Spring - eine Alternative oder eine Ergänzung zu JEE?

Von Michael Haas

Eigenständigkeitserklärung

versichere hiermit, dass ich meine Studienarbeit mit dem Thema

„Spring - eine Alternative oder eine Ergänzung zu Java EE?“

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____ _____

(Unterschrift) (Datum)

Inhaltsverzeichnis

I. Einführung

Das Ziel der Entwicklung des Spring Frameworks war es die Entwicklung Java Enterprise Edition Anwendungen vereinfachen. Basierend auf der im Jahr 1996 veröffentlichten JavaBeans-Spezifikation, was ein Softwarekomponentenmodell für Java ist. Es definierte einen Satz Kodierrichtlinien, die die Wiederverwendung von POJOs [Plain Old Java Objects] und deren Zusammenstellung zu komplexen Anwendungen gestattete. Der usprüngliche Sinn dieses Modells war es ein Allzweckmittel zur Definition wiederverwendbarer Anwendungskomponenten zu sein, was in der Entwickler-Community letztlich aber nur zur Erstellung von Oberflächen-Widgets gebraucht worden war. Sie wollten mehr.

Anspruchsvolle Anwendungen sind häufig auf Services angewiesen. Beispiele für Services sind Transaktionssupport, Sicherheit und verteilte Berehcnungen. Diese sind aber nicht direkt in der JavaBeans-Spezifikation vorhanden. Deswegen wurde im Jahr 1998 die erste Version der Enterprise-JavaBeans-Spezifikation (kurz: EJB) veröffentlicht.

Die EJBs erweiterten das Konzept auf die Serverseite und beinhaltete die notwendigen Unternehmensservices. Die Einfachheit, die in den zuvor definierten JavaBeans gegeben war, konnte in den EJBs nicht aufrechterhalten werden: Das deklarative Programmiermodell von EJB vereinfacht zwar viele Infrastrukturaspekte, verkomplizierte aber zugleich die Entwicklung, indem Interfaces statt Deployment-Diskriptoren und Fügecode für die Basis verwendet werden mussten.

Die Leistungsfähigkeit konnte durch die Einführung der objektorientieren Paradigmen *aspect-oriented-programming (kurz: AOP)* und *dependency injection (kurz: DI)* wieder in die EJBs eingebracht werden. Diese Paradigmen haben EJBs mit einem deklarativen Programmiermodell ausgestattet, auf das später noch eingegangen wird.

Zu dieser Zeit jedoch, war die Weiterentwicklung der EJB-Spezifikation zu langsam. Es hatten sich nämlich bereits andere Entwicklungs-Frameworks, die ebenfalls POJO-basiert waren, etabliert. Einer der führendsten war das Open-Source Spring Framework.

Diese Arbeit befasst sich, Jahre später, mit der Frage, ob Spring als POJO-basiertes Framework eine Alternative oder eine Ergänzung zu der Java EE Spezifikation bildet. Hierfür werden die Programmierungsparadigmas AOP und DI sowie das Pattern MVC in Spring erläutert. Danach wird auf ein eigens erstelltes Projekt eingegangen, welches das Arbeiten mit Spring erläutert. Letztlich wird ein Vergleich zwischen Java EE 7 und der neuesten Version des Spring Frameworks gezogen.

II. Spring Framework

1. Paradigmen

A. Dependency Injection

Martin Fowler prägt den Begriff indem er sich die Frage stellte, was genau in der „Inversion of Control“ invertiert wird und stellte fest, dass es der Abhängigkeitserwerb war. Daraufhin definierte er den Begriff „Dependency Injection“: Die Abhängigkeiten innerhalb eines Objekts werden von einer Dritten Instanz, einem Container, in das Objekt injiziert.

a. Beans in Spring

Das Prinzip der Dependency Injection greift innerhalb von Spring auf das Erstellen von Beans zurück. Beans sind Objekte, die vom IoC-Container des Frameworks zusammengestellt, erzeugt und kontrolliert werden. Sie werden entweder innerhalb einer XML-Konfigurationsdatei (*configuration metdata*), innerhalb einer Java-Klasse oder durch Annotationen konfiguriert.

```
<bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
</bean>
```

Codeausschnitt 1: Generische Beandefinition

Aus den Business Objekts der Applikation (in Spring POJOs [„Plain Old Java Objects“]) werden an Hand der Konfigurationsmetadaten Beans erstellt, die die Basis für alle auf das Spring Framework basierte Funktionen bereitstellen.

Initialisieren und Zerstören von Beans

Beans müssen gegebenenfalls auch initialisiert werden. Dies wird dadurch erreicht, dass ein Bean in einer XML-Konfigurationsdatei durch das Tag *„init-method“* mit dem Attribut *„=“[Methodenname]““* angereichert wird. Bei Initialisierung wird dann diese Methode ausgeführt.

```
<bean id="..." class="..."
        init-method=“[methodenname]“>
        <!-- collaborators and configuration for this bean go here -->
</bean>
```

Codeausschnitt 2: Bean-init

Ähnlich verhält es sich mit dem Zerstören von Beans. Hier muss das Tag *„destroy-method“* mit einer Methodenbezeichnung eingepflegt werden.

```
<bean id="..." class="..."
        destroy-method="[methodenname]">
        <!-- collaborators and configuration for this bean go here -->
</bean>
```

Codeausschnitt 3: Bean-destroy

Annotatonsbasierte Konfiguration

Folgende Annotationen werden häufig bei der annotationsbasierten Konfiguration gebraucht:

Annotation in Spring	**JSR-Standard**	
@Required	@Inject statt @Required: @Optional	Gibt an, dass das Bean in der XML-Konfigurationsdatei angegeben bei Konfigurationszeit angegeben werden muss.
@Autowired	@Qualifier @Named	Verschaltet Beans automatisert und bietet mehrere Konfigurationsmöglichkeiten, mittels der Annotation @Qualifier.
@Component	@Named	Markiert eine Klasse als Komponent. Dies bedeutet, dass für das Auto-Detecting des IoC-Containers sichtbar wird.
@Scope(„singleton“)	@Singleton	Gibt den Scope eines Beans an. In dem Beispiel wird singleton gewählt, d. h. das das Bean als Singleton innerhalb eines IoC Containers instanziiert wird. Der in JSR-330 voreingestellte Scope ähnelt dem „prototype“-Scope. Ein JSR-330 Bean in einem Spring-Container ist standardmäßig ein singleton.

Scopes von Beans

Beans können in ihrer Deklaration zusätzlich durch *Scopes* ergänzt werden:

Scope	Description
singleton	Es wird nur eine einzige Instanz pro IoC Container erzeugt
prototype	Es können beliebig viele Instanzen erzeugt werden
Die nachfolgenden Scopes sind nur für einen webbasierten Spring Applikationskontext gültig	
request	Ein Bean wird einem http-Request zugeschrieben
session	Ein Bean wird einer http-Session zugeschrieben
global-session	Ein Bean wird einer globalen http-Session zugeschrieben

b. IoC Container in Spring

Der IoC Container stellt die Funktionsweise der in Spring integrierten Dependency Injection bereit. Die Basis dafür findet sich in den Packages *org.springframework.beans* und *org.springframework.context* wieder.

Vom Interface *BeanFactory* innerhalb des Packages *org.springframework.beans* leiten sich die wichtigsten Mechanismen zur Kontrolle jeglicher Art von Beans ab. Sie folgt dem Factory Pattern (vergleiche Anhang). Heutzutage ist die BeanFactory jedoch nur noch Teil des Spring Frameworks um Rückwärtskompatibilität zu gewährleisten und wird nur mit der Einbindung auf Drittanbieter Frameworks verwendet.

Die historische Natur der BeanFactory wird deutlich durch den *ApplicationContext,* ein Interface das Bean-Instanziierung und -Vernetzung erlaubt und weitere Features mit sich bringt, überschattet. Daher wird der ApplicationContext eher zur Verwendung in Enterprise Anwendungen empfohlen.

Neben der Funktionalität Beans zu instanziieren, bietet der ApplicationContext folgende zusätzliche Funktionen:

- Automatische *BeanPostProcessor* Registrierung
- Automatische *BeanFactoryPostProcessor* Registrierung
- Einfacherer *MessageSource* Zugriff
- *ApplicationEvent* Veröffentlichung

Wie unter dem Punkt **Beans in Spring** beschrieben, konsumiert der IoC Container Konfigurationsmetadaten in Form einer XML-Datei. In Spring 2.5 wurde eine annotationsbasierte Konfiguration implementiert, in Spring 3.0 eine Java-basierte.

Instanziierung von Containern

Das Instanziieren eines Spring IoC Containers kann durch das Konstruiieren eines ApplicationContexts erreicht werden.

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});
```

Die übergegebenen Argumente sind die Namen der XML-Bean-Definitionen die auf unterschiedliche Weise aufgelöst (z. B. über den Classpath).

DI über den IoC Container

Der IoC Container injiziert Abhängigkeiten auf zwei unterschiedliche Arten:

- **Konstruktorbasierte Injizierung:** Diese Erstellungsart wird programmatisch dadurch erreicht, dass eine Konstruktormethode mit allen in der Klasse existierenden Feldern aufgebaut wird. Diese wird dann vom IoC-Container aufgerufen.
- **Setter-Methoden-basierte Injizierung:** Diese Erstellungsart wird programmatisch dadurch erreicht, dass der IoC Container einen leeren Konstruktor, d. h. einen Konstrukor ohne Parameterübergabe, aufruft und dann die für alle in dieser Klasse enthaltenen Felder die Setter-Methoden aufruft und dadurch die Feldwerte setzt.

B. Aspect Oriented Programming (AOP)

Aspect Oriented Programming (kurz: AOP) ist ein Programmierungsparadigma, das generische Funktionen über mehrere Instanzen und Klassen hinweg bereitstellt. Es entstand aus dem Bedürfnis, dass komponentenübergreifende Services mehrfach verwendet worden sind. Bei einer Veränderung einer dieser Services mussten die Änderungen auch in den daraufzugreifenden Klassen geändert werden.
Es ist eine Ergänzung des objektorientierten Programmierens. In AOP sind folgende Begriffe besonders wichtig:

- **Aspects:** Eine Modularisierung von Problemstellungen die sich über mehrere Klassen hinweg verteilen
- **Join Point:** Ein Methodenaufruf oder ein Exceptionwurf innerhalb des Programmablaufs
- **Advice:** Eine Aktion die von einem Aspect an einem Join Point ausgeführt wird. Ein Advice wird in vielen Frameworks (wie in Spring) auch als Interceptor-Pattern (vergleiche Anhang) umgesetzt.
- **Pointcuts:** Pointcuts grenzen die Anzahl der Joinpoints für einen Aspekt ein

a. AOP in Spring

Das Konzept des AOP ist in Spring, entgegen anderer Frameworks, nicht komplett integriert sondern ist sehr nah an den IoC Container integriert um Unternehmensprobleme zu lösen.
Aspekte werden durch die normale Syntax der Beandefinition erstellt. Dies unterscheidet sich sehr von anderen AOP Implementationen. Zudem sind die Funktionalitäten beschränkt und teilweise schwer umzusetzen. Spring integriert aber sowohl das IoC und das AOP mit AspectJ (vergleiche Anhang). Die Hauptaspekte von Spring im AOP Kontext sind:

- Klassische Proxy-basierte (vergleiche Anhang) Spring-AOP
- auf @AspectJ-Annotationen basierende Aspekte
- Pure-POJO-Aspekte
- Injizierte AspectJ-Aspekte

Die ersten drei genannten Aspekte sind Proxy-basiert und somit nur auf das Abfangen von Methoden beschränkt.
Die oben aufgeführten Sitchwörter wurden im Spring Framework wie folgt umgesetzt:

- **Spring-Aspekte:** Werden mittels der Annotation *@Aspect* oberhalb der Klassendefinition deklariert. Dies ermöglicht die Implementierung von Advices.
- **Spring-Advices:** Stellen Methodenaufrufe innerhalb einer Standard-Java-Klasse dar. Sie werden mittels Annotationen wie *@Before* und *@After* vor einer Methodendefinition implementiert. Die

Annotationen enthalten entweder mittels einer Wildcard-auffindbaren Methodendefinition oder eine konkrete Definition der Methode.

- *@Before:* Bevor eine Methode aufgerufen wird, wird die annotierte Methode ausgeführt.
- *@After:* Nachdem eine Methode aufgerufen wird, wird die annotierte Methode ausgeführt.
- *@AfterReturning:* Nachdem eine Methode einen korrekten Return-Wert geliefert hat, wird die annotierte Methode ausgeführt. Hier können mittels einer zusätzlichen Abfangvariablendeklaration innerhalb der Annotation Parameter empfangen werden.

  Beispiel:

```
@AfterReturning(pointcut="*.dataAccessOperation()",
returning="retVal")
public void doAccessCheck(Object retval) {
..
}
```

Codeausschnitt 4 - Aspect über Annotation mit Parameter

- *@AfterThrowing:* Nachdem eine Methode eine Exception geworfen hat, wird die annotierte Methode ausgeführt. Die Exception kann wieder mittels einer Deklaration innerhalb der Annotation abgefangen werden.

- **Pointcuts:** Werden in einer XML-Datei geschrieben oder mittels AspectJ vor einer Methode eingefügt. Sie definieren Join Points feingranularer, indem sie weitere Definitionen implementieren.
  - Bestimmen wann ein Advice ausgeführt wird
  - Annotiert an einer Methode
  - Besteht aus zwei Teilen: **Signatur** und **Pointcut Expression**

    z. B.: @Pointcut(„execution(*transfer(..))“). Hier wird bei der Ausführung jeglicher Methode, die „transfer“ im Namen enthält, die annotierte Methode aufgerufen.
  - Hat verschiedene Beschreibungen: *execution, within, this, target, args, etc.*
- **Objekte:** Erhalten Advices zur Laufzeit

Auf Grund dessen, dass das Spring AOP mittels Proxies realisiert worden ist, können *protecte* Methode und *JDK proxies* nicht abgefangen werden. Um dies dennoch zu bewerkstelligen sollte auf AspectJ zurückgegriffen werden.

Zu AOP zählt des Weiteren eines der zentralen Prinzipien des Spring Frameworks: *non-invasiveness*. Dies bedeutet, dass man nicht dazu gezwungen wird Klassen des Frameworks in die Business bzw. Domain Models zu integrieren sondern lediglich die Optionen dazu hat.

AOP Container

Der AOP Container liefert alle Werkzeuge um Aspect-Oriented Programming umzusetzen und ist in folgende Module aufgeteilt:

spring-aop	Erlaubt es Code, der aufzuteilende Funktionalität enthält, zu entkoppeln und um Verhaltensweisen informativ in den Code einzubinden.
spring-aspects	Erlaubt es AspectJ zu integrieren.
spring-instrument	Liefert Unterstützung von Klasseninstrumentalisierung und Integration von Classloadern um sie auf bestimmten Applikationsservern zu benutzen.
spring-instrument-tomcat	Enthält die Instrumentalisierung eines Tomcat-Servers

C. Model-View-Controller-Pattern

Um das Verständnis für das praktische Beispiel aufzubauen wird im Folgenden das MVC Pattern innerhalb von Spring erklärt.

Das Model-View-Controller Pattern ist bei vielen webbasierten Frameworks vorhanden und beschreibt den Zusammenhang, dass über eine View (eine inhaltliche Repräsentation der Anwendung) Aktionen an einen Controller geleitet werden, diese dann vom Controller auf die Modelle umgeleitet wird. Rückläufig bedeutet dies auch, dass eine View Updates vom Controller enthält, die wiederum von den Models abgeleitet werden können.

Der Controller arbeitet so gesehen als Bindeglied zwischen View und Model und kontrolliert so das Geschehen innerhalb der Web-Anwendung.

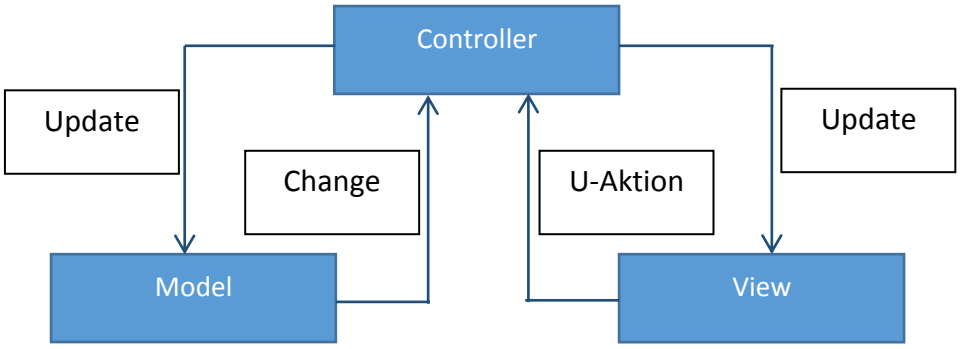


Abbildung 1: MVC Pattern

Im Spring Framework sind Controller schlichtweg POJOs (Plain Old Java Objects) und keine Servlets, was das Debugging erleichtern soll. Als Servlets beschreibt man Java-Klassen, die instanziiert die Kommunikation mit dem Client ermöglichen.

a. Dispatcher-Servlet

Das Dispatcher-Servlet dient in Spring dazu, http-Requests an Handler und Controller weiterzuleiten. Es wurde mit dem Front Controller Pattern (vergleiche Anhang) realisiert. Es leitet sich von der Klasse *FrameworkServlet* ab, welches die Grundzüge des Applikationskontextes in Spring, realisiert als JavaBean, schafft. Das *FrameworkServlet* wiederum leitet sich aus dem in der *javax.servlet* definierten HttpServlet ab.

b. Controller in Spring

Im nachfolgenden Abschnitt werden Controller an Hand eines praktischen Beispiels erläutert. Die Auswahl hierfür beschränkt sich lediglich auf den simplen Controller und einen RestController.

In Spring können Controller durch die diskriptive Annotationen *@Controller,* oder spezifischer *@RestController,* oberhalb der Klassendefinition definiert werden. Der Hauptunterschied der beiden liegt darin, dass eine Klasse, die mit *@Controller* annotiert worden ist, auf die View-Technologie basiert. Eine mit *@RestController* annotierte Klasse hingegen, liefert als Rückgabewert schlichtweg ein Objekt und ein Objekt wird auch direkt in den http-Response als JSON geschrieben.

```
@RestController
@EnableSwagger2
@Api("Handles alternative emails")
class EmailAlternativeController {
```

Codeausschnitt 5: Definition eines Controllers

```
@RequestMapping(value = "/v1/alternatives", method = RequestMethod.POST)
@ApiOperation(value = "Receives an alternative email")
public ResponseEntity<?> create(@RequestBody(required = true) Address sender,
                                @RequestBody(required = true) Body body,
                                @RequestBody(required = true) List<Address> recipients,
                                @RequestParam("files") List<MultipartFile> attachments) {
    def email = new EmailAlternative()
    email.setAttachments(createAttachmentsOutOfFiles(attachments))
    email.setRecipients(recipients)
    email.setSender(sender)
    email.setBody(body)
    emailDao.save(email)

    return ResponseEntity.ok(email)
}
```

Codeausschnitt 6 : Definiton einer Methode innerhalb eines REST-Controllers

In den Codeausschnitten finden sich weitere, für Spring typische Annotationen, die sich innerhalb eines Controllers befinden können:

- Die *@ResponseBody*-Annotation wird für Methoden verwendet und bewirkt, dass der Rückgabewert der Funktion direkt in den Body des http-Responses geschrieben wird.
- Die *@RequestMapping*-Annotation wird für Methoden verwendet und setzt den Pfad auf den der Controller horcht, d. h. sobald eine Nachricht auf den Pfad eingeht, wird sie vom Kontroller abgefangen. Das RequestMapping beschreibt ebenfalls das http-Verb (im obigen Beispiel „POST“).
- Die *@RequestParam*-Annotation wird innerhalb der Parameterklammer einer Methode verwendet und fängt Parameter ab, die mit der URL übertragen werden.

Die Methode „helloUser“ hingegen empfängt einen Parameter über die URL und gibt diesen als im String „hello [$name]“ im http-Response zurück.

c. Views in Spring

Views werden in Spring von dem Rest der Anwendung getrennt und können leicht konfiguriert werden, falls eine Anwendung von einem anderem Template wie Thymeleaf statt JSP Gebrauch machen soll. „Out of the box“ enthält Spring die Möglichkeit JSPs, Velocity Templates und XSLT Views zu implementieren. Im Folgenden wird eine Konfigurationsmöglichkeit hierfür beschrieben.

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.groovy();
    }

    @Bean
    public GroovyMarkupConfigurer groovyMarkupConfigurer() {
        GroovyMarkupConfigurer configurer = new GroovyMarkupConfigurer();
        configurer.setResourceLoaderPath("/WEB-INF/");
        return configurer;
    }
}
```

Codeausschnitt 7: Konfiguration GroovyMarkup

Durch die Integration dieser Konfigurationsklasse, wird ein GroovyMarkupTemplate Configurer als Bean initialisiert, sofern sich die Konfigurationsdateien im Ordner „/WEB-INF/“ befinden. Das hier im Beispiel verwendete Groovy-Markup-Template, bietet einige Schlüsselfunktionen wie **hierarchische Builder für XML-Content, Template-Einbindungen, Kompilierung von Templates zu Bytecode für eine schnellere Renderingzeit, usw..**

SpringBoot basiert auf Thymeleaf und bindet dies zugleich ein. Thymeleaf ist ein Template-Engine, welches darauf ausgelegt ist XHTML und HTML Views zu erzeugen. Es wird bei vielen Spring Projekten eingesetzt.

D. Andere Spring Module und Schwesterprojekte

Das Spring Framework besteht aus einigen, nach Funktionen klassifizierte und gruppierte Module. Diese sind von anderen entwickelt worden und im Nachhinein nach dem Prinzip „If you can't beat them join them" in Spring integriert worden:

- **Testing:** JUnit
- **Data Access/Integration:** EJB, JPA, Hibernate
- **Web:** JSF, Webservice, REST

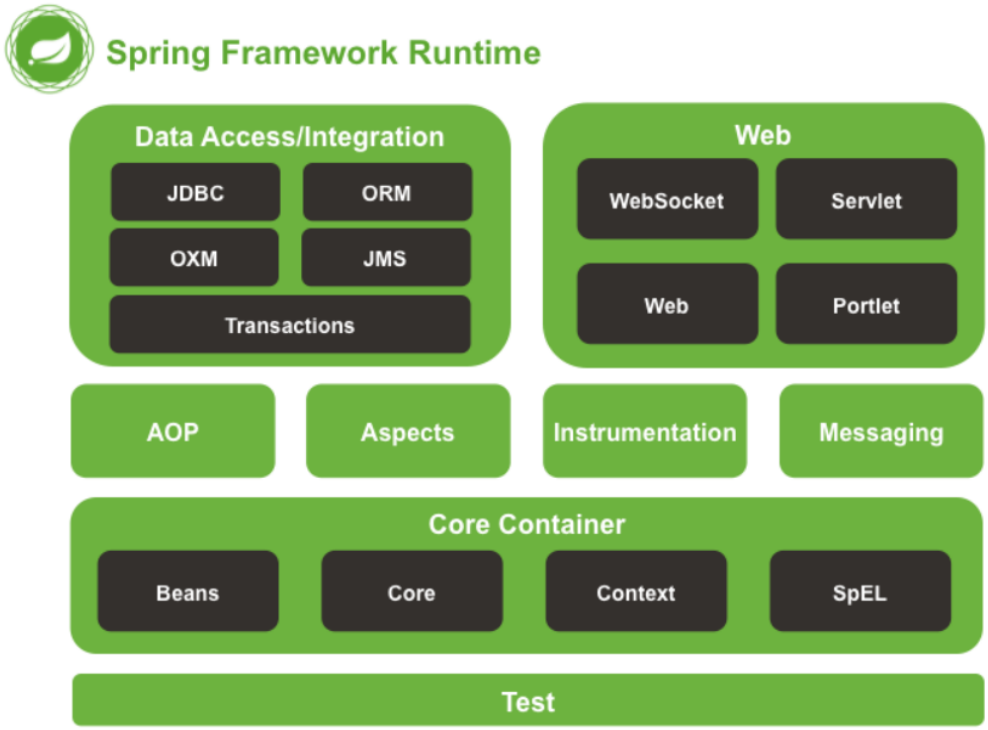


Abbildung 2: Spring Framework Module

Core Container

Der Core Container enthält alle Basiselemente auf die das Framework aufbaut und sich in folgende Module aufteilt:

spring-core	Bilden das Fundament des Frameworks und enthalten die Integration der Dependency Injection sowie die BeanFactory und ihre Ableitungen.
spring-beans	
spring-context	Abgeleitet aus dem Beans Modul, fügt der Kontext Internationalisierungs-Support, Eventverbreitung, Laden von Ressourcen und das transparente Konstruiieren von Contexts wie den Servlet container hinzu.
spring-context-support	Erlaubt es Bibliotheken Dritter in den Kontext der Springapplikation zu integrieren:

	Caching Mailing Scheduling Template Engines
spring-expression	Umfasst eine *Expression Language* um Objektgraphen während der Laufzeit anzufragen und zu manipulieren.

Data Access und Integration

Die Data Access/Integration Schicht besteht aus JDBC, ORM, OXM, JMS und Transaktionsmodulen und teilt sich wie folgt auf:

spring-jdbc	Stellt eine JDBC-Abstraktionsschicht bereit um wiederholten JDBC-bezogenen Code zu vermeiden
spring-orm	Umfasst Schichten für das Objekt-rationale-Mapping, die APIs wie JPA, JDO, Hibernate und iBatis enthalten.
spring-oxm	Ist eine Abstraktionsschicht für das Mapping von Objekten auf XML.
spring-jms	Enthält Funktionen zum Produzieren und Konsumieren von Messages
spring-transaction	Unterstützt die programmatischen und deklarativen Transaktionen für Klassen, die bestimmte Interfaces implementieren und für alle POJOs.

Spring JPA

Ein Subprojekt von Spring ist das Spring JPA (Java Persistence API). Es bietet Möglichkeiten zur einfacheren Integration von JPA in denen Repositorys dazu benützt werden automatisiert Querys zu erstellen und diese an die Datenbank weiterzuleiten. Die Querys sind nach einer Syntax aufgebaut, die einem SQL-Statement ähnelt (Structured Query Language).

```
public interface UserRepository extends Repository<User, Long> {
        List<User> findByFirstnameAndLastname(String firstname, String lastname);
}
```

Codeausschnitt 8: Repository

Das oben dargestellte Repository erweitert das Interface Repository um die Methode „findByFirstnameAndLastname“. Diese Methode ist in der Query-Syntax von Spring JPA aufgebaut und sucht in der Datenbank nach einem User mit den Feldern „firstname“ und „lastname“, was durch die Verkettung *FirstnameAndLastname* erreicht wird. Die übergebenen Parameter müssen der Feldnamen des Objekts entsprechen. Die Integration einer eigenen Search-Query wird durch diese Notation umgangen. Der Rückgabewert des Datentyps ist der einer Liste, die User enthält.

2. Arbeiten im Spring Framework

A. Einführung in das Projekt

Der von mir erstellte Webservice hat als Hauptfunktion Emails in JSON oder XML Format zu empfangen und diese nach einem Zeitplan abzuschicken. Die Integration des Webservices fand unter der Verwendung von der Distribution *Spring-Boot* statt. Er ist Teil einer größeren Microservice-Architektur (vergleiche Anhang: „Sam Newmans Microservices“) und enthält deshalb fast alle Schichten einer Applikation.

Das Projekt wurde in Spring Boot Version 2.4.0 umgesetzt. Der Build wird mittels Gradle vollzogen:

```
version '1.0-SNAPSHOT'

apply plugin: 'groovy'
apply plugin: 'java'
apply plugin: 'spring-boot'

sourceCompatibility = 1.8
targetCompatibility = 1.8

buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath('org.springframework.boot:spring-boot-gradle-plugin:1.3.3.RELEASE')
    }
}

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.11'
    testCompile group: 'junit', name: 'junit', version: '4.11'
    compile("org.springframework.boot:spring-boot-starter-web:1.3.5.RELEASE")
    compile("org.springframework.boot:spring-boot-starter-data-mongodb")
    compile('io.springfox:springfox-swagger2:2.4.0')
    compile('io.springfox:springfox-swagger-ui:2.4.0')
    // https://mvnrepository.com/artifact/javax.mail/mail
    compile group: 'javax.mail', name: 'mail', version: '1.4.1'
}
```

Codeausschnitt 9: gradle.build

Gradle ermöglicht es, wie Maven oder andere Build Tools, Dependencies leicht zu verwalten und vereinfacht es, das Projekt auf verschiedene Maschinen zu verteilen. Gradle ist Groovy basiert. Die *build.gradle*-Datei enthält ein Groovy-Skript, das alle nötigen Schritte zum Aufsetzen des Projekts beinhaltet (siehe Anhang).

Die Implementierung enthält keine Views. Business Layer und Persistenzschicht (oder auch Integrationsschicht) sind aber enthalten. Als Datenbank wurde MongoDB verwendet. Der Grund hierfür liegt in meiner persönlichen Neugier, eine NoSQL-Datenbank zu verwenden.

Desweiteren wurde für Testzwecke Swagger eingesetzt. Swagger ist ein Tool um aus einer REST-Schnittstelle automatisch eine View zu generieren. In dieser View werden Forms an Hand der übergegebenen Parameter innerhalb des REST-Controllers erstellt und weitere Deklarationen abgeleitet.

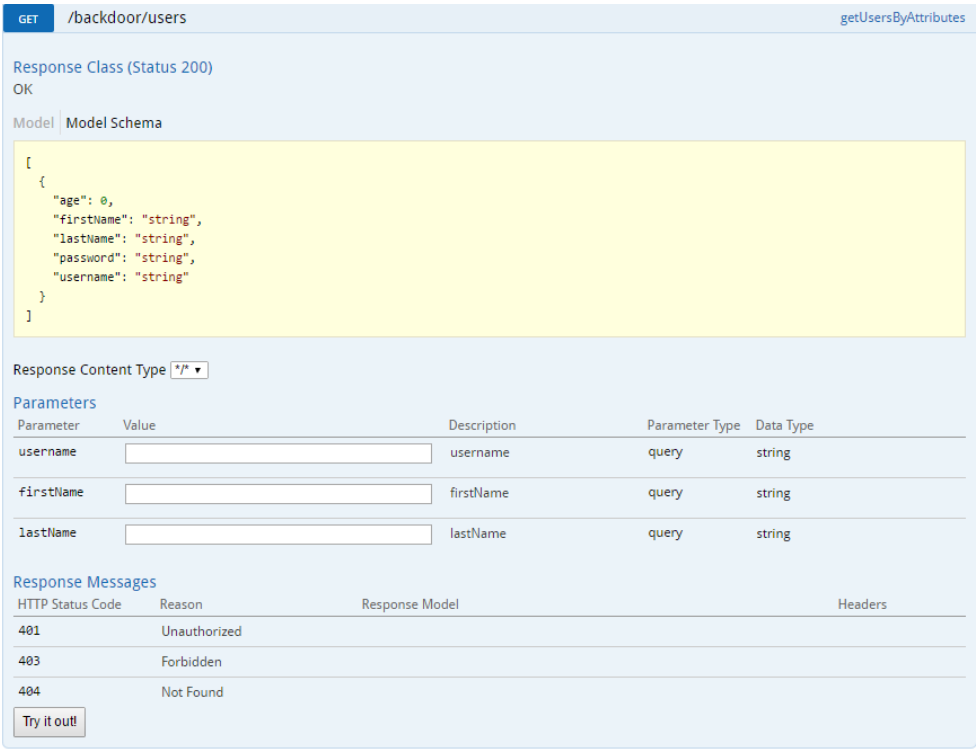
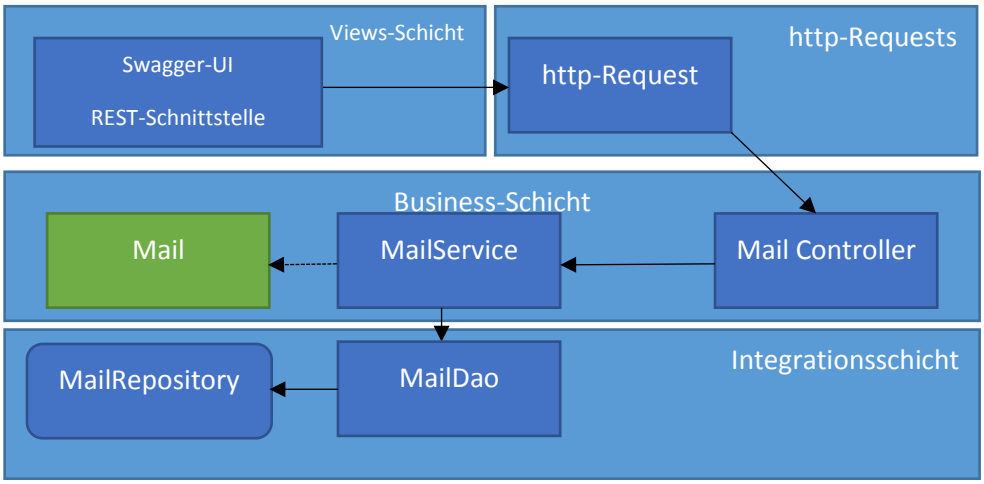


Abbildung 3 - REST-Schnittstelle in Swagger

Wie in der Abbildung zu erkennen ist, werden die Parameter *username, firstName* und *lastName* in der URL /backdoor/users entgegengenommen. Diese werden dann über das DispatcherServlet an den zuständigen Controller weitergeleitet und mit ihnen nach Einträgen innerhalb der Datenbank gesucht.

Schichtenmodell



Figur 1: Schichtenmodell

Aus dem Schichtenmodell lassen sich folgende Arbeitsschritte ableiten:

1. Es geht ein http-Request ein mit einem Body der einer E-Mail als JSON Format entspricht
2. Dieser Request wird über das DispatcherServlet an den MailController weitergeleitet
3. Der MailController erzeugt aus der JSON ein Objekt der Klasse Mail
4. Das Mail Objekt wird vom MailService entgegengenommen und entsprechend verändert
5. Die MailDao empfängt vom MailService das Objekt und schickt es an die MailRepository um das Objekt wiederum als JSON zu speichern

B. Integration

Der Projektbeginn bezieht sich ausschließlich auf die Einbettung in IntelliJ IDEA Community Edition 14.1.4.

Der Einstieg in eine auf dem Spring-Framework basierte Applikation ist das Erstellen eines Maven bzw. Gradle Projekts. Das Projekt wurde, wie erwähnt, in Gradle umgesetzt. Bei der Erstellung des Projekts kann direkt eine Auswahl zwischen Maven und Gradle getroffen werden und diese durch zusätzliche Optionen modifiziert werden.

Die Maven- bzw. Gradle Einbettungen finden sich unter http://projects.spring.io/spring-boot/. Für Maven-Benutzer muss folgende Deklaration in die *pom.xml*-Datei:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.5.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

Codeausschnitt 10: Maven Dependency

Nun da die Dependency zum Spring-Framework angegeben worden ist, kann die Projektentwicklung einer Standard-Applikation beginnen. Folgendes wurde zum Beginn des vorgestellten Projekts noch getan:

- Zur Anbindung an eine MongoDB muss MongoDB zunächst installiert werden. Dazu muss ein Paket von der Seite heruntergeladen und die darin enthaltene .exe-Datei ausgeführt werden. Das Installationsprogramm stellt alles so ein, wie es die Datenbank benötigt: Der Systempfad wird gesetzt und alle anderweitigen Komponenten werden installiert.
- Zur Integration von Swagger-UI muss Swagger als Dependency deklariert bzw. als Library heruntergeladen und eingetragen sein. Swagger ist ein Tool um automatisiert eine View-ähnliche Schnittstelle aus einer REST-Schnittstelle zu erstellen. Swagger-UI muss dann deklarativ in die Controller-Klassen eingebunden werden, was ausführlich in den kommenden Abschnitten erklärt wird. Swagger ist dazu dienlich um leicht http-Requests erstellen zu lassen und diese an den Controller zu schicken.

C. Controller

MailController

Der MailController ist wie folgt aufgebaut (Darstellung in Swagger 2.0):

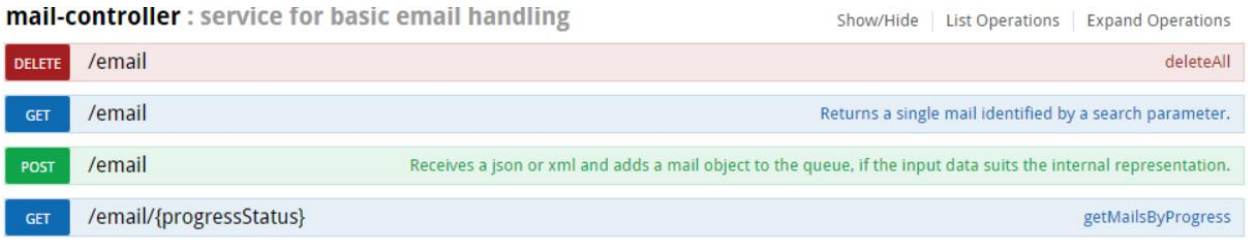


Abbildung 4: REST-Schnittstelle in Swagger 2.0

Verb	Pfad	Beschreibung
DELETE	/email	Für Testzwecke eingebunden: Löscht alle Emails innerhalb der Datenbank.
GET	/email	Gibt eine Email im Body der http-Antwort zurück, die eindeutig identifizierbar ist. Falls kein Suchkriterium angegeben ist, werden alle ausgegeben.
POST	/email	Empfängt eine Email im Body eines http-Requests in JSON- oder XML-Form und mapped diese Datei auf das entsprechende Model („Mail“). Das Mapping erfolgt entweder über den in Spring integriertem JSONMapper oder XMLMapper, welcher sich an der Struktur der

		Klasse des Objekts orientiert und daraus eine referenzielle JSON- oder XML-Struktur ableitet.
GET	/email/{progressStatus}	Gibt mehrere Emails zurück, die einem ProgressStatus entsprechen. Der ProgressStatus ist eine reine Datenbank-basierte Ergänzung und dient als Orientierung für den MailWorker.

Der MailController wird vom im Spring-Framework enthaltenen DispatcherServlet bedient. Alle einkommenden http-Requests werden nach Standard entgegengenommen.

D. Models

Mail

Alle Objekte der Klasse „Mail“ sind Domänenobjekte. Diese enthält alle relevanten Informationen für eine zu versendende Email. Sie wird als JSON vom MailController empfangen und als solche in die darunterliegende Datenbank aufgenommen. Die grundlegende Struktur der Mail-Objekte ist wie folgt:

- Eine Mail MUSS eine Adresse als Sender enthalten

  Adresse: Eine Adresse enthält einen String der Adresse, der nach Email-Konformität über ein Pattern-Matching angenommen wird, und einen Status, der den Status des Absendens der Email angibt.
- Eine Mail MUSS eine Adresse als Empfänger enthalten, kann aber auch mehrere Empfänger haben
- Eine Mail MUSS einen Titel haben
- Eine Mail bekommt einen MailStatus zugewiesen

  MailStatus: Der MailStatus ist eine Angabe über den gesamten Absendeprozess einer Mail, d. h. es enthält die Information ob eine Absender-Adresse nicht erreicht werden konnte.
- Eine Mail wird durch eine ID referenziert
- Eine Mail enthält einen Body

  Body: Ein Body enthält eine Spezifikation des Body-Typen im E-Mail-Kontext, d. h. es wird definiert ob es sich um den Typ HTML oder Plain-Text handelt. Der Inhalt wird als String gespeichert.

E. Services

MailService

Innerhalb der Applikation gibt es den „MailService“ der als Bindeglied zwischen Controller und MailDao fungiert. Der MailService ist ein Bean der mittels der Konfiguration @Autowired aus dem Controller heraus

instanziiert wird. Er empfängt Anfragen des Controllers, leitet sie an die *MailDao* weiter, erhält Datensätze über die MailDao und erweitert diese gegebenenfalls um Business-Aspekte.

F. Persistenz

MailDao

Die MailDao enthält die *MailRepository,* die mittels @Autowired als Bean definiert wird. Sie setzt die IDs der einzutragenen Objekte und führt Sortier- bzw. Gliederungsfunktionen auf Datensätze aus, die an den obenliegenden Service geleitet werden.

MailRepository

Die MailRepository stellt die Anbindung an die MongoDB bereit. Es ist baut auf das Spring Data JPA auf.

```
import ...

/**
 * Created by Michael on 04.05.2016.
 */
public interface MailRepository extends MongoRepository<Mail, String> {
    Mail findById(long id);
    List<Mail> findByMailStatusProgressStatus(ProgressStatus progressStatus);
}
```

Aus dem MailRepository werden Mails nach dem Status und dem Fortschritt zurückgegeben, genauso wie nach der ID. Man sieht hier, dass es leicht zu einer komplexen Wortkette kommen kann, wenn Felder in weitere Subfelder aufgeteilt werden müssen.

G. Start der Applikation

Die Main-Methode der Applikation ist wie folgt aufgebaut:

```
public class Application implements CommandLineRunner {


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

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .paths(PathSelectors.any())
                .build();
    }

    @Override
    public void run(String... args) throws Exception {

    }
}
```

Die Beandeklaration für Swagger wird mit @Bean annotiert. Hier werden alle Pfade der Applikation aufgelöst und zu der graphischen Oberfläche hinzugefügt.

Da der CommandLineRunner implementiert wird, wird die run Methode überschrieben. Warum der CommandLineRunner implementiert ist, ist nicht Teil dieser Ausarbeitung.

III. Vergleich von Spring und Java Enterprise Edition

Im AOP innerhalb von Spring finden sich lediglich Interceptoren in der Implementierung, die durch Annotationen oder einer XML-Definition eingebracht werden können. Hier stehen vor allem **Advices** und **Pointcuts** im Vordergrund. Erstere werden mittels den Pointcuts mit *bestimmten Methoden* verbunden. In Java EE können hingegen auch Dekoratoren angebracht werden, die vom gleichen Typ wie der Zieltyp sind. Sie werden über das Dependency Injection bereitgestellt.

Im Kontext der Dependency Injection gibt es wesentlich weniger Unterschiede. In beiden können die zu erstellenden Beans über XML oder Annotationen erzeugt werden, die resultierenden Beans können mittels Scopes und Qualifiern ferner definiert werden und die Injection wird mittels einem Container getätigt, welche Factory-Methoden hierfür bereitstellen. Die Unterschiede liegen hierbei vor allem in den Annotationen und dem Namespace.

Was für die Wahl eines geeigneten Frameworks jedoch eine Rolle spielen könnte, wäre die modulare Architektur von Spring die viele Möglichkeiten bietet Leightweight-Applikationen zu entwickeln. Auch das Arbeiten, das Debuggen im Spezifischen, vereinfacht sich durch das Werfen von vielen Exceptions während der Laufzeit.

Über die Zeit hinweg konnten Java Enterprise Edition und Spring ihre ursprünglichen Unterschiede meistenteils überbrücken. Vor allem in den respektiven Aspekten der Aspektorientierten Programmierung und der Dependency Injection kam es zu vielen Annäherungen. Über die Versionen hinweg konnten letztendlich die komplexen EJBs zu einfacheren, handlichen Beans reduziert werden, die vor allem durch die Einflüsse vom Spring Framework geprägt sind. Jedoch finden sich in der technischen Implementierung Unterschiede, die von Bedeutung sein könnten. Es bleibt aber dennoch Geschmackssache - und vielleicht auch eine organisatorische Entscheidung - welches Framework zur Verwendung kommen sollte.

IV. Anhang

Glossar

JPA	Java Persistence API
JSR	Java Specification Request
JDBC	Java Database Connectivity
AspectJ	Aspektorientierte Erweiterung für Java
SpringBoot	Leichtgewichtige Distribution des Spring Frameworks für Webservices

Design Patterns

Bezeichnung	Beschreibung
Singleton	Das Singleton-Pattern stellt sicher, dass von einer Klasse nur ein Objekt erzeugt wird. In Java kann dies erreicht werden indem sich eine statische create() (oder auch „getInstance()“) Methode sich innerhalb der Klasse befindet, der den Konstruktor (meist private) aufruft um ein Objekt der Klasse zu erzeugen. Dieses erzeugte Objekt wird einer statischen Variable zugeschrieben, die genau ein instanziiertes Objekt enthalten kann.
Factory	Das Factory-Pattern liefert ein Interface um Familien von zusammengehörige und voneinander abhängige Objekte zu erzeugen ohne die Klassen konkret zu spezifizieren.
Front Controller	Ein Front Controller dient als Einstiegspunkt in eine Webapplikation. Er leitet Anfragen an bestimmte andere Parteien innerhalb des Systems weiter.
Interceptor	Das Interceptor Pattern ist ein Verhaltensmuster, welches ein Framework erweitert ohne es selbst zu verändern. Methodenaufrufe können mittels eines Interceptors abgefangen werden (ähnlich Proxy).
Proxy	Das Proxy Pattern beschreibt die Steuerung eines Objekts auf ein vorgelagerts Stellvertreterobjekt.

V. Quellenverzeichnis

Generell

- „Design Patterns - Elements of Reusable Object-Oriented Software“,
  Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- „Spring im Einsatz“,
  Craig Walls
- https://spring.io/docs
  Zugriff am 06.06.2016, 07.06.2016, 09.06.2016

Dependency Injection

- http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html,
  Zugriff am 07.06.2016
- http://www.tutorialspoint.com/spring/spring_dependency_injection.htm,
  Zugriff am 09.06.2016

AOP

- http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html,
  Zugriff am 06.06.2016