Java Anwendungen mit dem Spring Framework

Informatica Feminale 2015

Christine Koppelt

Bremen, 24.8 - 26.8. 2015

Über mich

- Diplom Mathematikerin (FH)
- Senior Consultant bei der innoQ Deutschland GmbH
- Java seit 2002, Spring seit 2008
- Berufserfahrung seit 2007

Über euch

- Wie heißt ihr und woher kommt ihr?
- Was ist euer Hintergrund (Studium, Studienfach, Job, ...)?
- Was hab ihr bisher mit Java gemacht?
- Wie sind eure Vorkenntnisse bezüglich Maven und Git?
- Welche Erwartungen habt ihr an den Kurs?

Ablauf

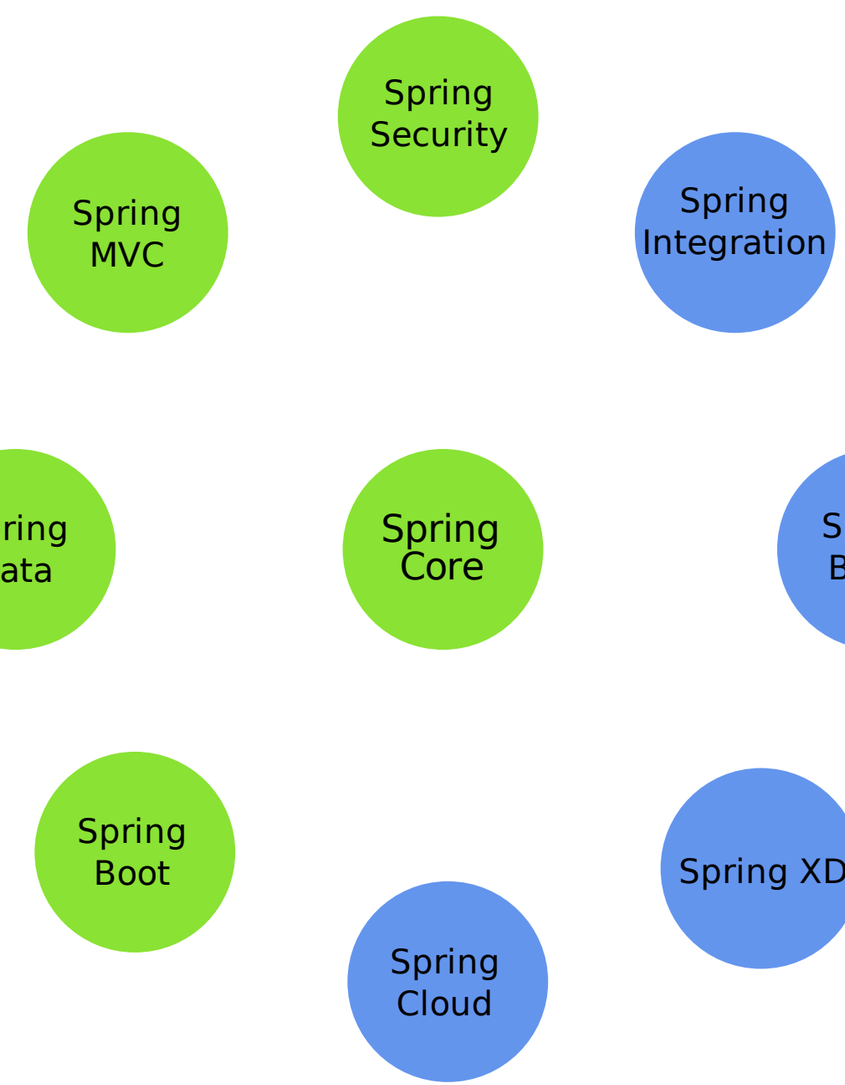
- Montag
  - Spring Grundlagen
  - Setup Beispielanwendung
- Dienstag
  - REST-Services
  - Datenbankzugriff
  - Security
- Mittwoch
  - weiterführende Themen

Spring

Was ist Spring

- Open-Source Framework für die Erstellung von Java Anwendung
- Basiert auf Dependency Injection und Inversion of Control
- Entstand ursprünglich als leichtgewichtige Alternative zum offiziellen J2EE Stack
- Viele Ideen aus Spring sind im Laufe der Zeit in J2EE/JEE eingeflossen
- Besteht aus einem Core Modul und zahlreichen Zusatzmodulen
- Innovative Themen werden in Spring immer noch schneller umgesetzt als in JEE (NoSQL, Cloud, Social, Mobile, etc)

Überblick Module



Core Container

- Wird immer benötigt
- Dependency Injection
- Validierung von Werten
- AOP

Spring MVC

- Webframework für die Erstellung von REST APIs oder Webseiten
- Unterstützt
  - JSON und XML APIs
  - verschiedener Templatesprachen und Webframeworks
  - Exception Handling
  - Localization

Spring Data

- Stellt APIs für den Zugriff auf relationale und nicht-relationale Datebanken bereit
  - Redis, ElasticSearch, MongoDB
- Vereinfacht die Java Persistence API (JPA) für relationale Datenbanken

Spring Security

- Authentifizierung
  - Formularbasiert, LDAP, HTTP Basic
- Autorisierung
  - von Webanfragen
  - von Methodenaufrufen

Spring Boot

- Ermöglicht schnellen Start, Vereinfacht die Verwendung von Spring
- Standard Konfigurationen für häufig genutzte Komponenten
  - Spring Boot Starters
- Ermöglicht es die Anwendung mit einem eingebetteten Applicationserver in ein ausführbares JAR zu verpacken

Gerüst einer Spring Boot Anwendung

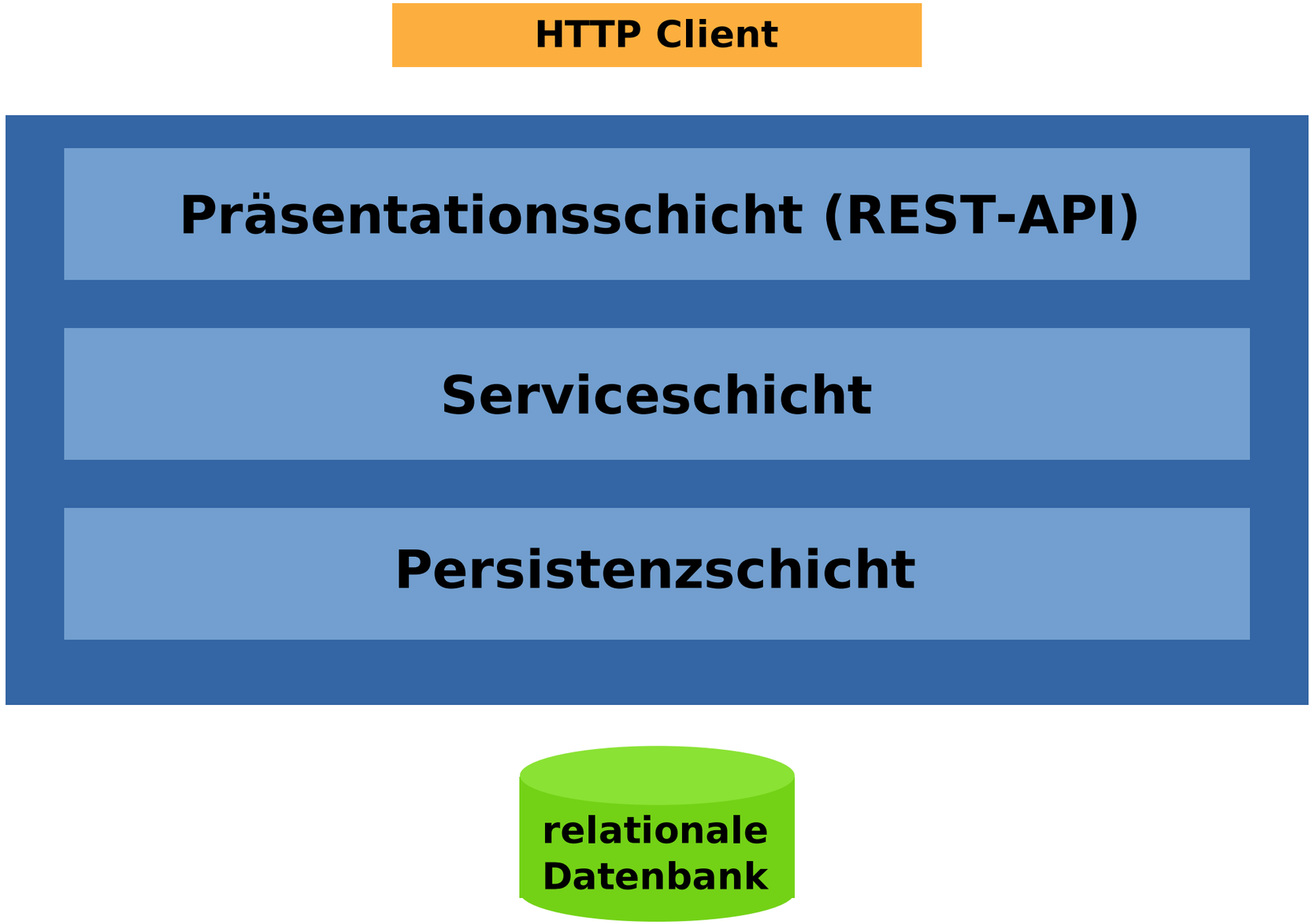
```
@EnableAutoConfiguration
public class App {

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

Übungsanwendung

- Verwaltung von Veranstaltungen oder eigene Anwendung
- Basiert auf Spring Boot
- Wird im Laufe des Kurses fortlaufend erweitert
- REST-API, keine GUI
- Unit-, Integrations- und funktionale Tests
- Zugriff auf relationale Datenbank
- Zugriffskontrolle

Drei-Schichten-Architektur



Software für die Übungsanwendung

- Git
- Java 8
- Maven 3
- Eclipse
- PostgreSQL

Übung 1

- Erstelle in Eclipse ein neues Maven Projekt
  - `File -> New -> Other -> Maven -> Maven Project`
  - Wähle dabei die Option `Create a simple project (skip archetype selection)`
- Ergänze die `pom.xml` wie auf der folgenden Seite beschrieben http://projects.spring.io/spring-boot/ und erstelle die dort aufgeführte Klasse `SampleController`
- Starte die Anwendung:
  - `SampleController.java -> Run As -> Java Application`
  - oder alternativ mit Maven `mvn spring-boot:run`
- Rufe die URL `http://localhost:8080` auf. Wenn alles korrekt funktioniert, erscheint ein `Hallo Welt!`
- Erzeuge ein GitHub-Repository und pushe die Anwendung

Dependency Injection

Worum geht es

- Klassen in einer Java Anwendung verwenden weitere Klassen als Komponenten
- Werden diese Klassen manuell instanziiert ist das oft umständlich, verringert die Flexibilität und erschwert die Testbarkeit
- Dependency Injection ist ein Design Pattern um dieses Problem zu adressieren

Allgemeines Beispiel Dependency Injection

Instanz der Klasse erzeugt Komponente selbst

```
public class ClientA {
    private MyService service;

    public ClientA() {
        this.service = new MyService();
    }

    public void send() {
        service.sendRequest();
    }
}
```

Instanz der Klasse bekommt Komponente injected

```
public class ClientB {
    private MyService service;

    public ClientB(MyService service) {
        this.service = service;
    }

    public void send() {
        service.sendRequest();
    }
}
```

Dependency Injection (DI)

- Design Pattern um Abhängigkeiten zwischen Objekten zur Laufzeit (anstatt zur Kompilierzeit) festzulegen
- Instanzen von Klassen erzeugen ihre abhängigen Komponenten nichts selbst, sondern bekommen sie von außen injected
- Klasse muss nur wissen was sie braucht, nicht wie sie es bekommt
- Ein DI Container übernimmt die Erzeugung und Verdrahtung der Komponenten
  - Garantiert, dass alle ahängigen Komponenten erzeugt und gesetzt werden
- Spring implementiert einen solchen DI Container
- Mechanismus wird auch in zahlreichen anderen Frameworks verwendet

Vorteile von Dependency Injection

- bessere Testbarkeit (Mocking)
- Programmierung gegen Interfaces, die konkrete Implementierung wird erst zur Laufzeit injected
- Klarerer Code, weniger Boilerplate
- Klasse muss nicht alle Details zu seinen abhängigen Komponenten wissen
- Abhängigkeiten können zwischen Objekten geteilt werden (Singleton)

Dependency Injection in Spring

- Funktioniert über Annotationen
  - Veraltet: XML
- Klassen können mit Hilfe spezieller Annotationen als Bean gekennzeichnet werden
- Beans können per Annotation in andere Klassen injected werden

Beispiel für Dependency Injection in Spring

Definition der Bean

```
@Service
public class MyService {

    public void sendRequest() {
        ...
    }
}
```

Injection der Bean

```
public class ClientB {

    @Autowired
    private MyService service;

    public void send() {
        service.sendRequest();
    }
}
```

Beans

- Werden vom Spring Container beim Start erzeugt
  - Default: Singletons
- Beans können selbst wieder Beans enthalten
- Default Annotationen um Beans zu erstellen
  - @Component: Generische Annotation
  - @Repository: Bean zum Zugriff auf externe Daten; Teil der Persistenschicht
  - @Service: Service als Teil der Serviceschicht
  - @Controller: Controller in der Präsentationsschicht
- @ComponentScan bewirkt, dass der Klassenpfad nach in Frage kommenden Klassen durchsucht wird

Injection von Beans

- Mittels der Annotationen `@Autowired` oder `@Inject`
- Constructor Injection vs Property Injection

Beispiel für verschiedene Injection Typen

Property Injection

```
@Autowired
private MyService service;
```

Constructor Injection

```
private MyService service;

@Autowired
public ClientB(MyService service) {
   this.service = service;
}
```

Vor- und Nachteile

- Constructor Injection
  - Änderungen der Abhängigkeiten schnell für Tests erkennbar
  - Sehr viele Argumente bei sehr vielen Abhängigkeiten
  - Final Felder möglich, d.h. immutable Objekte
  - Zu bevorzugen
- Property Injection
  - Für optionale Dependencies
  - Notwendig bei zirkulären Abhängigkeiten

Bean Lifecycle

- Mit `@PostConstruct` annotierte Methode wird aufgerufen, nachdem Abhängigkeiten injected wurden
- Mit `@PreDestroy` bevor das Bean gelöscht wird.
- Wenn `CommonAnnotationBeanPostProcessor` aktiv ist (wenn componenent scan genutzt wird)

Spring Container

- Der Spring Container sorgt für die Erzeugung, Verdrahtung, und Löschung von Objekten (Beans)
- Praktisch synonym zum Begriff ApplicationContext
- Muss normalerweise nicht manuell erzeugt werden
- Verschiedene Implementierungen:
  - Standalone `ClassPathXmlApplicationContext` und `FileSystemXmlApplicationContext`
  - Web: `AnnotationConfigWebApplicationContext`
  - Spring Boot benutzt z.B. den `TomcatEmbeddedServletContainerFactory`

Application Context & Spring Boot

```
SpringApplication.run(App.class, args);
```

- Startet, wenn nichts anderes konfiguriert, einen Tomcat Server
- Erstellt einen ApplicationContext und deployed ihn
- Lädt alle Singleton Beans

Übung 2

- Refaktoriere die Klasse `SampleController` aus der vorherigen Übung in dem Du sie in zwei Klassen aufteilst. Die Klasse `informatica.EventApp` enthält die main-Methode, die Klasse `informatica.controller.EventController` die Methode home. Ergänze in der Klasse `informatica.EventApp` noch die Annotation `@ComponentScan`, diese sorgt dafür, das der gesamte Klassenpfad nach Spring Beans durchsucht wird. Starte anschließend die Anwendung um zu prüfen ob noch alles funktioniert.
- Erstelle eine Klasse `informatica.service.EventService` und implementiere eine Methode `getAllFutureEvents` die eine Liste von Objekten vom Typ `informatica.domain.Event` zurückggibt. Annotiere diese Klasse mit `@Service`. `informatica.domain.Event` soll ein Attribut `name` vom Typ `String` enthalten.
- Injecte den Service in die Controller Klasse
- Implementiere eine neue Methode `getAllEvents` die eine Liste von Events zurückgibt und rufe dort die Methode `getAllFutureEvents` auf. Annotiere sie mit `@RequestMapping("/events")`. Starte die Anwendung unf rufe `/events/` im Browser auf.

Konfiguration

Worum geht es

- Konfigurationseinstellungen werden benötigt um die verwendeten Spring Features und umgebungsspezifische Parameter festzulegen
  - Festlegung zur Laufzeit
  - Festlegung zur Compile-Zeit
- Spring Boot
  - unterstützt eine XML-basierte (legacy) Konfiguration und eine Java-basierte Konfiguration
  - unterstützt Java Properties und YAML für Parameter

@EnableAutoConfiguration

```
@EnableAutoConfiguration
public class SampleController
```

- Spring Boot ermittelt aus den vorhandenen JARs (Logging Framework, Webcontainer, Datenbanktreiber) die "gewünschte" Konfiguration
  - spring-boot-starter-web: Spring MVC, Tomcat, Logback, Jackson, YAML-basierte Konfiguration, etc.
- Guter Start, kann später überschrieben werden
- 3 Konfigurationsmöglichkeiten
  - Property Files
  - Configuration Beans
  - Annotationen, die spezifische Features aktivieren/deaktivieren

Property Files

- Für Einstellungen die zur Laufzeit geändert werden sollen
  - Umgebungsabhängige Einstellungen z.B. Datenbankverbindung in Entwicklung, Test, Produktion
- Ursprünglich wurde das Properties Format genutzt
  - `src/main/resources/application.properties`
- Spring Boot unterstützt auch YAML Format
  - `src/main/resources/application.yml`
- Einbinden in den Code:
  - `@Value("${my.property}")` für einzelne Properties
  - `@ConfigurationProperties` typsicher

Vordefinierte Spring Boot Properties

- Spring Boot bietet eine vordefinierte Liste an Properties für die enthaltenen Komponenten
  - Logging, Application Server, Start Banner, JSON Serialisierung etc.
- Eine komplette Liste findet sich unter
  - http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#common-application-properties

@Configuration

- Kennzeichnet eine Konfigurations-Bean
- Nicht Spring Boot spezifisch
- Kann verwendet werden um Default-Konfiguration von Beans anzupassen
  - Dafür werden vorhandene Beans überschrieben
- Kann auch verwendet werden um Beans für Klassen aus externen Bibliotheken zu erzeugen die an mehreren Stellen der Anwendung verwendet werden sollen
- `@Bean` annotierte Methoden innerhalb einer `@Configuration` Klasse erzeugen eine Bean, die dann vom Container gemanangt wird

Beispiel: Vorhandene Bean RestTemplate überschreiben

```
@Configuration
public class RESTConfig {

    @Bean
    RestTemplate restTemplate(){
        RestTemplate restTemplate = new RestTemplate(clientHTTPRequestFactory()):
        restTemplate.setErrorHandler(new LoggingResponseErrorHandler());
        return restTemplate;
    }
}
```

```
@Autowired
private RestTemplate restTemplate;
```

Beispiel: Bean für Klasse aus externer Bibliothek erzeugen

```
@Configuration
public class AWSConfig {

  @Bean
  public AmazonSimpleEmailServiceAsyncClient awsSESClient() {
     AmazonSimpleEmailServiceAsyncClient emailClient = new AmazonSimpleEmailServiceAsyncClient();
     Region REGION = Region.getRegion(Regions.fromName("eu-west-1"));
     emailClient.setRegion(REGION);
     return emailClient;
  }
}
```

```
@Autowired
private AmazonSimpleEmailServiceAsyncClient emailClient;
```

Logging

Logging

- Via spring-boot-starter bzw. spring-boot-starter-looging sind die Frameworks SLF4j und Logback eingebunden
- Vorkonfiguiert: Logging auf der Konsole, Loglevel INFO
- Eigene Logback Konfiguration via `src/main/resources/logback.xml`
- Erzeugen eines Loggers

```
private final Logger LOG = LoggerFactory.getLogger(MyService.class);
...
LOG.debug(...);
```

Beispiel für eine Logback Konfiguration

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/base.xml"/>
    <logger name="org.mypackage" level="ERROR"/>
</configuration>
```

Übung 3

- Ergänze in der Klasse `EventService` ein debug-Logstatement, das bei jedem Aufruf der Methode die Anzahl der Events loggt.
- Starte anschließend die Anwendung und prüfe ob das Statement gelogged wird
- Ergänze eine Konfigurationsdatei für Logback. Setze das Loglevel für das package `org.informatica` auf `DEBUG`. Starte die Anwendung erneut. Was stellst Du fest?
- Ergänze eine YAML-basierte Properties Datei und setze das vordefinierte `spring.main.show_banner` auf den Wert `false`.
- Ergänze dort das Property `email.admin`, injecte das Property in den EventController und gib in der Methode `home` den Wert des Properties zurück.