten von HTTP-Requests . . . . .                                      | 13        |
| 2.2 Die vier CRUD-Operationen . . . . .                                    | 14        |
| <b>3 Das Architekturmmodell einer Webapplikation</b>                       | <b>15</b> |
| 3.1 Presentation Layer (Web Layer) . . . . .                               | 16        |
| 3.2 Service Layer . . . . .  | 16        |
| 3.3 Domain Layer . . . . .   | 16        |
| 3.4 Persistence Layer . . . . .  | 17        |
| 3.5 Subsysteme . . . . .   | 17        |
| <b>4 Eine erste Webapplikation als REST-API: Hi ...! Nice to meet you.</b> | <b>18</b> |
| 4.1 Anlegen des Projekts . . . . .   | 18        |
| 4.2 Vorgangsweise bei IntelliJ . . . . .                                   | 18        |
| 4.3 Starten der Webapplikation . . . . .                                   | 20        |
| 4.4 Wie bringt man unserer Webapplikation Grüßen bei? . . . . .            | 21        |
| 4.4.1 Anlegen einer Controller-Klasse . . . . .                            | 21        |
| 4.4.2 Implementierung des Grüßens . . . . .                                | 22        |
| 4.4.3 Ausprobieren des Codes . . . . .                                     | 23        |
| 4.5 Was Spring mit der Controller-Klasse tut . . . . .                     | 23        |
| 4.5.1 Annotations vor Klassen . . . . .                                    | 24        |
| 4.5.2 Annotations vor Methoden . . . . .                                   | 24        |
| 4.5.3 Annotations vor Methodenparametern . . . . .                         | 24        |
| 4.5.4 Rückgabewert der Controller-Methoden . . . . .                       | 25        |
| 4.6 Die Verwendung von HTTP-Parametern . . . . .                           | 25        |
| 4.6.1 Testen der Controller-Methode mit HTTP-Parameter . . . . .           | 26        |
| 4.7 Übungsaufgaben . . . . .   | 27        |
| <b>5 Eine "echte" Webapplikation: Spring MVC (Model-View-Controller)</b>   | <b>28</b> |
| 5.1 Frontend: Bootstrap und Thymeleaf . . . . .                            | 28        |
| 5.2 Die Buchklasse des Domain-Layers . . . . .                             | 28        |
| 5.3 Die Controllerklasse im Presentation-Layer . . . . .                   | 29        |

|           |  |           |
|-----------|--|-----------|
| 5.4       | Die Projektstruktur . . . . .  | 30        |
| 5.5       | Das HTML-File mit Bootstrap und Thymeleaf . . . . .                          | 30        |
| 5.6       | Testen der Anwendung . . . . .   | 32        |
| 5.7       | Übungsaufgaben . . . . .   | 32        |
| <b>6</b>  | <b>Ein Rest-Controller mit Post-, Put-, Get- und Delete-Requests</b>         | <b>33</b> |
| 6.1       | Erstellen des Projekts . . . . .   | 33        |
| 6.2       | Die Buchklasse des Domain-Layers . . . . .                                   | 33        |
| 6.3       | Der Rest-Controller des Presentation-Layers . . . . .                        | 33        |
| 6.4       | Testen der Http-Requests mit Postman . . . . .                               | 34        |
| <b>7</b>  | <b>Die Grundprinzipien von Spring</b>  | <b>37</b> |
| 7.1       | Application Context: Dort wo die Beans herkommen . . . . .                   | 37        |
| 7.1.1     | Ausgabe aller Beans beim Programmstart . . . . .                             | 39        |
| 7.1.2     | Zugriff auf vom Application Context erstellte Beans . . . . .                | 40        |
| 7.1.3     | Der Scope eines Beans (@Scope) . . . . .                                     | 41        |
| 7.2       | Dependency Injection (DI) . . . . .  | 42        |
| 7.2.1     | Arten von Dependency Injection . . . . .                                     | 42        |
| 7.2.2     | Informationen zu Dependency Injection . . . . .                              | 42        |
| 7.2.3     | Beispiel zur Dependency Injection: Das Sortieren von Büchern . . . . .       | 43        |
| 7.2.4     | Constructor Injection im Codebeispiel . . . . .                              | 47        |
| 7.2.5     | Merkregel: Der häufigste Fall von Constructor Injection . . . . .            | 50        |
| 7.2.6     | Setter Injection im Codebeispiel . . . . .                                   | 51        |
| <b>8</b>  | <b>Das Buildtool Maven</b>   | <b>52</b> |
| 8.1       | Übungen zu Maven . . . . .   | 54        |
| <b>9</b>  | <b>Loggen in Java mit s14j.Logger</b>  | <b>55</b> |
| 9.1       | Importieren der Logger-Klasse . . . . .                                      | 55        |
| 9.2       | Anlegen einer Loggerinstanz . . . . .  | 55        |
| 9.3       | Loggen von Variablenwerten . . . . .   | 55        |
| <b>10</b> | <b>SQL-Datenbanken anbinden mit Spring JDBC</b>                              | <b>56</b> |
| 10.1      | H2 Datenbank . . . . .   | 56        |
| 10.2      | Anlegen des Projekts . . . . .   | 56        |
| 10.3      | Die Application Properties ( <code>application.properties</code> ) . . . . . | 56        |
| 10.4      | Starten des Projekts . . . . .   | 57        |
| 10.5      | Anlegen und Befüllen einer Datenbanktabelle . . . . .                        | 58        |
| 10.6      | Erstellen einer Personenklasse im Domain Layer . . . . .                     | 60        |
| 10.7      | Implementierung des Persistence Layers mit JDBC . . . . .                    | 60        |
| 10.7.1    | SQL-Parameter Injection . . . . .  | 61        |
| 10.8      | Die JDBC-Repository Klasse . . . . .   | 62        |
| 10.8.1    | Implementierung eines RowMappers für Personen . . . . .                      | 63        |
| 10.9      | Testen der JDBC-Repository Klasse in der Application . . . . .               | 63        |
| 10.10     | Übungsaufgabe . . . . .  | 66        |
| 10.10.1   | Rest-Webservice API . . . . .  | 66        |
| 10.10.2   | Webservice . . . . .   | 66        |

|  |           |
|--|-----------|
| 10.10.3 HTML-Rendering der Daten . . . . .                                   | 67        |
| <b>11 SQL-Datenbanken verwenden mit Spring JPA</b>                           | <b>68</b> |
| 11.1 Entity Manager . . . . .  | 68        |
| 11.1.1 Nützliche Settings in <code>application.properties</code> . . . . .   | 69        |
| 11.2 Codebeispiel: Bookmark Verwaltung mit JPA und H2 . . . . .              | 69        |
| 11.3 Eine Bookmark Klasse im Domain-Layer . . . . .                          | 70        |
| 11.4 Ein Bookmark-Repository im Persistence-Layer . . . . .                  | 72        |
| 11.5 Die Klasse Bookmark Service der Businesslogik . . . . .                 | 74        |
| 11.6 Die Controller-Klasse im Presentation Layer . . . . .                   | 74        |
| 11.7 Das Rendern der Bookmarks durch Thymeleaf . . . . .                     | 76        |
| 11.8 Das File <code>error.html</code> zur Ausgabe von Fehlern . . . . .      | 77        |
| 11.9 Das File <code>style.css</code> zur Verschönerung der Ausgabe . . . . . | 78        |
| 11.10 Einstellungen für eine persistente H2-Datenbank . . . . .              | 78        |
| 11.11 Verwendung der Webapplication . . . . .                                | 78        |
| 11.12 Ausgabe einer Fehlerseite . . . . .                                    | 78        |
| 11.13 Zusammenfassung . . . . .  | 78        |
| <b>12 Aspect Oriented Programming (AOP)</b>                                  | <b>79</b> |
| 12.1 Performance Messungen . . . . .   | 80        |
| 12.2 Weitere AOP-Annotations . . . . .                                       | 81        |
| <b>13 Actuator</b>   | <b>82</b> |
| <b>14 Devtools</b>   | <b>83</b> |
| <b>Listings und Abbildungen</b>  | <b>84</b> |

# 1 Einleitung: Was Java Spring überhaupt ist

Java Spring ist ein sogenanntes *Application Framework*, mit dem man komplexe Applikationen entwickeln kann, wie sie typischerweise in großen Unternehmen verwendet werden. Solche Applikationen werden als **Enterprise Applications** bezeichnet.

## Java Spring

Java Spring ist ein Application Framework zur Entwicklung von Enterprise Applications.

Java Spring ist Open Source.

<https://spring.io>

## 1.1 Was bedeutet "Boot" in Java Spring Boot?

Seit 2012 gibt es eine Erweiterung für das Java Spring Framework, die als **Java Spring Boot**<sup>1</sup> bezeichnet wird. Java Spring Boot erleichtert zusätzlich die Verwendung des Spring Frameworks massgeblich.

## Java Spring Boot

Java Spring Boot ist eine Erweiterung von Java Spring die in vielen Entwicklungsabteilungen für die Entwicklung von Enterprise Applications verwendet werden.

Es ist **DAS** gebräuchlichste Framework und daher Thema dieses Skriptums.

<https://spring.io/projects/spring-boot>

## 1.2 Enterprise Applications

Als **Enterprise Applications (EA)** bezeichnet man alle Programme wie sie typischerweise in großen Unternehmen verwendet werden und vollständige Geschäftsprozesse abbilden. Sie dienen grundsätzlich dafür um die Produktivität und die Effizienz in Unternehmen zu erhöhen.

Typische Beispiele für Enterprise Applications sind:

- **Billing Systeme** (Bezahlsysteme)
- **Customer Relations Management Systeme**: Das sind Systeme die Kundenbeziehungen abbilden. Sie kommen beispielsweise im Online-Handel, im Versicherungs- oder Bankenbereich zum Einsatz. Andere Beispiele sind das Patientenmanagement in Gesundheitseinrichtungen, die Userverwaltung im Gaming Bereich und Marketingsysteme.
- **Unternehmensinterne Systeme** die den Daten- und Wissensaustausch von unterschiedlichen Abteilungen des Unternehmens ermöglichen und unterstützen.
- **Smart Home Systeme** die personenbezogene Daten speichern oder verarbeiten, wie sie beispielsweise über Sensoren im Haus oder Spracheingaben (Alexa, Google Home, ...) erfasst werden.

<sup>1</sup><https://spring.io/projects/spring-boot>

Java Spring ist ein Open Source Projekt und kostenlos verfügbar. Es gibt auch zahlreiche kommerzielle Produkte die die Entwicklung von Enterprise Applications unterstützen. Sie enthalten typischerweise einen Webserver. Beispiele dafür sind:

- BEA Web Logic
- IBM Websphere
- Oracle IAS (Internet Application Server)

Andere Frameworks für Enterprise Applications, die so wie Java Spring Open Source sind, sind Wildfly bzw. JBoss.

### 1.3 Was ist ein Application Framework?

Ein Application Framework in der Softwareentwicklung stellt Softwaremodule zur Verfügung, die einen Grundstock an Funktionalität anbieten, die der Entwickler/die Entwicklerin verwenden und erweitern kann. Ein Framework ist dementsprechend weder ein fertiges Programm, noch eine reine Sammlung von Klassen.

Solche Module werden in Java Spring als *Spring Module* bezeichnet und decken verschiedenste Aspekte von Anwendungen ab. Beispiele für solche Aspekte sind:

- Persistente Datenspeicherung in Datenbanken (Spring Data)
- Realisierung einer Web-API, also einer Webschnittstelle (Spring Web Services)
- Realisierung einer Model-View-Controller-Architektur (Spring MVC)
- Anbindung von Sozialen Netzwerken (Spring Social)
- Security, also Sicherheitsaspekte (Spring Security)

#### Application Framework vs. Software Development Kit

Ein **Application Framework** ist modular aufgebaut und ermöglicht es den EntwicklerInnen diese Module für eine Applikation gezielt auszuwählen.

Die größten Vorteile der Verwendung eines Frameworks sind ein schlanker Quellcode und ein geringer Aufwand für Programmanpassungen wenn sich die Anforderungen ändern.

Durch Spring Module wird allgemeine Funktionalität zur Verfügung gestellt und der Entwickler/die Entwicklerin muss nur den für die Applikation spezifischen Programmcode implementieren.

Ein **SDK (Software Development Kit)** ist dagegen eine Menge von Hilfsklassen und Hilfstoools. Ein Beispiel für ein SDK ist das JDK, das Funktionalität für Java zur Verfügung stellt.

Application Frameworks sind also gerade für die Verknüpfung von komplexen Funktionalitäten äußerst hilfreich und reduzieren die Implementierungsaufwände und -kosten drastisch, wenn die EntwicklerInnen in das Framework eingearbeitet sind.

Rund um Java haben sich ab dem Jahr 2000 einige Application Frameworks entwickelt, die zum großen Teil wieder verschwunden sind.

Java Spring ist das mit Abstand populärste dieser Frameworks, das von einer sehr lebendigen Open Source Szene auch laufend erweitert wird.

## 1.4 Open Source Projekte im Enterprise Bereich

Spring ist ein Beispiel für ein außerordentlich erfolgreiches Open Source Projekt. Die Liste aller Contributors ist öffentlich zugängig <https://spring.io/team>.

Darunter findet sich ein Österreicher an prominenter Stelle. **Jürgen Höller** ist ein Linzer, ein Co-Founder des Projekts und seit langer Zeit Project Lead.



Abbildung 1: Jürgen Höller auf der SpringOne Platform 2018 in Washington, D.C. (Klick zum Video)

**Michael Nitschinger** ist Absolvent der HTL-Donaustadt und als Principal Software Engineer bei Couchbase (<https://info.couchbase.com>) tätig. Couchbase ist eine High-Performance noSQL-Database und ebenso ein Open Source Project.



Abbildung 2: Michael Nitschinger auf der SPARK Summit 2016 in Brüssel (Klick zum Video)

## 1.5 Die Geschichte wie es zu Java Spring kam

Java und alle damit verbundenen Technologien waren von Beginn an für Software am Server gedacht. Der Start von Java in Verbindung mit dem Internet, vollzog sich etwas holprig, weil die frühen Browser Java gar nicht gleich unterstützten und die Internetanschlüsse damals nur unzureichende Bandbreiten boten.

### 1.5.1 Java am Client: Applets

So richtig aufmerksam wurde man auf Java Ende der 1990er Jahre durch die sogenannten Java-Applets<sup>2</sup>. Java-Applets waren Anwendungen die beim Browsen im Internet vom Server auf den Clients heruntergeladen wurden und im Browser ausgeführt wurden. Dies wurde beispielsweise häufig für Demonstrationen von Algorithmen im Internet verwendet.

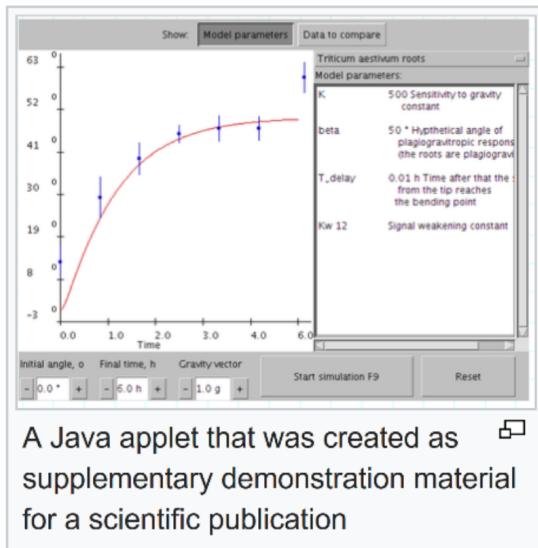


Abbildung 3: Screenshot eines Applets, Quelle: [https://en.wikipedia.org/wiki/Java\\_applet](https://en.wikipedia.org/wiki/Java_applet)

Java-Applets werden von modernen Browsern seit einigen Jahren nicht mehr unterstützt, weil die Technologie diverse Nachteile hat. Diese sind allen voran grobe Securitybedenken, da bei dieser Technologie lauffähiger Code auf den Client heruntergeladen und ausgeführt wird. Ein Artikel der diesen fatalen Nachteil beschreibt ist <https://www.mach.de/entwicklungs-blog-artikel/applets-und-ihre-fehlende-zukunft>.

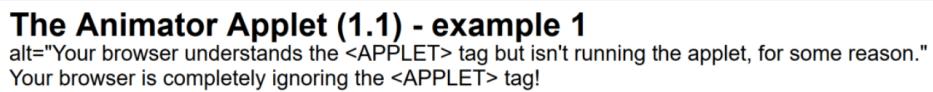


Abbildung 4: Browser Response beim Erkennen eines Applet Tags

Applets sind heutzutage von keinem seriösen Browser mehr ausführbar.

<sup>2</sup><https://de.wikipedia.org/wiki/Java-Applet>

## 1.6 Java am Server: Servlets

Die ersten Java-Programme am Server waren Java-Servlets<sup>3</sup>, die Pendants zu den Applets. Diese sind auch heute noch in breiter Verwendung, wenngleich sie auch bei weitem nicht mehr die einzigen serverseitigen Softwarekomponenten sind, so wie sie es in den Anfangstagen der Java-Serverentwicklungen waren.

Ein Servlet ist eine Klasse in Java die das Interface *javax.servlet.Servlet* implementiert. Servlets laufen am Server in eigenen Laufzeitumgebungen, in sogenannten Containern ab.

Trotz der großen Bedeutung von Java für den Serverbereich, muss festgehalten werden, dass gut und gerne rund 75 % aller Websites heutzutage die Skriptsprache PHP verwenden. Dafür sind die geringeren Entwicklungskosten verantwortlich.

Bei Enterprise Applications, für den Bankenbereich oder bei großen Content-Management-Systemen, sind im großen Maße Java-Entwicklungen üblich.

Die Vorteile von Java-Webanwendungen sind die Plattformunabhängigkeit und die einfache Integrationsfähigkeit zu Unternehmenssoftware.

## 1.7 Java Platform, Enterprise Edition (JEE)

Im Gegensatz zur Java Standard Edition (Java SE oder **JSE**), die für allgemeine Anwendungen (Desktop, Server) gedacht ist, ermöglicht die JEE die Entwicklung verteilter (distributed) und mehrstufiger (multi-tier) Enterprise Applications.

Die JEE ist also gewissermaßen der "große Bruder" der JSE.

**Java Platform, Enterprise Edition**, abgekürzt **Java EE**, ist die **Spezifikation** einer **Softwarearchitektur** für die **transaktionsbasierte** Ausführung von in **Java** programmierten Anwendungen und insbesondere **Webanwendungen**. Sie ist eine der großen **Plattformen**, die um den **Middleware**-Markt kämpfen. Größter Konkurrent ist dabei die **.NET-Plattform** von **Microsoft**.

Abbildung 5: Beschreibung der Java Enterprise Edition, Quelle: [https://de.wikipedia.org/wiki/Enterprise\\_Java](https://de.wikipedia.org/wiki/Enterprise_Java) (Stand: 2019-04-08)

Eine gut lesbarer Übersichtsartikel zum Aufbau der JEE ist <https://www.it-talents.de/blog/it-talents/was-ist-die-java-enterprise-edition>.

### 1.7.1 Enterprise Java Beans (EJB)

In Laufe der Zeit wurde zusätzlich zu den Java Servlets ein weiteres Komponentenmodell entwickelt, nämlich das der sogenannten Enterprise Java Beans.

Ein Enterprise Java Bean ist die Instanz einer Java Klasse die nur serverseitig verwendet wird und wesentlichen Erfordernissen einer Applikationsanwendung Rechnung trägt. Das EJB-Java-Package

<sup>3</sup><https://de.wikipedia.org/wiki/Servlet>

`javax.ejb` ist in <https://docs.oracle.com/javaee/6/api/javax/ejb/package-summary.html> dokumentiert. Enterprise Java Beans werden in sogenannten EJB-Containern ausgeführt und nicht in der JVM (=Java Virtual Machine).

## Enterprise JavaBeans

**Enterprise JavaBeans (EJB)** sind **standardisierte Komponenten** innerhalb eines **Java-EE-Servers** (Java Enterprise Edition). Sie vereinfachen die Entwicklung komplexer **mehrschichtiger verteilter Softwaresysteme** mittels **Java**. Mit Enterprise JavaBeans können wichtige Konzepte für Unternehmensanwendungen, z. B. Transaktions-, Namens- oder Sicherheitsdienste, umgesetzt werden, die für die **Geschäftslogik** einer Anwendung nötig sind.

Abbildung 6: Begriffserklärung Enterprise Java Beans, Quelle: [https://de.wikipedia.org/wiki/Enterprise\\_JavaBeans](https://de.wikipedia.org/wiki/Enterprise_JavaBeans) (Stand: 2019-04-08)

Enterprise-Anwendungen die in der JEE ablaufen und die EJBs verwenden, werden – analog zu den *jar*-Files der Java Standard Edition – in Enterprise Archive-Files gepackt, die die File Extension *ear* haben. Eine Beschreibung findet sich auf Wikipedia [https://de.wikipedia.org/wiki/Enterprise\\_Archive](https://de.wikipedia.org/wiki/Enterprise_Archive).

### Enterprise Development vor 2002

Enterprise Java Beans waren vor 2002 (fast) die einzige Alternative um Enterprise Applications unter Java zu entwickeln.

Die Entwicklung von Enterprise Java Beans erfordert sehr viel Expertenwissen, weil jede Klasse eines Beans verschiedensten Aspekten einer vernetzten sicheren Anwendung Rechnung tragen müssen. Der Implementierungsaufwand ist beträchtlich.

Eine typische Enterprise Java Bean-Klassen ist riesig, weil sie unzählige Interfaces implementieren und daher schwer zu warten ist.

### 1.7.2 Java Spring - Die Geburtsstunde

Rod Johnson und Jürgen Höller arbeiteten an der Entwicklung einer alternativen Architektur zu den Enterprise Java Beans.

Rod Johnson veröffentlichte 2002 das Buch *Expert One-on-One J2EE<sup>4</sup> Development without EJB*, das die Geburtsstunde von Java Spring einläutete. Das Java Spring Framework eröffnete einen vollständig anderen Weg der Entwicklung von Enterprise Applications.

<sup>4</sup>Hinweis: J2EE ist die damals gebräuchliche Abkürzung für die Java Enterprise Edition JEE

Einen lesenswerten Artikel zu diesem Buch ist <https://www.theserverside.com/news/1365316/J2EE-Without-EJB>.

### 1.7.3 POJO - Plain Old Java Objects

Im Gegensatz zu den aufwendigen Enterprise Java Beans-Klassen verwendet der Java Spring Ansatz sogenannte **POJOs**<sup>5</sup> (=Plain Old Java Object).

POJO ist ein Ausdruck, der für ganz "normale" Java Klassen eingeführt wurde. Der Begriff ist als Gegenentwurf zu den aufwendigen Enterprise Java Beans zu verstehen.

Java Klassen, die POJOs sind, ...

- ... müssen nicht unbedingt von anderen Klassen abgeleitet sind oder Interfaces implementieren.
- ... müssen public sein.
- ... können jeden beliebigen Access Modifier für Methoden und Instanzvariablen verwenden.
- ... können oder können auch nicht Getter und Setter Methoden haben.
- ... können einen Konstruktor mit Parameter haben.

---

<sup>5</sup>[https://de.wikipedia.org/wiki/Plain\\_Old\\_Java\\_Object](https://de.wikipedia.org/wiki/Plain_Old_Java_Object)

## 2 Das Prinzip einer Webapplikation

Eine Webapplikation läuft natürlich am Server ab. Die Kommunikation mit einem Programm, das am Client läuft erfolgt über HTTP-Requests.

Mit jedem Request des Clients können Daten und Parameter zum Server geschickt werden.

Webapplikationen haben häufig das Ziel Ressourcen, die am Server gespeichert sind, zu erzeugen, zu modifizieren, zu löschen und Informationen über sie dem Client zur Verfügung zu stellen.

Nach der Verarbeitung der Daten durch den Server, kann in einem HTTP-Response Daten zurück-schicken. Ein HTTP-Return-Code bzw. ein HTTP-Error-Code gibt dem Client Auskunft darüber, ob alles geklappt hat oder ob ein Fehler aufgetreten ist.

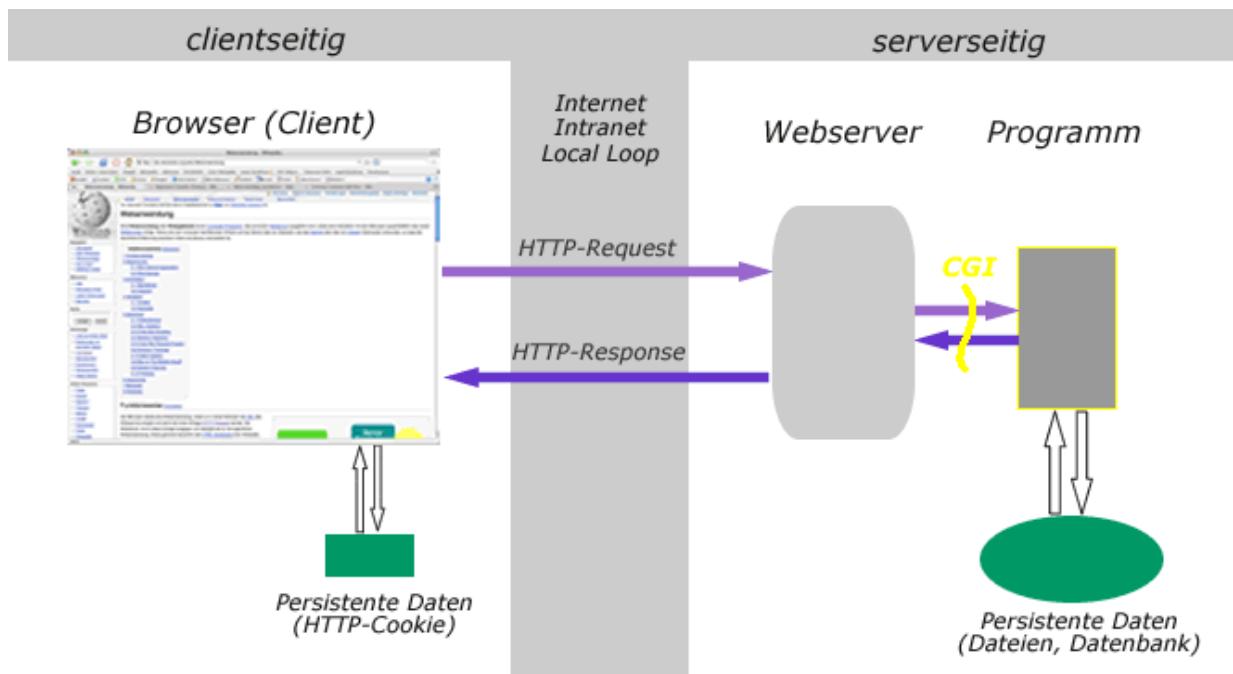


Abbildung 7: Prinzip einer Webapplikation

Quelle: <https://de.wikipedia.org/wiki/Webanwendung> (Stand 2019-04-08)

In Java Spring wird der Webserver, nämlich der Tomcat Server im JAR-File eingebettet. D. h. er muss nicht separat am Webserver installiert werden. Das ist ein großer Vorteil im Vergleich zum früheren Entwicklungsprozess vor Java Spring.

**Beispiel:** Ein Entwicklungsteam erstellt eine Webapplikation und möchte sie auf Linux deployen.

Früher (ohne Spring):

- Linux am Server installieren
- Java am Server installieren
- WebServer installieren, z. B. Tomcat
- War-File auf den Server deployen

Heute (mit Spring Boot):

- Linux am Server installieren
- Java am Server installieren
- Jar-File deployen mit einem embedded Server, z. B. Tomcat.

## 2.1 Arten von HTTP-Requests

In diesem Skriptum betrachten wir vier aller möglichen HTTP-Requests. Eine sehr einfach Kurzbeschreibung aller Arten von HTTP-Requests befindet sich z. B. auf <https://wiki.selfhtml.org/wiki/HTTP/Anfragemethoden>.

Die vier wichtigsten HTTP-Requests sind:

- **Get-Request:** Dient zur Anforderung von Informationen über Ressourcen vom Server. Die Auswahl der gewünschten Daten erfolgt zumeist über HTTP-Parameter die mit ? und & mitgeschickt werden können.  
Im Prinzip natürlich auch die URL selbst für die Übertragung der Werte herangezogen werden.
- **Post-Request:** Dienen dazu an den Server Daten zu schicken, die er abspeichert oder weiterverarbeitet. Im Allgemeinen werden dabei Ressourcen am Server erzeugt oder gelegentlich auch modifiziert. Bei einem Post-Request können auch große Datenmengen zum Server geschickt werden, z. B. Bilddaten.
- **Put-Request:** Ist ähnlich zum Post-Request. Put sollte dann verwendet werden, wenn die Ressource selbst bereits existiert, d. h. für das Updaten von Ressourcen.  
Kevin Sookocheff beschreibt in seinem Artikel <https://sookocheff.com/post/api/when-to-use-http-put-and-http-post/> den Unterschied zwischen Post und Put.  
Es gibt auch einen **Patch-Request**, der für das Aktualisieren von Daten verwendet werden kann, wobei ein solcher nur die Änderung einer Ressource enthalten soll.
- **Delete-Request:** Dient immer dazu, am Server Ressourcen zu löschen.

Die oben erklärten HTTP-Operationen stehen in einem losem Zusammenhang mit den vier CRUD-Operationen, die im nächsten Kapitel beschrieben werden.

Im Zusammenhang mit den HTTP-Requests liest man in der Literatur auch oft vom sogenannten **Verb** eines Requests. Damit ist lediglich die Art des Requests gemeint, nämlich GET, POST, PUT und DELETE.

## 2.2 Die vier CRUD-Operationen

In jeder realen Webapplikation werden bestimmte Dinge gespeichert, die angelegt, ausgelesen, verändert und gelöscht werden können. Beispielsweise in einem Online Shop.

Bestimmte Teile von Webapplikationen realisieren häufig die sogenannten CRUD-Operationen. Diese vier Operationen sind die Grundaktionen von Klassen und Softwaremodulen die Ressourcen persistent speichern und verwalten, sogenannten **Repositories**.

Die Tabelle gibt eine Übersicht über die vier Grundoperationen und zeigt, dass sie mit den vier oben erwähnten HTTP-Requests in einem logischen Zusammenhang stehen:

| Übersicht über die CRUD-Operationen |        |  |              |
|-------------------------------------|--------|--|--------------|
|                                     | Name   | Beschreibung   | HTTP-Request |
| <b>C</b>                            | Create | Das Erzeugen von neuen Ressourcen.                   | Post         |
| <b>R</b>                            | Read   | Das Zurückgeben von Informationen                    | Get          |
| <b>U</b>                            | Update | Das Anpassen (Updaten) von existierenden Ressourcen. | Put (Patch)  |
| <b>D</b>                            | Delete | Das Löschen von existierenden Ressourcen.            | Delete       |

Ein einfaches Beispiel für die CRUD-Operationen, ist z. B. ein Code-Repository in einem Versionsverwaltungsprogramm wie GitHub, diese Grundaktionen perfekt umsetzt:

Neue Files können erzeugt werden (=Create), auf einen Entwicklercomputer geklont werden (=Read), vom Entwicklungsteam geändert (=Update) und gegebenenfalls gelöscht (=Delete) werden.

<https://www.restapitutorial.com/lessons/httpmethods.html> gibt eine von zahlreichen Übersichten über CRUD.

## 3 Das Architekturmodell einer Webapplikation

Wir werden uns in diesem Skriptum schrittweise von völlig rudimentären Webanwendungen in den Abschnitten 4 und 5, die ausschließlich in einem Layer implementiert sind, schrittweise zu einer Webanwendung mit Datenbanken weiterarbeiten.

In realistischen Enterprise Applications werden die Daten natürlich in Ressourcen, wie Datenbanken oder Files gespeichert und bei Bedarf verwendet. Die dafür verantwortlichen Teile sind von der Web-API und der Business Logic streng zu trennen um fein aufgeteilte System zu ermöglichen und monolithische Ansätze zu vermeiden.

Die richtige Architektur von Webapplikationen ist sehr wesentlich in der Entwicklung, der Wartung und der Erweiterung von Systemen. Die hier beschriebene Architektur wird von Java Spring voll unterstützt. Die genaue Benennung der einzelnen Teile variieren in der Literatur.

Beispielsweise wird der Domain Layer und der Persistence Layer an manchen Stellen auch als **Data Layer** bezeichnet.

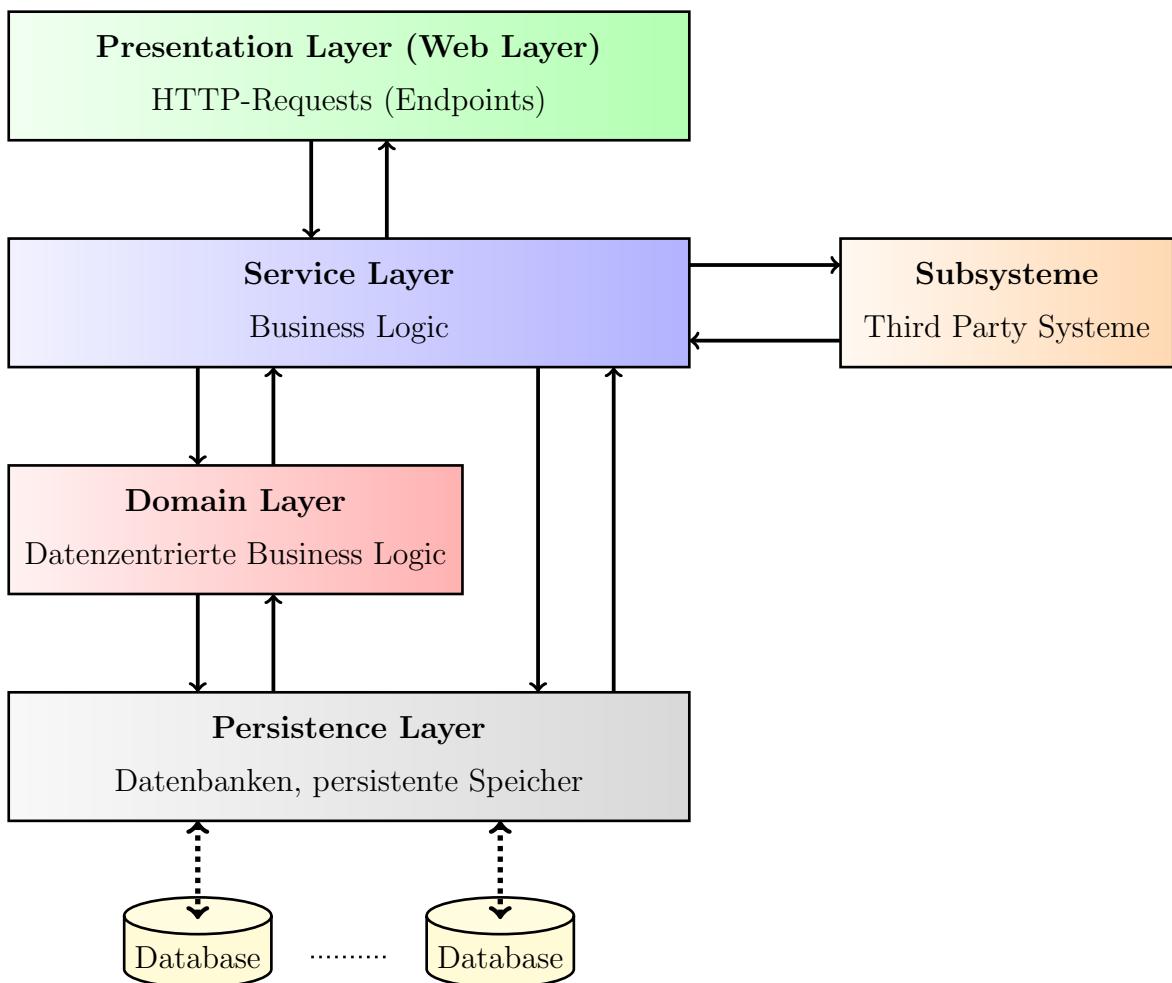


Abbildung 8: Architektur einer Enterprise Application

Klassen die in den einzelnen Layern angelegt werden, haben typischerweise bestimmte Annotations in Java Spring. Das ist sinnvoll, weil jedes Bean bei seiner Erstellung von Application Context spezielle Fähigkeiten erhält. Diese werden durch die Java Spring-Annotations oberhalb des Quellcodes der Klassen festgelegt.

| Layer              | Java Spring Annotations                            |
|--------------------|--|
| Presentation (Web) | @RestController, @Controller, ...                  |
| Service            | @Service, ...                                      |
| Domain             | @Entity, @Table, @Component, keine Annotation, ... |
| Persistence (Data) | @Repository, ...                                   |

### 3.1 Presentation Layer (Web Layer)

Der Presentation Layer definiert alle HTTP-Endpoints, d. h. das gesamte Web-API der Applikation. Im Allgemeinen sollte diese Schicht sehr dünn sein und nur jenen Code enthalten der für die Eingabe und der Ausgabe von Daten erforderlich ist.

Typischerweise enthält der Presentation Layer, der aus gutem Grund auch Web Layer genannt wird, alle Klassen, die HTTP-Requests und deren Parameter definieren.

### 3.2 Service Layer

Der Service Layer enthält die gesamte Logik der Geschäftsprozesse und ist gewissermaßen das Herzstück jeder Webapplikation. Die Logik der Geschäftsprozesse nennt man auch **Business Logic**.

In einem Webshop sind dies beispielsweise alle Abläufe die in einem Unternehmen mit dem Einkauf und Verkauf von Artikeln, sowie dem gesamten Kundenmanagement zu tun haben.

Bei der Business Logic unterscheidet man zwei Teile:

- **Application Services im Service Layer:** Das ist jener Teil, der die Abläufe unabhängig von der technischen Ressourcen abbildet.
- **Infrastructure Services im Domain Layer:** Das ist jener Teil, der mit den technischen Ressourcen, wie File System, Datenbanken oder Emailserver zu tun hat.

Der Service Layer enthält also nicht die gesamte Business Logik.

### 3.3 Domain Layer

Der Domain Layer enthält jenen Teil der Business Logik, der datenzentriert ist. Das ist die Funktionalität, die mit Daten zu tun hat, die mit den in den Datenbanken gespeicherten Tabellen in naher Verbindung stehen.

## 3.4 Persistence Layer

Der Persistenz Layer ist für die Kommunikation mit den technischen Ressourcen, wie Datenbanken oder Files zuständig. Dort befindet sich keine Art von Logik, sondern reine Zugriffsfunktionalität.

Diese technischen Ressourcen sind immer persistent, d. h. sie werden auf zeitlich dauerhaften Medien gespeichert.

Für den Persistenz Layer findet sich in der Literatur das Entwurfsmuster **DAO (Direct Access Object)**, dass in dem Blogbeitrag <https://www.baeldung.com/java-dao-pattern> konkret für Spring beschrieben ist.

Dahinter verbirgt sich in erster Linie das CRUD-Prinzip, das in Abschnitt 2.2 beschrieben wird.

## 3.5 Subsysteme

Als Subsystem bezeichnet man externe Computersysteme, die an die eigene Webapplikation angebunden sind und die man über ein API verwendet. Diese sind typischerweise an die Business Logic des Service Layers an die Applikation angebunden.

Beispiele für Subsysteme sind:

- SAP Systeme
- Die Cloud-Lösungen der diversen Anbieter, wie z. B. Google Cloud oder Azure von Microsoft
- Microsoft Sharepoint
- ...

## 4 Eine erste Webapplikation als REST-API: Hi ...! Nice to meet you.

Im folgenden Abschnitt schreiben wir eine erste kleine - aber feine - Webapplikation in Java mit Hilfe des Java Spring Boot Frameworks.

Wir realisieren diese Webanwendung als sogenanntes REST-Webservice. Ein solches dient dazu Daten zur Verfügung zu stellen. Um das Darstellen der Daten in HTML geht es dann im nächsten Abschnitt 5.

“Nichts begeistert mich in Java Spring mehr, als die überwältigende Einfachheit mit der man eine Webapplikation erstellen kann.”

[Zitat von Thomas Schlögl, als er sich 2019, anlässlich der Erstellung dieses Skriptums selbst interviewte]

### Was unsere erste Webapplikation können soll

Unsere erste Webapplikation soll die Möglichkeit bieten, über den Browser den Vor- und Nachname, sowie das Alter einer Person anzugeben und diese dann mit einer Hello-Meldung zu grüßen.

### 4.1 Anlegen des Projekts

Spring Projekte lassen sich sehr einfach mit Hilfe der Open Source Seite <https://start.spring.io> angelegt. In manchen IDEs (Integrated Development Environments), wie z. B. IntelliJ ist diese Seite bereits integriert. In Eclipse muss man ein zip-File über den Browser downloaden und dann in die IDE importieren.

### 4.2 Vorgangsweise bei IntelliJ

1. **Projekttyp auswählen:** Den Menüpunkt [File]→[New]→[Project...] aufrufen und den Eintrag [Spring Initializr] auswählen. Dann das JDK auswählen, den Default akzeptieren und auf [Next] klicken.

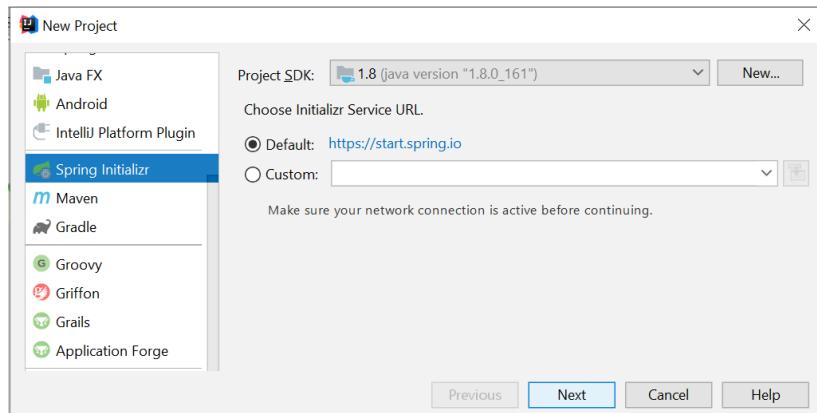


Abbildung 9: Anlegen eines Spring Projekts in IntelliJ

**2. Metadaten festlegen:** Die *Group* und das *Artifact* bilden gemeinsam das Package in dem die Mainklasse angelegt werden wird. Beides sind Begriffe die für das Buildtool **Maven** eine Rolle spielen, das in einem anderen Abschnitt näher beschrieben wird.

Bei *Type* lassen wir die Einstellung am vorgeschlagenen Wert [*Maven POM*]. Maven wird in einem eigenen Abschnitt in diesem Skriptum erklärt.

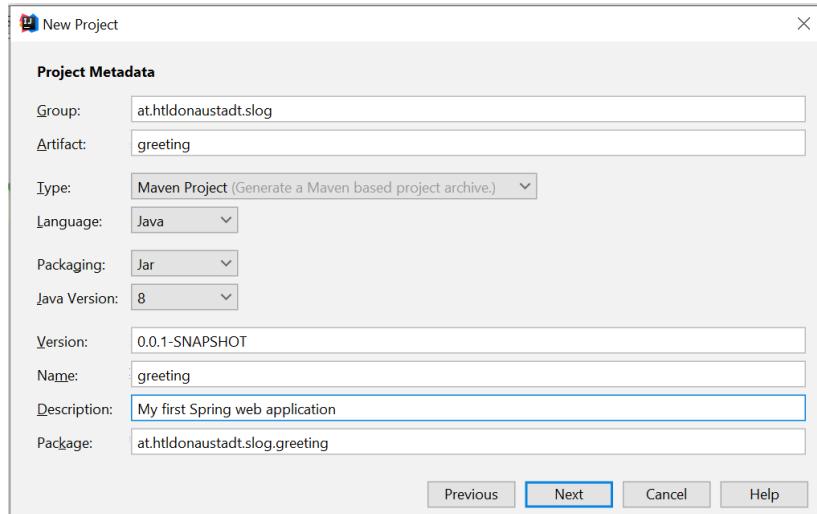


Abbildung 10: Festlegung der Metadaten eines Spring Projekts in IntelliJ

**3. Auswahl der Spring Module:** Dieser Schritt ist der interessanteste. Abhängig von der Applikation die man mit Spring schreiben möchte, werden nun alle erforderlichen Module ausgewählt, die man benötigt und die man bei der Entwicklung gebrauchen wollen.

Wir wählen für unsere erste Applikation lediglich zwei Module aus, *Lombok* und *Spring Web*.

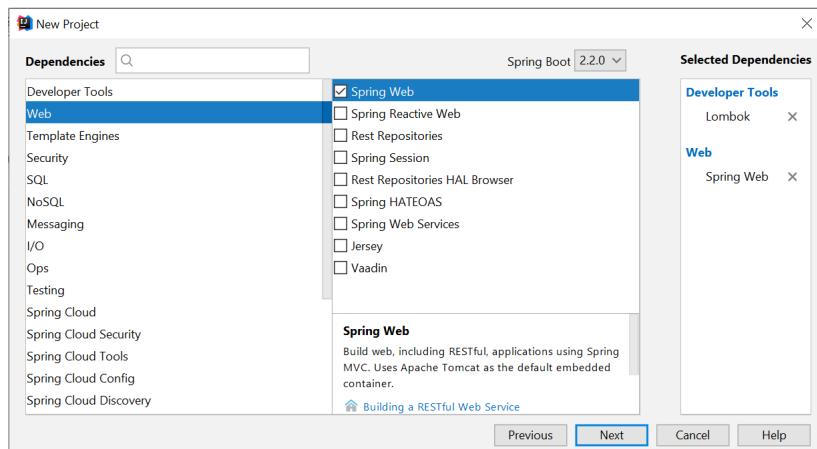


Abbildung 11: Auswahl der Spring Module in IntelliJ

*Lombok* ist eine sehr gebräuchliche Java Library die das Schreiben von Gettern, Settern, Konstruktoren, ... usw. sehr erleichtert.

*Spring Web* ist das Spring Modul, dass ein Programm zu einer Webapplikation macht und das Reagieren auf Webrequests ermöglicht.

#### 4. Angabe des lokalen Projektpfads und des Projektnames:

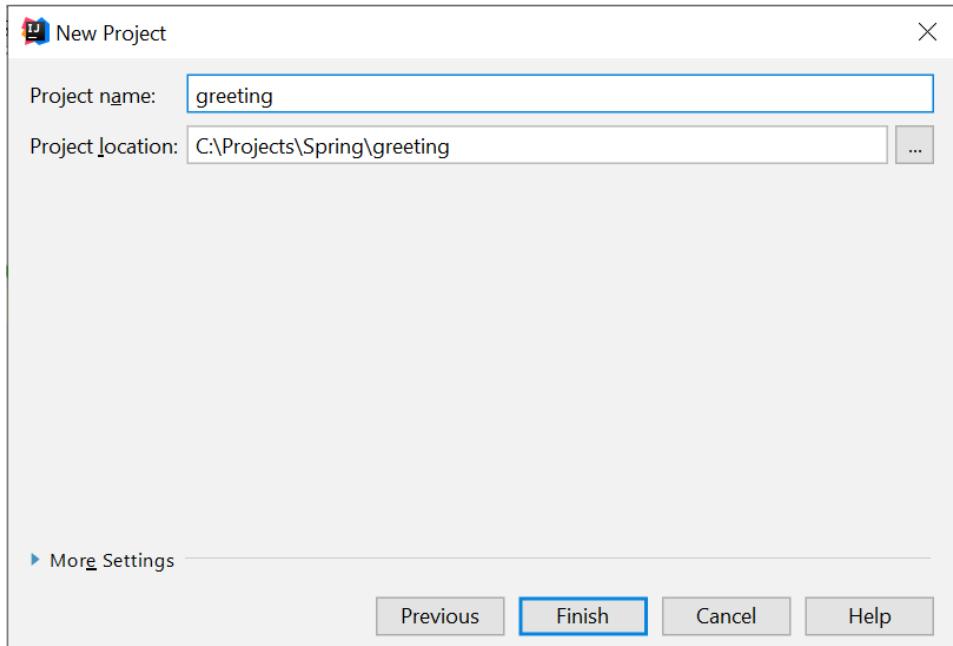


Abbildung 12: Angabe des Projektpfads in IntelliJ

Nach [Finish] wird das Projekt angelegt und der Projektbaum enthält unter src im Package at.htldonaustadt.slog.greeting die Main-Klasse GreetingApplication. Über der Klasse muss sich die Annotation @SpringBootApplication befinden.

#### 5. Maven Auto-Import aktivieren:

In IntelliJ poppt normalerweise ein Dialog auf, bei dem man sich aussuchen kann, ob Module die später hinzugefügt werden sollen, automatisch aktiviert werden sollen.

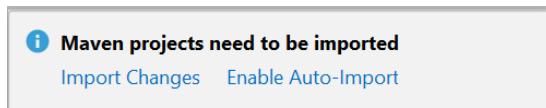


Abbildung 13: Auto-Import für Maven in IntelliJ aktivieren

Es ist höchst empfehlenswert *Enable Auto-Import* zu aktivieren.

### 4.3 Starten der Webapplikation

Beim Starten des Projekts werden auf der Konsole zahlreiche Log-Meldungen ausgegeben, die darüber Auskunft geben, was Java Spring zur Laufzeit gerade tut und ob alle Aktionen erfolgreich waren.

Die Meldung

**Tomcat started on port(s): 8080 (http) with context path "**

zeigt an, dass Spring einen Tomcat<sup>6</sup>-Webserver auf unserer Maschine gestartet hat und dieser am

<sup>6</sup>Tomcat ist ein weitverbreiteter Open Source Webserver von Apache [https://de.wikipedia.org/wiki/Apache\\_Tomcat](https://de.wikipedia.org/wiki/Apache_Tomcat).

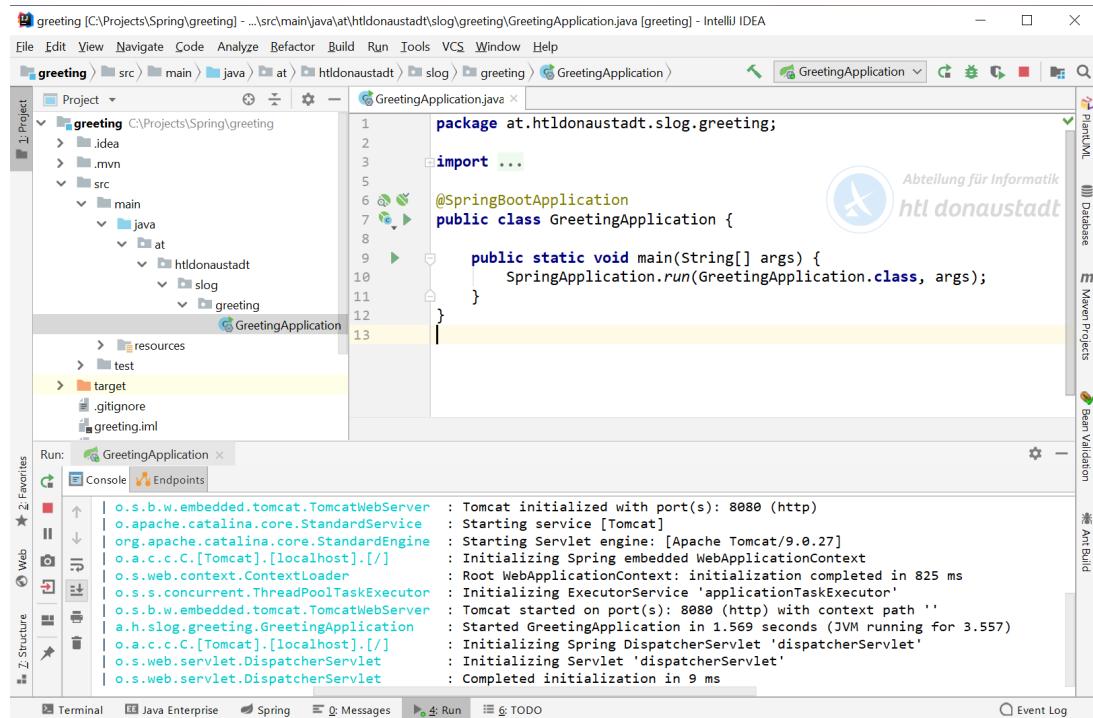


Abbildung 14: Start einer Web Application in IntelliJ

Port auf Http-Requests "lauscht".

Spring hat also dafür gesorgt, dass wir ohne auch nur einen Handgriff zu tun, eine voll einsatzfähige Webapplikation haben.

Jetzt geht es daran, dem Programm eine gewünschte Fähigkeit beizubringen, so dass es auf Http-Requests antworten kann.

## 4.4 Wie bringt man unserer Webapplikation Grüßen bei?

Dazu implementieren wir die Funktionalität in einer Klasse, die Http-Requests verarbeiten kann.

### 4.4.1 Anlegen einer Controller-Klasse

Für die eigentliche Funktionalität implementieren wir eine eigene Klasse `GreetingController` in einem neuen Package `presentation`.

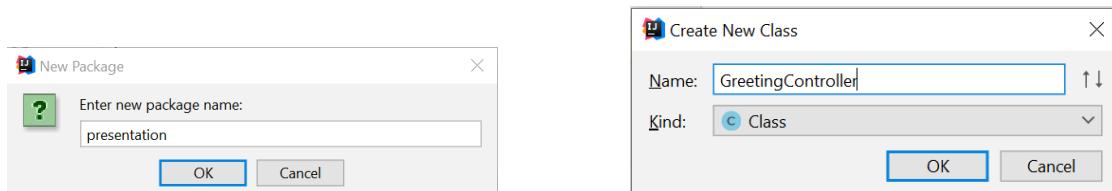


Abbildung 15: Anlegen einer Controller-Klasse in einem neuen Package

**Wichtig!** Der Folder des Packages *presentation* muss ein direktes Kind des Packages *greeting* sein. Java Spring scannt beginnend mit der Spring Boot Application (also der Klasse GreetingApplication) nach *Components*. Was eine *Spring Component* genau ist, wird in einem späteren Abschnitt erklärt.

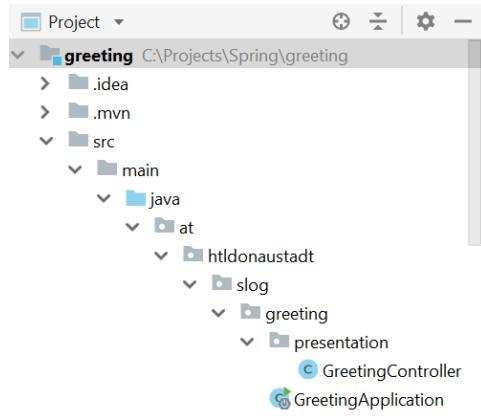


Abbildung 16: Projektstruktur

#### 4.4.2 Implementierung des Grüßens

Damit unserer Webapplikation nicht einfach nur läuft, sondern auch auf HTTP-Requests reagieren kann, muss man eine sogenannte Controllerklasse implementiert, in unserem Fall eines sogenannten REST-Controller.

##### Controller Klasse

Eine Controllerklasse einer Webapplikation ist für die HTTP-Kommunikation zuständig.

In Java Spring werden Controllerklassen durch die Annotation `@Controller` bzw.  
`@RestController` gekennzeichnet.

```
@RestController
public class GreetingController {

    public static final String PREFIX = "/api";

    @GetMapping(PREFIX + "/greet")
    public String greet() {
        return "Hi! I am ready to say hello!";
    }

    @GetMapping(PREFIX + "/{someName}")
    public String greetByName(@PathVariable("someName") String name) {
        return String.format("Hi %s! Nice to meet you.", name);
    }
}
```

Listing 1: Ein REST-Controller der grüßen kann

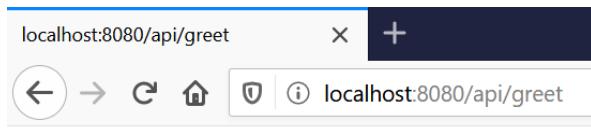
### POJO statt EJB

Die Controller-Klasse `GreetingController` ist ein Beispiel für eine POJO-Klasse, wie sie im Spring Framework üblich ist. Im Gegensatz zu den in der Einleitung besprochenen Enterprise Java Beans (EJBs) implementiert sie keinerlei Interfaces und braucht nur geringe formale Anforderungen erfüllen.

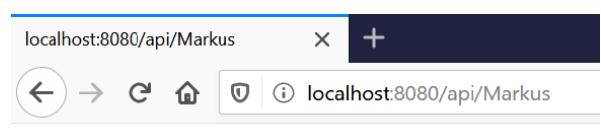
#### 4.4.3 Ausprobieren des Codes

Wenn wir das Programm nun starten, kann unsere Applikation zwei Arten von Http-Get-Requests verarbeiten.

Davon kann man sich leicht überzeugen, wenn man entsprechende URLs im Browser eingibt.



(a) Ohne Parameter



(b) Mit einem Path-Parameter

Abbildung 17: Aufruf der beiden Methoden des allerersten REST-Controllers

#### 4.5 Was Spring mit der Controller-Klasse tut

### Instantierung der Klasse

Spring erstellt zur Laufzeit eine Instanz unserer Klasse. Das macht das Framework völlig automatisch im sogenannten *ApplicationContext* der in einem weiteren Abschnitt beschrieben wird. Man beachte, dass wir keinen Code geschrieben haben, der den Konstruktor der Klasse aufgerufen hat.

Nur durch das Hinzufügen einer Klasse mit bestimmten Annotations ist Spring in der Lage zur Laufzeit eine Instanz dieser Klasse zu erstellen und mit den notwendigen Funktionalitäten auszustatten, damit die Methoden automatisch aufgerufen werden, sobald durch den Browser ein HTTP-Request erfolgt.

Es gibt grundsätzlich drei Arten von Annotations:

- Annotations vor Klassen
- Annotations vor Methoden
- Annotations vor Parameter von Methoden

#### 4.5.1 Annotations vor Klassen

Spring geht beim Start jeder Applikation her und scannet das gesamte Projekt nach Annotations die oberhalb von Klassen stehen. Alle solche Klassen erhalten von Spring zusätzliche Funktionalität.

Jede Klasse mit einer Annotation wird zu einem sogenannten **Bean** (Spring Bean) oder einer **Component**. Dieses Scannen des Programmcodes wird in einem weiteren Abschnitt genauer beschrieben.

In unserem Fall ist jede Instanz der Klasse *GreetingController* - unsere Applikation verwendet aber nur eine - ein sogenannter REST-Controller und kann Http-Requests verarbeiten.

#### 4.5.2 Annotations vor Methoden

Ähnlich wie die Annotation @RestController verknüpft Spring jede Methode mit einer Annotation intern mit weiterem Code.

```
@GetMapping(PREFIX+”/greet”)
```

ist eine Kurzschreibweise für das früher gebräuchliche

```
@RequestMapping(path=PREFIX+”/greet”,method=RequestMethod.GET)
```

und dient dazu um die Methode an einen HTTP-Get-Request zu binden.

Wie wir später sehen werden, gibt es auch die Annotations @PutMapping und @PostMapping und HTTP-Put-Requests und HTTP-Post-Requests in Klassen zu realisieren.

#### 4.5.3 Annotations vor Methodenparameter

Um einzelne Teile einer URL an Java Methodenparameter zu binden, sind zusätzliche Annotations vor diesen Parametern notwendig.

```
@GetMapping(PREFIX+”/{someName}”)  
public String greetByName(@PathVariable(“someName”) String name)
```

Der URL-Pathparameter *someName* wird durch den Code an den Java-Parameter *name* gebunden. D. h. der String der der URL enthalten ist, wird von Spring automatisch der Variable *name* zugewiesen.

##### Path Variables (@PathVariable)

Path variables und die damit verbundene Verwendung der Annotation @PathVariable ist eine von zwei Möglichkeiten, wie man über eine URL Informationen an eine Controller-Klasse weitergibt.

Die zweite Möglichkeit stellen HTTP-Parameter dar, die über die Annotation @RequestParam übergeben werden können. Sie sind weiter unten in diesem Abschnitt beschrieben.

In unserem Beispiel mit der URL

localhost:8080/api/Markus

wird *someName* der Wert *Markus* zugewiesen und dann weiters der Variablen *name*. Damit ist der Wert aus der URL im Java-Code "angekommen" und kann dort verwendet werden.

#### 4.5.4 Rückgabewert der Controller-Methoden

Der Rückgabewert der mit @GetMapping markierten Methoden entspricht dem HTTP-Response. Da es sich bei der Klasse um einen REST-Controller handelt, wird der Rückgabewert einfach eins-zueins als HTML-Body angezeigt.

### 4.6 Die Verwendung von HTTP-Parametern

Neben den Path Variables gibt es noch eine häufig verwendete Möglichkeit um über URLs Werte an einen Controller zu übergeben. Diese Möglichkeit sind HTTP-Parameter.

Eine hervorragende Beschreibung aller Möglichkeiten findet sich auf der Seite hervorragenden Seite von Eugen Paraschiv <https://www.baeldung.com/spring-request-param>.

**Beispiel:** In der URL

http://localhost:8080/api/greetWithAge?firstname=Hugo&age=25

gibt es zwei URL-Parameter:

- *firstname* mit dem Wert Hugo
- *age* mit dem Wert 25

Diese sind mit einem ? an den Domainteil der URL angehängt. Mehrere Parameter können mit & konkateniert werden.

#### HTTP-Variables (@RequestParam)

HTTP-Parameter werden über die Annotation @RequestParam an die Methoden der Controller-Klasse übergeben.

Um diese zu demonstrieren, erweitern wir unsere GreetingController Klasse um die folgende Methode:

```
@GetMapping(PREFIX+{/greetWithAge})
public String greetByNameAndAge(
    @RequestParam("firstname") String name,
    @RequestParam int age) {
    final String word = age > 18 ? "Good day" : "Hello";
    return String.format("%s %s! Nice to meet you.", word, name);
}
```

Listing 2: HTTP-Parameter in der Controller-Klasse

Wie man sieht werden für beide Parameter die Annotations @RequestParam verwendet. Optional kann man in der Annotation den Namen des Parameters gesondert angeben, wenn er nicht mit dem Namen des Methodenparameters übereinstimmt.

Außerdem ist es auch möglich, einen Default-Wert zu hinterlegen.

#### 4.6.1 Testen der Controller-Methode mit HTTP-Parameter

Nach der Eingabe von unterschiedlichen Altersangaben in zwei URLs zeigt sich der gewünschte Response im Browserfenster.

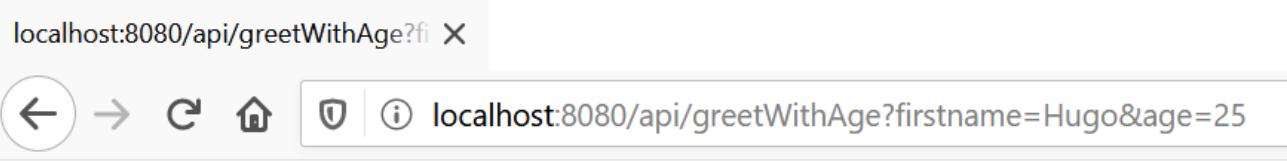
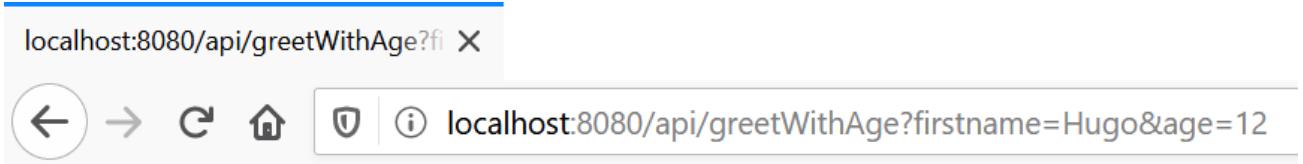


Abbildung 18: Übergabe des Namens und des Alters an die Controller-Methode über HTTP-Parameter

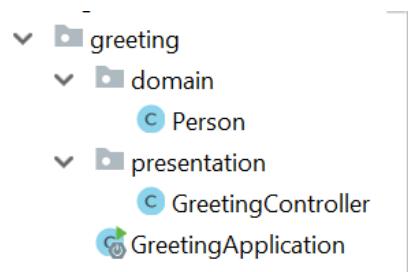
## 4.7 Übungsaufgaben

1. Füge der Klasse `GreetingController` weitere Methoden hinzu, die keinen String sondern

- eine Zahl
- einen Boolean
- eine `List<String>`

zurückgibt. Überprüfe was als Resultat im Browser angezeigt wird.

2. Erstelle ein Package `domain` mit einer Klasse `Person`.



Die Klasse `Person` soll typische Eigenschaften einer Person speichern und zwar zumindest:

- Vorname (required)
- Nachname (required)
- Körpergröße in Zentimeter
- Geburtstag als `LocalDate`

Verwende für die Erstellung - wenn Du möchtest - die Erweiterung Lombok um unnötigen Code zu sparen.

Erstelle nun zwei Methoden in der Klasse `GreetingController`. Eine soll eine Instanz von `Person` zurückgeben und die andere Methode eine `List<Person>`. Beide Methoden dürfen hardcoded Instanzen zurückgeben und brauchen keine Inputparameter verwenden.

Überprüfe was als Resultat im Browser angezeigt wird. Welches Format hat die Ausgabe?

## 5 Eine "echte" Webapplikation: Spring MVC (Model-View-Controller)

Das im Abschnitt 4 REST-Service ist dafür gedacht, über ein wohldefiniertes API Daten zur Verfügung zustellen. Die Daten werden - wie man in den Übungen sieht - als JSON zurückgeliefert und sind daher ausgezeichnet zur Weiterverarbeitung geeignet.

Bei den Seiten auf denen man tagtäglich im Internet surft, werden die Daten von der Webapplikation zum Client geschickt und im Browser zu einer gefälligen Darstellung gerendert.

Ein kleines Codebeispiel zeigt, wie man eine Bücherliste (die einstweilen noch hartkodiert im Presentation-Layer angelegt wird) von einer Spring-Controllerklasse zurückgegeben wird und dann in HTML gerendert an den Client zurückgeschickt wird.

### 5.1 Frontend: Bootstrap und Thymeleaf

Wir werden dabei für den Frontendteil **Bootstrap** und **Thymeleaf** verwenden um die Daten zu rendern.

**Bootstrap** <https://getbootstrap.com> ist ein Frontend-Framework mit dem man Webseiten gestalten kann. Bootstrap stellt HTML- und CSS-Templates für eine große Menge von GUI-Elementen zur Verfügung, die man sehr einfach in HTML einbetten kann.

**Thymeleaf** <https://www.thymeleaf.org> ist eine sogenannte Java-Template-Engine, die unter anderem Sprachelemente zur Verfügung stellt um Daten eines Modells in HTML-Code umzuwandeln. Diese beiden Libraries werden nur aufgrund der Einfachheit gewählt. Es gibt eine Riesenmenge an verschiedenen Java Script Libraries.

Eine sehr nützliche Seite, die alternative JavaScript-Libraries für die Frontentwicklung enthält, ist <https://www.webjars.org>.

### 5.2 Die Buchklasse des Domain-Layers

Die Buchklasse wird wie üblich mit Lombok angelegt.

```
@Data  
@AllArgsConstructor  
class Book {  
    private final long id;  
    private final String title;  
    @NonNull private String author;  
    private LocalDate publicationDate;  
}
```

Listing 3: Eine Klasse mit einigen Buchdaten

## 5.3 Die Controllerklasse im Presentation-Layer

Analog zu REST-Controller in Listing 10.6, hat auch unsere Webapplikation eine Controllerklasse (siehe Listing 4).

Diese ist ähnlich aufgebaut wie ein REST-Controller, verwendet aber zusätzlich eine Klasse **Model** oder **ModelAndView** <https://www.javapedia.net/Spring-MVC-Interview-questions/862>:

- Der Konstruktor der Klasse erhält einen Pfad zu einem HTML-File. Dieser Pfad wird - wie man später sehen wird - im Projekt abgebildet.
- Mit `addAttribute` können beliebige Daten an das HTML-File weitergegeben werden.

```
@Controller
public class BooksController {

    @GetMapping("/books")
    public ModelAndView showBooks() {
        List<Book> books = new ArrayList<>(
            Arrays.asList(
                new Book(80, "C++ Programmieren für Einsteiger",
                    "Michael Bonacina", LocalDate.of(2018,9,5)),
                new Book(41, "Spring Boot in Action",
                    "Craig Walls", LocalDate.of(2015, 11, 30)),
                new Book(50, "Java Design Patterns",
                    "Vaskaran Sarcar", LocalDate.of(2018, 12, 7)),
                new Book(38, "Java Socket programming",
                    "Mazhar Hussain Malik", LocalDate.of(2014, 12, 1)),
                new Book(62, "Schrödinger programmiert Java",
                    "Philip Ackermann", LocalDate.of(2017, 7, 31))
            )
        );
        // localhost:8080/books/list.html
        ModelAndView modelAndView = new ModelAndView("books/list");
        modelAndView.addAttribute("books", books);
        return modelAndView;
    }
}
```

Listing 4: Ein Controller für Buchdaten

Das Codebeispiel zeigt nur eine ganze einfache Variante Daten zu transportieren. Java Spring bietet darüber hinaus noch viel mehr an Funktionalität.

Im Internet sind dazu Unmengen an Tutorials verfügbar.

## 5.4 Die Projektstruktur

Zusätzlich zu den Klassen im Domain- und Presentation-Layer wird in den Ressourcen ein HTML-File angelegt, das später die gerenderten Buchdaten enthalten wird.

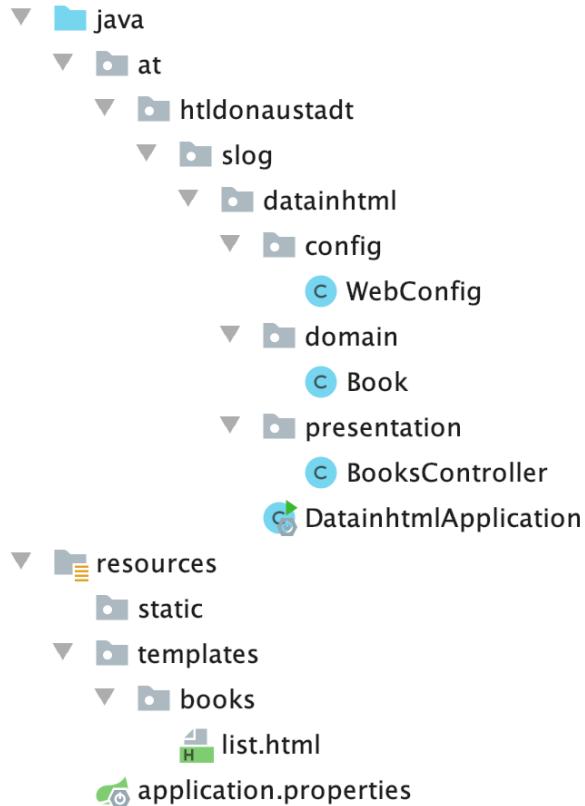


Abbildung 19: Die Projektstruktur der Webanwendung mit einem HTML-File list.html

## 5.5 Das HTML-File mit Bootstrap und Thymeleaf

Das von IntelliJ erzeugte HTML-File wird schrittweise erweitert:

- Einfügen der Maven-Dependencies für Bootstrap und Thymeleaf in das File `pom.xml`. Die Dependencies findet man unter <https://www.webjars.org>.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>4.3.1</version>
</dependency>
```

Listing 5: Bootstrap- und Thymeleaf-Dependencies in pom.xml

- Einfügen eines <div> Containers mit einer <table>.
- Der XML-Namespace `th` wird im html-Root Element hinzugefügt. Der Wert ist der thymeleaf-Dokumentation zu entnehmen. `xmlns:th="http://www.thymeleaf.org"`. Über die `class`-Attribute kann in Thymeleaf das Aussehen der einzelnen Element bestimmt werden.
- Das Einfügen der Daten aus dem Model erfolgt durch die Schleife `<tr th:each='book: $allbooks'>`. `allbooks` ist das Modelattribut der Bücherliste, die in der Schleife ausgelesen wird. Die Bezeichnungen `identifier` in `book.identifier` leitet sich aus dem Namen der Instanzvariable der Klasse Book ab.

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>My favourite books</title>
</head>
<body>
  <div class="container">
    <table class="table table-dark">
      <thead class="thead-dark">
        <tr>
          <th scope="col">Identifier</th>
          <th scope="col">Title</th>
          <th scope="col">Author</th>
          <th scope="col">Publication Date</th>
        </tr>
      </thead>
      <tbody>
        <!-- model.addObject("allbooks", books); -->
        <tr th:each="book: ${allbooks}"> <!-- foreach ( var book : allbooks ) -->
          <td th:text="${book.identifier}" />
          <td th:text="${book.title}" />
          <td th:text="${book.author}" />
          <td th:text="${book.publicationDate}" />
        </tr>
      </tbody>
    </table>
  </div>
</body>
</html>

```

Listing 6: Das HTML-File mit Thymeleaf-Anweisungen

## 5.6 Testen der Anwendung

Nach dem Start und dem Aufruf der Webapplikation über den Browser durch die Eingabe von `http://localhost:8080/books` werden die gerenderten Buchdaten angezeigt.

| <b>Identifier</b> | <b>Title</b>                     | <b>Author</b>        | <b>Publication Date</b> |
|-------------------|----------------------------------|----------------------|-------------------------|
| 80                | C++ Programmieren für Einsteiger | Michael Bonacina     | 2018-09-05              |
| 41                | Spring Boot in Action            | Craig Walls          | 2015-11-30              |
| 50                | Java Design Patterns             | Vaskaran Sarcar      | 2018-12-07              |
| 38                | Java Socket programming          | Mazhar Hussain Malik | 2014-12-01              |
| 62                | Schrödinger programmiert Java    | Philip Ackermann     | 2017-07-31              |

Abbildung 20: HTML Ausgabe von `http://localhost:8080/books`

## 5.7 Übungsaufgaben

1. Erweitere das HTML-File `list.html` um eine Zeile in der steht, wie viel Bücher insgesamt gefunden wurden, also z. B.  
**5 Bücher wurden gefunden.**  
Erstelle dafür ein weiteres Modelattribut.
2. Falls die Liste nur ein einziges Buch enthält, soll nicht die Liste von vielen Büchern angezeigt werden, soll am Client eine spezielle Detailansicht dieses einzigen Buches.  
Implementiere diese Funktionalität in dem du eine zweite HTML-Seite mit der Detailansicht einbaust und im Controller-Code Modellattribute mit den Daten dieses einzigen Buchs befüllst.

# 6 Ein Rest-Controller mit Post-, Put-, Get- und Delete-Requests

In unserem ersten Beispiel in diesem Skriptum programmierten wir einen Rest-Controller der grüßen konnte. Dafür implementierten wir Get-Requests.

Für reale Webanwendungen ist das natürlich viel zu wenig. Die Realisierung der drei anderen wichtigen Http-Request-Arten, nämlich Post, Put und Delete, wird daher in diesem Kapitel demonstriert.

## 6.1 Erstellen des Projekts

Unser Projekt verwendet die Spring Module Lombok und Spring Web.

## 6.2 Die Buchklasse des Domain-Layers

Die Buchklasse wird wie üblich mit Lombok angelegt.

Diesmal wird ein Default-Konstruktor implementiert und keines der Instanzvariablen `final` gemacht. Das ist erforderlich um Instanzen der Klasse `Book` aus JSON heraus bauen zu können. Wir werden sehen, dass das für Post- und Put-Requests notwendig ist.

```
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
class Book {  
    private long id;  
    private String title;  
    private String author;  
    private LocalDate publicationDate;  
}
```

Listing 7: Eine Klasse mit einigen Buchdaten

## 6.3 Der Rest-Controller des Presentation-Layers

Die Klasse demonstriert die vier Http-Requests nur grundsätzlich. Wesentlich ist, dass beim Post- und Put-Request die Instanz eines Buches als Inputparameter steht.

Zu beachten ist, dass dieser Code (**fast**) **reiner Fun-Code** ist und demonstrieren soll, wie man einen Post-Request realisiert, der eine ganze Instanz einer Klasse als Inputparameter mitschickt.

Der Post-Request schickt eine JSON-Spezifikation des Buches mit und Spring serialisiert die JSON-Werte automatisch in eine Instanz.

Diese Serialisierung wird automatisch von Java Spring durchgeführt. Dazu ist es erforderlich, dass die zu serialisierende Klasse den Default-Konstruktor implementiert. Das erledigt in Lombok die Annotation `@NoArgsConstructor`.

```

@RestController
public class BooksRestAPI {

    private final Logger logger = LoggerFactory.getLogger(this.getClass());
    private final static String API = "/api";

    @PostMapping(value = API + "/createbook", consumes = "application/json",
                 produces = "application/json")
    public ResponseEntity<String> createBook(@RequestBody Book book) {
        logger.info("Http-POST: Book was sent: {}", book);
        return ResponseEntity.status(HttpStatus.OK).build();
    }

    @PutMapping(value = API + "/createbook", consumes = "application/json",
               produces = "application/json")
    public ResponseEntity<String> putBook(@RequestBody Book book) {
        logger.info("Http-PUT: Book was sent: {}", book);
        return ResponseEntity.status(HttpStatus.I_AM_A_TEAPOT).build();
    }

    // http://localhost:8080/api/deletebook?id=15
    @DeleteMapping(value = API + "/deletebook")
    public ResponseEntity<String> deleteBook(@RequestParam int id) {
        logger.info("Http-DELETE of book {}", id);
        return ResponseEntity.ok("Very nice!");
    }

    // http://localhost:8080/api/get?id=15
    @GetMapping(API + "/get")
    public ResponseEntity<String> getSomething(@RequestParam int id) {
        logger.info("Http-GET!");
        return ResponseEntity.ok("Very nice!");
    }
}

```

Listing 8: Der Rest-Controller mit allen vier Requests

## 6.4 Testen der Http-Requests mit Postman

Postman kann von hier <https://www.getpostman.com> heruntergeladen und installiert werden. Es ist eines von vielen Tools mit denen man Http-Requests abschicken kann und das es erlaubt, die Requests und Responses ganz genau zu überprüfen.

In Postman kann man die Http-Requests auswählen, die URL angeben und alle Daten im Header und Body angeben.

Wir möchten einen Post-Request absetzen, um zu sehen, ob die Erstellung der Inputinstanz der Methode `createBook` korrekt funktioniert.

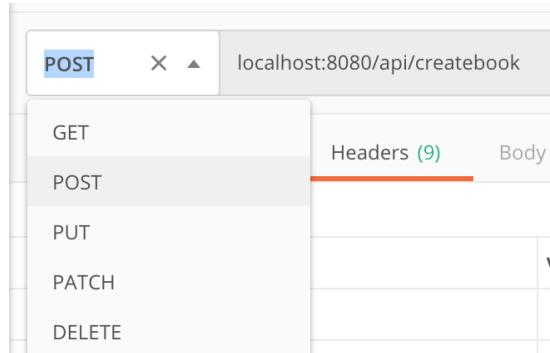


Abbildung 21: Auswahl der Art des Http-Requests und Eingabe der URL

Weiters setzen wir im Header des Requests den Content-Type, so dass wir im Body JSON-Daten mitschicken kann.

A screenshot of the Postman interface focusing on the 'Headers' tab. The tab is highlighted with a red underline. Below the tab, it says 'Headers (9)'. Underneath that, there is a section titled '▼ Headers (1)'. A table is shown with two columns: 'KEY' and 'VALUE'. There is one row in the table with a checked checkbox in the 'KEY' column and the value 'Content-Type' in the 'VALUE' column. The 'BODY' tab is also visible to the right of the 'Headers' tab.

Abbildung 22: Daten im Request-Header

Im Body des Requests schicken wir nun die Daten eines Buches im JSON-Format mit (Abbildung 23).

Die Namen der Keys im JSON-Object müssen den Namen in der Klasse Book entsprechen.

Nach dem Senden des Requests durch das Anklicken von **Send**, wird unsere Webapplikation - wenn gerade läuft - auf den Post-Request reagieren und auf der Konsole die Meldung des Loggers mit den Daten ausgeben.

*Konsolenausgabe nach dem Abschicken des Post-Requests:*

```
2019-11-17 21:32:58.850 INFO 9515 — [nio-8080-exec-7] a.h.s.d.presentation.BooksRestAPI :  
Http-POST: Book was sent: Book(identifier=5000, title=Streit um Asterix, author=Rene Gos-  
cinny, publicationDate=2020-01-02)
```

Postman zeigt die erfolgreiche Übertragung beim Http-Statuscode an (Abbildung 24).

Auf völlig analoge Art und Weise - das Umschalten auf PUT reicht - lässt sich der Put-Request unseres Controllers testen.

The screenshot shows the Postman interface. At the top, a 'POST' method is selected and a URL 'localhost:8080/api/createbook' is entered. Below the header, there are tabs for 'Params', 'Authorization', 'Headers (9)', 'Body' (which is currently active), and 'Pre-request'. Under 'Body', there are four options: 'none', 'form-data', 'x-www-form-urlencoded', and 'raw'. The 'raw' option is selected. The request body contains the following JSON data:

```
1 {  
2   "identifier": 5000,  
3   "title": "Streit um Asterix",  
4   "author": "Rene Goscinny",  
5   "publicationDate": "2020-01-02"  
6 }
```

Abbildung 23: Bücherdaten im Request-Body

Status: 200 OK

Abbildung 24: Juhu!!! It did work.

Auch den Get- und Delete-Request kann man über Postman der Webapplikation schicken.

## 7 Die Grundprinzipien von Spring

In diesem Abschnitt werden zwei Grundprinzipien beschrieben, die im Spring Framework eine große Rolle spielen.

1. Application Context: Die Erstellung von Beans
2. Dependency Injection (DI) und Inversion of Control (IOC)

### 7.1 Application Context: Dort wo die Beans herkommen

Wie bei der Webapplikation in Abschnitt 2 aufgefallen ist, erzeugt das `Main` keine Instanz der Controllerklasse. Überhaupt wird der Konstruktor der Klasse `GreetingController` nicht von einer Codestelle unseres Projekts aufgerufen. Trotzdem gibt es eine Instanz, die auf einen HTTP-Request reagieren kann. Wie ist das möglich? Woher kommt die her?

Die Antwort auf diese Fragen liefert uns der Application Context.

#### Spring Beans

Instanzen von Klassen können vom Spring Framework automatisch beim Starten angelegt werden und zwar durch den sogenannten *Application Context* angelegt. Das erfolgt aber nicht für alle Klassen, sondern nur wenn eine Klasse durch eine Annotation als Component bzw. als Bean markiert ist.

Durch die Annotation erhält diese Klasse zusätzliche "Fähigkeiten eingimpft".

Eine vom Spring Framework automatisch angelegte Instanz nennt man **Component**, **Bean** oder **Spring Bean**.

Spring Beans werden zur Laufzeit in einem eigenen Spring Container verwaltet.

Die Annotation `@RestController` sorgt also dafür, dass für die nachfolgende Klasse eine Component bzw. ein Bean des Typs RestController angelegt wird<sup>7</sup>.

```
@RestController  
public class GreetingsController {  
    ...  
}
```

Listing 9: HTTP-Parameter in der Controller-Klasse

Es gibt eine ganze Menge an Annotations, die aus einer einfachen Klasse ein Spring Bean machen. Einige Beispiele dafür sind **Component**, **Controller**, **Service** und **Repository**.

<sup>7</sup>In der Fachliteratur wird zwischen Component und Bean unterschieden. Diese Unterscheidung wird in diesem Skriptum (derzeit) völlig außer Acht gelassen.

Welche Annotations es gibt und wo Spring in einem Projekt überhaupt nach Annotations sucht, erklärt Eugen Paraschiv in dem Artikel <https://www.baeldung.com/spring-bean-annotations>.

In diesem Skriptum wird in einem weiteren Abschnitt erklärt, wie man dieses abscannen des Codes nach Bean-Klassen mit der Annotation `@ComponentScan` beeinflussen kann.

Die folgende Graphik illustriert die zwei Arten von Instanzen die zur Laufzeit in einer Spring Applikation existieren. Des gibt solche Instanzen die von einem expliziten Konstruktorauftrag in unserem Code stammen und solche die vom Application Context automatisch (meist beim Programmstart) angelegt werden.

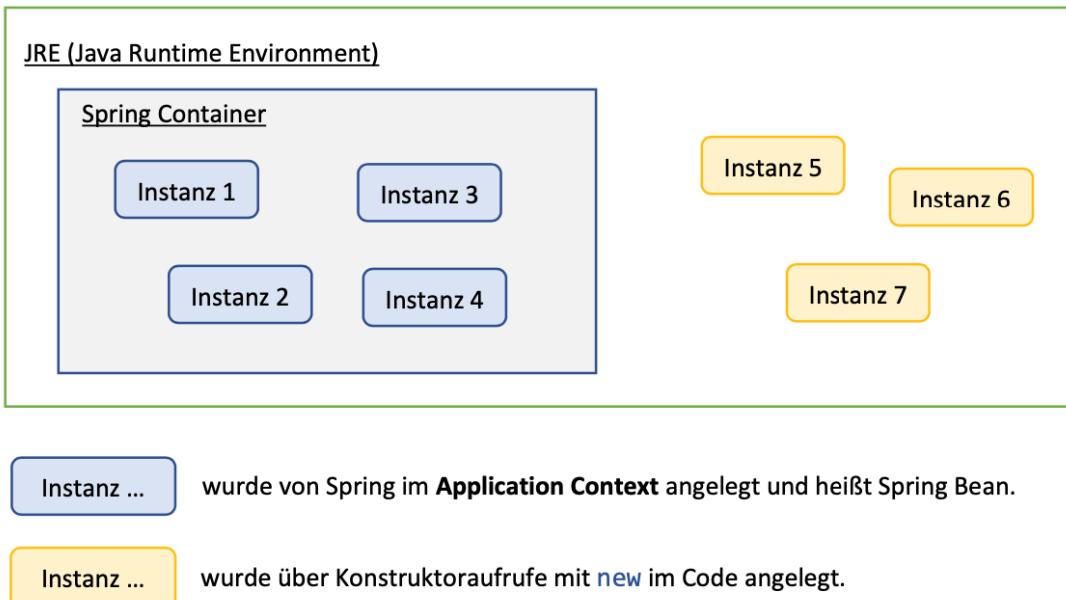


Abbildung 25: Spring Beans und ”normale“ Instanzen in einer Spring Applikation

Wo kommt der Application Context her? Ein Blick in unser Main beantwortet diese Frage. Der Aufruf der statischen Methode `run` im `main` erzeugt einen Application Context und gibt ihn sogar als Rückgabewert zurück.

```
@SpringBootApplication
public class GreetingsApplication {

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

Listing 10: Instantiieren des Application Contexts

### 7.1.1 Ausgabe aller Beans beim Programmstart

Um ein besseres Gefühl für den Application Context und seine Machenschaften beim Starten einer Spring Applikation zu bekommen, erweitern wir unser Main um Code der uns Zugang zu den erzeugten Spring Beans gibt.

```
@SpringBootApplication
public class GreetingsApplication {

    public static void main(String[] args) {
        ApplicationContext appContext =
            SpringApplication.run(GreetingsApplication.class, args);

        // print names of beans
        String[] beans = appContext.getBeanDefinitionNames();
        Arrays.sort(beans);
        for ( String bean : beans )
            System.out.println(bean);

        // print number of all beans (=automatically created instances)
        System.out.printf("%s beans were created.%n",
                           appContext.getBeanDefinitionCount());
    }
}
```

Listing 11: Ausgabe aller beim Start erzeugten Spring Beans

Der Konsolenoutput beim Starten der Spring Applikation zeigt, dass der Application Context eine ganze Menge an Beans erzeugt.

Auch unsere Klasse `GreetingController` wurde in eine Bean umgewandelt und mit `greetingController` ausgegeben. Sogar unsere Application Klasse `GreetingsApplication` wurde durch die Annotation `@SpringBootApplication` in ein Bean umgewandelt worden.

Konsolenausgabe nach dem Programmstart:

```
applicationTaskExecutor
basicErrorController
beanNameHandlerMapping
beanNameViewResolver
...
greetingController
greetingsApplication
...
websocketServletWebServerCustomizer
welcomePageHandlerMapping
124 beans were created.
```

### 7.1.2 Zugriff auf vom Application Context erstellte Beans

Der Application Context liefert uns nicht nur die Anzahl und die Namen von Beans, sondern man kann auch eine der zahlreichen `getBean`-Methoden auch das Bean selbst erhalten.

Um das zu demonstrieren erweitern wir unser Main um folgenden Codezeilen. Wir lassen uns im Main vom Application Context über den Klassennamen das dazugehörige Bean zurückgeben.

```
@SpringBootApplication
public class GreetingsApplication {

    public static void main(String[] args) {
        ...

        GreetingController gc1 = appContext.getBean(GreetingController.class);
        System.out.println(gc1);

        GreetingController gc2 = appContext.getBean(GreetingController.class);
        System.out.println(gc2);
    }
}
```

Listing 12: Zweimaliger Aufruf von `getBean` im Main

Nach dem Starten des Programms gibt die Konsole die Instanz unseres GreetingsControllers zweimal aus. Der Hash `2cae9b8` zeigt, dass beide Methodenaufrufe von `getBean` immer die gleiche Instanz zurückgeben.

*Konsolenausgabe:*

```
at.htldonaustadt.slog.greetings.presentation.GreetingController@2cae9b8
at.htldonaustadt.slog.greetings.presentation.GreetingController@2cae9b8
```

Das mag nicht verwunderlich sein, da wir in unserem Programm nur einen `GreetingController` benötigen.

Bei guten ProgrammiererInnen und LeserInnen dieses Skriptums ergibt sich wohl folgende Frage:

“Alles gut und schön! Spring erstellt also für meine annotierten Klassen automatisch eine Instanz. Was mache ich aber, wenn ich von einer solchen Klasse nicht nur eine Instanz, sondern mehrere benötige?”

[Gute ProgrammiererIn und LeserIn beim Lesen dieses Skriptums]

Die Antwort auf die Frage liefert im Spring Framework der sogenannte **Scope** eines Beans, der im nächsten Abschnitt besprochen wird.

### 7.1.3 Der Scope eines Beans (@Scope)

Ein Bean hat einen bestimmten Scope. Es gibt zwei verschiedenen Arten, die bei der Klasse des Beans über die Annotation `@Scope` ausgewählt werden kann.

- **Singleton:** Von einer Klasse mit dem Scope *Singleton* existiert innerhalb eines Application Contexts stets nur eine einzige Instanz<sup>8</sup>. Es ist unmöglich eine weiteres Bean anzulegen.  
**Der Scope *Singleton* ist der Default für alle Beans.**
- **Prototype:** Von einer Klasse mit dem Scope *Prototype* kann vom Application Context jederzeit eine weitere Instanz erzeugt werden.

In unserem Beispiel des `GreetingController`s fehlt die Annotation `@Scope`, da der Scope *Prototype* der Default ist und stets weggelassen werden kann.

```
@RestController  
public class GreetingController {  
    ...  
}
```

Durch das Hinzufügen der Annotation verändert sich das Verhalten also nicht.

```
@RestController  
@Scope(ConfigurableBeanFactory(SCOPE_SINGLETON))  
public class GreetingController {  
    ...  
}
```

Ändern wir den Scope auf *Prototype* um, so zeigt sich bei der Ausführung des Programms 12 ein wesentlicher Unterschied.

```
@RestController  
@Scope(ConfigurableBeanFactory(SCOPE_PROTOTYPE))  
public class GreetingController {  
    ...  
}
```

Konsolenausgabe:

```
at.htldonaustadt.slog.greetings.presentation.GreetingController@588f63c1  
at.htldonaustadt.slog.greetings.presentation.GreetingController@5a6fa56e
```

Die ausgegebenen Hash-Werte der beiden Instanzen `gc1` und `gc2` sind nicht mehr gleich!

<sup>8</sup>Spring Applications können grundsätzlich mehrere Application Contexte besitzen.

Beim zweiten Aufruf von `getBean` erstellt der Application Context also eine neue Instanz und gibt sie uns zurück.

Festzuhalten ist also, Beans werden nicht notwendigerweise beim Start einer Spring Application erstellt, sondern können auch während der Programmausführung angelegt werden.

## 7.2 Dependency Injection (DI)

Zuerst die Definition dieses Begriffs.

### Dependency Injection

**Dependency Injection (DI)** ist ein Entwurfsmuster, dass allgemein in der objektorientierten Programmierung anwendbar ist, existiert also auch abseits von Spring.

Dependency Injection bedeutet, dass Klassen ihre Abhängigkeiten - das können Informationen (=Daten) oder ein bestimmtes Verhalten (=Methoden) sein - von einer anderen, externen Instanz zugewiesen bekommen.

Diese "Injektion" kann von einem bestimmten Framework - wie etwa dem Spring oder im Gaming Bereich dem Unity-Framework - durchgeführt werden.

### 7.2.1 Arten von Dependency Injection

Man unterscheidet drei Arten von Dependency Injection [https://de.wikipedia.org/wiki/Dependency\\_Injection](https://de.wikipedia.org/wiki/Dependency_Injection) von denen die ersten beiden in einem Code Beispiel beleuchtet werden:

- **Constructor Injection:** Die Abhängigkeit wird über einen Konstruktor der Klasse gesetzt.
- **Setter Injection:** Die Abhängigkeit wird über einen Setter einer Instanzvariable gesetzt.
- **Interface Injection:** Die Klasse muss ein Interface implementieren, in das dann ein Verhalten von außen "injiziert" wird.

Alles klar? ... Hinter dem Begriff *Dependency Injection* und *Inversion of Control* versteckt sich ein Prinzip, das - soweit die Meinung des Autors - theoretisch erklärt weit schwerer zu begreifen ist, als wenn man sie in Code gegossen sieht.

Bevor ein sehr einfaches **Codebeispiel** das Prinzip zeigt, noch einige Bemerkungen zu Informationen, die man über dieses Entwurfsmuster findet.

### 7.2.2 Informationen zu Dependency Injection

Der Begriff geht ursprünglich auf Martin Fowler zurück, der ihn 2004<sup>9</sup> das erste Mal gebrauchte und den Begriff im lesenswerten Buch Clean Code beschreibt.

<sup>9</sup><https://martinfowler.com/articles/injection.html>

Interessante Artikel mit Erklärungen und Erläuterungen finden sich zu hauf im Internet.  
<https://www.microsoft.com/de-de/techwiese/know-how/was-ist-eigentlich-dependency-injection.aspx>  
<https://www.dev-insider.de/was-ist-dependency-injection-a-814452>  
etc.  
Speziell für Spring erklärt Eugen Paraschiv in seinem Blog ausführlich was Dependency Injection ist  
<https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>.

Das Prinzip wird nun anhand eines sehr einfachen, aber aussagekräftigen Beispiels gezeigt.

### 7.2.3 Beispiel zur Dependency Injection: Das Sortieren von Büchern

In einem Programm sollen Bücher sortiert werden. Die Sortierung ist Teil der Business Logik einer Enterprise Application.

Es gibt zwei Sortierreihenfolgen:

- Sortierung nach dem Buchtitel
- Sortierung nach dem Buchauthor

Es gibt zunächst folgende Klassen, die implementiert werden:

- Book: mit Instanzvariablen für einen Identifier, den Titel und den Author des Buches
- ComparerByID: Wird vom Interface Comparer<Book> abgeleitet und enthält den Vergleich für das aufsteigende Sortieren nach dem Identifier der Bücher.
- ComparerByName: Ist wie ComparerByID, vergleicht aber die Namen der Bücher.

1. **Anlegen des Projekts:** Wir erstellen ein Spring Projekt und wählen nur das Modul Lombok aus. Mehr Module sind nicht erforderlich, weil wir für dieses einfache Programm in dem das Prinzip der Dependency Injection gezeigt werden soll, nur das Main benötigen und keine Controller-Klassen. GroupId = at.htldonaustadt.slog und Artifact = di

2. **Anlegen eines Packages domain**

3. **Anlegen der Klassen:** Book und die beiden Comparer-Klassen ComparerByID und ComparerByName im Package domain.

Die fertige Packagestruktur des Projekts soll am Ende folgendermassen aussehen:

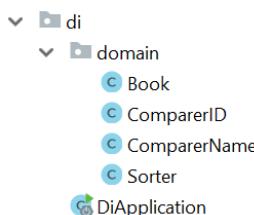


Abbildung 26: Packagestruktur des Demobeispiels für Dependency Injection

```

package at.htldonaustadt.slog.di.domain;

import lombok.*;

@Data
@AllArgsConstructor
public class Book {
    @NonNull private long id;
    @NonNull private String title;
    @NonNull private String author;
}

```

```

package at.htldonaustadt.slog.di.domain;

import java.util.Comparator;

public class ComparerByID implements Comparator<Book> {
    @Override
    public int compare(Book a, Book b) {
        if ( a == null )
            return b == null ? 0 : 1;
        else if ( b == null )
            return -1;

        return Long.compare(a.getId(), b.getId());
    }
}

```

```

package at.htldonaustadt.slog.di.domain;

import java.util.Comparator;

public class ComparerByTitle implements Comparator<Book> {
    @Override
    public int compare(Book a, Book b) {
        if ( a == null )
            return b == null ? 0 : 1;
        else if ( b == null )
            return -1;

        return a.getTitle().compareTo(b.getTitle());
    }
}

```

Listing 13: Die Klassen Book, ComparerByID und ComparerByTitle

#### 4. Anlegen einer Klasse **Sorter** die Bücher sortieren kann.

Die Klasse **Sorter** stellt das Sortieren von Büchern zu Verfügung. Das konkrete Verhalten kennt sie selbst aber nicht, weil sie nur eine Instanzvariable enthält, die festlegt, wie das Sortieren tatsächlich durchgeführt wird.

Die Klasse **Sorter** ist also abhängig von **Comparator<Book>**.  
**Comparator<Book>** ist also eine Dependency von **Sorter**.

```
package at.htldonaustadt.slog.di.domain;

import java.util.List;
import java.util.ArrayList;
import java.util.Comparator;

public class Sorter {
    private Comparator<Book> comparator;

    public Sorter(Comparator<Book> comparator) {
        this.comparator = comparator;
    }

    public List<Book> sort(List<Book> books) {
        // copy input list to new list and sort it
        List<Book> sorted = new ArrayList<>(books);
        sorted.sort(comparator);
        return sorted;
    }
}
```

Listing 14: domain/Sorter.java

Im konkreten Fall, wird diese Dependency über den Konstruktor "injiziert". Man spricht daher von **Constructor Injection**.

Hätte die Klasse nur den argumentlosen Default-Konstruktor und stattdessen einen public Setter für die Instanzvariable **comparator**, würde man von einer **Setter Injection** sprechen.

#### 5. Im **Main** wird eine Liste von Büchern erstellt und die Sortierung ausprobiert.

**Beachte:** Alle Instanzen werden zunächst mit Konstruktoraufrufen angelegt, d. h. keine der Instanzen ist ein Spring Bean! Der Code würde in einer Konsolenanwendung exakt gleich funktionieren.

Dem Konstruktor wird eine Instanz von `ComparerByID` "injiziert", daher sortiert die Methode `sort` nach der ID der Bücher.

```
public static void main(String[] args) {  
    SpringApplication.run(DiApplication.class, args);  
  
    // create instance without Spring  
    Sorter sorter = new Sorter( new ComparerByID() );  
  
    List<Book> books = new ArrayList<>( Arrays.asList(  
        new Book(80, "C++ Programmieren für Einsteiger", "Bonacina"),  
        new Book(41, "Spring Boot in Action", "Walls"),  
        new Book(50, "Java Design Patterns", "Sarcar"),  
        new Book(38, "Java Socket programming", "Malik"),  
        new Book(62, "Schrödinger programmiert Java", "Ackermann")  
    ));  
  
    List<Book> sortedBooks = sorter.sort(books);  
  
    for ( Book book : sortedBooks )  
        System.out.println(book);  
}
```

Listing 15: DiApplication.java mit dem Main

Konsolenausgabe:

```
Book(id=38, title=Java Socket programming, author=Mazhar Hussain Malik)  
Book(id=41, title=Spring Boot in Action, author=Craig Walls)  
Book(id=50, title=Java Design Patterns, author=Vaskaran Sarcar)  
Book(id=62, title=Schrödinger programmiert Java, author=Philip Ackermann)  
Book(id=80, title=C++ Programmieren für Einsteiger, author=Michael Bonacina)
```

"Injiziert" man dem Konstruktor dagegen eine Instanz von `ComparerByTitle` sortiert die Methode `sort` natürlich nach den Buchtiteln.

```
// create instance without Spring  
Sorter sorter = new Sorter( new ComparerByTitle() );
```

Konsolenausgabe:

```
Book(id=80, title=C++ Programmieren für Einsteiger, author=Michael Bonacina)  
Book(id=50, title=Java Design Patterns, author=Vaskaran Sarcar)  
Book(id=38, title=Java Socket programming, author=Mazhar Hussain Malik)  
Book(id=62, title=Schrödinger programmiert Java, author=Philip Ackermann)  
Book(id=41, title=Spring Boot in Action, author=Craig Walls)
```

Im nächsten Abschnitt wird erklärt, wie Dependency Injection in Spring funktioniert.

#### 7.2.4 Constructor Injection im Codebeispiel

Da die Sortierung - wie am Beginn erwähnt - Teil der Business Logik einer Enterprise Application ist, sollen die Klassen `Sorter`, `ComparerByID` und `ComparerByTitle` Spring Components bzw. Spring Beans sein und vom Application Context verwaltet werden.

Damit die Klassen Teile einer Business Logik einer Spring Applikation werden, müssen sie mit Annotations versehen werden, damit der Application Context beim Programmstart, Instanzen automatisch anlegen kann.

Aber nicht nur das Erstellen der Instanzen, sondern auch das "Injizieren" der Dependency muss vom Application Context beim Erstellen des Sorters automatisch erledigt werden.

Wie später im Skriptum erklärt wird, erhalten alle Klasse der Business Logic die Annotation `@Service`. Da die beiden Comparer, nur Hilfsklassen sind, die nicht einmal über Instanzvariablen verfügen, wählen wir für diese die einfachste Annotation `@Component`.

```
@Component  
public class ComparerByID implements Comparator<Book> {  
    ...  
}
```

```
@Component  
public class ComparerByTitle implements Comparator<Book> {  
    ...  
}
```

```
@Service  
public class Sorter {  
    private Comparator<Book> comparator;  
  
    public Sorter(Comparator<Book> comparator) {  
        this.comparator = comparator;  
    }  
    ...  
}
```

Listing 16: Annotations vor den Klassen `Sorter`, `ComparerByID` und `ComparerByTitle`

Im Main ersetzen wir den Konstruktorauftruf mit dem im letzten Abschnitt die vom Application Context erstellte Instanz holt mit `getBean` holt.

```

@SpringBootApplication
public class DiApplication {

    public static void main(String[] args) {

        ApplicationContext appContext =
            SpringApplication.run(DiApplication.class, args);

        Sorter sorter = appContext.getBean(Sorter.class);
        //Sorter sorter = new Sorter(new ComparerByTitle());
        ...

    }
}

```

Listing 17: Annotations vor den Klassen Book, ComparerByID und ComparerByTitle

Ein (hoffnungsvoller) Start des Compilers zeigt uns allerdings, dass wir einen wichtigen Punkt übersehen haben. Der Konstruktor unseres Sorters benötigt einen Inputparameter, nämlich eine Instanz eines Comparers.

*Fehlerausgabe auf der Konsole beim Programmstart:*

```

Parameter 0 of constructor in at.htldonaustadt.slog.di.domain.Sorter required a
single bean, but 2 were found:
- comparerByID: defined in file [.../domain/ComparerByID.class]
- comparerByTitle: defined in file [..../domain/ComparerByTitle.class]

```

### Mehrdeutigkeiten beim Anlegen von Beans

Der Application Context ruft beim Erstellen eines Beans bzw. einer Component den Konstruktor der Klasse auf. Dabei müssen natürlich alle Inputparameter des Konstruktoraufrufs einen Wert haben.

Der Application Context sucht beim Anlegen aller Beans selbstständig aufgrund des Datentyps nach möglichen Beans, die es als Inputparameter beim Konstruktorauftrag "injizieren" kann. Wenn mehr als ein Kandidat gefunden wird, terminiert das Programm mit einem Fehler.

Für die Lösung dieses Problems stehen in Spring viele Wege offen. Die einzige Möglichkeit die wir hier in diesem Skriptum besprechen, ist die Qualifizierung der Componenten mit Hilfe der `@Qualifier` Annotation.

Mit der Annotation `@Qualifier` kann man jeder Klasse einen eigenen Namen geben, auf den man im Code von anderen Klassen referenzieren kann. In unserem Beispiel

```
@Component
@Qualifier("comparer-by-id")
public class ComparerByID implements Comparator<Book> {
    ...
}
```

```
@Component
@Qualifier("comparer-by-title")
public class ComparerByTitle implements Comparator<Book> {
    ...
}
```

```
@Service
public class Sorter {
    private Comparator<Book> comparator;

    public Sorter(@Qualifier("comparer-by-id") Comparator<Book> comparator) {
        this.comparator = comparator;
    }
    ...
}
```

Listing 18: Verwendung der Annotation `@Qualifier` zum Auflösen von Mehrdeutigkeiten bei Konstruktoraufrufen

Nach diesem Verknüpfen des Inputparameters im Konstruktor von `Sorter` mit der Component `ComparerByID` kann der Application Context nun das Bean für den Sorter anlegen.

### 7.2.5 Merkregel: Der häufigste Fall von Constructor Injection

Bei sehr vielen Klassen in der Praxis tritt das Problem mit der Mehrdeutigkeit gar nicht auf. Häufig gibt es zu einem Datentyp ohnehin nur ein Bean und der Application Context kann dieses gar einfach selbst bestimmten und dem Konstruktor übergeben.

Dadurch entfällt die Angabe von `@Qualifier` beim Konstruktorparameter und man kann Lombok-Annotations verwenden.

Wenn nur eine Comparer-Klasse da ist, kann man den Sorter mit den Lombok-Konstruktor- Annotations sehr einfach implementieren.

Wenn die einzige Instanzvariable bei einem existierenden Bean geändert werden könnte soll, ist folgender Code die kürzeste Variante für Constructor Injection.

```
@AllArgsConstructor  
@Setter  
@Service  
public class Sorter {  
  
    private Comparator<Book> comparator;  
  
    public List<Book> sort(List<Book> books) {  
        // copy input list to new list and sort it  
        List<Book> sorted = new ArrayList<>(books);  
        sorted.sort(comparator);  
        return sorted;  
    }  
}
```

Listing 19: Constructor Injection mit einer einzelnen, nicht `final` Instanzvariable

Wenn die einzige Instanzvariable nur im Konstruktor gesetzt werden darf - also, wenn sie `final` ist, ist folgender Fall sinnvoll.

```
@RequiredArgsConstructor  
@Service  
public class Sorter {  
  
    private final Comparator<Book> comparator;  
  
    public List<Book> sort(List<Book> books) {  
        // copy input list to new list and sort it  
        List<Book> sorted = new ArrayList<>(books);  
        sorted.sort(comparator);  
        return sorted;  
    }  
}
```

Listing 20: Constructor Injection mit einer einzigen `final` Instanzvariable

### 7.2.6 Setter Injection im Codebeispiel

Die Klasse `Sorter` in unserem Codebeispiel ist nicht sehr flexibel. Der Comparer der im Konstruktor übergeben wird, kann nicht verändert werden. Beispielsweise sind Database Repositories oder JPA-Repositories nicht mehrdeutig, sondern eindeutig bestimmt.

In realistischen Applikationen kann es sinnvoll sein, den Comparer auch umsetzen zu können, indem man einen public Setter hinzufügt. Um Setter Injection demonstrieren zu können, entfernen wir den Konstruktor vollständig. Es existiert daher nur der Default-Konstruktor.

```
@Service
public class Sorter {

    private Comparator<Book> comparator;

    public void setComparator(Comparator<Book> comparator) {
        this.comparator = comparator;
    }

    public List<Book> sort(List<Book> books) {
        // copy input list to new list and sort it
        List<Book> sorted = new ArrayList<>(books);
        sorted.sort(comparator);
        return sorted;
    }
}
```

Listing 21: Die Klasse `Sorter` mit einem Setter statt einem Konstruktor

Das Main ist nach wie vor unverändert, wie in Listing 17. Der Application Context erzeugt daher ein Bean und setzt die Instanzvariable gar nicht.

Beim Starten des Programms, stürzt das Sortieren daher ab, weil in `sorted.sort(comparator);` `null` übergeben wird.

Dafür gibt es die Lösung, dass man die Instanzvariable fix mit dem gewünschten Comparer "verdrahtet". Das funktioniert mit der Annotation `@Autowired`. Da es mehr als einen Comparer gibt, müssen wir zusätzlich mit `@QName` angeben, welches Bean wir fest verknüpfen wollen. Dieses Verknüpfen wird im Jargon von Java Spring als "autowiren" bezeichnet.

```
@Service
public class Sorter {
    @Autowired
    @Qualifier("comparer-by-id")
    private Comparator<Book> comparator;
    ...
}
```

Listing 22: Autowiring einer Instanzvariable

## 8 Das Buildtool Maven

Build Tools dienen dazu das Kompilieren, das Linken und das Zusammenpacken (Packaging) einer Applikation oder Library zu automatisieren.

Warum ist das notwendig? In kleinen Projekten ist das Zusammenbauen von Executables meist kein Problem. Bei großen Projekten benötigt man dagegen eine genaue Übersicht über alle Komponenten und deren Versionen. Es ist wesentlich, dass der Build Prozess konsistent und sauber abläuft. Sie sind daher im Entwicklungsprozess von fundamentaler Wichtigkeit.

Die beiden bekanntesten und verbreitetsten Build Tools für Java Projekte sind

- **Maven** <https://maven.apache.org> und
- **Gradle** <https://gradle.org>.

Bei Maven werden alle Bibliotheken als Dependency (Abhängigkeiten) im **Project Model File** (*pom.xml*) eingetragen.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  ...
</dependencies>
```

Listing 23: Zwei Dependencies im Project Model File einer Spring Applikation

Dependencies werden am einfachsten beim Anlegen des Projekts auf <https://start.spring.io> ausgewählt und dann automatisch ins *pom.xml* File eingetragen. In IntelliJ ist diese Seite bereits in die IDE integriert.

Beim Eintragen einer Dependency ladet Maven automatisch die betroffene Bibliothek (Package) aus dem Internet auf sein Repository herunter, falls das entsprechende JAR-File in der korrekten Version noch nicht im lokalen Repository vorhanden ist.

Die meisten JAR-Files werden von CENTRAL heruntergeladen, das die Adresse <https://repo1.maven.org/maven2> hat.

Ein Verzeichnis für die Dependencies befindet sich im Maven Repository unter der Adresse <http://mvnrepository.com>.

Im Project Explorer findet man alle von Maven heruntergeladenen Libraries unter *External Libraries* in IntelliJ.

Dort sind alle im Projekt eingebundenen Bibliotheken (Packages, Spring Module) ersichtlich.

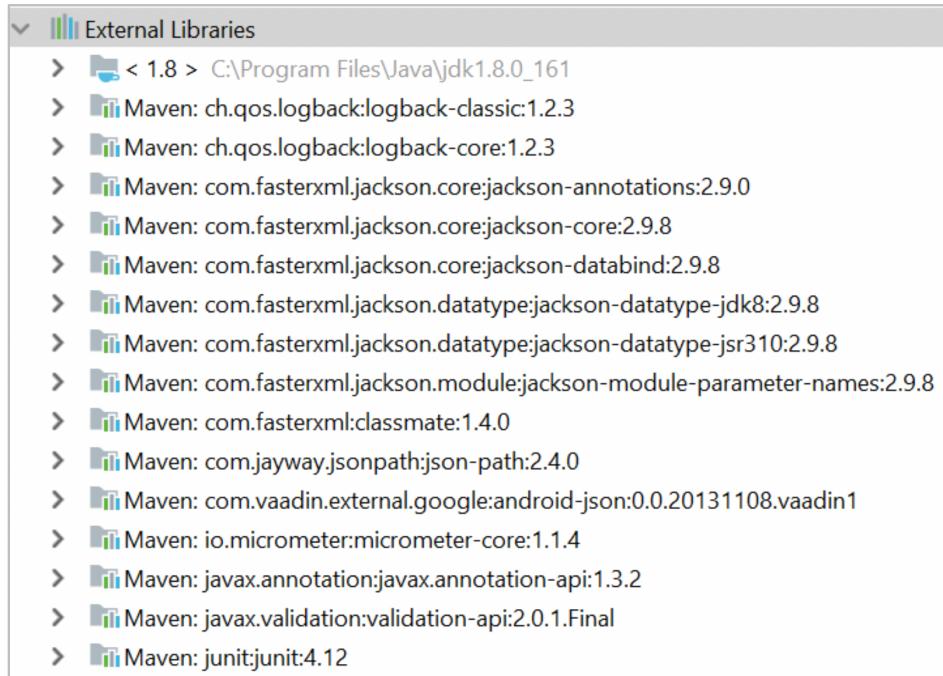


Abbildung 27: External Libraries in einem Maven Projekt

Maven speichert alle externen Libraries in einem lokalen Repository ab <https://www.baeldung.com/maven-local-repository>.

| Default Pfade der Maven Repositories |                          |
|--------------------------------------|--------------------------|
| Windows                              | C:\Users\<user-name>\.m2 |
| Linux                                | /home/<user-name>/.m2    |
| MacOS                                | /Users/<user-name>/.m2   |

## 8.1 Übungen zu Maven

1. Öffne das *pom.xml* File eines bestehenden Spring Projekts das Maven verwendet.
2. Gehe auf <http://mvnrepository.com>, suche Dir die JSON-Library *org.json* heraus und wähle die letztgültige Version der Library durch Anklicken an.

The screenshot shows the Maven dependency page for the *org.json* library. At the top, there's a navigation bar with links to Home, org.json, json, and the specific version 20190722. Below this is a large header "JSON In Java > 20190722" with a logo. A brief description follows: "JSON is a light-weight, language independent, data interchange encoders/decoders in Java. It also includes the capability to correctly implement. There is a large number of JSON packages in Java, choose carefully. The license includes this restriction: 'The software is provided 'as-is' without warranty of any kind, either expressed or implied'." Below the description is a table with the following data:

|                     |   |
|---------------------|---|
| <b>License</b>      | JSON  |
| <b>Categories</b>   | JSON Libraries  |
| <b>HomePage</b>     | <a href="https://github.com/douglascrockford/JSON-java">https://github.com/douglascrockford/JSON-java</a> |
| <b>Date</b>         | (Aug 07, 2019)  |
| <b>Files</b>        | <a href="#">bundle (63 KB)</a> <a href="#">View All</a>   |
| <b>Repositories</b> | Central   |
| <b>Used By</b>      | 3,374 artifacts   |

Below the table are tabs for Maven, Gradle, SBT, Ivy, Grape, Leiningen, and Buildr. Underneath these tabs is a code snippet of the Maven dependency XML:

```
<!-- https://mvnrepository.com/artifact/org.json/json -->
<dependency>
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20190722</version>
</dependency>
```

Abbildung 28: Die Maven Dependency der Library *org.json* auf <https://mvnrepository.com>

3. Füge dann die Dependency im *pom.xml*-File an die richtige Stelle ein. Kontrolliere dann, dass das JAR-File für *org.json* unter den *External Libraries* erscheint.
4. Gehe dann auf Deinem Computer in das Verzeichnis des lokalen Maven Repositories und navi- giere an die Stelle wo das JAR-File von *org.json* gespeichert ist, also */Users/<user-name>/ .m2/repository/org/json/json/20190722*. Lösche dann das File oder das ganze Verzeichnis weg. Gehe dann in deine IDE und aktualisiere Maven.

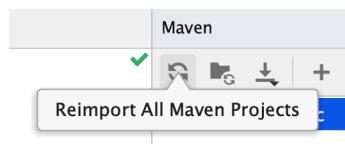


Abbildung 29: Aktualisieren des lokalen Maven Repositories im Maven-Fenster von IntelliJ

# 9 Loggen in Java mit `slf4j.Logger`

Loggen bedeutet Statusinformationen über Variablen während des Programmlaufs gemeinsam mit einem Zeitstempel (Timestamp) auf der Konsole auszugeben. Logs sind wichtige Hilfsmittel um zu späteren Zeitpunkten auf Aktionen eines Programms rückschließen zu können. Besonders wertvoll ist dies natürlich während einer Fehlersuche.

Anstatt ein Programm während der Entwicklungsphase mit `System.out.print` zuzupflastern, die dann vor der Release (Auslieferung) des produktiven Systems wieder entfernt werden müssen, ist es sinnvoll gleich einen Logger zu verwenden.

Eines der einfachsten Logger-APIs in Java, ist `slf4j.Logger` <https://www.slf4j.org/docs.html>.

## 9.1 Importieren der Logger-Klasse

```
import org.slf4j.Logger;
```

## 9.2 Anlegen einer Loggerinstanz

```
Logger logger = LoggerFactory.getLogger(SorterApplication.class);
```

## 9.3 Loggen von Variablenwerten

Die Methode `info` ersetzt die `System.out.print` Aufrufe.

```
logger.info(sorter.toString());
// instead of System.out.println(sorter.toString());
```

oder als formatierte Ausgabe

```
logger.info("Sorter: {}", sorter.toString());
// instead of System.out.println("Sorter: " + sorter.toString());
```

Ausgabe auf der Konsole:

```
23:58:07.775 [main] INFO at.htldonaustadt.slog.didemo.SorterApplication - Sorter:
at.htldonaustadt.slog.didemo.Sorter@353352b6
```

# 10 SQL-Datenbanken anbinden mit Spring JDBC

Unsere beiden bisher erstellten Webapplikationen liefern nur hartkodierte Daten zurück. Es ist nun endlich an der Zeit unsere Daten aus Datenbanken zu extrahieren.

Wir werden dazu Daten von Personen in einer Datenbank speichern, über Endpoints hinzufügen, verändern und entfernen.

Dafür müssen wir nicht nur den Presentation Layer, sondern auch den darunterliegenden Service Layer und Persistence Layer.

Im ersten Anlauf werden wir eine Applikation erstellen, die alle Datenzugriffe mit SQL-Statements realisieren und dafür JDBC verwenden.

Das ist ein - gegenüber JPA<sup>10</sup> - weitaus aufwendigere Weg, der aber sehr deutlich die Aufgabe der einzelnen Layer zeigt.

## 10.1 H2 Datenbank

Im ersten Schritt verwenden wir die In-Memory Datenbank H2 (<https://www.h2database.com/html/main.html>). H2 ist ein in Java geschriebenes relationales Datenbanksystem, das über SQL und JDBC verwendet werden kann. Die Datenbank wird in der Entwicklung von Spring Applikationen häufig eingesetzt, weil sie im JAR-File embedded wird und nicht eigens installiert werden muss. Man kann damit sehr rasch Dinge austesten.

Einen Überblick über H2 und Spring gibt der Blogbeitrag <https://www.baeldung.com/spring-boot-h2-database>

## 10.2 Anlegen des Projekts

Wir erstellen eine Applikation mit einer H2-Datenbank und beginnen damit, dass wir ein Projekt anlegen, dass die folgenden Maven-Dependencies hat:

- **Lombok:** Zur Reduktion von boiler-plate Code in Klassen.
- **Web:** Webapplikation
- **JDBC:** Einbindung der Java Database Connectivity um auf die Datenbank zugreifen zu können.
- **H2:** Einbindung der H2-Datenbank

## 10.3 Die Application Properties (`application.properties`)

In Java Spring Applikationen können mit Application Properties gestartet werden. Über diese Properties werden bestimmte Parameter in den Spring Modulen gesetzt um diese zu konfigurieren. Damit können zum Beispiel Fähigkeiten freigeschalten werden.

---

<sup>10</sup>[https://de.wikipedia.org/wiki/Java\\_Persistence\\_API](https://de.wikipedia.org/wiki/Java_Persistence_API)

Diese Properties werden über das Textfile `application.properties`, das sich im `resources`-Folder befindet editiert. Das File ist normalerweise leer. Eine Übersicht über alle möglichen Application Properties sieht man z. B. hier <https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html>.

In unserem Fall möchten wir die Datenbank unserer Webapplikation über den Browser zugreifbar machen.

`application.properties:`

```
spring.h2.console.enabled=true
```

## 10.4 Starten des Projekts

Ohne auch nur sonst irgendetwas implementiert zu haben, starten wir den Browser und surfen auf die Datenbankkonsole.

*Adresszeile des Browsers:*

```
http://localhost:8080/h2-console
```

Die Applikation liefert dann dem Browser als Response den Loginscreen in die H2-Datenbank.

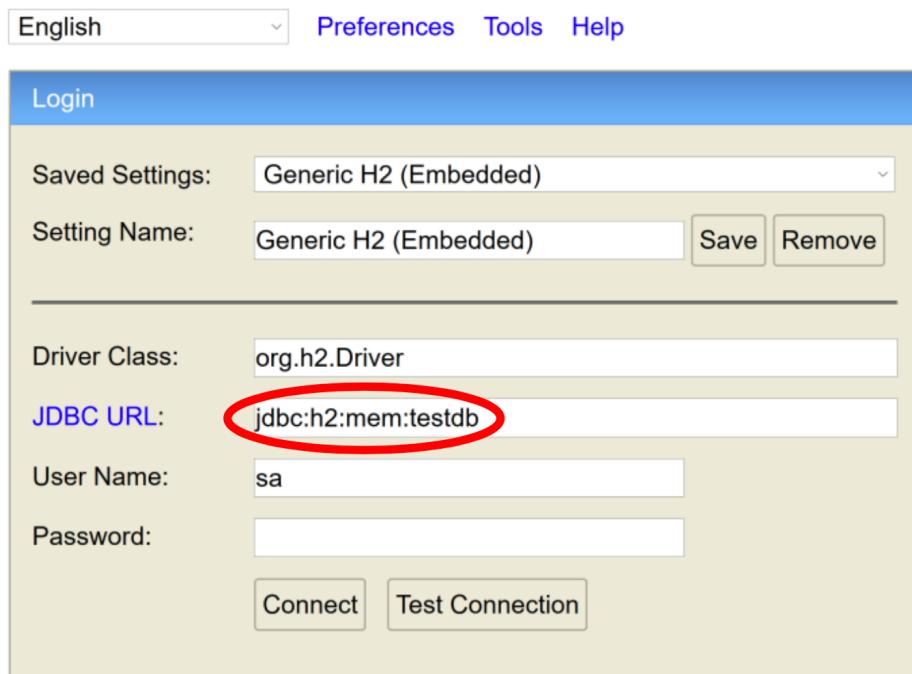


Abbildung 30: Login Screen in die H2-Testdatabase

Für ein erfolgreiches Einloggen in unsere H2-Datenbank, muss die JDBC-URL auf `jdbc:h2:mem:testdb` geändert werden!

Die Login-Daten können über das Application Property File angepasst werden.

Was wir dann erhalten ist ein vollständiger Database-Client.

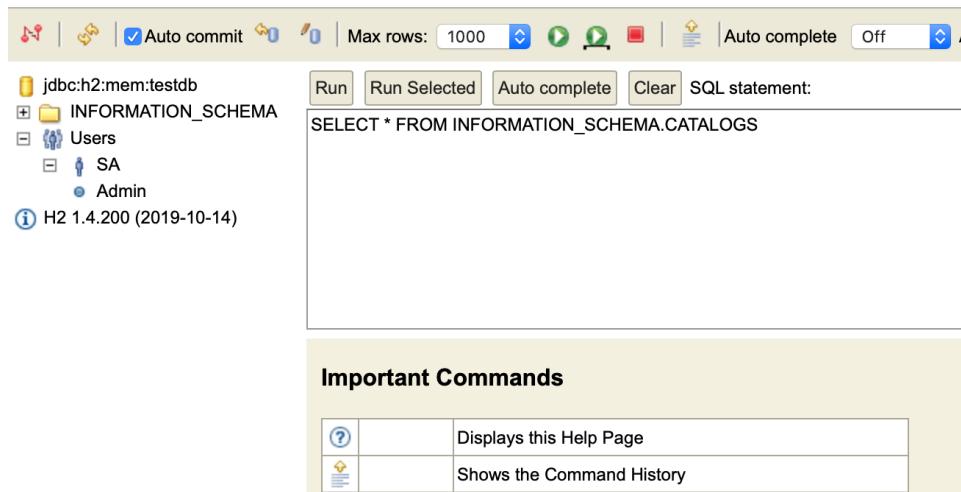


Abbildung 31: Database Client der Test-H2-Datenbank im Browser

## 10.5 Anlegen und Befüllen einer Datenbanktabelle

Da H2 eine In-Memory Database ist, wird sie jedesmal gelöscht, wenn unsere Webapplikation beendet wird.

Daher müssen die Tabellen, die von der Applikation benötigt werden, beim Programmstart frisch angelegt werden und über einen SQL-Skript am Beginn insertiert werden. Unter JDBC besteht die Möglichkeit, die SQL-Create-Statements selbst zu schreiben.

Dafür wird im Folder `resources` das File `schema.sql` angelegt. Dieses wird automatisch beim Start von Spring ausgeführt.

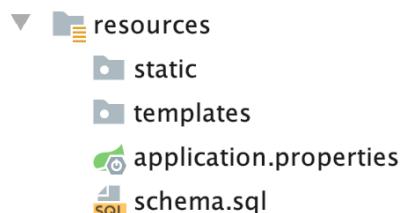


Abbildung 32: Der SQL-Skript `schema.sql` im Verzeichnisbaum.

```
create table person (
    id integer not null,
    name varchar(255) not null,
    location varchar(255),
    primary key(id)
);
```

Listing 24: Inhalt von `schema.sql`

Nach einem **Restart der Applikation** und einem erneuten Anmelden in der Datenbank, zeigt der Browser die Tabelle und alle Columns an.

Nun können wir - wie in jedem Database-Client auch - die Tabelle mit einigen Datensätzen befüllen.

The screenshot shows a database interface with the following details:

- Database Connection:** jdbc:h2:mem:testdb
- Schemas:** PERSON, INFORMATION\_SCHEMA, Users
- Version:** H2 1.4.199 (2019-03-13)
- SQL Statement:** A series of four INSERT statements into the 'PERSON' table:
  - insert into person values (10000,'Donald Duck','Entenhausen');
  - insert into person values (10001,'Daisy Duck','Entenhausen');
  - insert into person values (10002,'Micky Mouse','Wunderhaus');
  - insert into person values (10003,'Asterix','Aremorica');
- Execution Results:**
  - insert into person values (10000,'Donald Duck','Entenhausen'); Update count: 1 (0 ms)
  - insert into person values (10001,'Daisy Duck','Entenhausen'); Update count: 1 (0 ms)
  - insert into person values (10002,'Micky Mouse','Wunderhaus'); Update count: 1 (0 ms)

Abbildung 33: Befüllen der Tabelle PERSON im Browser

Anschließend, vergewissern wir uns, dass H2 sinnvolle Daten insertiert hat.

The screenshot shows a database interface with the following details:

- Database Connection:** jdbc:h2:mem:testdb
- Schemas:** PERSON, INFORMATION\_SCHEMA, Users
- Version:** H2 1.4.199 (2019-03-13)
- SQL Statement:** SELECT \* FROM PERSON
- Execution Results:**
  - SELECT \* FROM PERSON;

| ID    | NAME        | LOCATION    |
|-------|-------------|-------------|
| 2000  | Obelix      | Rom         |
| 10000 | Donald Duck | Entenhausen |
| 10001 | Daisy Duck  | Entenhausen |
| 10002 | Micky Mouse | Wunderhaus  |
| 10003 | Asterix     | Aremorica   |

  - (5 rows, 6 ms)

Abbildung 34: Anzeige aller Datenbank Records der Tabelle PERSON

**Beachte:** Die Insertierung von Daten über einen solchen Skript hat in der Praxis den großen Nachteil, dass auch automatisch erzeugte Primärschlüsseln der Tabelle (autoincremented primary keys) in den SQL-Insert-Statements mit eingetragen werden müssen.

Nach einem erneuten Start der Applikation, ist der Tabelleninhalt natürlich wieder gelöscht, da H2 ja eine reine InMemory-Datenbank ist. Um nach dem Starten der Applikation die Einträge wieder fix in der Datenbank zu haben, müssen die entsprechenden Insert-SQL-Statements in einem zusätzlichen SQL-Skript `data.sql` hinzugefügt werden.

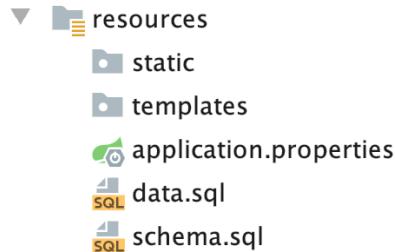


Abbildung 35: Die SQL-Skripts `data.sql` und `schema.sql` im Verzeichnisbaum

```
insert into person values(10000,'Donald Duck','Entenhausen');
insert into person values(10001,'Daisy Duck','Entenhausen');
insert into person values(10002,'Mickey Mouse','Wunderhaus');
insert into person values(10003,'Asterix','Aremorica');
```

Listing 25: Inhalt von `data.sql`

## 10.6 Erstellen einer Personenklasse im Domain Layer

Im Java Code muss eine Klasse für eine Person angelegt werden. Eine Instanz dieser Klasse entspricht einem Datenbankrecord, also einer Person in der Datenbank.

Das folgende Listing ist die Klasse `Person` im Domain Layer, also im Package `domain`. Die Namen der Instanzvariablen werden automatisch an die Felder der Datenbanktabelle gebunden.

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Person {
    private final int id;
    private String name;
    private String location;
}
```

## 10.7 Implementierung des Persistence Layers mit JDBC

Eine Klasse des Persistenz Layers abstrahiert den Zugriff auf ein Datenobjekt, wie z. B. auf eine Datenbank. In Spring wird so eine Klasse mit `@Repository` markiert. Ein Repository unterstützt - wie weiter vorne beschrieben - immer die CRUD-Operationen (=Create, Read, Update und Delete).

Unsere Repository-Klasse setzt mit JDBC, alle Zugriffe auf die Datenbank mit SQL-Statements ab. Dabei werden bei Queries (SQL-Select) die erhaltenen Datenrecords über einen **RowMapper** in Instanzen von Klassen umgewandelt.

| CRUD-Operationen in einer Datenbank |                           |  |
|-------------------------------------|---------------------------|--|
| Name                                | Beschreibung              | SQL-Statement als Beispiel                                       |
| Create                              | Einfügen von Datenrecords | <code>insert into person values (10020, 'Max', 'Vienna');</code> |
| Read                                | Lesen von Datenrecords    | <code>select * from person where location='Vienna';</code>       |
| Update                              | Ändern von Datenrecords   | <code>update person set location='Graz' where id=10020;</code>   |
| Delete                              | Löschen von Datenrecords  | <code>delete from person where location=10001;</code>            |

Wir implementieren nun die Klasse `PersonJdbcRepository` so, dass die Methoden diese SQL-Statements realisieren.

### 10.7.1 SQL-Parameter Injection

Gerade bei Where-Klauseln in SQL ist es notwendig, Parameterwerte für die Filterung von außen mitgeben zu können. Das Injecten der Parameterwerte in ein SQL-Statement erfolgt durch Parameterbinding über ?. Anstelle des Parameterwerts wird ein ? im SQL-Statement eingefügt. Das JDBC-Template fügt dann den Wert aus einem Object-Array ein. Die Anzahl der ? muss exakt der Anzahl der Werte im Object[] entsprechen. Bei Queries (SQL-Select-Statements) werden die erhaltenen Datenrecords über einen RowMapper in Instanzen von Klassen umgewandelt.

## 10.8 Die JDBC-Repository Klasse

Die Repository Klasse die im Persistence Layer für den Zugriff auf die H2-Datenbank verantwortlich ist, verwendet die Spring Klasse JdbcTemplate um die Datenbank-CRUD-Operationen aufzurufen.

```
@Repository
public class PersonJdbcRepository {
    @Autowired
    JdbcTemplate jdbcTemplate;

    public List<Person> selectAll() {
        return jdbcTemplate.query("select * from person",
            new BeanPropertyRowMapper<Person>(Person.class)
        );
    }

    public List<Person> selectByLocation(String location) {
        return jdbcTemplate.query( "select * from person where location=?",
            new Object[]{ location },
            new BeanPropertyRowMapper<Person>(Person.class)
        );
    }

    public void insert(Person person) {
        jdbcTemplate.update("insert into person values (?,?,?);",
            new Object[]{person.getId(),person.getName(),person.getLocation()}
        );
    }

    public void update(Person person) {
        jdbcTemplate.update("update person set name=?, location=? where id=?;",
            new Object[]{person.getName(),person.getLocation(),person.getId()}
        );
    }

    public void delete(int id) {
        jdbcTemplate.update("delete from person where id=?;",
            new Object[]{ id }
        );
    }
}
```

Listing 26: Die Repository Klasse für die Personen Tabelle der Datenbanks

### 10.8.1 Implementierung eines RowMappers für Personen

Bei Queries, ermöglicht der `BeanPropertyRowMapper` eine einfache Umwandlung von Datenbank-Records in Instanzen.

Die Umwandlung kann über das Interface `RowMapper<T>` auch selbst implementiert werden. Der untenstehende Code zeigt wie das geht.

`ResultSet` ist der JDBC-Typ für ein von der Datenbank zurückgegebenes Datenbank-Resultset bei einem SQL-Select-Statement. Der Zugriff auf die einzelnen Spalten jedes Records erfolgt über den Namen der Spalte oder den Index.

```
class PersonRowMapper implements RowMapper<Person> {
    @Override
    public Person mapRow(ResultSet resultSet, int i) throws SQLException {
        Person person = new Person();
        person.setId(resultSet.getInt("id"));
        person.setName(resultSet.getString("name"));
        person.setLocation(resultSet.getString("location"));
        return person;
    }
}
```

Listing 27: Definition eines RowMappers zum Umwandeln von Database Records auf Instanzen

```
@Repository
public class PersonJdbcRepository {
    @Autowired
    JdbcTemplate jdbcTemplate;

    public List<Person> selectAll() {
        return jdbcTemplate.query("select * from person",
            new PersonRowMapper()
        );
    ...
}
```

Listing 28: Verwendung des selbstdefinierten RowMappers in einer Query-Methode

## 10.9 Testen der JDBC-Repository Klasse in der Application

Um das Repository zu testen, verwenden wir zunächst das Main. Da dieses allerdings statisch ist, ist das Anlegen eines Beans für das `PersonJdbcRepository` nicht ohne weiteres möglich, da die Dependency Injection nur für (nicht statische) Instanzvariablen funktioniert und nicht für (statische) Klassenvariablen.

Eine oft verwendete Hilfestellung bietet das Interface `CommandLineRunner`, von dem man die Application ableitet und die `run`-Methode des Interfaces implementiert.

Die Methode `run` erhält die Commandline-Argumente als Inputparameter.

```
import org.slf4j.Logger;

@SpringBootApplication
public class PersondbApplication implements CommandLineRunner {

    public static void main(String[] args) {
        SpringApplication.run(PersondbApplication.class, args);
    }

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired
    PersonJdbcRepository personRepo;

    @Override
    public void run(String... args) throws Exception {
        // CREATE
        Person p = new Person(2000, "Obelix", "Aremorica");
        personRepo.insert(p);
        logger.info("insert Obelix -> {}", personRepo.selectAll());

        // READ
        List<Person> allpersons = personRepo.selectAll();
        logger.info("\n\nselect * from person -> {}", allpersons);

        List<Person> somepersons = personRepo.selectByLocation("Entenhausen");
        logger.info("\n\nselect all from Entenhausen -> {}", somepersons);

        // UPDATE: Obelix übersiedelt nach Rom
        p.setLocation("Rom");
        personRepo.update(p);
        logger.info("\n\nupdate location of Obelix -> {}",
                    personRepo.selectAll());

        // DELETE: Mickey Mouse entfernen
        personRepo.delete(10002);
        logger.info("\n\ndelete from person where id=10002 -> {}",
                    personRepo.selectAll());
    }
}
```

Listing 29: Testen der CRUD-Operationen der Repository-Klasse in der Methode `run` des `CommandLineRunner`

*Konsolenausgabe nach dem Programmstart:*

```
2019-11-16 00:10:06.087 INFO 4100 — [ main] :  
select * from person → [  
Personid=10000, name='Donald Duck', location='Entenhausen',  
Personid=10001, name='Daisy Duck', location='Entenhausen',  
Personid=10002, name='Micky Mouse', location='Wunderhaus',  
Personid=10003, name='Asterix', location='Aremorica']  
  
2019-11-16 00:10:06.101 INFO 4100 — [ main] :  
select all from Entenhausen → [  
Personid=10000, name='Donald Duck', location='Entenhausen',  
Personid=10001, name='Daisy Duck', location='Entenhausen']  
  
2019-11-16 00:10:06.102 INFO 4100 — [ main] :  
insert Obelix → [  
Personid=2000, name='Obelix', location='Aremorica',  
Personid=10000, name='Donald Duck', location='Entenhausen',  
Personid=10001, name='Daisy Duck', location='Entenhausen',  
Personid=10002, name='Micky Mouse', location='Wunderhaus',  
Personid=10003, name='Asterix', location='Aremorica']  
  
2019-11-16 00:10:06.106 INFO 4100 — [ main] :  
update location of Obelix → [  
Personid=2000, name='Obelix', location='Rom',  
Personid=10000, name='Donald Duck', location='Entenhausen',  
Personid=10001, name='Daisy Duck', location='Entenhausen',  
Personid=10002, name='Micky Mouse', location='Wunderhaus',  
Personid=10003, name='Asterix', location='Aremorica']  
  
2019-11-16 00:10:06.108 INFO 4100 — [ main] :  
delete from person where id=10002 → [  
Personid=2000, name='Obelix', location='Rom',  
Personid=10000, name='Donald Duck', location='Entenhausen',  
Personid=10001, name='Daisy Duck', location='Entenhausen',  
Personid=10003, name='Asterix', location='Aremorica']
```

Die Ausgabe zeigt, dass die vier CRUD-Operationen in der Repository-Klasse `PersonJdbcRepository` korrekt ausgeführt werden.

## 10.10 Übungsaufgabe

Schreibe eine Webapplikation mit Java Spring Boot, die das Verwalten von Internet Bookmarks ermöglicht.

Verwende JDBC und eine H2-Datenbank und füge beim Start zumindest 15 Bookmarks in die Datenbank ein.

Ein Bookmark besteht aus mindestens:

- Name
- URL
- Creation Time: siehe auch die Links  
<https://www.baeldung.com/hibernate-date-time>  
<https://stackoverflow.com/questions/14138532/h2-database-string-to-timestamp>  
[http://www.h2database.com/html/datatypes.html#timestamp\\_type](http://www.h2database.com/html/datatypes.html#timestamp_type)
- Schlagwörter (Tags), z. B. als CSV-Liste

Die Webanwendung soll folgende Endpoints anbieten:

### 10.10.1 Rest-Webservice API

- Einfügen eines neuen Bookmarks
- Löschen eines Bookmarks anhand des Namens
- Umändern der URL eines existierenden Bookmarks anhand des Namens
- Rückgabe aller Bookmarks sortiert nach dem Namen
- Rückgabe aller Bookmarks für ein bestimmtes Schlagwort
- Anzeige der n zuletzt eingefügten Bookmarks alphabetisch sortiert nach dem Namen

### 10.10.2 Webservice

- Ausgabe einer Bookmarkliste wahlweise sortiert nach dem Namen, der URL, oder der Creation Time. Die Sortierung soll der URL über einen HTTP-Parameter mitgegeben werden können.
- Auslesen aller Bookmarks die vor einem bestimmten Datum angelegt wurden.
- Detailansicht eines einzigen Bookmarks der über den Namen bestimmt wird.

### **10.10.3 HTML-Rendering der Daten**

Implementiere im Webservice einen Endpoint, der die Seite *input.html* öffnet. Diese Seite beinhaltet Bootstrap Elemente wie Eingabefelder und Buttons.

Diese Eingabeseite soll Eingabefelder für Name, URL und die Schlagwörter haben.

Wenn der Benutzer die Daten eines Bookmarks eingetragen hat, soll man über einen Submit-Button einen Post-Request an die Webapplikation absetzen und diesen in der H2-Datenbank eintragen.

Im Internet findet sich diverse Links die Unterstützung bieten:

[https://www.w3schools.com/tags/att\\_button\\_formmethod.asp](https://www.w3schools.com/tags/att_button_formmethod.asp)

<https://getbootstrap.com/docs/4.0/components/buttons>

...

# 11 SQL-Datenbanken verwenden mit Spring JPA

JPA ist die Abkürzung für *Java Persistence API* und ist derzeit die modernste Art Datenbanken in Java Spring anzubinden.

JPA ist eine Alternative für die im vorigen Abschnitt beschriebene Anbindung über JDBC. JPA ermöglicht Datenbankzugriffe ohne SQL-Statements, kann diese aber optional verwenden.

JPA verwendet in Spring drei Elemente:

- Persistence Entity: Ein Persistence Entity ist eine Klassen, die einer Tabelle oder View der Datenbank entsprechen. Die Instanzvariablen entsprechen den Felder der Tabelle.
- Objektrelationale Metadaten: Die Relationen zwischen einzelnen Tabellen werden über Annotations realisiert.
- JPQL (Java Persistence Query Language): Das ist eine Abfragesprache, die SQL ähnelt. Sie bezieht sich aber nicht auf die Datenbankobjekte (Tabellen, Views, ...), sondern auf die Entitäten (Persistence Entities).

Das hat den enormen Vorteil, dass die verwendete Datenbank einfach ausgetauscht werden kann, ohne SQL-Statements umschreiben zu müssen<sup>11</sup>.

JPA ist ein ORM<sup>12</sup> Framework in Spring. Es entspricht eher einem "Interface", dass durch andere Technologien implementiert wird.

Die populärste und bekannteste "Implementierung" dieses "Interfaces" ist Hibernate <https://hibernate.org/>. Hibernate ist äußerst populär und war schon vor Spring weitverbreitet.

JPA definiert Annotations und Interfaces, die in einer Springapplikation verwendet werden können, damit Hibernate die Entities und Relations richtig verwenden kann.

## 11.1 Entity Manager

Wenn man beim Anlegen eines Spring Projects das Spring Modul JPA als Dependency mit einbindet und das Programm startet, sieht man, dass der Application Context eine Instanz einer sogenannten EntityManagerFactory erstellt.

```
...
2019-11-22 22:12:14.783 INFO 8516 --- [ main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
```

Listing 30: Log Output beim Programmstart eines Spring Programms mit JPA

<sup>11</sup>Unterschiedliche Datenbanken haben oft nicht dieselbe SQL-Syntax, sondern haben unterschiedliche SQL-Dialekte.

<sup>12</sup>ORM = Object Relational Mapping [https://de.wikipedia.org/wiki/Objektrelationale\\_Abbildung](https://de.wikipedia.org/wiki/Objektrelationale_Abbildung)

## Entity Manager

Eine EntityManagerFactory kann ein Bean der Klasse EntityManager erstellen, wenn diese Klasse auto-injected wird. Ein Entity Manager sorgt zur Laufzeit dafür, dass Datenbankobjekte (Tabellen, Spalten, ...) auf Klassen und Instanzvariablen gemappet werden, ohne dass dafür SQL-Statements geschrieben werden müssen. Die SQL-Statements werden vom EntityManager selbst generiert.

### Was ist der Vorteil von JPA und Entitäten?

In komplexen Anwendung gibt es leicht sehr viele Datenbanktabellen werden die SQL-Statements leicht sehr aufwendig. Bei JPA müssen die SQL-Statements nicht erstellt werden. In JPA definiert man Entitäten fest und die SQL-Statements werden von Spring automatisch erstellt und ausgeführt.

#### 11.1.1 Nützliche Settings in application.properties

Bei der Entwicklung gibt es nützliche Settings, die man im File application.properties eintragen kann.

spring.jpa.show-sql=true zeigt im Log alle SQL-Statements an, die Java Spring durch JPA automatisch erzeugt und gegen die Datenbank absetzt. Das ist gerade am Beginn eine große Hilfe.

application.properties:

```
spring.jpa.show-sql=true
```

## 11.2 Codebeispiel: Bookmark Verwaltung mit JPA und H2

Unser Projekt verwendet die Spring Module Lombok, Spring Web, JPA, Thymeleaf, H2 und Actuator.

Die Struktur des fertigen Projekts ist in Abbildung 36 ersichtlich. Das Anlegen aller Files wird in den nächsten Abschnitten beschrieben.



Abbildung 36: Projektstruktur des IntelliJ-Projekts

Das Projekt hat alle drei Layer: Presentation Layer (package `presentation`), Business Layer (package `service`) und Persistence Layer (package `persistence`). Die Entitäten werden im Package `domain` angelegt.

Als Resources gibt es ein statisches File `style.css`, das HTML-File für die korrekte Darstellung von Fehler `error.html`, sowie unsere Bookmarkseite `list.html`, die auch das Anlegen von neuen Bookmarks mittels eines Post-Request erlaubt.

## 11.3 Eine Bookmark Klasse im Domain-Layer

In Anlehnung an die Übung in Abschnitt 10.10 erstellen wir eine Datenbanktabelle für Bookmarks.

Dafür definiert man in JPA eine Klasse, die von Spring als Bean angelegt werden soll und für die eine Tabelle in unserer H2-Datenbank erstellt werden soll.

Dafür verwendet man die Annotation `@Entity`. Wir definieren eine Klasse zur Speicherung eines Bookmarks durch den Code in Listing 31.

Für jede Klasse die die Annotation `@Table` hat, wird in der Datenbank eine Tabelle mit dem in Klammern angegebenen Namen angelegt.

Die Ableitung von der abstrakten Klasse `AbstractPersistable` ist für jede Entität sinnvoll (`@Entity`). Durch diese Ableitung wird automatisch eine Spalte für den Primary Key `ID` angelegt. Der Datentyp wird in den Spitzklammern definiert. `Long` bedeutet also, dass der erstellte Primary Key `ID` den Datentyp `Long` hat.

Wie man sieht, gibt es eine ganze Menge von Annotations, die scheinbar auch mehrfach vorkommen. Dass `name` nicht null sein darf, ist eigentlich dreifach vorhanden.

Die Annotation `@Id` legt den Primary Key der Tabelle fest. Wenn mehrere Columns den Primary Key bilden, muss die Annotation `@EmbeddedId` eingesetzt werden. <https://www.baeldung.com/jpa-composite-primary-keys> beschreibt, wie man dabei vorgeht.

`@GeneratedValue` kann verwendet werden, um in der Datenbank den Wert automatisch generieren zu lassen. Es gibt verschiedene Strategien dazu. <https://www.objectdb.com/java/jpa/entity/generated> und <https://thoughts-on-java.org/jpa-generate-primary-keys/> erläutert diese.

Die Annotation `@Column` kommt von JPA und definiert die Tabelle in der Datenbank näher. `nullable = false` wird also beim Anlegen der Datenbank verwendet.

`@Size` und `@NotNull` sind Annotations die eine Validierung durchführen. D. h. Spring selbst stellt sicher, dass beim Einfügen eines Namens, die definierte kleinste und größte Länge erfüllt sind.

`@NonNull` kommt von Lombok und definiert, dass der Konstruktor der alle erforderlichen Instanzvariablen setzt, einen Inputparameter für `name` hat.

Näheres zu diesen Annotation findet man hier

<https://www.baeldung.com/jpa-size-length-column-differences>.

@Version legt in der Datenbanktabelle eine zusätzliche Spalte an, die wichtig wird, wenn mehrere Threads zugleich auf die Tabelle insertieren wollen. Siehe auch <https://www.byteslounge.com/tutorials/jpa-entity-versioning-version-and-optimistic-locking>.

```
@Entity
@Table(name = "bookmark")
public class Bookmark extends AbstractPersistable<Long> {

    private static final int LEN_MAX_NAME = 30;
    private static final int LEN_MIN_NAME = 2;
    private static final int LEN_MAX_TAGS = 50;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // @Column is for the database creation
    @Column(name = "name", length = LEN_MAX_NAME, nullable = false)
    @Size(min = LEN_MIN_NAME, max = LEN_MAX_NAME) // for validation
    @NotNull // for validation
    @NonNull // for lombok for @RequiredArgsConstructor
    private String name;

    @Column(name = "url")
    private String address;

    @Column(name = "tags", length = LEN_MAX_TAGS, nullable = false)
    private String tags;

    @Column(name = "creationtime")
    private LocalDateTime creationTime;

    @Version // used internally in Spring for logging
    @NotNull
    private Integer version;
}
```

Listing 31: Entität zur Speicherung eines Bookmarks

Ein Start der Applikation und ein Einloggen auf die H2-Console zeigt, dass Spring eine Tabelle für diese Entität erstellt hat.

Auf der Konsole sieht man das SQL-Statement, das beim Starten ausgeführt wurde um die Tabelle zu erzeugen.

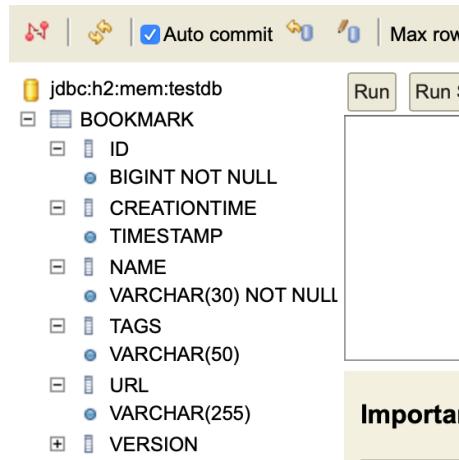


Abbildung 37: Die Datenbanktabelle Bookmark, die für die Entity-Klasse beim Start der Applikation erzeugt wurde. Für die Erstellung werden die Datentypen der Annotation @Column berücksichtigt.

#### *Konsolenausgabe nach dem Start:*

```
Hibernate: drop table bookmark if exists
Hibernate: drop sequence if exists hibernate_sequence
Hibernate: create sequence hibernate_sequence start with 1 increment by 1
Hibernate: create table bookmark (id bigint not null, creationtime timestamp, name varchar(30) not null, tags varchar(50), url varchar(255), version integer not null, primary key (id))
```

## 11.4 Ein Bookmark-Repository im Persistence-Layer

Für den Zugriff auf die H2-Datenbank wird nun statt einem JDBC-Template wie im letzten Abschnitt, eine JPA-Repository Klasse verwendet.

Das Repository soll neben dem Anlegen von neuen Bookmarks, diverse Queries erlauben:

- Rückgabe aller Bookmarks
- Rückgabe eines Bookmarks mit einem bestimmten Namen
- Rückgabe aller Bookmarks, die ein bestimmtes Tag enthalten
- Rückgabe alle Bookmarks, die während eines bestimmten Zeitintervalls angelegt wurden.

### JPA Repository Methoden

Spring JPA implementiert alle Methoden automatisch, d. h. beim Kompilieren wird der Programmcode jeder Methode aufgrund des Namens erzeugt. JPA verwendet dabei die Methodennamen, die alle Auswahlkriterien der Queries enthalten müssen. Je nach der Methode, müssen entsprechende Inputparameter übergeben werden. IntelliJ unterstützt die Eingabe der Methodennamen äußerst effizient, wie man in Abbildung 38 sieht.

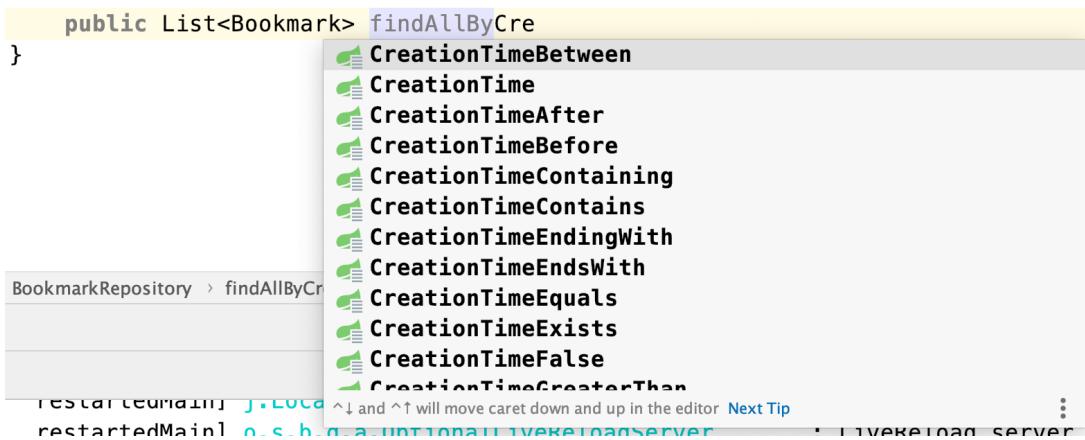


Abbildung 38: Eingabehilfe von IntelliJ beim Eingeben eines Methodennamens

Die Methoden selbst müssen gar nicht implementiert werden! Das macht Java Spring für uns. Wie das Listing 32 zeigt, können beliebige Kombinationen erzeugt werden.

Eine von vielen Erklärungen findet man hier [https://www.amitph.com/spring-data-jpa-query-methods/#6\\_Query\\_Methods\\_to\\_SQL\\_Queries](https://www.amitph.com/spring-data-jpa-query-methods/#6_Query_Methods_to_SQL_Queries).

```
@Repository
public interface BookmarkRepository extends JpaRepository<Bookmark, Long>
{
    public List<Bookmark> findAll();

    public Bookmark findByName( String name );

    public List<Bookmark> findAllByTagsContains( String tag );

    public List<Bookmark> findAllByCreationTimeBetween( LocalDateTime from,
                                                       LocalDateTime to);

    public List<Bookmark> findTop3ByCreationTimeBefore( LocalDateTime beforeDate);

    public List<Bookmark> findBookmarksByUrlMatchesRegexAndNameIsStartingWithAndCreationTimeAfter(
        String urlPattern, String namePrefix, LocalDateTime afterDate
    );
}
```

Listing 32: JPA-Repository Klasse für den Zugriff auf die Bookmarktabelle

Außerdem unterstützt die Klasse `JpaRepository<Bookmark, Long>` alle anderen CRUD-Operationen ohne und mit Auswahlkriterien.

Das Anlegen von neuen Bookmarks, wird beispielsweise durch die Methode `save` erledigt, wie wir noch sehen werden.

## 11.5 Die Klasse Bookmark Service der Businesslogik

In einer richtigen Webanwendung, wird ein Repository über die Business Logik des Service Layers und den Domain Layer angesprochen, niemals über den Persistence Layer.

Der Application Context injected eine Instanz des Bookmark Repositories in die Service Klasse.

In unserem Fall enthält die Business Logik ausschließlich die Zugriffe auf das eine Bookmark Repository. In realen Systemen enthält die Business Logik oft die Verknüpfung von zahlreichen Repositories und die Einbindung von Subsystemen.

```
@Service
public class BookmarkService {

    @Autowired
    private BookmarkRepository repository;

    public void create( Bookmark bookmark ) {
        repository.save(bookmark);
    }

    public List<Bookmark> getAllBookmarks() {
        return repository.findAll();
    }

    public Bookmark getBookmarkByName( String name ) {
        return repository.findByName(name );
    }

    public List<Bookmark> getAllBookmarksContainingText(String text) {
        return repository.findAllByTagsContains(text);
    }
}
```

Listing 33: Service Klasse des Business Layers zum Zugriff auf die Repository Klasse

Das Anlegen eines neuen Bookmarks erfolgt durch Aufruf der Methode `save` der Repository Klasse.

Diese Service Klasse wird im Persistenz Layer verwendet, der im nächsten Abschnitt implementiert wird.

## 11.6 Die Controller-Klasse im Presentation Layer

Wir implementieren exemplarisch zwei Http-Requests, einen für das Auslesen aller Bookmarks und einen für das Anlegen eines neuen Bookmarks.

Die Darstellung der Bookmarks ist mit Thymeleaf im File list.html implementiert. Ebenso wird dort der Post-Request für das Anlegen eines neuen Bookmarks realisiert.

Dieser Code ist ähnlich zu den Controllern der vorigen Kapiteln.

Neu ist die Annotation `@RequestMapping` oberhalb der Klasse, die eine gemeinsame Basis URL für alle Controller-Methoden festlegt. Im Http-Get-Request wird ein Model-Attribute gesetzt, in dem alle Bookmarks an Thymeleaf zum Rendern im HTML-File transportiert werden.

Die Liste der Bookmarks wird über das Bookmark-Service geholt. Der Http-Post-Request verwendet die vorhin implementierte `create`-Methode des Bookmark-Services.

```
@Controller
@RequestMapping( "/bookmarks" )
public class BookmarkController {

    @Autowired
    private BookmarkService service;

    private final Logger log = LoggerFactory.getLogger(this.getClass());

    @GetMapping("get")
    public ModelAndView getAllBookmarks()
    {
        ModelAndView modelAndView = new ModelAndView("list");
        modelAndView.addObject( "allbookmarks" , service.getAllBookmarks() );
        return modelAndView;
    }

    @PostMapping("create")
    public String addBookmark( Bookmark bookmark )
    {
        bookmark.setCreationTime( LocalDateTime.now() );
        service.create( bookmark );
        // let's show the get page after the successful post
        return "redirect:/bookmarks/get";
    }
}
```

Listing 34: JPA-Repository Klasse für den Zugriff auf die Bookmarktabelle

Der Rückgabewert `"redirect:/bookmarks/get"` hat zur Folge, dass nach dem erfolgreichen Posten eines Bookmarks, unsere Web Applikation wieder den Get-Request ausführt und alle Bücher anzeigt.

## 11.7 Das Rendern der Bookmarks durch Thymeleaf

Wir implementieren exemplarisch zwei Http-Requests, einen für das Auslesen aller Bookmarks und einen für das Anlegen eines neuen Bookmarks.

Die Darstellung der Bookmarks ist mit Thymeleaf im File list.html implementiert. Ebenso befinden sich dort die Eingabefelder und der Submit-Button für den Post-Request .

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Spring-Boot-MVC-Thymeleaf-Webanwendung</title>
    <link rel="stylesheet" th:href="@{/style.css}" />
</head>
<body onload='document.f.text.focus();'>
    <hr/>
    <h2>Bookmarksverwaltungsprogramm mit Spring, JPA und Thymeleaf</h2>
    <hr/>
    <h3>Erstelle ein neues Bookmark</h3>
    <form method="POST" name='f' action="/bookmarks/create">
        <input th:if="${_csrf}" th:name="${_csrf.parameterName}"
               th:value="${_csrf.token}" type="hidden"/>
        <label for="name">Name:</label>
        <input name="name" size="30" type="name"/>
        <br>
        <label for="url">Adresse:</label>
        <input name="url" size="100" type="url"/>
        <br>
        <label for="tags">Schlagwörter:</label>
        <input name="tags" size="50" type="tags"/>
        <br>
        <input class="btn btn-primary" type="submit" value="Submit" />
    </form>
    <hr/>
    <h3>Gespeicherte Bookmarks</h3>
    <div th:if="#lists.isEmpty( allbookmarks )">
        <p>Keine Datenelemente vorhanden.</p>
    </div>
    <div th:unless="#lists.isEmpty( allbookmarks )">
        <table>
            <tr>
                <th>Name</th>
                <th>Adresse</th>
                <th>Schlagwörter</th>
                <th>Erstellungszeitpunkt</th>
            </tr>
```

```

<tr th:each="bm : ${allbookmarks}">
    <td th:text="${bm.name}">Name</td>
    <td th:text="${bm.url}">Adresse</td>
    <td th:text="${bm.tags}">Schlagwörter</td>
    <td th:text="${bm.creationTime}">Erstellungszeitpunkt</td>
</tr>
</table>
</div>
<hr/>
</body>
</html>

```

Listing 35: Das list.html File mit dem Post-Request und der Verarbeitung aller Bookmarks

## 11.8 Das File error.html zur Ausgabe von Fehlern

Im Fehlerfall wird die Webseite /error aufgerufen. Um Fehlermeldungen lesbar darstellen zu können, legen wir folgendes File an.

```

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Hoppala!</title>
    <link rel="stylesheet" th:href="@{/style.css}"></link>
</head>

<html>
    <h1>Hoppala!</h1>
    <p>Tja, das ging wohl daneben mit dem Request <span th:text="${path}" />
    </p>
    <p th:text="${'Fehler: ' + status + ', ' +
        error + ', ' + exception}" />
    <p th:text="${'Meldung: ' + message}" />
    <p th:text="${'Zeitpunkt: ' +
        #dates.format( timestamp, 'yyyy-MM-dd HH:mm:ss' )}" />
</html>

```

Listing 36: Das error.html File mit den Fehlermeldungen lesbar dargestellt werden können

## 11.9 Das File style.css zur Verschönerung der Ausgabe

CSS-Files müssen in den Folder `static` erzeugt werden, wie man bei der eingangs erwähnten Folderstruktur sieht.

```
body { font-family: arial,helvetica,sans-serif; }
table, th, td { border: 1px solid black;
                border-collapse: collapse;
                padding: 5px;
                text-align: left; }
th { background-color: #eeeeee }
```

Listing 37: Das error.html File mit dem Fehlermeldungen lesbar dargestellt werden können

## 11.10 Einstellungen für eine persistente H2-Datenbank

Um zu verhindern, dass die H2-Datenbank nach jedem Start wieder frisch befüllt werden muss, kann man die JDBC-URL entsprechend anpassen und einige Settings im File `application.properties` setzen.

Wir ändern die Einstellungen so, dass die Datenbank im Pfad `database/h2-db.mv.db` gespeichert wird und nach dem Beenden nicht gelöscht wird.

`application.properties:`

```
...
spring.datasource.url = jdbc:h2:./database/h2-db;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.username = sa
spring.datasource.password =
spring.jpa.hibernate.ddl-auto = update
```

## 11.11 Verwendung der Webapplication

Mit `http://localhost:8080/bookmarks/get` kommt man zur Übersichtsseite aller Bookmarks. Auf dieser kann man Bookmarks einfügen und über dem Submit-Button in die Datenbank eingeben (Abbildung 39).

## 11.12 Ausgabe einer Fehlerseite

Gibt man im Browser den ungültigen Endpunkt `http://localhost:8080/bookmarks/slog` ein, verarbeitet unser `error.html` File den Fehler zu folgender Ausgabe (Abbildung 40).

## 11.13 Zusammenfassung

Wie das gesamte Codebeispiel zeigt, ist JPA eine extrem effiziente Art und Weise Repositories zu implementieren.

Zur weiteren Vertiefung ist es angeraten, sich insbesondere mit den zahlreichen Methoden der Klasse `JpaRepository<..., ...>` zu beschäftigen und im Internet zu recherchieren.

**Bookmarksverwaltungsprogramm mit Spring, JPA und Thymeleaf**

**Erstelle ein neues Bookmark**

Name:   
 Adresse:   
 Schlagwörter:

**Gespeicherte Bookmarks**

| Name           | Adresse                                  | Schlagwörter            | Erstellungszeitpunkt       |
|----------------|--|-------------------------|----------------------------|
| Baeldung       | https://www.baeldung.com                 | Java, Spring            | 2019-11-23T22:17:58.652119 |
| HTL-Donaustadt | http://www.htl-donaustadt.at             | HTL, Informatik         | 2019-11-23T22:18:26.161143 |
| ORF            | http://www.orf.at                        | doku, tv                | 2019-11-23T22:33:37.845498 |
| Instagram CCC  | https://www.instagram.com/codingcontest/ | Coding Contest          | 2019-11-23T22:34:24.826056 |
| Kurier         | http://www.kurier.at                     | sport, fußball, politik | 2019-11-23T22:44:50.170429 |
| Die Presse     | http://www.diepresse.at                  | Politik                 | 2019-11-23T22:48:44.626168 |

Abbildung 39: Übersicht über alle Bookmarks und Eingabe eines weiteren Bookmarks

## 12 Aspect Oriented Programming (AOP)

Eine gute Einführung in das was ein Aspekt ist und wozu es gut ist, findet sich in <https://www.baeldung.com/spring-aop>.

AOP ist der Softwaransatz um gesamtheitliche Aspekte einer Anwendung zu implementieren. Man bezeichnet diese auch als Cross-Cutting Concerns.

Das sind Aspekt einer Anwendung die für alle Layer gelten.

Beispiele für solche Aspekte sind:

- Security
- Performance

Ein Aspect in Spring wird durch eine Klasse repräsentiert, die mit der Annotation `@Aspect` markiert ist. Weiters kann man Methoden definieren, die in Methodenaufrufe von Pakete "eingehängt" werden.

Das ist mit den Annotations `@Before...` möglich. Die Syntax ist etwas gewöhnungsbedürftig, aber sehr mächtig. Der Ausdruck in der Klasse heißt **PointCut**.

Im unteren Beispiel wird vor jedem Methodenaufruf des angegebenen Packages die Methode `before()` aufgerufen.

# Hoppala!

Tja, das ging wohl daneben mit dem Request /bookmarks/slog

Fehler: 404, Not Found, null

Meldung: No message available

Zeitpunkt: 2019-11-24 01:49:06

Abbildung 40: Fehlerausgabe bei Eingabe einer ungültigen URL

```
@Aspect
@Configuration
public class BeforeAspect {
    private Logger logger = LoggerFactory.getLogger(this.getClass());

    // What is intercepted?
    // execution(* PACKAGE.*.*(..))
    @Before("execution (* at.htldonaustadt.slog.demo.*.*(..))")
    public void before(JoinPoint joinPoint) {
        logger.info("Check for user access");
        logger.info(" Intercepted Method Call: {}", joinPoint);
    }
}
```

Listing 38: Programmierung einer Methode die sich in die Methodencalls eines Packages einhängt.

Eine Änderung bewirkt, dass alle Subpackages mitgetrackt werden.

```
@Before("execution (* at.htldonaustadt.slog.demo..*.*(..))")
```

Die Methoden der Aspect-Klasse werden also ausgeführt, ohne dass die eigentlichen Methoden der Business Logic das wissen!

Ein Ausdruck der das zum Ausdruck bringt, ist Weaving bzw. der Code dazu, ist Weaver.

## 12.1 Performance Messungen

Es gibt eine ganz Reihe an Annotations mit denen Methoden in den Aufruf anderer Methoden "eingewebt" werden können. Ein der nützlichsten Annotations ist @Around. Damit kann beispielsweise die Performance gemessen werden.

```
@Around(value="execution (* at.htldonaustadt.slog.demo.*.*(..))")
public void around(ProceedingJoinPoint joinPoint) throws Throwable {
```

```

long startTime = System.currentTimeMillis();
joinPoint.proceed();
long timeTaken = System.currentTimeMillis() - startTime;
logger.info("Time taken by {} is {}", joinPoint, timeTaken);
}

```

Listing 39: Codebeispiel für eine Performance Messung mit AOP

## 12.2 Weitere AOP-Annotations

Weitere sind @After, @AfterReturning und @AfterThrowing.

```

@After(value="execution (* at.htldonaustadt.slog.demo.*.*(..))")
public void after(JoinPoint joinPoint) {
    logger.info("after execution of {}", joinPoint);
}

@AfterReturning(value="execution (* at.htldonaustadt.slog.demo.*.*(..))",
               returning = "result")
public void afterReturning(JoinPoint joinPoint, Object result) {
    logger.info("{} returned with value {}", joinPoint, result);
}

@AfterThrowing(value="execution (* at.htldonaustadt.slog.demo.*.*(..))",
               throwing = "exception")
public void afterThrowing(JoinPoint joinPoint, Exception exception) {
    logger.info("{} thrown exception value {}", joinPoint, exception);
}

```

Listing 40: Codebeispiel für die Performance Messung mit AOP

## 13 Actuator

Die Einbindung des Activators in ein Java Spring Projekt aktiviert in einer Webapplikation zusätzliche Endpoints, die über ein Webapplikation Status- und Laufzeitdaten zur Verfügung stellen.

Siehe [https://o7planning.org/de/11757/überwachungsanwendung-mit-spring-boot-actuator](https://o7planning.org/de/11757/uberwachungsanwendung-mit-spring-boot-actuator) für Details.

Dafür muss im File application.properties eine Zeile hinzugefügt werden.

application.properties:

```
management.endpoints.web.exposure.include=*
```

Interessante Endpoints sind z. B.

- actuator (Abbildung 41)
- actuator/metrics
- actuator/env (Abbildung 42)

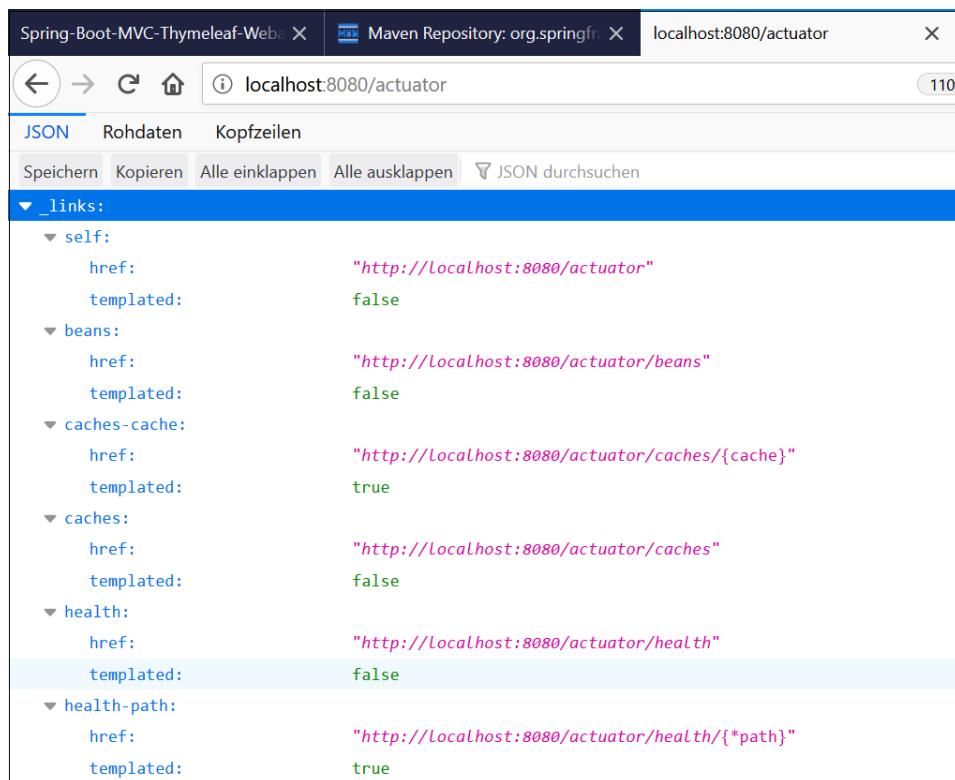


Abbildung 41: Anzeige aller durch das Modul Actuator hinzugefügten Endpoints im Browser

```

{
  "activeProfiles": [],
  "propertySources": [
    {
      "name": "server.ports",
      "value": "8080"
    },
    {
      "name": "...",
      "value": "..."
    },
    {
      "name": "...",
      "value": "..."
    },
    {
      "name": "systemEnvironment",
      "value": "TAFELSPITZ"
    }
  ],
  "properties": {
    "configsetroot": {
      "value": "C:\\WINDOWS\\ConfigSetRoot",
      "origin": "System Environment Property \\\"configsetroot\\\""
    },
    "USERDOMAIN_ROAMINGPROFILE": {
      "value": "TAFELSPITZ",
      "origin": "System Environment Property \\\"USERDOMAIN_ROAMINGPROFILE\\\""
    },
    "PROCESSOR_LEVEL": {
      "value": "6",
      "origin": "System Environment Property \\\"PROCESSOR_LEVEL\\\""
    },
    "VS140COMTOOLS": {
      "value": "C:\\Program Files (x86)\\Microsoft Visual Studio 14.0\\Comm",
      "origin": "System Environment Property \\\"VS140COMTOOLS\\\""
    }
  }
}

```

Abbildung 42: Anzeige von Environmentsinformationen der Webapplikation

## 14 Devtools

Die Einbindung des Spring Moduls Devtools in ein Java Spring Projekt ermöglicht einige nützliche Optionen.

Siehe <https://docs.spring.io/spring-boot/docs/current/reference/html/using-spring-boot.html#using-boot-devtools> für eine Liste aller Optionen.

Durch das Einbinden des Moduls startet die Applikation immer automatisch, sobald im Code etwas geändert wurde. Möchte man dies auf alle Folder im Projekt erweitern, ist folgender Eintrag in den Settings erforderlich.

```
application.properties:
spring.devtools.restart.additional-paths=.
```

# Listings

|    |  |    |
|----|--|----|
| 1  | Ein REST-Controller der grüßen kann . . . . .  | 22 |
| 2  | HTTP-Parameter in der Controller-Klasse . . . . .  | 25 |
| 3  | Eine Klasse mit einigen Buchdaten . . . . .  | 28 |
| 4  | Ein Controller für Buchdaten . . . . .   | 29 |
| 5  | Bootstrap- und Thymeleaf-Dependencies in pom.xml . . . . .   | 30 |
| 6  | Das HTML-File mit Thymeleaf-Anweisungen . . . . .  | 31 |
| 7  | Eine Klasse mit einigen Buchdaten . . . . .  | 33 |
| 8  | Der Rest-Controller mit allen vier Requests . . . . .  | 34 |
| 9  | HTTP-Parameter in der Controller-Klasse . . . . .  | 37 |
| 10 | Instantieren des Application Contexts . . . . .  | 38 |
| 11 | Ausgabe aller beim Start erzeugten Spring Beans . . . . .  | 39 |
| 12 | Zweimaliger Aufruf von <code>getBean</code> im Main . . . . .  | 40 |
| 13 | Die Klassen <code>Book</code> , <code>ComparerByID</code> und <code>ComparerByTitle</code> . . . . .                           | 44 |
| 14 | <code>domain/Sorter.java</code> . . . . .  | 45 |
| 15 | <code>DiApplication.java</code> mit dem Main . . . . .   | 46 |
| 16 | Annotations vor den Klassen <code>Sorter</code> , <code>ComparerByID</code> und <code>ComparerByTitle</code> . . . . .         | 47 |
| 17 | Annotations vor den Klassen <code>Book</code> , <code>ComparerByID</code> und <code>ComparerByTitle</code> . . . . .           | 48 |
| 18 | Verwendung der Annotation <code>@Qualifier</code> zum Auflösen von Mehrdeutigkeiten bei Konstruktoraufrufen . . . . .          | 49 |
| 19 | Constructor Injection mit einer einzelnen, nicht <code>final</code> Instanzvariable . . . . .                                  | 50 |
| 20 | Constructor Injection mit einer einzigen <code>final</code> Instanzvariable . . . . .  | 50 |
| 21 | Die Klasse <code>Sorter</code> mit einem Setter statt einem Konstruktor . . . . .  | 51 |
| 22 | Autowiring einer Instanzvariable . . . . .   | 51 |
| 23 | Zwei Dependencies im Project Model File einer Spring Applikation . . . . .   | 52 |
| 24 | Inhalt von <code>schema.sql</code> . . . . .   | 58 |
| 25 | Inhalt von <code>data.sql</code> . . . . .   | 60 |
| 26 | Die Repository Klasse für die Personen Tabelle der Datenbanks . . . . .  | 62 |
| 27 | Definition eines RowMappers zum Umwandeln von Database Records auf Instanzen . . . . .   | 63 |
| 28 | Verwendung des selbstdefinierten RowMappers in einer Query-Methode . . . . .   | 63 |
| 29 | Testen der CRUD-Operationen der Repository-Klasse in der Methode <code>run</code> des <code>CommandLineRunner</code> . . . . . | 64 |
| 30 | Log Output beim Programmstart eines Spring Programms mit JPA . . . . .   | 68 |
| 31 | Entität zur Speicherung eines Bookmarks . . . . .  | 71 |
| 32 | JPA-Repository Klasse für den Zugriff auf die Bookmarktabelle . . . . .  | 73 |
| 33 | Service Klasse des Business Layers zum Zugriff auf die Repository Klasse . . . . .   | 74 |
| 34 | JPA-Repository Klasse für den Zugriff auf die Bookmarktabelle . . . . .  | 75 |
| 35 | Das <code>list.html</code> File mit dem Post-Request und der Verarbeitung aller Bookmarks . . . . .                            | 76 |
| 36 | Das <code>error.html</code> File mit den Fehlermeldungen lesbar dargestellt werden können . . . . .                            | 77 |
| 37 | Das <code>error.html</code> File mit den Fehlermeldungen lesbar dargestellt werden können . . . . .                            | 78 |
| 38 | Programmierung einer Methode die sich in die Methodencalls eines Packages einhängt . . . . .                                   | 80 |
| 39 | Codebeispiel für eine Performance Messung mit AOP . . . . .  | 80 |
| 40 | Codebeispiel für die Performance Messung mit AOP . . . . .   | 81 |

# Abbildungsverzeichnis

|    |  |    |
|----|--|----|
| 1  | Jürgen Höller Presentation . . . . .   | 7  |
| 2  | Michael Nitschinger Presentation . . . . .   | 7  |
| 3  | Screenshot eines Applets, Quelle: <a href="https://en.wikipedia.org/wiki/Java_applet">https://en.wikipedia.org/wiki/Java_applet</a> . . . . .  | 8  |
| 4  | Browser Response beim Erkennen eines Applet Tags . . . . .   | 8  |
| 5  | Beschreibung der Java Enterprise Edition, Quelle: <a href="https://de.wikipedia.org/wiki/Enterprise_Java">https://de.wikipedia.org/wiki/Enterprise_Java</a> (Stand: 2019-04-08) . . . . .          | 9  |
| 6  | Begriffserklärung Enterprise Java Beans, Quelle: <a href="https://de.wikipedia.org/wiki/Enterprise_JavaBeans">https://de.wikipedia.org/wiki/Enterprise_JavaBeans</a> (Stand: 2019-04-08) . . . . . | 10 |
| 7  | Prinzip einer Webapplikation Quelle: <a href="https://de.wikipedia.org/wiki/Webanwendung">https://de.wikipedia.org/wiki/Webanwendung</a> (Stand 2019-04-08) . . . . .                              | 12 |
| 8  | Architektur einer Enterprise Application . . . . .   | 15 |
| 9  | Anlegen eines Spring Projekts in IntelliJ . . . . .  | 18 |
| 10 | Festlegung der Metadaten eines Spring Projekts in IntelliJ . . . . .   | 19 |
| 11 | Auswahl der Spring Module in IntelliJ . . . . .  | 19 |
| 12 | Angabe des Projektpfads in IntelliJ . . . . .  | 20 |
| 13 | Auto-Import für Maven in IntelliJ aktivieren . . . . .   | 20 |
| 14 | Start einer Web Application in IntelliJ . . . . .  | 21 |
| 15 | Anlegen einer Controller-Klasse in einem neuen Package . . . . .   | 21 |
| 16 | Projektstruktur . . . . .  | 22 |
| 17 | Aufruf der beiden Methoden des allerersten REST-Controllers . . . . .  | 23 |
| 18 | Übergabe des Namens und des Alters an die Controller-Methode über HTTP-Parameter . . . . .   | 26 |
| 19 | Die Projektstruktur der Webanwendung mit einem HTML-File list.html . . . . .   | 30 |
| 20 | HTML Ausgabe von <a href="http://localhost:8080/books">http://localhost:8080/books</a> . . . . .   | 32 |
| 21 | Auswahl der Art des Http-Requests und Eingabe der URL . . . . .  | 35 |
| 22 | Daten im Request-Header . . . . .  | 35 |
| 23 | Bücherdaten im Request-Body . . . . .  | 36 |
| 24 | Juhu!!! It did work. . . . .   | 36 |
| 25 | Spring Beans und "normale" Instanzen in einer Spring Applikation . . . . .   | 38 |
| 26 | Packagestruktur des Demobeispiels für Dependency Injection . . . . .   | 43 |
| 27 | External Libraries in einem Maven Projekt . . . . .  | 53 |
| 28 | Die Maven Dependency der Library org.json auf <a href="https://mvnrepository.com">https://mvnrepository.com</a> . . . . .  | 54 |
| 29 | Aktualisieren des lokalen Maven Repositories im Maven-Fenster von IntelliJ . . . . .   | 54 |
| 30 | Login Screen in die H2-Testdatabase . . . . .  | 57 |
| 31 | Database Client der Test-H2-Datenbank im Browser . . . . .   | 58 |
| 32 | Der SQL-Skript schema.sql im Verzeichnisbaum. . . . .  | 58 |
| 33 | Befüllen der Tabelle PERSON im Browser . . . . .   | 59 |
| 34 | Anzeige aller Datenbank Records der Tabelle PERSON . . . . .   | 59 |
| 35 | Die SQL-Skripts data.sql und schema.sql im Verzeichnisbaum . . . . .   | 60 |
| 36 | Projektstruktur des IntelliJ-Projekts . . . . .  | 69 |
| 37 | Die Datenbanktabelle Bookmark, die für die Entity-Klasse beim Start der Applikation erzeugt wurde. Für die Erstellung werden die Datentypen der Annotation @Column berücksichtigt. . . . .         | 72 |
| 38 | Eingabehilfe von IntelliJ beim Eingeben eines Methodennamens . . . . .   | 73 |
| 39 | Übersicht über alle Bookmarks und Eingabe eines weiteren Bookmarks . . . . .   | 79 |

|    |   |    |
|----|---|----|
| 40 | Fehlerausgabe bei Eingabe einer ungültigen URL . . . . .                          | 80 |
| 41 | Anzeige aller durch das Modul Actuator hinzugefügten Endpoints im Browser . . . . | 82 |
| 42 | Anzeige von Environmentsinformationen der Webapplikation . . . . .                | 83 |

## Noch fehlende Teile im Skriptum

- Ein Beispiel für `@ComponentScan`
- Das Einloggen in eine Webapplikation: Spring Security.
- Anbindung einer bestehenden Datenbank, z. B. Postgres
- Behandlung von Enums in JPA
- Embeddables
- Beispiel mit Datenbank Relationen
- ...