

Java und Spring

Patrick Schulz
Universität Osnabrück

In dieser Ausarbeitung befassen wir uns mit einer Programmiersprache, in der wir uns schon seit Informatik-A-Zeiten auskennen. Die Rede ist von Java. Allerdings werden wir dieses Mal keine Java-typischen Standalone-Programme schreiben, sondern Web-Applikationen, die das Potenzial haben unsere Welt digital zu verknüpfen.

Inhaltsverzeichnis

| | |
|---------------------------|-----------|
| 1. Main..... | 2 |
| 2. Motivation..... | 3 |
| 2.1 Webserver..... | 3 |
| 2.2 Appserver..... | 4 |
| 2.3 TwoParties..... | 5 |
| 3. Spring..... | 7 |
| 3.1 Framework..... | 7 |
| 3.2 Ingredence..... | 8 |
| 3.3 FileSystem..... | 9 |
| 3.4 SpringBoot..... | 11 |
| 4. HalloWelt..... | 13 |
| 4.1 SpringInit..... | 13 |
| 4.2 Annotation..... | 14 |
| 4.3 Application..... | 15 |
| 4.4 Cronjobs..... | 16 |
| 4.5 WebController..... | 18 |
| 4.6 Templates..... | 19 |
| 4.7 RESTController..... | 20 |
| 4.8 Repository..... | 21 |
| 5. Advanced..... | 25 |
| 5.1 votequiz..... | 25 |
| 5.2 JDBCTemplate..... | 25 |
| 5.3 WebSocket..... | 26 |
| 5.4 CrossOrigin..... | 28 |
| 6. Fazit..... | 29 |

1. Main

2. Motivation

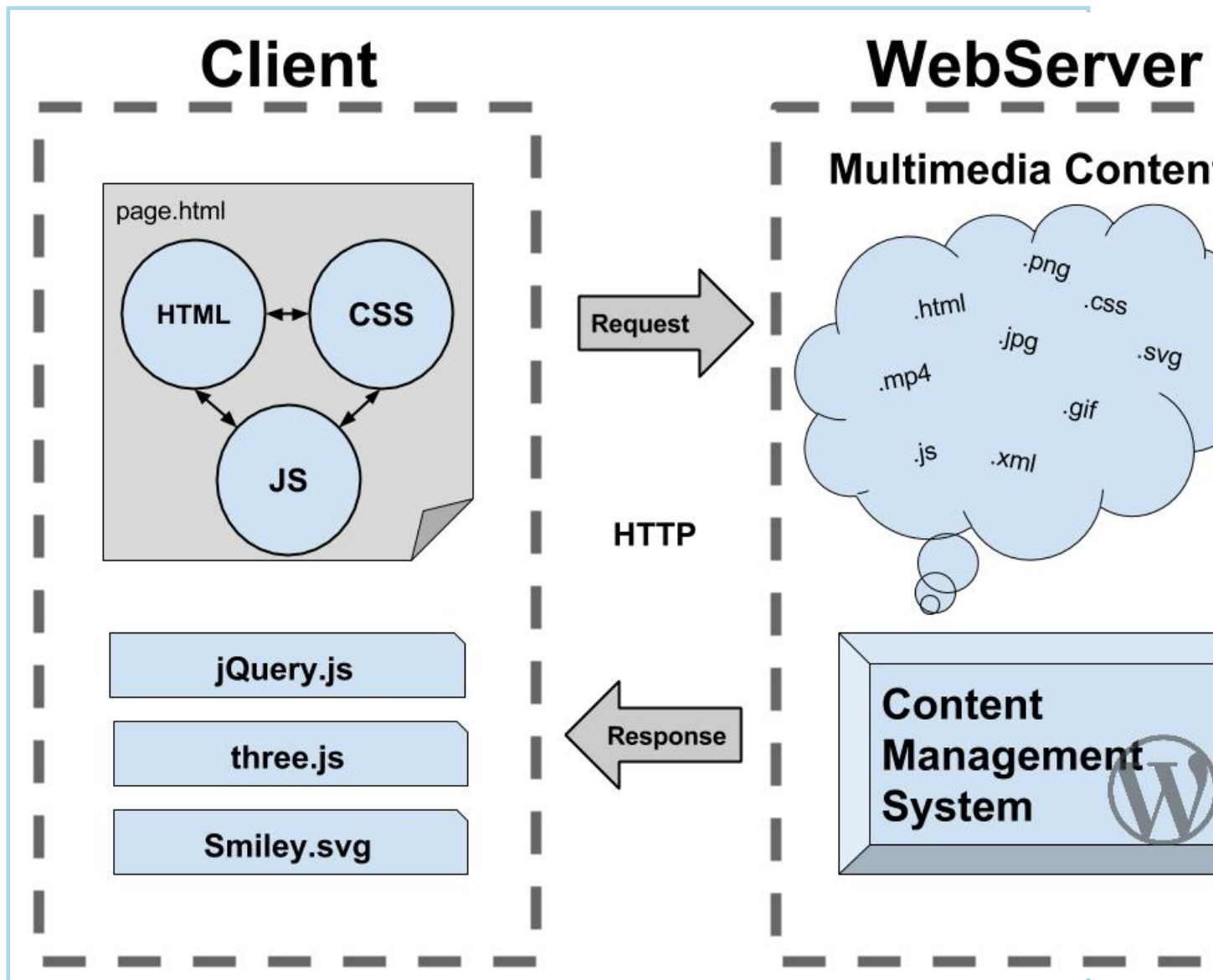
Im Abschnitt "Motivation" wird, bevor wir uns mit dem eigentlichen Thema beschäftigen, erstmal weit ausgeholt: Wir leiten unseren Fokus vom WebServer zum AppServer um und betrachten kurz eine Alternative zu Spring.

2.1 Webserver

Vom WebServer ...

In den bisherigen Kapiteln des Web Publishing Seminars haben wir uns mit Webtechnologien befasst, die auf einem WebServer deployed werden. Der Client kann mithilfe eines Browsers einen oder mehrere HTTP-Requests an den WebServer senden. Anhand der mitgelieferten URL weiß der WebServer dann welchen Content er dem Client als HTTP-Response zurücksenden soll. Meistens handelt es sich initial bei dem Content um HTML-Seiten, die in sich selbst Referenzen zu weiterem Content beinhalten - wie zB. Bilder oder Skripte - und durch den Browser asynchron vom WebServer nachgeladen werden. Durch die Kombination aus dem HTML-Grundgerüst, den CSS-Stylesheets und JavaScript-Bibliotheken sind wir in der Lage unseren Multimedialen-Content auf dynamischen und schönen Webseiten präsentieren zu können.

In der Regel reicht uns das auch aus...



2.2 Appserver

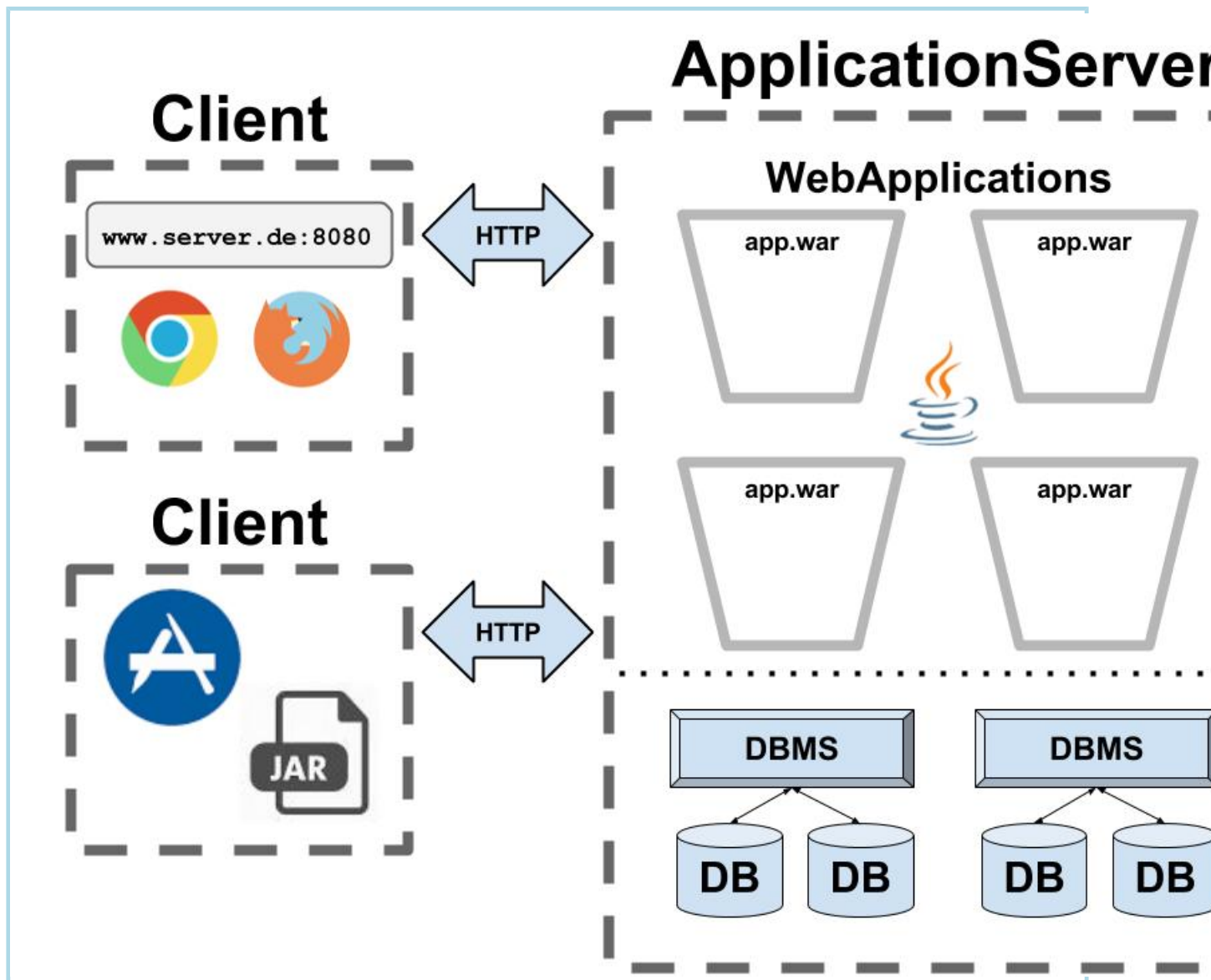
... zum AppServer

Nun gibt es aber auch Situationen, wo die client-seitige Daten-Verarbeitung auf dem Endgerät nicht mehr zumutbar sein kann. Gute und prägnante Beispiele hierfür sind Aufgaben, die einen höheren Rechenaufwand erfordern oder der Umfang der Daten, die für eine Aufgabe benötigt werden, so hoch sind, als das es absurd wäre diese Daten für die Verarbeitung zu übertragen. Ein konkreteres Beispiel ist die Berechnung einer kürzesten Strecke zwischen zwei Standorten auf einer WebKarte: Es lohnt sich nicht das aktuell benötigte Straßennetz - in Form einer 200 Megabyte XML-Datei - auf den Client zu übertragen. Gerade in unserer aktuellen Zeit der leistungs-eingeschränkten Smartphones sind solche Aufgaben dazu prädestiniert *server-seitig* gelöst zu werden und man nur die Visualisierung der Ergebnisse dem Client-Endgerät überlässt.

Wir wollen neben den WebServern heute eine andere Art von Servern betrachten - nämlich die *ApplicationServer* (fort an in der Ausarbeitung als "*AppServer*" bezeichnet). Mit einem AppServer ist der Fokus auf die serverseitige Bearbeitung von Aufgaben gesetzt. Für die Realisierung von solchen App-Servern sind in der Java-Welt die *Tomcat* und *Jetty* Projekte und zu erwähnen. Beide Projekte bieten innerhalb der *Java Runtime Environment* (JRE) einen *WebContainer* an, worin mehrere WebApplikationen *deployed* werden können. Jede einzelne WebApplikation für sich besitzt dann seine eigene WebSchnittstelle, um Anfragen vom Client empfangen zu können. Dabei ist es unerheblich, ob die Anfragen vom Browser oder von einer Desktop-Applikation stammen. Für die Persistenz von Daten existiert in den meisten Fällen noch ein Datenbanksystem.

Um den Umfang der Ausarbeitung nicht zu sprengen, machen wir für's Erste die Annahme, dass diese Rahmenbedingungen vom *WebContainer* und dem Datenbanksystem bereits zur Verfügung gestellt sind und werden auch nicht weiter auf die Einrichtung dieser Komponenten eingehen. Es gibt genügend Tutorials¹ dazu im Netz, wenn das Interesse hierfür besteht. Unser Fokus liegt heute ganz auf den WebApplikationen selbst. Jedoch bevor man mit der Programmierung solcher *WebApps* beginnen kann, hat jeder Einsteiger eine bestimmte Entscheidung zu treffen ...

¹ <https://www.google.de/webhp?q=Tomcat+einrichten>



2.3 TwoParties

Zwei Frameworks

Wenn wir uns nun mit Java Web-Applikationen auseinandersetzen wollen, dann gilt es am Anfang eine wichtige Entscheidung zu treffen. Denn es existieren für die Programmiersprache Java zwei Frameworks, die einem für die Umsetzung zur Verfügung stehen. Zwei Frameworks, die im Prinzip das gleiche können, aber es über die Jahre geschafft haben die Java-Community zu spalten. Die Rede ist von der **Java Enterprise Edition** und **Spring**.

Die Java Enterprise Edition wird von Oracle gepflegt und repräsentiert somit die native Basis, woran viele Entwickler, die Wert auf Minimalismus legen, ihre Argumentation sehen mit der von Java hauseigenen Erweiterung zu arbeiten. Das Spring Framework entstand zu einem Zeitpunkt, als die ersten Release-Iterationen der Java Enterprise Edition veröffentlicht wurden und die Entwicklung nicht die versprochene Produktivität erzielte. Bis heute findet man noch ein provozierendes Zitat auf der Homepage von Spring, welches seit Anbeginn der Entwicklung als Kampfansage verstanden werden kann: "*Let's build a better enterprise*"².

² <http://spring.io/>

Mittlerweile hat die Java EE über die Jahre nachgelegt und somit kann behauptet werden, dass beide Frameworks gleichermaßen produktiv sind. Den entscheidenden Vorteil jedoch bezieht das Spring Framework in seiner Einsteigerfreundlichkeit. Pivotal - die Firma, die hinter der Entwicklung von Spring steckt - legt einen sehr großen Wert darauf, dass die Lernkurve nicht zu steil ist. Es ist ein legitimes Geständnis, dass nicht jeder Softwareentwickler ein IT-Ninja ist, der sich in alles reinhacken kann. Dies ist auch der Grund, wieso neue Start-Up Unternehmen und andere Einsteiger bei der großen Entscheidung eher zu Spring tendieren.

In dieser Ausarbeitung werden wir uns auch deswegen nur mit der WebApp-Entwicklung mit Spring befassen und behalten im Hinterkopf, dass es eine Alternative gibt, die für eine gesunde Wettkampf-Atmosphäre sorgt und beide Frameworks in ihrer Entwicklung vorantreibt - im Kampf um die User...



3. Spring

In diesem Abschnitt versuchen wir ohne ein Stückchen Quellcode zu beschreiben, wozu Spring gut ist und was man alles am Anfang dazu wissen sollte, um durchzustarten: Vom Framework über die Vorbereitungen, dem typischen Projektverzeichnis und eine Erklärung zu SpringBoot.

3.1 Framework

Spring in a Nutshell

Spring ist ein Java-Framework, entwickelt von der Ideenschmiede Pivotal. Das Framework ist in mehrere Aspekte aufgeteilt, die alle dazu dienen auf effiziente Weise WebApps zu erstellen. Es folgt nun eine kurze, prägnante Aufzählung von Aspekten, die von Interesse sein können:

Spring Web (Logo: links)

Der für unser Seminar wohl wichtigste Aspekt ist Spring Web. Dieses Framework definiert die Controller für die Web-Schnittstellen unserer Web-Applikation. In anderen Worten wird jegliche Kommunikation zum Client über diesen Aspekt verwaltet.

Spring Data (Logo: unten links)

Spring Data beinhaltet alle Interfaces, um die Kommunikation zu Datenbanken und anderen Frameworks zu erstellen, die für die Abspeicherung von jeglichen Daten verantwortlich sind.

Spring Security (Logo: unten)

Jegliche Form von Zugriffsbeschränkung - ob einfache User/Passwort oder SSH-Schlüssel-Authentifizierung - lässt sich mithilfe des Spring Security Aspektes auf die gewünschten Pfade anwenden.

Spring Social (Logo: oben links)

Spring Social bietet High-Level-Interfaces für den Zugriff auf Twitter und Facebook-Userdaten. Voraussetzung hierfür ist natürlich eine vorangehende Registrierung der App bei der entsprechenden Webseite, um den Zugriff mit einem App-Schlüssel zu legitimieren.

Spring Boot (Logo: oben)

Spring Boot ist wahrscheinlich das Hauptargument, wieso Spring im Vergleich zu Java EE eine grundlegende Abkürzung in der WebApp-Entwicklung bietet und damit ein Totschlag-Argument liefert gegen Java EE mit Hinsicht auf die Einsteigerfreundlichkeit. Was nun genau dahinter steckt, wird im noch folgenden Kapitel gezeigt.

Spring Roo (Logo: oben rechts)

Ist weniger ein Aspekt, als mehr ein Konsolen-basierter Code-Generator um die Erstellung von Basis-AppElementen nochmals zu beschleunigen. In dieser Ausarbeitung werden wir uns nicht mit Spring Roo befassen. Es lohnt sich aber dennoch Roo zu erwähnen, vor allem wegen³ dessen Potenzial massig Zeit einsparen zu können bei der Entwicklung von neuen WebApps. Verweis³

Spring Core (Logo: mitte)

Der kleinste gemeinsame Nenner des Spring Frameworks. Die Elemente und Klassen aus dem Core werden von allen anderen Aspekten benötigt und eingesetzt. Der Entwickler selbst arbeitet mit den Klassen vom Spring Core eher indirekt. Die Core ist zum Beispiel dafür verantwortlich, dass die *Dependency Injection* umgesetzt wird.

Unter **Dependency Injection** versteht man das Prinzip, dass der Softwareentwickler für bestimmte Programmiererelemente nur Interfaces schreiben muss und die dazugehörige Implementation dann dem Framework überlassen wird. Das Spring Framework ist durch die "Convention over Configuration" Namensgebung der definierten Interface-Methoden in der Lage selbst zu interpretieren, was die vorliegende Methode machen soll. In den Codebeispielen zu den Repositories werden wir diesen *Zauber* dann bewusst ausnutzen.

³ <http://docs.spring.io/spring-roo/docs/2.0.0.M1/reference/html/#beginning>

Zuvor noch fassen wir zusammen, welche Komponenten benötigt werden, damit wir mit der WebApp-Entwicklung starten können.



3.2 Ingredence

Vorbereitung

Was brauchen wir alles, um mit der Entwicklung von WebApplikationen mit Spring zu starten:

Java Developer Kit

Das Java Developer Kit ist das Fundament für die jegliche Entwicklung von Java-Programmen. Auf Ubuntu lässt sich die aktuelle JDK Version 8 über ein Paket-Repository sehr angenehm herunterladen und einrichten:

```
sudo apt-add-repository ppa:webupd8team/java
sudo apt-get install oracle-java8-installer
```


Das Installations-Setup dann noch durchwinken und euer System ist bereit, Java zu kompilieren und ausführen.

Apache Maven

Maven ist ein mächtiges Projekt-Management und Build-Tool. Mit Hilfe von Maven werden unsere WebApps kompiliert, zusammengebaut und in ein WebArchiv (.war) verpackt:

```
sudo apt-get install maven
```

Für unsere Code-Beispiele erfordert es keine explizite Einarbeitung in Maven selbst. Unsere Interaktionen mit diesem Tool werden sich auf zwei Kommandozeilen beschränken. Es steht jedem frei auch ein anderes Build-Tool zu verwenden. Die Entwickler von Spring schlagen als passende Alternative zum Beispiel Gradle⁴ vor. Für die Beispielprojekte in dieser Ausarbeitung und auch in der Mehrheit aller anderen Tutorials im Web wird man an Maven nicht herumkommen.

Eine beliebige IDE deines Vertrauens

Als Entwicklungsumgebung stellt Pivotal eine Eclipse-Erweiterung für die explizite Entwicklung von Spring-Apps. Diese Erweiterung hört auf den Namen Spring Tool Suite⁵. Der generelle Vorteil dieser Erweiterung besteht in der Auto-Vervollständigung und in den speziellen Project-Wizards, sowie weiterem "Schnickschnack". Wer aber schon seine eigene Lieblings-Programmierungsumgebung besitzt, der braucht sich auch von ihr nicht trennen, denn das Bauen und Testen der WebApp funktioniert am Besten immer noch über die Konsole. Eine persönliche Empfehlung für eine Eyecandy-IDE ist sublimeText3⁶.

Das war's!

Das Spring Framework selbst muss von uns nicht explizit heruntergeladen werden. Die benötigten Aspekte werden dynamisch für jedes Projekt von unserem Build-Tool geladen. Konkret auf Maven bezogen existiert für jede uns erdenkliche Bibliothek ein Eintrag im Maven-Repository⁷. Von dort bezieht Maven dann alle Ressourcen, die als Abhängigkeiten im Projekt angegeben sind.

Schauen wir uns nun ein typisches Projektverzeichnis an von Spring...

3.3 FileSystem

Das Projektverzeichnis

Der Aufbau einer WebApplikation in Spring besitzt ein gleichbleibendes, übersichtliches Schema:

Im obersten Level des Projektverzeichnisses befindet sich die *pom.xml* - eine Builddatei, womit wir unser Projekt kompilieren. Sie beinhaltet Projekt-Metadaten, wie Projektname, Versionsnummer und Bibliotheksabhängigkeiten. Das kompilierte Projekt lässt sich dann im *target*-Ordner finden. Zum Bauen des Projektes mit Maven braucht man nur die folgende Zeile im Projektverzeichnis in der Konsole eingeben:

```
mvn package
```

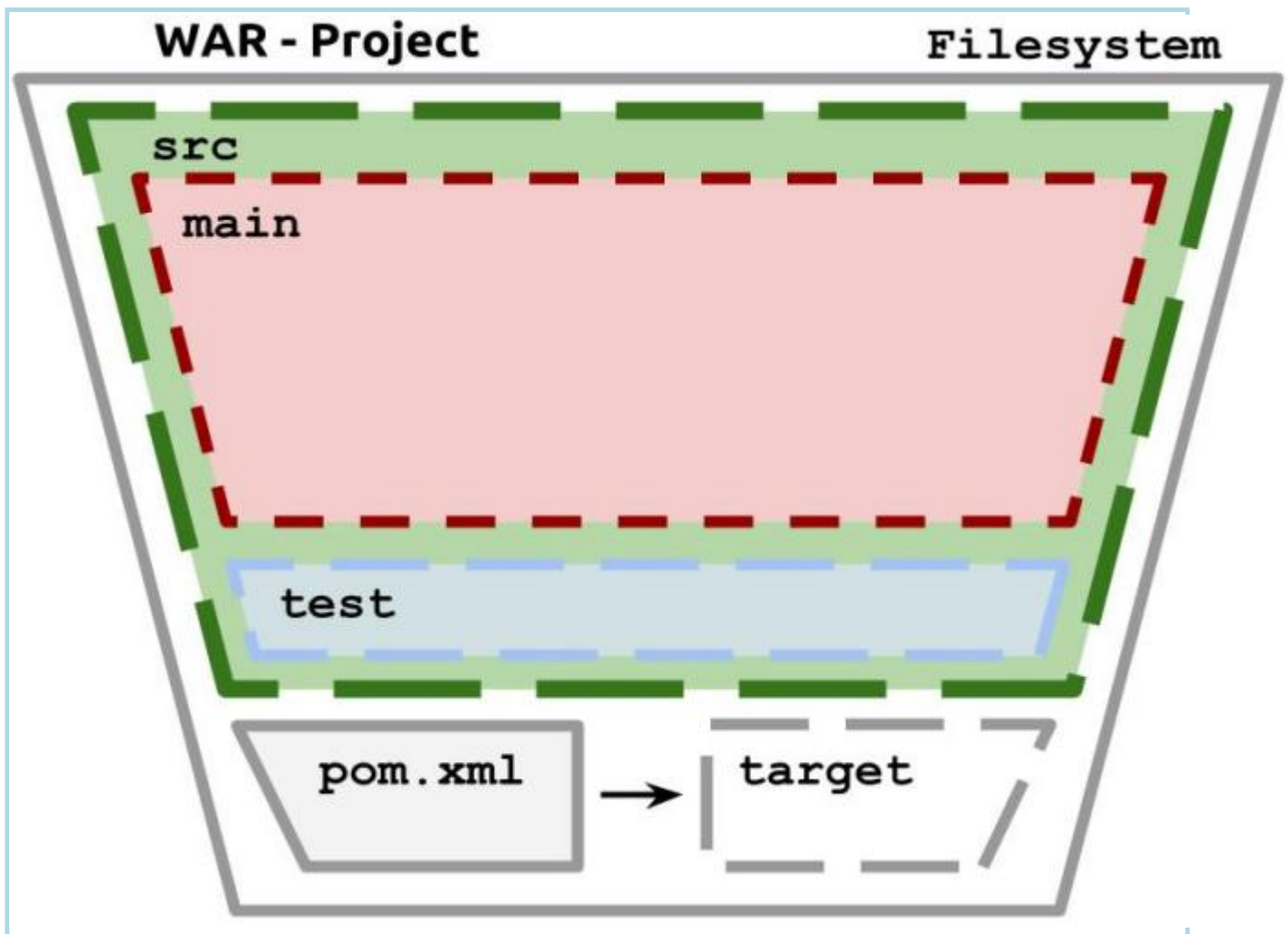
Im *src*-Ordner befindet sich alles, was inhaltstechnisch mit der WebApplikation zu tun hat. Der *src-Ordner* innerhalb spaltet sich in einen *main*-Ordner und einen *test*-Ordner auf. Im *test*-Ordner befinden sich die *jUnit*-Klassen, die nach jedem Kompilieren vor jedem Start-Prozess die Funktionalitäten des Projektes überprüfen sollen. Der Entwickler braucht die Tests nicht explizit zu starten. Auf eine weitergehende Einführung in *jUnit*-Tests wird in dieser Ausarbeitung verzichtet.

⁴ <http://gradle.org/>

⁵ <https://spring.io/tools>

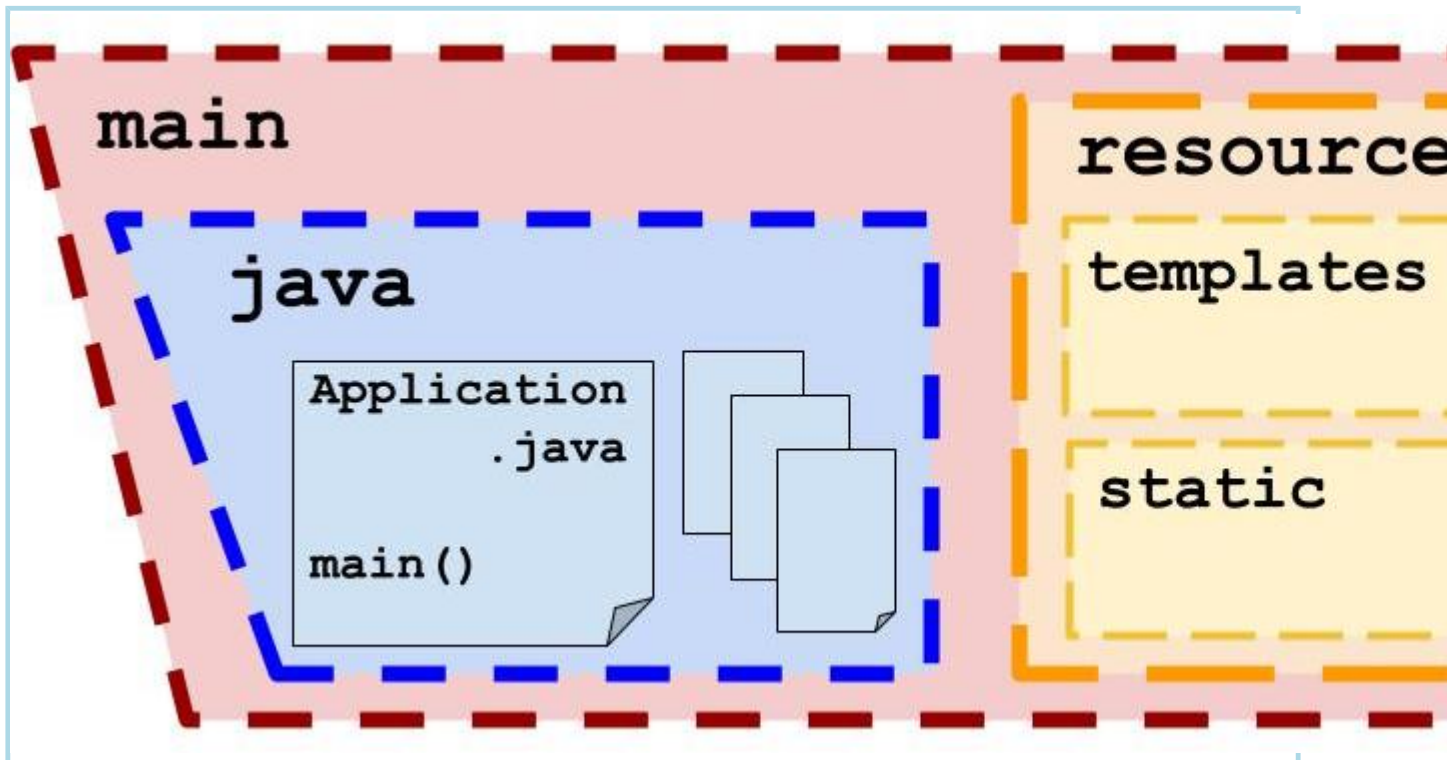
⁶ <http://www.sublimetext.com/>

⁷ <http://mvnrepository.com/>



Eine Ebene tiefer im *main*-Verzeichnis wird der Pfad ein weiteres Mal in einen *java*-Ordner und einem *resources*-Ordner geteilt. Wie die Bezeichnung es bereits andeutet, befinden sich im *java*-Ordner die Java-Klassendateien, in denen die Logik und Funktionalität der Applikationen umgesetzt sind. Die *java*-bekannte *main-Methode* befindet sich aus Konvention in der *Application.java*, die für die generelle Initialisierung der WebApp zuständig ist.

Im *resource*-Ordner befinden sich die Views und WebContent der Applikation. Die Unterteilung in einen *static*-Ordner und einen *template*-Ordner, hält die Ressource unter dem folgenden Aspekt auseinander, dass die HTML-Seiten im *template*-Ordner serverseitig vorverarbeitet werden, bevor sie dem Client zugeschickt werden. Ein Beispiel hierfür folgt im Kapitel . Alle anderen Ressourcen, die von der Applikation nicht direkt beeinflusst werden, landen im *static*-Ordner. Den *static*-Ordner kann man als "WebServer im AppServer" wahrnehmen, der jeglichen URL-Path, der nicht explizit von WebControllern besetzt ist, auf den Content im *src*-Ordner mapped. Wenn also im "*src/main/src*"-Pfad eine *index.html* vorhanden ist, dann wird diese auch als Standardseite der Web-Application festgelegt.



3.4 SpringBoot

Spring Boot

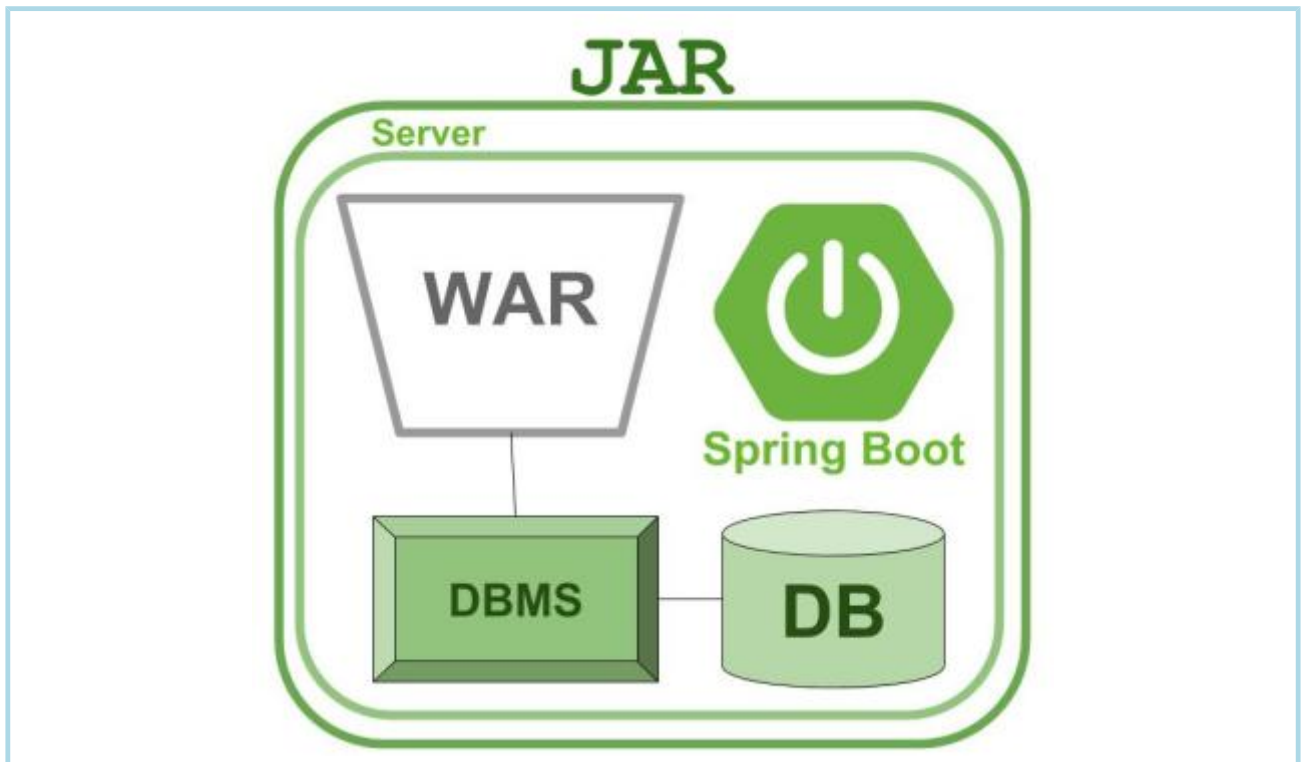
Angenommen wir hätten eine WebApp geschrieben und wären bereit für die ersten Tests, dann würden wir dafür einen funktionierenden AppServer benötigen, wo wir unsere WebApp dann deployen ... *eigentlich*. An dieser Stelle kommt nämlich der Zauber von Spring Boot ins Spiel und ist auch eines der Hauptgründe, wieso das Entwickeln einer WebApplication mit Spring als einfach bezeichnet werden kann. Spring Boot sorgt dafür, dass basierend auf unseren Projektabhängigkeiten alle Module bereitgestellt werden (Darunter zählt zB. eine eigene interne Datenbank). Der benötigte WebContainer wird simuliert. Oben drauf wird dann das ganze Projekt mit all seinen benötigten Komponenten in eine Über-JavaArchiv (*.jar*) verpackt.

Die Vorteile sind rigoros: Zum Testen einer Applikation auf dem Arbeitsrechner wird nur die folgende Kommandozeile für Maven benötigt:

```
mvn spring-boot:run
```

Nach wenigen Sekunden ist die WebApp dann zum Testen über "*localhost:8080*" erreichbar. Die Vorteile der WebApp als JAR besteht darin, dass diese auf **jedem Rechner** gestartet werden kann, wo auch die Java Runtime Environment läuft. Jeder Computer wird automatisch zu einem Server mit dem Aufruf einer einzigen JAR-Datei.

Deswegen: "*Make jar, not war.*"



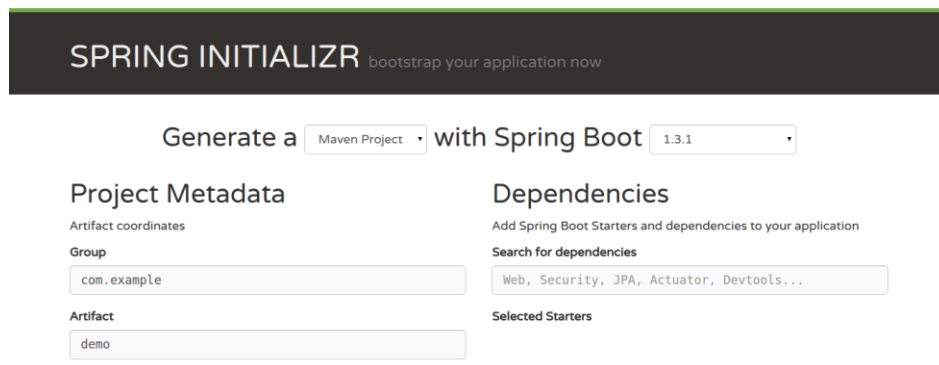
4. HalloWelt

In diesem Abschnitt geht es nun endlich um Sourcecode. Wir beziehen unseren Projektrahmen mit dem *Spring-Initializr* und starten mit einem "Hello World!"-Projekt. Anschließend erweitern wir unsere WebApp schrittweise um weitere Elemente. Bei den Elementen handelt es sich um Cronjobs, WebController, HTML-Templates, speziell nochmal der RestController und eine Einführung in die Repositories.

4.1 SpringInit

Spring Initializr

Damit wir nicht komplett bei Null anfangen und den Projektrahmen selber zusammenbasteln müssen, bietet die Spring-Homepage einen initialen Projekt-Generator, der unter dem Namen *Spring Initializr* zu finden ist. Öffnet den Link zur App indem ihr auf den Screenshot unterhalb des Absatzes klickt:



Sobald ihr das Formular auf dem Browser seht, wie auf dem Screenshot oben, dann können wir nun Schritt für Schritt durchgehen, was die Formularelemente bedeuten:

- **Generate a Maven Project with Spring Boot 1.3.1**
Die ersten beiden Einträge sind ziemlich selbsterklärend: Wir wollen ein Spring Project mit dem Build-Tool Maven und der aktuellsten, stabilen Version 1.3.1 von Spring Boot drin haben.
- **Group = webpub Artifact = HalloWelt**
Das *package*, worin sich unsere Java-Klassen befinden, soll "webpub" heißen und der Name des Projekts heißt traditionell "HalloWelt".
- **Dependencies = Web, Thymeleaf, JPA, H2**
Bei der Textbox für die Dependencies handelt es sich um eine auto-vervollständigende Suchbox, wo ihr jede einzelne Abhängigkeit mit der Entertaste zu bestätigen habt. **Web** benötigen wir für die WebController, Thymeleaf für unsere dynamischen HTML-Templates, **JPA** (Java Persistenz API) für die Repositories und **H2** als die zugrundeliegende Datenbank, wo die Entitäten der Repositories abgespeichert werden.

Wer genau wissen möchte, was alles an ein Spring Projekt angeknüpft werden kann, der kann sich das Formular auch in "full version" anschauen. Für den Anfang aber reichen uns die vier obigen Angaben. Der Spring Initializr macht im Grunde genommen nichts anderes, als die *pom.xml* für uns zu schreiben, damit wir uns selbst nicht mit Maven großartig beschäftigen müssen. Alle Angaben können wir im späteren Verlauf in der pom.xml bei Bedarf noch umändern - die Angaben sind für unser Projekt nicht absolut.

Drückt abschließend auf den dicken grünen Button *Generate Project* und entpackt die heruntergeladene Zip-Datei in eurem Arbeitsverzeichnis. Schauen wir uns nun an, wie unser erstes Projekt aussieht...

4.2 Annotation

Die Projekt-Build-Datei

Nachdem wir unser Startprojekt von der Spring Initializr heruntergeladen und entpackt haben, erkennen wir eine vertraute Verzeichnisstruktur, wie sie bereits in einem vorherigen Kapitel beschrieben wurde. Werfen wir zuerst einen Blick in unsere Build-Datei pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>webpub</groupId>
  <artifactId>HalloWelt</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>HalloWelt</name>
  <description>Demo project for Spring Boot</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.1.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
  </properties>

  ...

</project>
```

Innerhalb des *project*-Tags sollten alle Einträge zu sehen sein, die wir auch im "Spring Initializr"-Formular getätigt haben. Es folgt nun eine kurze Erläuterung der restlichen Elemente:

- **modelVersion** beschreibt die genutzte Version vom Maven Build-Tool
- **version** beschreibt eine manuell verwaltete Versionsnummer des eigenen Projektes und wird als Suffix später an den Archivnamen angehängt.
- **packaging** definiert, in welchem Archivformat unsere WebApp später verpackt werden soll. Generell gilt: Eine WebApp im *war*-Format braucht einen WebContainer (zB. Glassfish), dafür können mehrere WebApps in einem solchen WebContainer deployed werden. Eine WebApp im *jar*-Format kann direkt auf jeder Java-Maschine ausgeführt werden, da sie - dank Spring Boot - den WebContainer simuliert, zur Einschränkung, dass gleichzeitig nur eine 'jar' aktiv ausgeführt werden kann.
- **parent** definiert, um was für eine Art Programm es sich bei unserem Projekt handelt und welche Eigenschaften es besitzt. Wir lesen darin, dass unser Projekt ein "spring-boot-starter"-Projekt ist und dadurch auch die entsprechenden Eigenschaften übertragen bekommt.
- innerhalb **properties** sind dann noch die Zeichencodierungen der Dateien, sowie die verwendete Java Version angegeben.

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

```

</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
</dependencies>

```

Im nächsten Block unter **dependencies** finden wir unsere vier gewählten Abhängigkeiten unseres Projektes wieder. Jedes dieser Abhängigkeiten lässt sich auch im Maven-Repository⁸ wiederfinden, von wo sie nach dem ersten Kompilieren auf den Arbeitsrechner heruntergeladen werden.

Schauen wir im nächsten Schritt nun unsere Application.java an ...

4.3 Application

Hallo Welt!

Der erste Blick auf die Application.java inklusiver Main-Methode.

Application.java

```

package webpub;

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

@SpringBootApplication
public class Application {

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

}

```

Die Annotation **@SpringBootApplication** bewirkt den Zauber, der bereits in einem anderen Kapitel kompakt erwähnt wurde. Zusätzlich scannt Spring Boot jede Java-Klasse im Projekt und versucht durch Autokonfiguration zu verstehen, welche Klasse im Projekt für was zuständig ist. Dieser Art von Autokonfiguration kann man auch mit Hilfe weiterer @-Annotationen entgegenkommen, denn alle selbst definierten Klassen erben nur von POJO's⁹ - Klassen, die nur von der obersten Java-Klasse *Object* erben. @-Annotationen erleichtern auch die Verständlichkeit über ihre Aufgabe für andere Entwickler, die sich in den Code einlesen und ihn nachvollziehen müssen.

Die run()-Methode initialisiert anschließend alle Elemente der WebApp.

Damit wir mit unserer App den ersten großen Meilenstein erreichen, fügen wir einen Logger der App-Klasse hinzu, um eine Konsolen-Ausgabe tätigen zu können:

Application.java

⁸ <http://mvnrepository.com/>

⁹ https://de.wikipedia.org/wiki/Plain_Old_Java_Object

```

package webpub;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HalloWeltApplication {

    static final Logger log = LoggerFactory.getLogger(HalloWeltApplication.class);

    public static void main(String[] args) {
        SpringApplication.run(HalloWeltApplication.class, args);
        log.info("Hallo, Welt!");
    }

}

```

Das gewöhnliche `println()` sollte auf jeden Fall vermieden werden, weil sich in der Laufzeitumgebung meist verschiedene Objekte aus verschiedenen Threads über den Logger zu Wort melden. Würden wir normal `println()`, dann würde man die einzelnen Aussagen der Prozesse nur schwer auseinander halten können. Die Loggerfabrik liefert für jede Klasse einen persönlichen Logger, der mit jedem `log()`-Aufruf auch die Signatur mit ausgibt.

Sobald der Logger geschrieben wurde, können wir die App in der Konsole starten mit:

```
mvn spring-boot:run
```

Nach (hoffentlich) wenigen Sekunden erscheint dann nach der Initialisierung aller Komponenten unsere gewünschte Nachricht:

Konsole

```

2016-01-20 20:59:17.198 INFO 19771 --- [main] o.s.w.s.c.a.WebMvcConfigurerAdapter :
Adding welcome page: class path resource [static/index.html]
2016-01-20 20:59:17.254 INFO 19771 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping :
Root mapping to handler of type [class org.springframework.web.servlet.mvc.ParameterizableViewController]
2016-01-20 20:59:17.292 INFO 19771 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping :
Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2016-01-20 20:59:17.292 INFO 19771 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping :
Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2016-01-20 20:59:17.350 INFO 19771 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping :
Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2016-01-20 20:59:18.375 INFO 19771 --- [main] o.s.j.e.a.AnnotationMBeanExporter :
Registering beans for JMX exposure on startup
2016-01-20 20:59:18.538 INFO 19771 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer :
Tomcat started on port(s): 8080 (http)
2016-01-20 20:59:18.549 INFO 19771 --- [main] webpub.Application :
Started Application in 8.59 seconds (JVM running for 17.306)
2016-01-20 20:59:18.553 INFO 19771 --- [main] webpub.Application :
Hallo, Welt!

```

Herzlichen Glückwunsch zur ersten, funktionierenden Spring-App. Bevor wir unsere erste WebSchnittstelle einrichten, gehen wir zuvor noch auf Cronjobs ein.

4.4 Cronjobs

Cronjobs einrichten

Eine Spring-Applikation besitzt zwei Möglichkeiten intern Aufgaben zu starten ohne irgendeinen Außeneinfluss.

- Entweder unmittelbar nach der Initialisierung der App in der main()-Methode
- oder durch Cronjobs.

Wichtig: Damit Cronjobs verwendet werden dürfen, muss in der *Application.java* eine **@EnableScheduling** Notation vor der Klasse angegeben werden.

Cronjobs werden für Aufgaben eingerichtet, die eine gewisse Routine mit sich bringen. Ein Beispiel hierfür wäre die Freischaltung bzw. Sperrung von bestimmten Dienstleistungen, weil das Personal hierfür nur zu gewählten Zeiten erreichbar ist. Eine regelmäßige Erstellung eines Backups von der Datenbank ist ebenfalls ein wichtiger Routinejob. In unserem Beispiel halten wir den auszuführenden Job einfach und bauen uns eine Nervensäge, die alle 5 Sekunden die Uhrzeit in der Konsole ausgibt.

Nagger.java

```
package webpub;

import java.text.SimpleDateFormat;
import java.util.Date;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

@Component
public class Nagger {

    private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");

    private static final Logger log = LoggerFactory.getLogger(Nagger.class);

    @Scheduled(fixedRate = 5000)
    public void currentTime() {
        log.info("Hey World! Its " + dateFormat.format(new Date()) + " now ^^");
    }
}
```

Die **@Component** Notation indiziert die Klasse als eine arbeitende Komponente des Projekts. Die **@Scheduled** Notation vor der Methode beschreibt, in welchem zeitlichen Rhythmus die Methode ausgeführt werden soll. Mehrere Funktionsaufrufe werden parallel ausgeführt, wenn die Zeitabstände verhältnismäßig geringer sind als der Zeitaufwand innerhalb der Methode.

Wenn man zuvor noch das **@EnableScheduling** an die *Application.java* angehängt hat, sollte sich beim Starten des Projektes in der Konsole der Chronjob in der Nagger-Klasse melden:

Konsole

```
2016-01-21 09:02:55.397 WARN 3686 --- [main] o.s.b.a.t.ThymeleafAutoConfiguration :
Cannot find template location: classpath:/templates/ (please add some templates or check your Thymel
eaf configuration)
2016-01-21 09:02:56.056 INFO 3686 --- [main] o.s.j.e.a.AnnotationMBeanExporter :
Registering beans for JMX exposure on startup
2016-01-21 09:02:56.113 INFO 3686 --- [pool-2-thread-1] webpub.Nagger :
Hey World! Its 09:02:56 now ^^
2016-01-21 09:02:56.213 INFO 3686 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer :
Tomcat started on port(s): 8080 (http)
2016-01-21 09:02:56.227 INFO 3686 --- [main] webpub.HelloWeltApplication :
Started HelloWeltApplication in 6.364 seconds (JVM running for 11.755)
2016-01-21 09:02:56.227 INFO 3686 --- [main] webpub.HelloWeltApplication :
Hallo, Welt!
2016-01-21 09:03:01.108 INFO 3686 --- [pool-2-thread-1] webpub.Nagger :
Hey World! Its 09:03:01 now ^^
2016-01-21 09:03:06.108 INFO 3686 --- [pool-2-thread-1] webpub.Nagger :
Hey World! Its 09:03:06 now ^^
2016-01-21 09:03:11.108 INFO 3686 --- [pool-2-thread-1] webpub.Nagger :
Hey World! Its 09:03:11 now ^^
2016-01-21 09:03:16.108 INFO 3686 --- [pool-2-thread-1] webpub.Nagger :
Hey World! Its 09:03:16 now ^^
```

Was einem hier auffallen sollte ist, dass der Cronjob bereits startet - noch bevor die *Application.java* fertig mit der Initialisierung aller Komponenten war. Um ein solches Verhalten vorzubeugen, kann man als Parameter auch einen *initialDelay=10000* mitgeben.

Neben der einfachen Variante zur Rhythmenangabe in Millisekunden kann man auch die für Cronjobs bekannte komplexere und mächtigere Variante verwenden:

```
@Scheduled(cron = "0 15 9-17 * * MON-FRI")
```

Die Cron-Syntax im String beinhaltet 6 Angaben zur gewählten Sekunde, Minute, Stunde, Monatstag, Monat und nochmal extra die Wochentage. Die Methode wird erst dann ausgeführt, wenn alle 6 Zeitkriterien erfüllt sind. In diesem Beispiel wird die zugrundeliegende Methode zu jeder 15. Minute einer vollen Stunden von 9 bis 18 Uhr ausgeführt - und das alles nur von Montag bis Freitag.

Eine genauere Ausführung zu der Cronjob-Syntax finden sie im großen, allmächtigen Wikipedia¹⁰ oder in der Spring Doku¹¹.

Nachdem wir nun die App lange genug mit sich selbst beschäftigt haben, wird es an der Zeit die erste Webschnittstelle zu bauen.

4.5 WebController

Webschnittstelle einrichten

Damit die WebApp Anfragen von Clients empfangen kann, benötigen wir eine Klasse mit der **@Controller** Notation vorne dran:

GreetingController.java

```
package webpub;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.ui.Model;

@Controller
public class GreetingController {

    @RequestMapping("/hallo")
    public String greeting3(@RequestParam(value="name", defaultValue="World") String name, Model model) {
        model.addAttribute("name", name);
        return "hello_temp";
    }

}
```

Jede Schnittstelle nach außen erhält eine **@RequestMapping** Notation, in der angegeben wird, auf welchen Pfad die Funktion gemappt werden soll. In unserem Code wird die WebApp während des Testens über die Adresse **localhost:8080/hallo** erreichbar sein.

Die Methode kann mithilfe der **@RequestParam** Notation *URL-Queries* auswerten und die Werte als Methoden-Parameter verwenden. Neben den Query-Parametern muss in den meisten Fällen noch ein **Model**-Objekt mitgegeben werden. Diesem Model-Objekt sollen dann alle Informationen und Daten mitgegeben werden, die im **HTML-Template** ausgewertet werden sollen. Die obige Methode erwartet zum Beispiel, dass ein Name mitgegeben wird und somit der Client im Response mit seinem Namen begrüßt werden kann. Ansonsten wird die Annahme gemacht, dass der Client *die Welt* ist.

¹⁰ <https://de.wikipedia.org/wiki/Cron>

¹¹ <https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/scheduling/support/CronSequenceGenerator.html>

Der Rückgabewert ist ein String, welcher die Position des HTML-Templates im `/src/main/resources/templates`-Verzeichnis angibt. Der String in unserem Codebeispiel bedeutet, dass das gewünschte Template in `/src/main/resources/templates/hello_temp.html` zu finden ist.

Bevor wir die Webschnittstelle testen können, müssen wir uns noch zuletzt mit dem HTML-Template beschäftigen ...

4.6 Templates

HTML-Templates

Im Standard Web-Controller wird als Rückgabewert in den meisten Fällen ein HTML-Template als Referenz angegeben. In dem folgenden Codebeispiel wird der mitgelieferte `name`-Parameter vom Model ausgegeben. Das verwendete Framework hierfür heißt **Thymeleaf**.

hello_temp.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Hallo Welt mit HTML und Thymeleaf</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    <p th:text="'Hello, ' + ${name} + '!'" />
  </body>
</html>
```

Es fällt auf, dass die Datenverarbeitung bei Thymeleaf in der Attributliste der HTML-Tags eingebettet ist. Diese Herangehensweise macht es Designern leichter, mit dem Code zu arbeiten. Die Alternativen - wie PHP und Java Server Pages - müssen, um Veränderungen anzusetzen, den HTML-Fluss mit eigenen Code-Blöcken unterbrechen. Ein guter Vergleich hierfür ist das Template für die Ausgabe von Listen, die wir nachher bei den Repositories verwenden werden:

guestlist.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Hallo Welt mit HTML und Thymeleaf</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    <p>Liste aller Greetings:</p>
    <ul>
      <li th:each="greet : ${namelist}" th:text="${greet.id} + ': ' +
        ${greet.content}" />
    </ul>
  </body>
</html>
```

Dazu das Equivalent in PHP:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Hallo Welt mit HTML und PHP</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
```

```

<body>
  <p>Liste aller Greetings:</p>
  <ul>
    <?php
      foreach ($namelist as $greet) {
        echo "<li>$greet.id $greet.content</li>";
      }
    <?>
  </ul>
</body>
</html>

```

Wenn die `hello_temp.html` geschrieben ist, kann die WebApp gestartet werden. Über `localhost:8080/hallo`¹² sollte dann die WebApp einen herzlichen Gruß zurückschicken. Alternativ kann das zu erwartende Ergebnis über die URL `http://vm496.rz.uos.de:8080/hallo` einer vorbereiteten WebApp betrachtet werden. Das QueryMapping sollte ebenfalls funktionieren: `localhost:8080/hallo?name=Bernd`¹³ bzw. `http://vm496.rz.uos.de:8080/hallo?name=Bernd`

Neben dem Standard-WebController existiert noch der REST-Controller als eine weitere Variation ...

4.7 RESTController

REST-Controller

WebSchnittstellen in Spring sind nicht nur in der Lage HTML-Seiten zurückzuschicken, sondern auch JSON-Objekte, die aus JavaKlassen abgeleitet werden. Solche Schnittstellen werden mit einem **@RestController** umgesetzt:

GreetingRestController.java

```

package webpub;

import java.util.concurrent.atomic.AtomicLong;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.ui.Model;

@RestController
public class GreetingRestController {

    private static final String template = "Hallo, %s!";
    private final AtomicLong counter = new AtomicLong();

    //simpler String output
    @RequestMapping("/hallo1")
    public String greeting1(@RequestParam(value="name", defaultValue="World") String name) {
        return String.format(template, name);
    }

    //JSON output
    @RequestMapping("/hallo2")
    public Greeting greeting2(@RequestParam(value="name", defaultValue="World") String name) {
        return new Greeting(counter.incrementAndGet(),
            String.format(template, name));
    }
}

```

¹² `http://localhost:8080/hallo`

¹³ `http://localhost:8080/hallo?name=Bernd`

Die `@RequestMapping` Notation sorgt dafür, dass nur *GET*-Anfragen behandelt werden. Diese Einstellung lässt sich mit einem weiteren Parameter sensibilisieren :

```
@RequestMapping(value="/path", method=RequestMethod.POST)
```

Der Rückgabewert der `greeting2()`-Methode ist ein `Greeting`-Objekt, welches ebenfalls definiert werden muss:

Greeting.java

```
package webpub;

public class Greeting {

    private final long id;
    private final String content;

    public Greeting(long id, String content) {
        this.id = id;
        this.content = content;
    }

    public long getId() {
        return id;
    }

    public String getContent() {
        return content;
    }
}
```

Basierend auf dem Aufbau der `Greeting`-Model-Klasse, versucht Spring die Attribute der Klasse in ein JSON-Objekt umzukonvertieren. Der Entwickler muss nicht genau angeben, wie die Konvertierung umgesetzt werden soll. Das Framework, welches Spring hierfür verwendet, heißt Jackson 2¹⁴.

Die Ergebnisse lassen sich dann testen:
einfacher String-Output: Localhost¹⁵ bzw. TestServer¹⁶
JSON-Output von `Greeting`: Localhost¹⁷ bzw. TestServer¹⁸

Es folgt nun als nächstes eine Einführung in Repositories ...

4.8 Repository

Repository

Mit Hilfe von Repositories sind wir in der Lage Java-Objekte zu speichern. Die Umsetzung hierfür ist verblüffend einfach:

- man benötigt nur ein **Repository-Interface**
- und eine **Entität**, die im Repository gespeichert werden soll.

Als Programmbeispiel werden wir nun jeden Gruß, den der Server an die Clients zurückschickt, abspeichern.
#Datenvorratspeicherung

¹⁴ <http://wiki.fasterxml.com/JacksonRelease20>

¹⁵ <http://localhost:8080/hallo1>

¹⁶ <http://vm496.rz.uos.de:8080/hallo1>

¹⁷ <http://localhost:8080/hallo2>

¹⁸ <http://vm496.rz.uos.de:8080/hallo2>

Hierfür beginnen wir mit der Entität und ergänzen unser Greeting-Modell um zwei Annotationen:

Greeting.java

```
package webpub;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;

@Entity
public class Greeting {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;
    private String content;

    //Why JPA, Why!!!
    protected Greeting(){}

    public Greeting(String content) {
        this.content = content;
    }

    public long getId() {
        return id;
    }

    public String getContent() {
        return content;
    }
}
```

Neben der **@Entity** Notation vor der Klasse und der **@Id** Notation vor dem Id-Attribut haben wir zusätzlich die Auto-Inkrementierung des Identifiers vom RestController in das Modell verlagert. Zuvor hätte man das Problem gehabt, dass jeder Controller selbst eine ID hochzählt, welches zu unerlaubten Doppeleinträgen im Repository führen kann.

Steht die Entität, kann das dazugehörige Interface für das Repository geschrieben werden:

GreetingRepository.java

```
package webpub;

import org.springframework.data.repository.Repository;
import java.util.List;

interface GreetingsRepository extends Repository<Greeting, Long> {

    //Folgende Methoden sollen zur Verfügung stehen:
    //Speichen einer Entity
    void save(Greeting entity);
    //Zugriff auf alle abgespeicherten Entities im Repo
    List<Greeting> findAll();
}
```

Das Repository-Interface erwartet als generische Parameter den Namen der Entität und das Format seines Identifiers. Innerhalb des Interfaces werden dann nur die benötigten Methoden deklariert. Aufgrund der Namenskonvention, weiß das Spring Framework was die Methode zu tun hat. Der Entwickler muss die Methoden selbst nicht schreiben. Diesen *Zauber* kennen wir inzwischen unter dem Namen **Dependency Injection**.

Zum Abschluss binden wir das Repository in die Controller ein mit der **@Autowired** Notation vorne dran:

GreetingRestController

```

package webpub;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.beans.factory.annotation.Autowired;

@RestController
public class GreetingRestController {

    private static final String template = "Hallo, %s!";

    @Autowired
    GreetingsRepository repo;

    //JSON output
    @RequestMapping("/hallo2")
    public Greeting greeting2(@RequestParam(value="name", defaultValue="World") String name) {
        Greeting greet = new Greeting( String.format(template, name));
        //Speichern
        repo.save(greet);

        return greet;
    }
}

```

Die **@Autowired** Notation überlässt dem Spring Core Framework die Instantiierung vom Repository. Das Greeting-Objekt in der *greeting2()*-Methode wird nur noch mit dem Textinhalt konstruiert. Außerdem wird vor dem Abschicken des Grußes noch das Objekt im Repository mit *save(greet)* abgespeichert.

Zusätzlich zur Abspeicherung bauen wir noch eine weitere WebSchnittstelle zur Ausgabe aller bisherigen Grüße:

GreetingController

```

package webpub;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.ui.Model;

@Controller
class ListController{

    @Autowired
    GreetingsRepository repo;

    //Liste aller Greetings
    @RequestMapping("/list")
    public String greetingList(Model model) {
        model.addAttribute( "namelist", repo.findAll() );
        return "guestlist";
    }
}

```

Das dazugehörige HTML-Template *guestlist.html* haben wir bereit im Kapitel HTML-Template betrachtet.

Die Ergebnisse lassen sich dann testen:
 Gruß Schnittstelle: localhost¹⁹ bzw. TestServer²⁰

¹⁹ <http://localhost:8080/hallo2>

²⁰ <http://vm496.rz.uos.de:8080/hallo2>

Liste der Grüße: localhost²¹ bzw. TestServer²²

Mit diesem Kapitel haben wir die Basics von Spring Web durchgenommen. Im letzten Abschnitt wird noch eine Beispiel-WebApp vorgestellt, wofür man das Spring Framework explizit verwenden kann.

²¹ <http://localhost:8080/list>

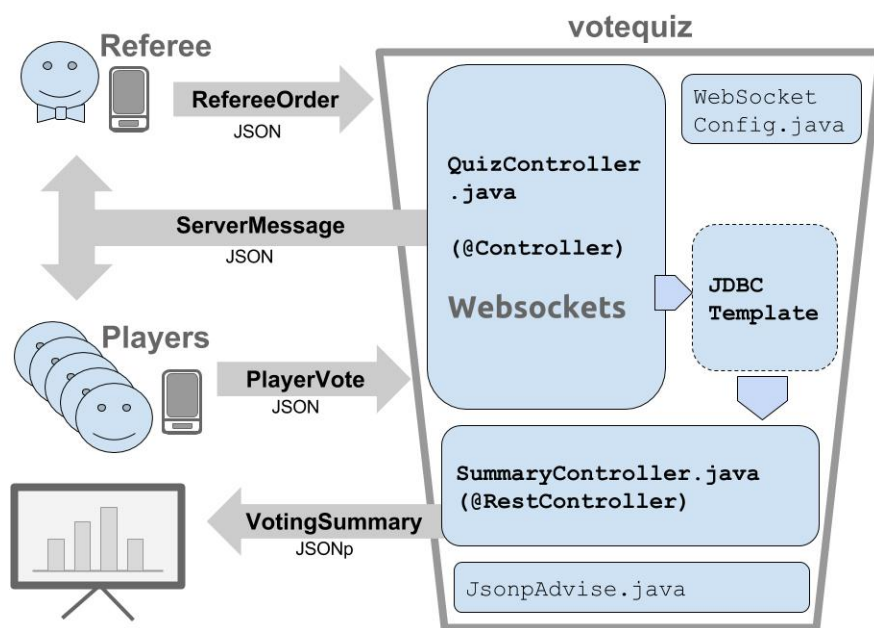
²² <http://vm496.rz.uos.de:8080/list>

5. Advanced

In dem letzten Abschnitt wird ein einfaches und praktisches Anwendungsbeispiel vorgestellt, welches die Stärken des Spring Frameworks nochmal aufzeigen soll. Zu den bestehenden Features, die im vorherigen Abschnitt erläutert wurden, kommen hier noch WebSockets, JDBC-Templates und Rückgabewerte als JSONP hinzu.

5.1 votequiz

Anwendungsbeispiel "VoteQuiz"



Im Anwendungsbeispiel "votequiz" geht es darum ein Quiz zu organisieren. Die Spieler ("player") können per Smartphones ein Voting abgeben, welches in einer Datenbank abgespeichert wird. Das Ergebnis des Votings kann auf Anfrage abgefragt werden. Ein Quizmaster ("referee") regelt den Spielfluss, wenn eine neue Votingrunde beginnt und gibt die Lösung auf den Smartphones der Spieler frei.

Das Projekt kann auf GitHub²³ heruntergeladen werden, um eine Einsicht in den Code zu erhalten.

Zum Abspeichern der Votes sind anstelle von Repositories ein JDBC-Verbindungen verwendet worden.

5.2 JDBCTemplate

JDBCTemplate

Eine nostalgische *Java Database Connection* lässt sich in Java Spring über das JDBCTemplate realisieren:

²³ https://github.com/PattyDePuh/webpub_spring/tree/master/samples/votequiz

```
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.beans.factory.annotation.Autowired;

@Service
class VoteHandler{
    @Autowired
    JdbcTemplate jdbcTemplate;

    public void addVote(String letter){
        String query = "UPDATE votes SET counter = counter + 1 WHERE option = ?";
        Object[] params = { letter };
        int[] types = { Types.VARCHAR };
        int effect = jdbcTemplate.update( query, params, types );
    }
}
```

Wie bei den Repositories, fällt es auf, dass durch die **@Autowired** Notation der Entwickler sich nicht um die Einrichtung der Datenbankverbindung kümmern muss. Die Referenz zum **JdbcTemplate** kann direkt verwendet werden. In dem Codebeispiel erkennt man, wie eine SQL-Query aufgebaut wird und anschließend ausgeführt wird. Solche Konstrukte, wo Queries abhängig von Nutzereingaben sind, sind beliebte Angriffsstellen für SQL-Injections²⁴. Deswegen existiert für das Abschicken von Queries eine erweiterte Methode, um Parameter-Binding²⁵ zu realisieren.

Die API²⁶ des JdbcTemplates ist auf jeden Fall ein Hingucker wert. Da das JdbcTemplate eine eigene Bibliothek ist, muss diese Abhängigkeit auch in der pom.xml ergänzt werden, um sie nutzen zu können:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
</dependency>
```

Weiter gehts mit den Websockets ...

5.3 WebSocket

WebSockets

WebSockets sind TCP-Verbindungen im HTTP-Wrapper. Server und Client sind über WebSockets in der Lage schnell und frequent sich gegenseitig Daten zuzuschicken. Diese Art von Verbindung wird sehr häufig benötigt, um zB. Online-Spiele umzusetzen, die Echtzeit-Reaktionszeiten erfordern.

WebSockets im SpringFramework werden im WebController folgendermaßen aufgebaut:

WebSocketController.java

```
package quiz;

import org.springframework.stereotype.Controller;
import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.messaging.handler.annotation.SendTo;

@Controller
class WebSocketController {
```

²⁴ <https://de.wikipedia.org/wiki/SQL-Injection>

²⁵ <http://use-the-index-luke.com/de/sql/where/bind-variablen>

²⁶ <http://docs.spring.io/spring/docs/current/javadoc-api/org.springframework.jdbc.core/JdbcTemplate.html>

```

    @PostMapping("/player")
    @SendTo("/channel/all")
    public ServerMessage getVote(PlayerVote vote) throws Exception {
        //Hier PlayerVote verarbeiten
        return ServerMessage("Vote erhalten");
    }
}

```

Mit **@PostMapping** definiert man die Empfängerseite des Servers. Mit **@SendTo** wird definiert auf welchem Kanal eine Nachricht zurückgesendet werden soll nach der Verarbeitung des Signaleingangs. Bei dem ClientVote- und dem ServerMessage-Objekt handelt es sich wieder um POJO's, bei denen die Jackson 2-Bibliothek weiß wie das Java-Objekt in JSON umkonvertiert werden muss (und umgekehrt). Auf der Clientseite im JavaScript ist die WebSocket-Verbindung via SockJS²⁷ und STOMP²⁸ geregelt:

brain.js

```

var stompClient = null;

function join(){
    //Verbindungsaufbau
    var socket = new SockJS('/player');
    stompClient = Stomp.over(socket);
    stompClient.connect({}, function(frame) {
        //WebSocketListener
        stompClient.subscribe('/channel/all', function(server_message){
            parseMessage(server_message);
        });
    });
}

//Auswahl an den Server schicken.
stompClient.send("/quiz/player", {}, JSON.stringify({ 'choice': choice }));

```

Damit die WebSockets serverseitig auch aktiviert werden, ist die Ergänzung einer Konfigurationsklasse für die Websockets noch erforderlich:

WebSocketConfig.java

```

package quiz;

import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.simp.config.MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation.AbstractWebSocketMessageBrokerConfigurer;
import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;

@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        //Einrichtung des WebSocketNamespaces
        config.enableSimpleBroker("/channel");
        //Präfix fuer die eingehenden WebSocket-Nachrichten
        config.setApplicationDestinationPrefixes("/quiz");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        //Deklariere die WebSocket-Endpunkte
        //"/quiz/candidate" und "/quiz/referee"
    }
}

```

²⁷ <https://github.com/sockjs>

²⁸ <http://stomp.github.io/>

```
        registry.addEndpoint("/player", "/referee").withSockJS();  
    }  
}
```

Nun sollte dem Verbindungsaufbau von WebSockets nichts mehr im Wege stehen.

Zu guter Letzt folgt noch ein kleiner Block zu JSONP ...

5.4 CrossOrigin

JSON with padding

In der Beispiel WebApp *votequiz* existiert eine WebSchnittstelle, in der das Ergebnis des QuizVotings in Form eines JSON-Objekts ausgegeben wird. Leider hindert die Same-Origin-Policy²⁹ daran, dass andere Webseiten in der Lage sind die Ergebnisse abzufragen. Dieses Problem lässt sich umgehen mit **JSONP**³⁰ (*JSON with padding*). Im Prinzip verlagert man die Daten in die Parameterangaben einer Funktion, denn Skripte sind von der Same-Origin-Policy nicht betroffen.

Damit der JSONP-Trick auch für die *votequiz*-App funktioniert, muss die folgende Java-Klasse geschrieben werden mit einer `@ControllerAdvice`³¹ Notation:

JsonpAdvice.java

```
package quiz;  
  
import org.springframework.web.servlet.mvc.method.annotation.AbstractJsonpResponseBodyAdvice;  
import org.springframework.web.bind.annotation.ControllerAdvice;  
  
@ControllerAdvice  
public class JsonpAdvice extends AbstractJsonpResponseBodyAdvice {  
    public JsonpAdvice() {  
        super("callback");  
    }  
}
```

Mit dem obigen Code sind nun AJAX-Requests vom Client möglich, sobald in der URL-Query das Schlüsselwort *"callback"* steht.

Das wars auch schon mit den Beispielen! Es bleibt das Fazit ...

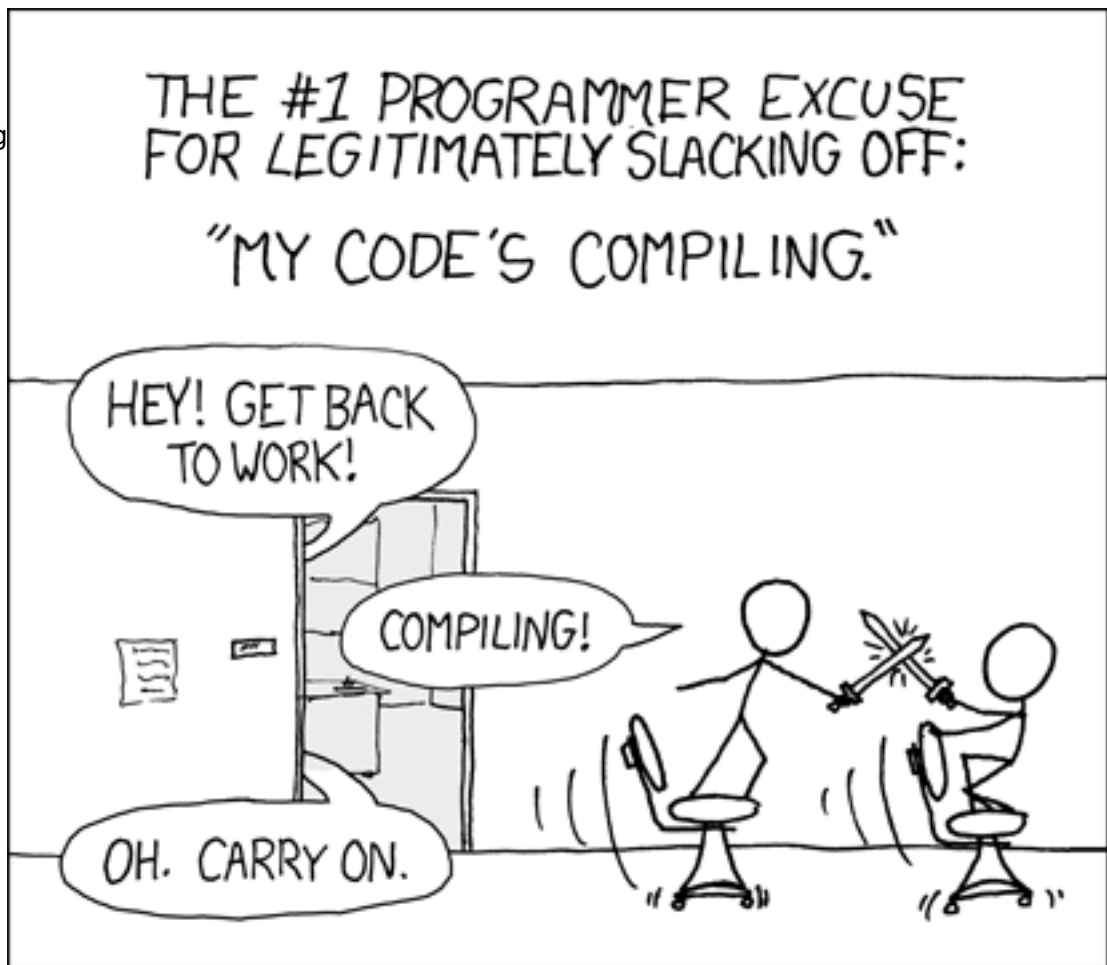
²⁹ <https://de.wikipedia.org/wiki/Same-Origin-Policy>

³⁰ <https://en.wikipedia.org/wiki/JSONP>

³¹ <https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ControllerAdvice.html>

6. Fazit

Spring ist eine gelungene und produktive Umsetzung für die Entwicklung von



Java-WebApplikationen. Die Interoperabilität mit anderen bestehenden Frameworks funktioniert, dank spezieller Interfaces, hervorragend. Der Code schreibt sich dank "Convention over Configuration" und "Dependency Injection" quasi von selbst.

Der einzige Störfaktor, der sich bei der Entwicklung bemerkbar macht, ist die Tatsache, dass es sich bei Java um eine Compiler-Sprache handelt und daher während des Testens viel Zeit verschwendet, wann immer der Java-Code kompiliert werden muss. Um diesen Störfaktor zu entgehen ohne auf Spring verzichten zu müssen bietet *JRebel*³² eine Möglichkeit zur Beschleunigung, indem nur die veränderten Codezeilen neu kompiliert werden müssen und diese in die bestehenden Binaries einpflegt. Eine andere Möglichkeit wäre die Verwendung von *Groovy*³³ im Spring-Projekt. Auf die Faust gesagt ist Groovy eine Skriptsprache, die von der Java Runtime Environment interpretiert wird und mit jeder konventionellen Java-Bibliothek kompatibel ist.

Alle Codebeispiele dieser Ausarbeitung finden sich im folgenden Git-Repo³⁴. Ansonsten existieren noch viele weitere ausführliche Tutorials³⁵ auf der Spring-Homepage. Man merkt sehr, dass die Einsteigerfreundlichkeit einen hohen Stellenwert hat bei Pivotal. Wann immer man auf die Idee kommt einen Webservice einzurichten, sollte immer in Erwägung ziehen das Projekt mit Spring aufzubauen.

Und wenn es für den einen oder anderen Spring dann doch zu umständlich ist, was die Einrichtung von dynamischen Webseiten angeht, und eher etwas in Richtung *Ruby on Rails* sucht: Es gibt auch Grails³⁶!

³² <https://zeroturnaround.com/software/jrebel/>

³³ <http://www.groovy-lang.org/>

³⁴ https://github.com/PattyDePuh/webpub_spring/tree/master/samples

³⁵ <http://spring.io/guides>

³⁶ <https://grails.org/>