# Enterprise Java Apps mit Spring Boot

Dozent

**Michael Theis**

Fach

**Aktuelle Technologien zur Entwicklung verteilter Java Anwendungen**

Autor

**Philipp Konopac**

E-Mail

**konopac@hm.edu**

Matrikelnummer

**---**

Studiengruppe

**IF6**



München, 25.05.2018

# Inhaltsverzeichnis

Einleitung ........ 2

Enterprise Applications ........ 3

- Definition ........ 3
- Grundlagen ........ 3

Spring Boot ........ 4

- Definition ........ 4
- Unterschiede zu Spring ........ 4
- Spring Boot Enterprise Application ........ 5
  - Einbindung in ein Maven-Projekt ........ 5
  - Start-Konfiguration ........ 5
  - Dependency Injection und Beans ........ 6
  - Komponenten ........ 7
  - Persistenzierung ........ 8
  - Services ........ 9
  - REST-Controller ........ 9
  - Security und Authentifizierung ........ 11

Fazit ........ 13

Literaturverzeichnis ........ 14

Abbildungsverzeichnis ........ 15

Tabellenverzeichnis ........ 17

Anhang ........ 18

# Einleitung

Sicherheit: Eines der Grundbedürfnisse jedes Menschen. Was vor einigen Jahrhunderten nach einem simplen Bedürfnis klang, ist heutzutage ein viel komplexerer Wunsch. In unserer modernen Gesellschaft lauern Gefahren hinter jeder Ecke. Das geht so weit, dass sogar im Internet, oder vielleicht eher gerade dort, viel Energie investiert werden muss, um sicher zu sein. Auch wenn es ein Großteil der Bevölkerung überhaupt nicht mitbekommt, passieren täglich unfassbar viele Cyber-Angriffe, die abgewehrt werden müssen. Laut Angaben des Konzern-Chefs zählt die Telekom täglich eine Millionen Angriffe auf ihr Netz. (n-tv Nachrichtenfernsehen GmbH)

Werden Internet- oder Webseiten-Anbieter erfolgreich gehackt, so können deren Daten und die Daten ihrer Nutzer gestohlen werden. Das Speichern von Nutzerdaten birgt also ein Risiko, das große Verantwortung mit sich bringt. Wie also sollen möglichst alle Webseiten so abgesichert sein, dass kein Angreifer eine Chance hat? Die Antwort: Ein totaler Schutz ist unmöglich, jedoch können schwache Angriffe herausgefiltert werde. Selbst diese Aufgabe zu meistern ist schwer, doch hier kommt Spring Boot ins Spiel: Das Web-Entwicklungs-Framework kann grundlegende Funktionen, wie zum Beispiel einen sicheren Login, übernehmen und dem Entwickler so Freiraum für andere Themen schaffen. Diese und weitere Funktionen von Spring Boot sollen im Folgenden in Anwendung auf eine Enterprise-Java-Anwendung vorgestellt werden.

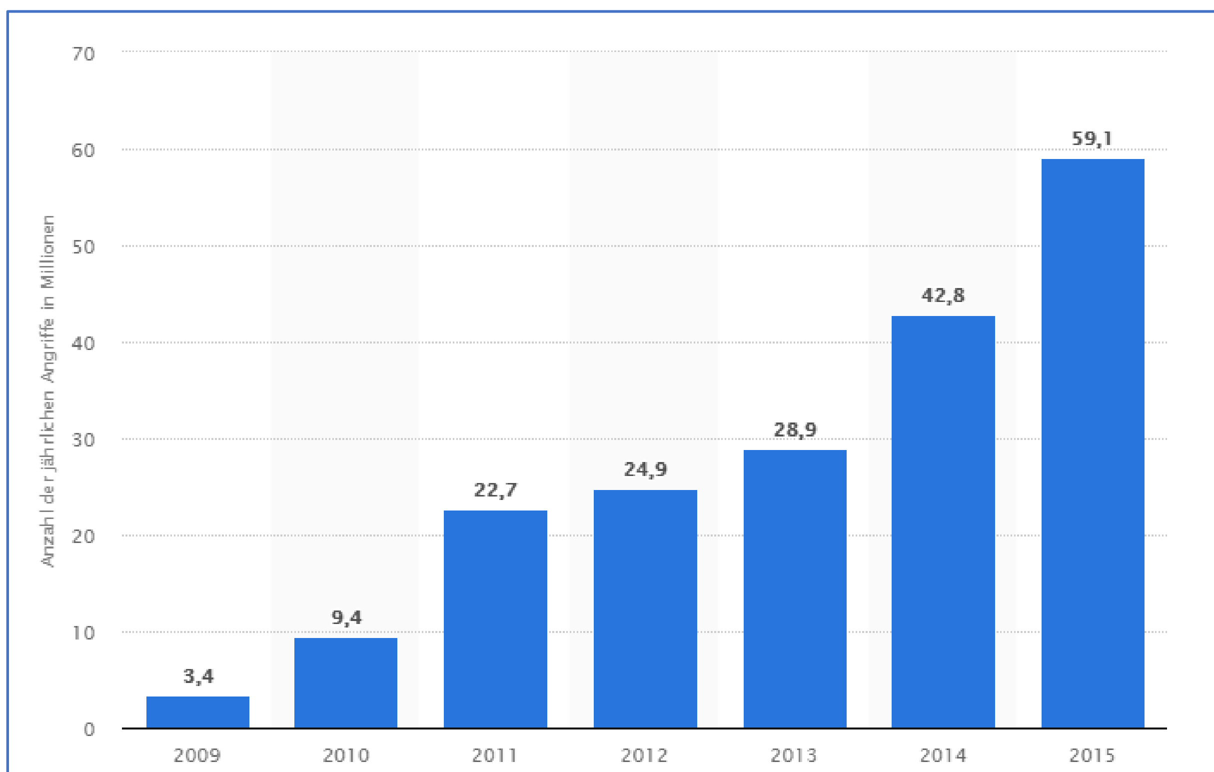


*Abbildung 1 - Anzahl der jährlichen Cyberangriffe weltweit in den Jahren 2009 bis 2015 (in Millionen) (Statista GmbH)*

## Enterprise Applications

Um das Thema dieser Arbeit genauer zu verstehen, wird es in zwei Bereiche unterteilt. Zu Beginn wird der Begriff „Enterprise Application" – oder auf Deutsch „Unternehmensanwendung" – analysiert.

### Definition

Enterprise Applications sind hoch komplexe Anwendungen, die von Unternehmen genutzt werden. Sie sind oft sehr groß und meistens auch alt, da sie schwer ausgetauscht werden können. (Microsoft)

### Grundlagen

Enterprise Applications werden normalerweise so entworfen, dass sie über mehrere Netzwerke hinweg mit anderen Enterprise Applications kommunizieren oder zusammenarbeiten können. Dabei muss auf hohe Sicherheit und Administrierbarkeit geachtet werden, weshalb diese großen Anwendungen oft unternehmensintern betrieben werden. Allerdings gibt es auch moderne Ansätze wie zum Beispiel software-as-a-service. Hier wird die Anwendung von einem Dienstleister nur angekauft beziehungsweise gemietet. Für Sicherheit und andere Anforderungen ist dann der Entwickler, nicht das einkaufende Unternehmen, verantwortlich. Ein weiterer Trend geht in Richtung Cloud-Computing, wodurch jegliche Infrastruktur in die Cloud verschoben wird.

Ein paar typische Bereiche einer Enterprise Application sind ein Zahlungsprozess, automatisches E-Mail-Versand-System, Kundenbetreuung, Abbildung der Geschäftslogik des Unternehmens und Human Resources Management. (Beal, V.)

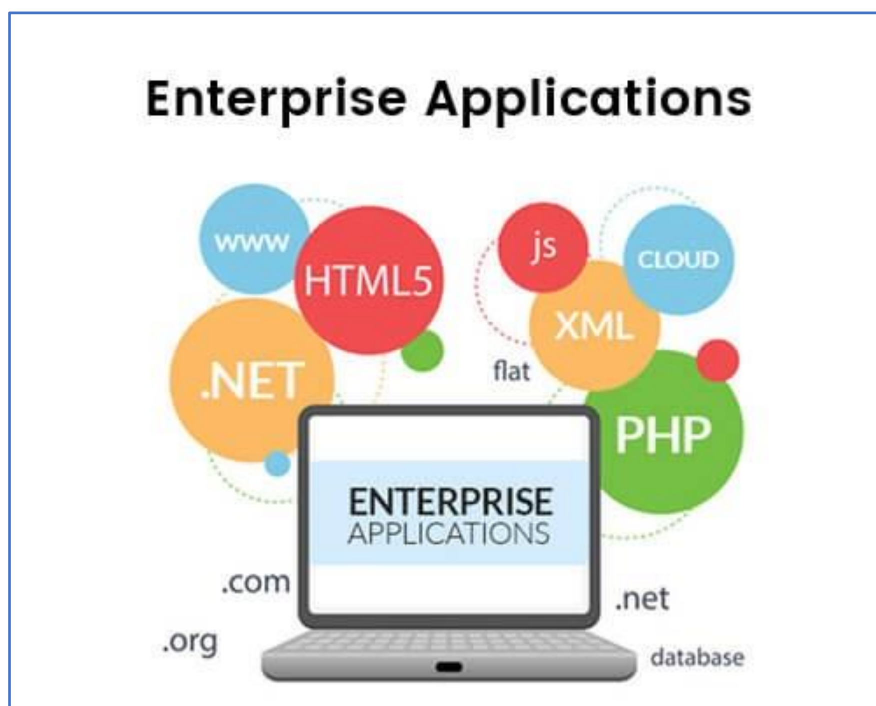


*Abbildung 2 - Häufig genutzte Plattformen für die Entwicklung von Enterprise Applications (Alam, A.)*

# Spring Boot

Es folgt der Hauptteil dieser Arbeit, in dem Spring Boot unter die Lupe genommen und mit dem zuvor definierten Begriff „Enterprise Application“ in Verbindung gebracht wird.

## Definition

„Ein Webframework […] ist eine Software, die für die Entwicklung von dynamischen Webseiten, Webanwendungen oder Webservices ausgelegt ist. Sich wiederholende Tätigkeiten werden vereinfacht und die Wiederverwendung von Code und die Selbstdokumentation der Software-Entwicklung gefördert. […] Durch vordefinierte und vorgefertigte Klassen werden häufig gebrauchte Funktionen wie Mailversand, sichere Authentifizierung und Authentisierung, Sicherheitsfunktionen, Lokalisierung, Performance (z. B. HTTP Caching) oder grundlegende Funktionen für Webformulare vom Framework mitgebracht.“ (Wikimedia Foundation Inc.)

Da es ein Webframework ist, trifft diese Definition auf Spring Boot sehr gut zu. Wie genau diese Features umgesetzt und angewendet werden, wird im Folgenden erklärt.

## Unterschiede zu Spring

Das Framework Spring MVC ist sehr mächtig. Es bietet Lösungen für viele Probleme, wie zum Beispiel Dependency Injection, Security, Schnittstellen und das Einbinden von anderen Frameworks. Leider bringt diese Masse an Funktionalität auch Kosten mit sich: Konfiguration und ein schnell überfülltes Projekt. Laut Angaben des Entwicklers soll Spring Boot genau diese Schwachstellen ausmerzen. Spring Boot macht es möglich, eine spring-basierte Anwendung innerhalb von Minuten aufzusetzen und zu starten. Außerdem soll dafür kaum bis gar keine Konfiguration nötig sein. Große Module von Spring MVC werden standardmäßig weggelassen, sodass eine leichtgewichtige Anwendung entsteht.

Man sollte also nicht nach Unterschieden zwischen Spring und Spring Boot suchen, sondern lieber untersuchen, welche Probleme von Spring durch Spring Boot gelöst werden.

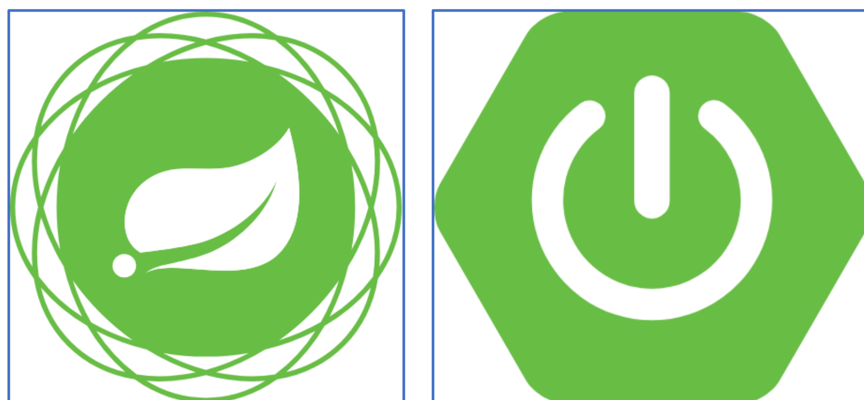

*Abbildung 3 - Spring und Spring Boot Icon (Pivotal Software Inc.)*

## Spring Boot Enterprise Application

Um die beiden Begriffe „Spring Boot" und „Enterprise Application" endgültig zusammenzuführen, wird in diesem Kapitel der grundlegende Aufbau einer Spring Boot Anwendung im Enterprise Umfeld beispielhaft erklärt. Dafür werden allgemeine Kenntnisse über Java, Maven, HTTP und Datenbanken vorausgesetzt. Begleitend zum Text findet sich im Anhang ein Beispielprojekt, welches die besprochenen Themen veranschaulicht.

### Einbindung in ein Maven-Projekt

Zu Beginn muss ein gewöhnliches Maven-Projekt erstellt werden. In die von Maven generierte Konfigurationsdatei müssen nun wenige Werte eingetragen werden, um die noch leere Anwendung in eine Spring Boot Anwendung zu verwandeln. Durch Einstellen des Parent auf *spring-boot-starter-parent* (Abbildung 4) werden alle Standardeinstellungen von Spring Boot geerbt. Die einzige benötigte Maven-Dependency ist dann *spring-boot-starter-web* (Abbildung 4). Nun werden alle Standard-Spring-Boot-Libraries von Maven geladen. (Pivotal Software Inc.)

```
<!-- Inherit defaults from Spring Boot -->
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.0.BUILD-SNAPSHOT</version>
</parent>

<!-- Add typical dependencies for a web application -->
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
</dependencies>
```

*Abbildung 4 - Maven Starter Konfiguration (Pivotal Software Inc.)*

### Start-Konfiguration

Eines der großen Ziele von Spring Boot ist es, die XML-Konfiguration von Spring zu vereinfachen und die Entwickler von Spring Boot haben dieses Konzept bis auf die Spitze getrieben: Zum Starten einer Spring Boot Anwendung werden nur zwei Code-Fragmente benötigt (Abbildung 5).

1. Eine main-Methode, wie sie in Java üblich ist, mit dem Aufruf
   *SpringApplication.run(Application.class, args);*
2. Eine Annotation *@SpringBootApplication* an der „Main-Klasse"

```java
@SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentScan
public class Application {

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

}
```

*Abbildung 5 - Start Konfiguration einer Spring Boot Anwendung (Pivotal Software Inc.)*

Die Annotation stellt eine Konfigurations-Sammlung dar, die eigentlich aus den drei Annotationen *@EnableAutoConfiguration*, *@ComponentScan* und *@Configuration* besteht. Diese Annotationen sind also gleichzusetzen mit *@SpringBootApplication*. (Pivotal Software Inc.)

*Tabelle 1 - Springs Standard-Annotations zur Konfiguration (Konopac, P.)*

| Annotation | Funktion |
|---|---|
| **@EnableAutoConfiguration** | Aktiviert Spring Boots automatischen Konfigurationsmechanismus |
| **@ComponentScan** | Aktiviert Spring Boots Scannen nach Komponenten. Hier kann ein Package-Pfad angegeben werden. Wenn kein Package angegeben wird, so wird das, in dem die Klasse mit dieser Annotation liegt, und dessen Unterpackages durchsucht. |
| **@Configuration** | Markiert eine Konfigurations-Klasse |
| **@SpringBootApplication** | Dieselbe Funktion wie die oberen drei Annotationen in ihren Standardeinstellungen |

### Dependency Injection und Beans

„Als Dependency Injection […] wird in der objektorientierten Programmierung ein Entwurfsmuster bezeichnet, welches die Abhängigkeiten eines Objekts zur Laufzeit reglementiert: Benötigt ein Objekt beispielsweise bei seiner Initialisierung ein anderes Objekt, ist diese Abhängigkeit an einem zentralen Ort hinterlegt – es wird also nicht vom initialisierten Objekt selbst erzeugt.“ (Wikimedia Foundation Inc.) Um dieses Modell zu realisieren, müssen alle von Spring verwalteten Klassen – auch Beans genannt – ein Interface nach dem Prinzip von Inversion of Control implementieren. Das bedeutet, dass keine Klasse auf Beans einer anderen Architektur-Schicht direkt zugreifen darf, sondern immer nur auf dessen Interface. Die eigentliche Instanz wird erst zur Laufzeit von Spring eingefügt. (Abbildung 6)

Es gibt verschiedene Arten von Beans, zu denen *@Component*, *@Service*, *@Repository* und *@Controller* zählen. Auf diese Beans wird in den folgenden Abschnitten eingegangen.

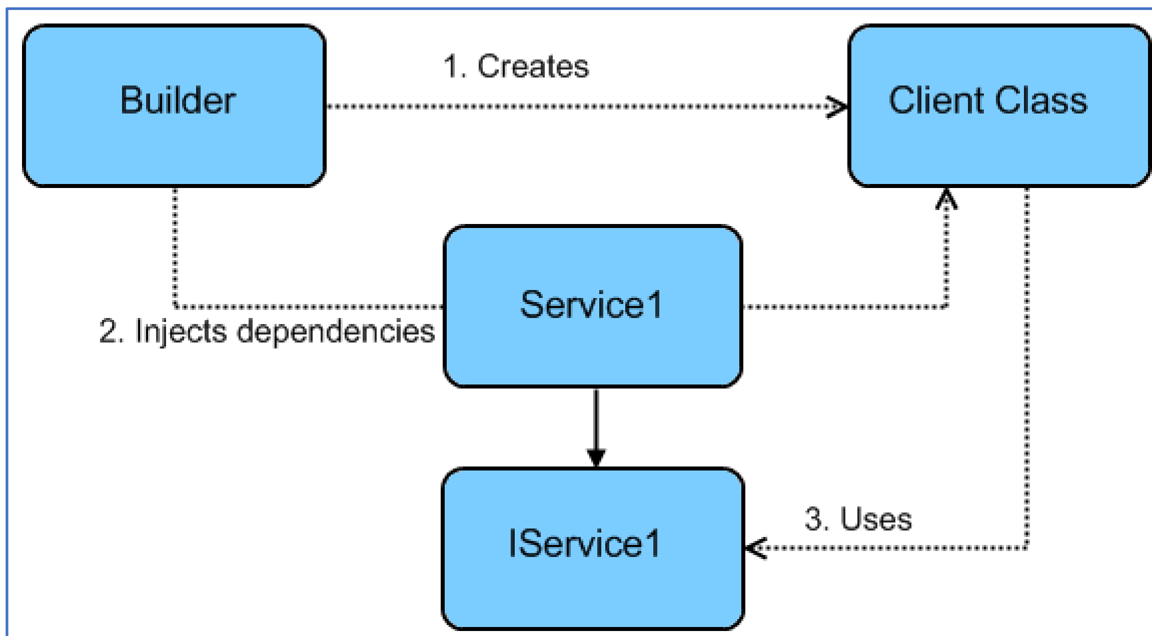


*Abbildung 6 - Dependency Injection (Millar, E.)*

## Komponenten

Die Annotation *@Component* markiert eine Komponente, welche von Spring gesucht wird, wenn *@ComponentScan* aktiv ist (Tabelle 1). *@Service*-, *@Repository*- und *@Controller*-Klassen werden deshalb gefunden, weil diese Annotationen wiederum die *@Component*-Annotation haben. Während die drei „Unterkomponenten“ spezielle Aufgabenbereiche abdecken, ist die Anwendung von Komponenten nicht definiert, weshalb sie kaum verwendet werden. Mit der Annotation wäre es dennoch möglich, weitere Komponenten zu definieren.

```
@Component
public @interface MyComponent {
    ...
}
```

*Abbildung 7 - Eine eigene Spring-Komponente (Konopac, P.)*

Die selbst geschriebene Annotation *@MyComponent* wäre gleichrangig mit *@Service*, *@Repository* und *@Controller* (Abbildung 7).

*Tabelle 2 - Komponenten Typen von Spring (Konopac, P.)*

| Annotation | Funktion |
|---|---|
| **@Component** | Allgemeine Komponente, die von Spring gesucht wird |
| **@Repository** | Datenbank-Zugriffs-Klasse, ist eine Komponente |
| **@Service** | Geschäftslogik-Klasse, ist eine Komponente |
| **@Controller** | Kommunikation mit Außenwelt, ist eine Komponente |

### Persistenzierung

Das MVC-Prinzip von Spring beachtend gibt es eine Datenbank-Zugriffs-Schicht. Sogenannte DatabaseAccessObject-Klassen werden in Spring mit @Repository markiert. Zusätzlich bindet diese Annotation den *PersistenceExceptionTranslationPostProcessor* ein, welcher für das Umwandeln von plattformspezifischen Exceptions in Spring Exceptions verantwortlich ist.

```java
@Repository
public class DemoUserRepository implements UserRepository {

	private final Map<String, User> users;

	public DemoUserRepository() {
		this.users = new HashMap<>();

		// insert users
		this.users.put("philipp", new User("philipp", "123"));
		this.users.put("jasmin", new User("jasmin", "456"));
	}

	@Override
	public Optional<User> findByUsername(String username) {
		return Optional.of(getUsers().get(username));
	}

	private Map<String, User> getUsers() {
		return users;
	}
}
```

*Abbildung 8 - Beispiel einer Repository Klasse (Konopac, P.)*

Eine Beispiel-Implementierung ist in Abbildung 8 zu sehen, in der eine Klasse *DemoUserRepository* ein Interface *UserRepository* implementiert und mit der Annotation *@Repository* markiert ist. Das Interface ist erforderlich um dem zuvor erwähnten Prinzip Inversion of Control gerecht zu werden. Das Interface bietet eine Methode *findByUsername(String)* für andere Architektur-Schichten an, die von der Demo-Klasse implementiert wird. Zur Vereinfachung wird hier eine private Map-Variable als „Datenbank“ verwendet. Zugriffe auf diesen Speicher sind im Anwendungsfall durch Zugriffe auf eine echte Datenbank zu ersetzen.

### Services

Die nächst höhere Schicht ist die Controller-Schicht beziehungsweise die Service-Schicht, wie Spring sie nennt. Services beinhaltet die Geschäftslogik der Anwendung und sind somit der Kern des Projekts. Alle anderen Architektur-Schichten sollten austauschbar sein. Markiert ist eine solche Klasse mit der Annotation *@Service*.

```
@Service
public class DemoUserService implements UserService, UserDetailsService {

        @Autowired
        private UserRepository userRepository;

        @Override
        public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
                return userRepository.findByUsername(username).orElseThrow(() ->
                        new UsernameNotFoundException("User with username " + username + " not found"));
        }
}
```

*Abbildung 9 - Beispiel einer Service Klasse (Konopac, P.)*

In dem Beispiel aus Abbildung 9 implementiert die mit *@Service* annotierte Klasse *DemoUserService* zwei Interfaces. *UserService* ist dabei ein Interface, das definiert wurde um Inversion of Control einzuhalten. *UserDetailsService* wird von Spring vorgegeben, da im Beispiel der Login-Mechanismus von Spring verwendet werden soll. Dieses Interface definiert eine Methode *loadUserByUsername(String)*, welche von der Demo-Klasse implementiert wird. Die *UserRepository*-Instanz, die in dieser Methode verwendet wird, wird per Dependency Injection dynamisch von Spring geladen. Dafür ist die Annotation *@Autowired* verantwortlich. Wichtig ist hier, dass nur auf Methoden des Repository-Interface, welches vorher definiert wurde, zugegriffen werden kann.

### REST-Controller

Die höchste Schicht im Schichtenmodell von Spring ist die Controller-Ebene. Klassen, die mit *@Controller* markiert werden, können Daten nach außen weitergeben. Typische Anwendungen haben zum Beispiel eine HTML und JavaScript basierte Oberfläche, die mittels HTTP-Anfragen auf diese REST-Schnittstelle zugreift. Nur Klassen, die *@Controller* oder *@RestController*, eine spezielle Art von Controller, zugewiesen haben, können die Annotationen *@RequestMapping*, *@GetMapping* und *@PostMapping* nutzen. Diese können HTTP-Requests anhand von URLs an Klassen und Methoden weiterleiten. Nach Verarbeitung durch eine solche Methode kann eine Antwort, standardmäßig im JSON-Format, an den Anfragenden gesendet werden.

Es können allerdings nicht nur Daten gesendet, sondern auch empfangen werden. Mit den Annotationen *@RequestParam* und *@RequestBody* vor den Parametern der Rest-Methoden können

HTTP-Request-Parameter oder sogar ganze Objekte an die Anwendung übergeben werden. Während Request-Parameter direkt in der URL stehen, werden Objekte vorher im JSON-Format zusammengebaut und dann an die HTTP-Anfrage gehängt. Dies ist in der Oberfläche meistens mit JavaScript realisiert.

*Abbildung 10 - Beispiel einer HTTP-Anfrage (Konopac, P.)*

In Abbildung 10 erkennt man die Darstellung der URL einer möglichen HTTP-Anfrage. Sie ist aus folgenden Elementen aufgebaut:

- Protokoll
- Adresse der Anwendung, an die die Anfrage gerichtet wird
- Request Mapping (*@RequestMapping*, *@GetMapping* oder *@PostMapping*)
- Key-Value-Paare für die Request Parameter

Das Beispiel aus Abbildung 11 zeigt die Klasse *UserController*, welche keine Interfaces im Sinne von Inversion of Control definieren muss, da keine andere Klasse direkt auf eine Controller-Klasse zugreifen sollte. Die Annotationen *@RestController* und *@RequestMapping(„/rest/user“)* sorgen dafür, dass Spring alle Anfragen auf diese Anwendung mit der URL */rest/user* an diese Klasse weiterleitet. Eine weitere Annotation *@GetMapping(„/hello“)* zwingt eine Methode schließlich dazu, alle GET-Anfragen mit der Request-URL */rest/user/hello* zu bearbeiten. Diese Methode gibt lediglich einen String in Form einer HTTP-Response zurück.

```
@RequestMapping("/rest/user")
@RestController
public class UserController {

        @GetMapping("/hello")
        public String index() {
                return "Hello World!";
        }

}
```

*Abbildung 11 - Beispiel einer Controller Klasse (Konopac, P.)*

## Security und Authentifizierung

Da die Anwendung nach den bisherigen Schritten zwar funktioniert, allerdings auch jeder darauf zugreifen kann, wird in diesem Abschnitt eine Authentifizierung eingebaut. Dabei wir das Spring Boot Security Plugin verwendet, welches mittels der Maven Dependency *spring-boot-starter-security* eingebunden wird (Abbildung 12).

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

*Abbildung 12 - Maven Dependency für Spring Boot Security (Konopac, P.)*

Um das Security Plugin zu aktivieren, muss eine Konfigurations-Klasse angelegt werden. Diese zeichnet sich durch eine Annotation *@Configuration* aus. Eine weitere Annotation *@EnableWebSecurity* aktiviert die Authentifizierung. Nun werden standardmäßig alle Ressourcen von Spring mit einem Login geschützt. Bei einem Aufruf des oben definierten Controllers wird jetzt auf eine Login-Seite weitergeleitet. Nutzer und Passwort werden von Spring bei Anwendungsstart generiert und in der Konsole ausgegeben. Nach erfolgreichem Anmelden kann auf die Ressource zugegriffen werden.

Da die meisten Anwendungen verlangen, ihre User selbst zu verwalten, bietet Spring Boot eine einfache Möglichkeit, dies zu bewerkstelligen: Erweitert die geschaffene Konfigurationsdatei die Klasse *WebSecurityConfigurerAdapter*, so kann die Methode *configure(AuthenticationManagerBuilder)* überschrieben werden. Dort wird ein User-Service an Spring übergeben. Im Beispiel von Abbildung 13 ist das eine Instanz des in einem vorherigen Abschnitt erstellten *DemoUserService*. Alle Nutzerdaten werden nun durch diesen Service und nichtmehr durch Spring verwaltet.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService).passwordEncoder(getPasswordEncoder());
}
```

*Abbildung 13 - Konfiguration eines eigenen User-Service (Konopac, P.)*

```
@Override
protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("/**").authenticated()
                .and()
                .formLogin().permitAll()
                .and()
                .logout().logoutRequestMatcher(new AntPathRequestMatcher("/logout"));
}
```

*Abbildung 14 - Standard Einstellung von Spring Security (Konopac, P.)*

Neben der Einstellung der Nutzerdaten bietet Spring auch die Möglichkeit, spezielle REST-URLs mit einem Login zu schützen und andere wiederum nicht, aber auch Zugriffsrechte auf Nutzer zu verteilen. So kann es zum Beispiel sinnvoll sein, alle URLs, die mit */admin* beginnen, nur gewissen Nutzern zu öffnen. In der einfachsten Konfiguration von Spring Security werden alle URLs geschützt, jedoch nicht die Login-URL, da diese für noch nicht angemeldete Nutzer verfügbar sein muss (Abbildung 14).

# Fazit

Die Frage nach den Problemen von Spring, die durch Spring Boot gelöst werden können, sollte nun leicht beantwortet werden können: Der Konfigurationsaufwand von Spring Boot Anwendungen ist schwindend gering. Um eine Spring Boot Anwendung startfähig aufzusetzen werden nur wenige Schritte benötigt, die innerhalb von wenigen Minuten abgearbeitet werden können. Auch das Einbinden von komplexeren Funktionalitäten wie ein Login-System erfolgt leicht und gut konfigurierbar.

Abschließend kann festgehalten werden, dass Spring Boot als unterstützendes Framework meistens eine bessere Wahl als Spring und viele andere Web-Frameworks ist, da der Trend in der Anwendungsentwicklung eindeutig Richtung „leichtgewichtige Projekte“ mittels Cloud-Computing und Micro-Services geht.

# Literaturverzeichnis

**Beal, V. - QuinStreet Inc.**

enterprise application (2018) - *https://www.webopedia.com/TERM/E/enterprise_application.html* (Abgerufen: 18.05.2018)

**Microsoft.**

Chapter 1: What is an Enterprise Application? (2018) - *https://msdn.microsoft.com/en-us/library/aa267045(v=vs.60).aspx* (Abgerufen: 18.05.2018)

**n-tv Nachrichtenfernsehen GmbH.**

Eine Million Attacken pro Tag: Telekom fürchtet immer mehr Cyber-Angriffe (2014) - *https://www.n-tv.de/wirtschaft/Telekom-fuerchtet-immer-mehr-Cyber-Angriffe-article13892991.html* (Abgerufen: 12.05.2018)

**Pivotal Software Inc.**

Spring Boot Reference Guide (2018) - *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started* (Abgerufen: 18.05.2018)

Using the @SpringBootApplication Annotation (2018) - https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-using-springbootapplication-annotation.html (Abgerufen: 18.05.2018)

**Wikimedia Foundation Inc.**

Dependency Injection (2018) - *https://de.wikipedia.org/wiki/Dependency_Injection* (Abgerufen: 18.05.2018)

Webframework (2018) - *https://de.wikipedia.org/wiki/Webframework* (Abgerufen: 18.05.2018)

# Abbildungsverzeichnis

**Alam, A.**

Enterprise Applications (2015) - *https://qph.ec.quoracdn.net/main-qimg-8b974d656ea2a3035f89d731c493180f* (Abgerufen: 18.05.2018)

**Konopac, P.**

Beispiel einer HTTP-Anfrage (2018) – *Im Rahmen dieser Arbeit vom Autor selbst erstellt*

Beispiel einer Repository Klasse (2018) – *Im Rahmen dieser Arbeit vom Autor selbst erstellt*

Beispiel einer Service Klasse (2018) – *Im Rahmen dieser Arbeit vom Autor selbst erstellt*

Eine eigene Spring-Komponente (2018) – *Im Rahmen dieser Arbeit vom Autor selbst erstellt*

Konfiguration eines eigenen User-Service (2018) – *Im Rahmen dieser Arbeit vom Autor selbst erstellt*

Maven Dependency für Spring Boot Security (2018) – *Im Rahmen dieser Arbeit vom Autor selbst erstellt*

Standard Einstellung von Spring Security (2018) – *Im Rahmen dieser Arbeit vom Autor selbst erstellt*

**Millar, E.**

How Dependency Injection (DI) Works In Spring Java Application Development (2016) - *https://dzone.com/articles/how-dependency-injection-di-works-in-spring-java-a* (Abgerufen: 18.05.2018)

**Pivotal Software Inc**.

Maven Installation (2018) - https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started (Abgerufen: 22.05.2018)

Spring Boot (2018) - *https://spring.io/img/homepage/icon-spring-boot.svg* (Abgerufen: 18.05.2018)

Spring Framework 5 (2018) - *https://spring.io/img/homepage/icon-spring-framework.svg* (Abgerufen: 18.05.2018)

Using the @SpringBootApplication Annotation (2018) - *https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-using-springbootapplication-annotation.html* (Abgerufen: 18.05.2018)

**Statista GmbH**.

Anzahl der jährlichen Cyberangriffe weltweit in den Jahren 2009 bis 2015 (in Millionen) (2015) - *https://de.statista.com/statistik/daten/studie/348766/umfrage/jaehrliche-anzahl-von-internetangriffen-weltweit/* (Abgerufen: 12.05.2018)

# Tabellenverzeichnis

**Konopac, P.**

Komponenten Typen von Spring (2018) – *Im Rahmen dieser Arbeit vom Autor selbst erstellt*

Springs Standard-Annotations zur Konfiguration (2018) – *Im Rahmen dieser Arbeit vom Autor selbst erstellt (Informationsquelle: Pivotal Software Inc.)*

# Anhang

Die wichtigsten Komponenten des Beispielprojekts sind hier in Form von Screenshots aufgelistet. Das gesamte und eventuell aktuellere Projekt ist unter https://github.com/konopac/spring-boot-demo zu finden.

**1) pom.xml des Beispielprojekts**

```
1   <project xmlns="http://maven.apache.org/POM/4.0.0"
2           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
4
5           <modelVersion>4.0.0</modelVersion>
6           <groupId>edu.hm.konopac</groupId>
7           <artifactId>spring-boot-demo</artifactId>
8           <packaging>war</packaging>
9           <version>0.0.1-SNAPSHOT</version>
10          <name>Spring Boot Demo Application</name>
11
12          <parent>
13                  <groupId>org.springframework.boot</groupId>
14                  <artifactId>spring-boot-starter-parent</artifactId>
15                  <version>2.0.0.RELEASE</version>
16          </parent>
17
18          <dependencies>
19                  <dependency>
20                          <groupId>org.springframework.boot</groupId>
21                          <artifactId>spring-boot-starter-web</artifactId>
22                  </dependency>
23
24                  <dependency>
25                          <groupId>org.springframework.boot</groupId>
26                          <artifactId>spring-boot-starter-security</artifactId>
27                  </dependency>
28
29                  <dependency>
30                          <groupId>junit</groupId>
31                          <artifactId>junit</artifactId>
32                          <scope>test</scope>
33                  </dependency>
34          </dependencies>
35
36          <build>
37                  <finalName>spring-boot-demo</finalName>
38          </build>
39
40  </project>
```

**2) Main Implementierung des Beispielprojekts**

```
1   package edu.hm.konopac.boot;
2
3   import org.springframework.boot.SpringApplication;
4   import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6   @SpringBootApplication
7   public class Main {
8
9           public static void main(String[] args) {
10                  SpringApplication.run(Main.class, args);
11          }
12  }
```

## 3) Repository Implementierung des Beispielprojekts

### a) Interface

```
1   package edu.hm.konopac.boot.repository;
2
3   import java.util.Optional;
4
5   import edu.hm.konopac.boot.entity.User;
6
7   public interface UserRepository {
8
9           public Optional<User> findByUsername(String username);
10
11  }
```

### b) Implementierung

```
1   package edu.hm.konopac.boot.repository;
2
3   import java.util.HashMap;
4   import java.util.Map;
5   import java.util.Optional;
6
7   import org.springframework.stereotype.Repository;
8
9   import edu.hm.konopac.boot.entity.User;
10
11  @Repository
12  public class DemoUserRepository implements UserRepository {
13
14          private final Map<String, User> users;
15
16          public DemoUserRepository() {
17                  this.users = new HashMap<>();
18
19                  // insert users
20                  this.users.put("philipp", new User("philipp", "123"));
21                  this.users.put("jasmin", new User("jasmin", "456"));
22          }
23
24          @Override
25          public Optional<User> findByUsername(String username) {
26                  return Optional.of(getUsers().get(username));
27          }
28
29          private Map<String, User> getUsers() {
30                  return users;
31          }
32  }
```

## 4) Service Implementierung des Beispielprojekts

### a) Interface

```
1    package edu.hm.konopac.boot.service;
2
3    public interface UserService {
4
5    }
```

### b) Implementierung

```
1   package edu.hm.konopac.boot.service;
2
3   import org.springframework.beans.factory.annotation.Autowired;
4   import org.springframework.security.core.userdetails.UserDetails;
5   import org.springframework.security.core.userdetails.UserDetailsService;
6   import org.springframework.security.core.userdetails.UsernameNotFoundException;
7   import org.springframework.stereotype.Service;
8
9   import edu.hm.konopac.boot.repository.UserRepository;
10
11  @Service
12  public class DemoUserService implements UserService, UserDetailsService {
13
14          @Autowired
15          private UserRepository userRepository;
16
17          @Override
18          public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
19                  return userRepository.findByUsername(username).orElseThrow(() ->
20                          new UsernameNotFoundException("User with username " + username + " not found"));
21          }
    }
```

**5) Controller Implementierung des Beispielprojekts**

```
1   package edu.hm.konopac.boot.controller;
2
3   import org.springframework.web.bind.annotation.GetMapping;
4   import org.springframework.web.bind.annotation.RequestMapping;
5   import org.springframework.web.bind.annotation.RestController;
6
7
8   @RequestMapping("/rest/user")
9   @RestController
10  public class UserController {
11
12          @GetMapping("/hello")
13          public String index() {
14                  return "Hello World!";
15          }
16
17  }
```

## 6) Security Implementierung des Beispielprojekts

```java
package edu.hm.konopac.boot.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.util.matcher.AntPathRequestMatcher;

@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

        @Autowired
        private UserDetailsService userDetailsService;

        /**
         * Configure authentication.
         * @see org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter
         */
        @Override
        protected void configure(AuthenticationManagerBuilder auth) throws Exception {
                auth.userDetailsService(userDetailsService).passwordEncoder(getPasswordEncoder());
        }

        /**
         * Configure Security.
         * @see org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter
         */
        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http.authorizeRequests()
                        .antMatchers("/**").authenticated()
                        .and()
                        .formLogin().permitAll()
                        .and()
                        .logout().logoutRequestMatcher(new AntPathRequestMatcher("/logout"));   // logout url
        }

        private PasswordEncoder getPasswordEncoder() {
                return new PasswordEncoder() {

                        @Override
                        public boolean matches(CharSequence chars, String s) {
                                return true;
                        }

                        @Override
                        public String encode(CharSequence chars) {
                                return chars.toString();

                        }
                };
        }
}
```