

# Agenda

- Teil 1:
  - Einleitung
  - Dependency Injection
  - Spring Boot
  - Externe Konfiguration
- Teil 2
  - MVC
  - RESTful Services
  - DevOps Aspekte
- Teil 3
  - Spring Data
  - Spring Reactive
  - Spring Security

# Einführung in Spring und Spring Boot

1. Das Spring Framework
2. Spring Boot

# Die Geschichte von Spring (1)

- 2002: Rod Johnson veröffentlicht einen Prototypen zusammen mit dem Buch "Expert One-on-One J2EE Design and Development"
- Juni 2003: Erster Release unter der Apache license version 2.0
- 2004: Release 1.0 des Spring Framework
- 2006: Spring 2.0 vereinfacht die XML Konfigurationsdateien

## Die Geschichte von Spring (2)

- 2007: Spring 2.5 mit annotationsbasierter Konfiguration
- 2012: Spring 3.2 mit Java Konfiguration, Unterstützung für Java 7, Hibernate 4, Servlet 3.0
- 2014: Spring 4.0 unterstützt Java 8
- 2014: Spring Boot wird eingeführt
- 2020: Erste Alpha-Releases von Spring Native for GraalVM

# Überblick

- Das Spring Framework stellt im Java Kontext eine Infrastruktur bereit, die die Entwicklung von Java Anwendungen umfassend unterstützt
- Erlaubt es, sich ganz auf die Domäne zu fokussieren
- Ermöglicht es, Anwendungen auf Basis von "plain old Java objects" (POJOs) zu erstellen
- <https://spring.io>

## Beispiele für Vorteile für den Entwickler

- Abstrahiert von der Objekterzeugung durch Dependency Injection
- Abstrahiert vom (Datenbank) Transaktionsmanagement
- Abstrahiert Web-APIs
- Abstrahiert von JMX, JMS, JPA und viel mehr

# Spring Projekte

- Wir betrachten vor allem das "Spring Framework" und "Spring Boot"
- Es gibt aber eine Vielzahl weiterer:
  - Spring Cloud
  - Spring Data
  - Spring Security
  - Spring Integration
  - ....

# Spring Einsatzszenarien

- Von kleinen Kommandozeilenanwendungen zu wirklich riesigen Enterpriseanwendungen
- Man kann mit Spring eine Menge machen, aber Spring wird auch schnell sehr komplex, wenn man es falsch macht
- Typischer Einsatz: Spring-Anwendung, die als Webanwendung in einem Servlet-Container läuft

# Typischer Vollausbau einer Spring Webanwendung

# Einführung in Spring und Spring Boot

1. Das Spring Framework
2. **Spring Boot**

# Motivation

- Für die Ausführung einer klassischen Spring-Anwendung benötigt man einen Servlet Container
- Dazu packt man die Spring-Anwendung als WAR-Archiv und deployt das dann in einen Servlet Container (Tomcat)
- Dann muss man die Anwendung innerhalb des Servlet Containers konfigurieren
- Diese Einrichtung (Servlet Container und Spring Konfiguration) muss für jeden Micro-Service wiederholt werden

Vielleicht ist es ja möglich, den Container und die Spring-Anwendung in einem Framework zu vereinen?

# Die Geschichte von Spring Boot (1)

- Oktober 2012: Mike Youngstrom erstellt einen Feature-Request im Spring JIRA, der containerlose Webanwendungen mit dem Spring Framework ermöglicht. Er sprach dabei davon, die Konfiguration des Web-Containers mit einer Spring-Konfiguration aus der `main`-Methode vorzunehmen!

# Die Geschichte von Spring Boot (2)

Aus <https://jira.spring.io/browse/SPR-9888>:

*"I think that Spring's web application architecture can be significantly simplified if it were to provide tools and a reference architecture that leveraged the Spring component and configuration model from top to bottom. Embedding and unifying the configuration of those common web container services within a Spring Container bootstrapped from a simple main() method."*

# Spring Boot Features

- Ein lauffähiges JAR mit Servlet Container und Webanwendung
- Vordefiniertes Dependency Management
- Bietet 'Starter' Lösungen, die die Konfiguration von Dependencies erleichtern
- Konfiguriert so viel wie möglich automatisch
- Liefert produktionsrelevante Funktionen wie Metriken, Gesundheitschecks und externe Konfiguration out-of-the-box
- Keine Codegenerierung und keine XML Konfiguration notwendig

# Spring Boot Dependency Management

- Ein Problem traditioneller Spring-Anwendungen: Das Management von Spring- und Thirdparty-Dependencies
- Eine typische Enterprise-Anwendung mit einem großen Tech-Stack hat unzählige Dependencies
- 'Starter' Artefakte binden typische Thirdparty-Bibliotheken ein
- für Build und Dependency Management können Maven oder Gradle genutzt werden

# Einsatz von Spring Boot als 'Parent' in Maven

- Benutze Spring Boot Starter als Maven Parent
- Liefert ein vordefiniertes Dependency Management
- Liefert ein vordefiniertes Plugin Management
- Ist der bevorzugte Weg zu starten

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.6.6</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

# Einsatz von Spring Boot über das Dependency Management in Maven

- Benutze Spring Boot Dependency als Maven Dependency Import
- Liefert ein vordefiniertes Dependency Management
- Liefert ***kein*** vordefiniertes Plugin Management

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <!-- Import dependency management from Spring Boot -->
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>2.6.6</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

# Einsatz von Spring Boot in Gradle

- Benutze Spring Dependency Management Plugin
- Liefert ein vordefiniertes Dependency Management
- Liefert ein vordefiniertes Plugin Management

```
plugins {  
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'  
}
```

# Starter Module (1)

- Benutze 'Starter' Module, um Features einfach hinzuzufügen (insgesamt > 50)
- Offizielle 'Starter' Module heißen: `spring-boot-starter-*`
- Übersicht: <http://docs.spring.io/spring-boot/docs/2.6.6/reference/htmlsingle/#using-boot-starter>
- `spring-boot-starter-data-elasticsearch`
- `spring-boot-starter-data-jpa`
- `spring-boot-starter-mail`
- `spring-boot-starter-security`
- `spring-boot-starter-thymeleaf`
- `spring-boot-starter-web-services`
- ...

# Starter Module (2)

Maven

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

Gradle

```
dependencies {
  implementation 'org.springframework.boot:spring-boot-starter'
  implementation 'org.springframework.boot:spring-boot-starter-web'
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

# Lauffähiges Spring Boot JAR

- Benutze das Spring Boot Plugin zum Erstellen des Bundles
- Erzeugt das lauffähige JAR und fasst alle benötigten Dependencies zusammen
- Einfach das Plugin zur Build-Konfiguration hinzufügen, es ist keine weitere Konfiguration notwendig

## Maven

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

## Gradle

```
plugins {
    id 'org.springframework.boot' version '2.6.6'
}
```

# Übung: Erste Schritte mit Spring Boot

1. Erzeuge ein Projekt, indem Du entweder

- <https://start.spring.io> aufrufst und eine Spring Boot Anwendung konfigurierst und runterlädst,
- eine kurze Einführung zum Aufsetzen einer Spring Boot Anwendung unter <https://spring.io/guides/gs/spring-boot/> anschau, oder
- in IntelliJ unter Datei->Neues Projekt->Spring Initializr ein neues Project anlegst
- in der Spring Tool Suite (STS) IDE, ein “New Spring Starter Project” anlegst

2. Mit der IDE:

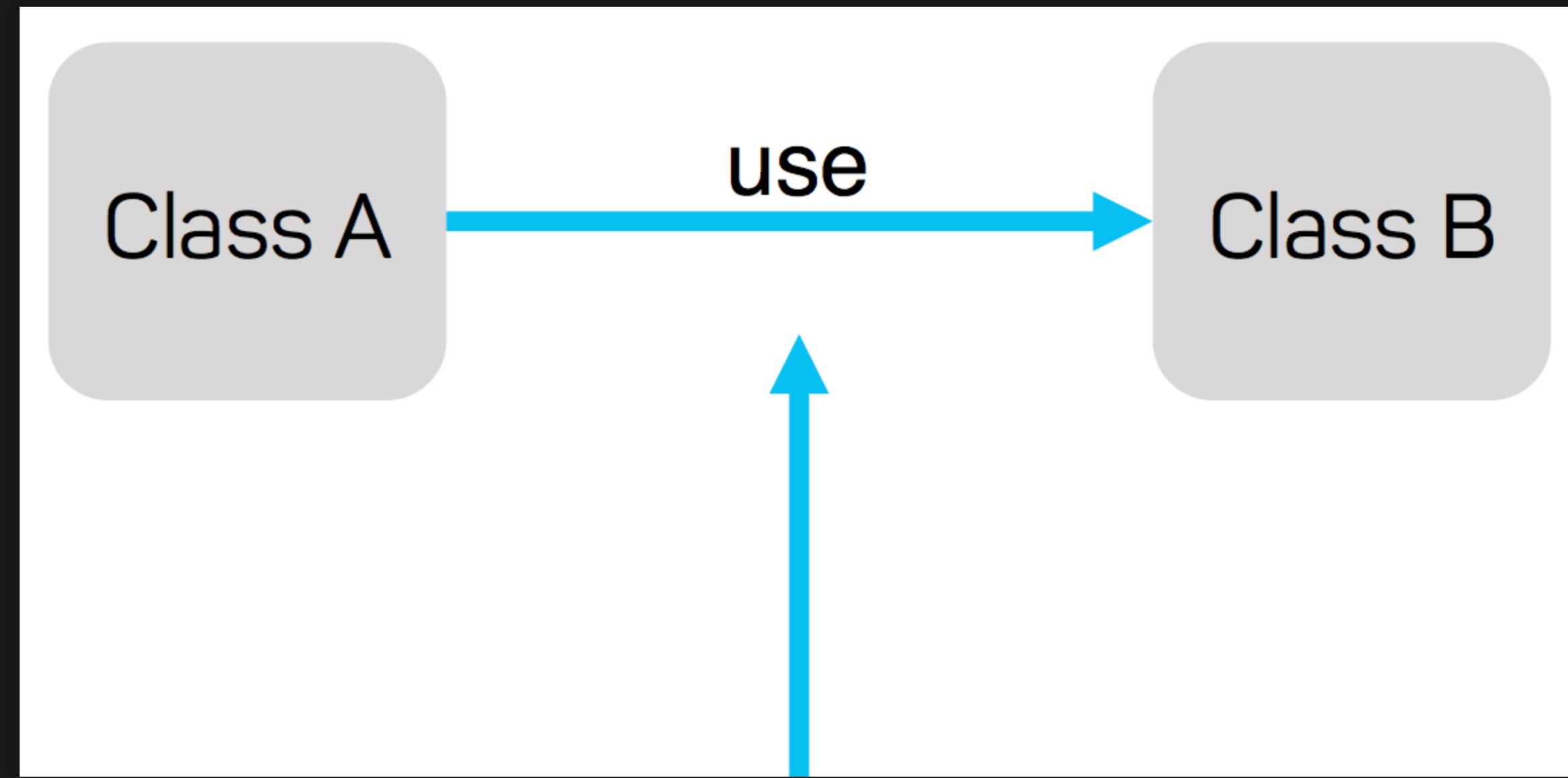
- Importieren, Compilieren und Starten der App

3. Auf der Kommandozeile:

- Compilieren und zusammenpacken: `./mvnw package` oder `./gradlew build`
- Starten: `java -jar target/hello-world-0.0.1-SNAPSHOT.jar`
- Alternativ: `./mvnw spring-boot:run` oder `./gradlew bootRun`

# Dependency Injection (DI)

1. **Wieso Dependency Injection?**
2. Spring und Dependency Injection
3. Spring Konfiguration



# Problem

- Objekterzeugung und -abhängigkeiten können sehr komplex sein
- Ziel: Trenne Objekterzeugung und -abhängigkeiten von der Domänenlogik, um den Fokus auf die Domäne zu behalten
  - ⇒ Verlagere Objekterzeugung und -abhängigkeiten aus den Domänenklassen in eine spezielle Komponente (üblicherweise ein Framework)
- Framework benutzt Dependency Injection (DI) zur Unterstützung der Domänenklassen
- Alternative: Domänenklassen suchen die abhängige Komponente

# Beispiel für DI

```
UserService service = new UserService(new UserRepository())
```

- Dependencies werden außerhalb der Klasse erzeugt
- Dependencies werden der Klasse übergeben
- Der `UserService` muss nicht wissen, wie das `UserRepository` erzeugt wurde
- `UserService` sollte nicht für die Erzeugung des `UserRepository` verantwortlich sein, hält aber natürlich eine Referenz darauf
- Außerdem: Dependencies sind privat und sollten nicht geteilt werden; *kein* `getUserRepository()` im `UserService`

# Abstraktion

- Code sollte nicht von konkreten Klassen abhängen, sondern von bereitgestellter Funktionalität
- Das heißt: benutzende Klassen sollten ein **interface** erwarten, und eine konkrete Implementierung wird injiziert
- Erlaubt die flexible Zusammenstellung von Verhalten
- Wird als "Best Practise" betrachtet

# Inversion of Control (IoC)

- Ohne IoC: Die Anwendung benutzt ihre eigenen Funktionen und Thirdparty Bibliotheken
  - Mit IoC: Das Framework benutzt die Funktionen der Anwendung
  - auch bekannt als "Das Hollywood Prinzip": "Don't call us, we'll call you"
- 
- Die Anwendung gibt ihre Funktionen dem Framework bekannt
  - Das Framework steuert, was aufgerufen/injiziert wird
  - Spring ist *ein* solches Framework
- 
- DI erlaubt uns IoC anzuwenden

# Verschiedene Arten: konstruktorbasiert

- Pro: Die Klasse kann nicht ohne ihre Abhängigkeiten instanziert werden
- Pro: Abhängigkeiten können als `final` deklariert werden
- Con: Viele Abhängigkeiten blähen den Konstruktor auf
- Pro: Allerdings ist ein großer Konstruktor auch ein Hinweis auf unsauberen Code
- Con: Man muss den Code für den Konstruktor schreiben, nicht nur die Annotation (vielleicht hilft [lombok](#))
- Code Beispiel: nicht Spring, sondern DI händisch machen

```
public class UserService {  
    private final UserRepository repo;  
  
    public UserService(UserRepository repo) {  
        this.repo = repo;  
    }  
}
```

## Verschiedene Arten: methodenbasiert ("setter")

- Con: Kann zu unsauberer Zuständen führen
  - Obacht bei Code im Konstruktor!
- Wird oft in Spring Anwendungen bevorzugt

```
public class UserService {  
    private UserRepository repo;  
  
    public void setUserRepository(UserRepository repo) {  
        this.repo = repo;  
    }  
}
```

## Verschiedene Arten: feldbasiert

- Con: benötigt Reflection
- Con: wegen Reflection: müssen bei Instanzierung in den Tests unterstützt werden
- Con: kann zu unsauberer Zuständen führen

```
public class UserService {  
    private UserRepository repo;  
}
```

## Verschiedene Arten: Empfehlung

- Für alle Pflichtabhängigkeiten konstruktorbasierte DI benutzen
- Für alle optionalen Abhängigkeiten `Optional as Wrapper` benutzen
- Alle per DI initialisierte Variablen `final` setzen
- Die Folien verwenden feldbasierte DI aus Platzgründen

# Übung: DI ohne Container

1. Download via

```
git clone https://github.com/GROSSWEBER/hb-spring
```

2. Gehe ins Verzeichnis ***01\_di\_without\_container***
3. Implementiere die beiden Aufgaben aus der Readme
4. Optional: Implementierte die Bonusaufgabe aus der Readme

# Dependency Injection (DI)

1. Wieso Dependency Injection?
2. **Spring und Dependency Injection**
3. Spring Konfiguration

# Dependency Injection mit Spring

- Dependency Injection ist eine Kernfunktion von Spring
- Das Spring Framework bestimmt die Abhängigkeiten zur Laufzeit
- Erstellte und konfigurierte Objekte werden "Beans" genannt
  - Beans sind per default Singletons
- Application Context:
  - Erzeugt und verdrahtet Objekte entsprechend einer Vorgabe
  - Kennt alle Beans

# Application Lifecycle

1. Code in `main()` bekommt die Steuerung
2. Der Application Context wird gestartet
  - Die Konfiguration wird eingelesen
  - Die Beans werden erzeugt und initialisiert
  - Optional: Server Threads werden gestartet
  - Die Anwendung startet nicht, wenn in dieser Phase ein Problem auftritt
3. restlicher Code in `main()` wird ausgeführt (üblicherweise Validierungen/Tests)
4. Die `main()` Methode wird verlassen
5. JVM wird beendet (oder auch nicht)

## Code Dive: Spring Context

Macht einen ***git pull*** und schaut euch das Verzeichnis ***02\_spring\_context*** an.

# Dependency Injection (DI)

1. Wieso Dependency Injection?
2. Spring und Dependency Injection
3. **Spring Konfiguration**

# Spring Container Konfiguration

- Der Container benötigt Informationen über die Beans und ihre Abhängigkeiten
- Historische Reihenfolge:
  1. XML
  2. Java Annotationen
  3. Programmatische Konfiguration
- Heute relevante Reihenfolge:
  1. Java Annotationen
  2. Programmatische Konfiguration
  3. XML

# Übersicht Konfigurationen: XML

- War die erste Variante
- Nett: Die Konfiguration kann angepasst werden, ohne den Code anzufassen und neu zu bauen
- Keine Typsicherheit
- Kontextwechsel durch den Wechsel zwischen Code und Konfiguration
  - IDE-Unterstützung ist notwendig, bietet aber z.B. IntelliJ

# Übersicht Konfigurationen: Annotationen

- Seit Spring 2.5 (2007)
- Typsichere Alternative zu XML
- Die Konfiguration wandert zu ihrer Komponente

## Übersicht Konfigurationen: Java Code

- Seit Spring 3.2 (2012)
- Manchmal ist die deklarative Konfiguration (XML / Java Annotationen) nicht dynamisch genug
- Ermöglicht die programmatische Konfiguration von Komponenten in Spring

# Spring Konfiguration mit Annotationen

# Java Annotationen

- Stellen Metadaten bereit: von Entwickler zu Entwickler oder Entwickler zu Software
- Wird üblicherweise für einen der folgenden Anwendungsfälle benutzt:
  1. Dem Compiler helfen, Warnungen zu generieren (@Deprecated)
  2. Codegenerierung zur Buildzeit auszulösen (@Data aus lombok)
  3. Zur Laufzeit (@Test)
- Es ist möglich, Klassen, Interfaces, Methoden, Methodenparameter, Felder und lokale Variablen zu annotieren

# Spring Annotationen: Komponenten

Eine Klasse kann mit einer der folgenden Annotationen als Bean gekennzeichnet werden:

- `@Component`: Generischer Stereotyp; alle anderen sind Spezialisierungen von `@Component`
- `@Service` (analog einem DDD Service):
  - "an operation offered as an interface that stands alone in the model, with no encapsulated state"
- `@Repository` (analog einem DDD Repository):
  - "a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects"
- `@Controller` (Web Controller)

```
@Component
public class SayHelloWorldImpl implements SayHelloWorld {

    @Override
    public String greet() {
        return "Hello World!";
    }
}
```

# Spring Annotationen: DI

- Platzieren von `@Autowired` an einem Konstruktor, Feld oder Setter für die entsprechende DI Variante

```
@Component
public class WithCtorBasedInjection {

    private final SayHelloWorld sayHelloWorld;

    @Autowired
    public WithCtorBasedInjection(SayHelloWorld sayHelloWorld) {
        this.sayHelloWorld = sayHelloWorld;
    }
}
```

- Seit Spring Framework 4.3 (2016): `@Autowired` muss nicht am Konstruktor stehen, wenn es nur einen Konstruktor gibt

```
@Component
public class WithSetterBasedInjection {

    private SayHelloWorld sayHelloWorld;

    @Autowired
    public void setSayHelloWorld(SayHelloWorld sayHelloWorld) {
        this.sayHelloWorld = sayHelloWorld;
    }
}
```

```
@Component
public class WithFieldBasedInjection {
    @Autowired
    private SayHelloWorld sayHelloWorld;
}
```

# Spring Programmatische Konfiguration

# Programmatische Konfiguration

- Anwendungsszenarien:
  - Konfiguration von @Beans in Abhängigkeit von bestimmten Bedingungen
  - Integration von Thirdparty-Bibliotheken in eine Spring Anwendung
  - Im Zusammenspiel mit @Configuration in einem Test-Setup
- Ist dem Benutzer der Abhängigkeiten transparent
- Wie: Erstelle eine Klasse mit @Configuration als Annotation und Methode, die mit @Bean annotiert sind, um Beans zu erzeugen:

```
@Configuration
public class GreetConfig {

    @Bean
    public SayHelloWorld sayHelloWorld() {
        return new SayHelloWorldImpl();
    }
}
```

# Programmatische Konfiguration und Dependencies

- Dependencies werden in die @Bean Methoden injiziert

```
@Configuration  
@Import({UserRepoConfig.class})  
public class UserServiceConfig {  
  
    @Bean  
    public UserService getService(UserRepository repo) {  
        return new UserServiceImpl(repo);  
    }  
}
```

# Spring XML Konfiguration

# XML basierte Konfiguration (1)

- Pro: Die Re-Konfiguration der Beans kann ohne Codeänderung stattfinden
- Con: Ist erst einmal nicht typsicher
- Con: XML und Java sind zwei verschiedene Sprachen und zwei verschiedene Kontexte

## XML basierte Konfiguration (2)

In neuen Projekten ***keine*** XML Konfigurationen verwenden!

- Manchmal ist XML notwendig bei der Integration von Legacy Anwendungen
- Spring Security hatte eine sehr mächtige XML Konfiguration
- IDEs: IntelliJ IDEA und STS unterstützen Spring XML Konfigurationen

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="..." xmlns:xsi="..." xsi:schemaLocation="...">
    <bean name="sayHelloWorld"
        class="inc.monster.app.hello.greet.SayHelloWorldImpl"/>

    <bean name="helloWorldService"
        class="inc.monster.app.hello.service.HelloWorldServiceImpl">
        <constructor-arg name="sayHelloWorld" ref="sayHelloWorld"/>
    </bean>

    <bean name="otherHelloWorldService"
        class="inc.monster.app.hello.service.OtherHelloWorldServiceImpl">
        <property name="sayHelloWorld" ref="sayHelloWorld"/>
    </bean>

</beans>
```

# Schnell zurück zu Annotationen

# Injection nach Typ oder Name

- `@Autowired` sucht standardmäßig nach einem passenden Typen
- Alternative: Suche nach Name
  - `@Resource` verwenden oder `@Qualifier` hinzufügen

```
@Component("myFirstDatabase")
public class Database1 implements Database {}

@Component
public class DataLoader {
    private final Database db1;

    @Resource(name = "mySecondDatabase")
    private Database db2;

    @Autowired
    public DataLoader(@Qualifier("myFirstDatabase") Database db1) {
        this.db1 = db1;
    }
}
```

## Andere Annotationen

- **@PostConstruct**: Eine derartig annotierte Methode wird aufgerufen, nachdem alle Abhängigkeiten und Properties gesetzt wurden. Kann bspw. zur Validierung und abschließendem Setup genutzt werden.
- **@PreDestroy**: Eine derartig annotierte Methode wird aufgerufen, bevor der Application Context beendet wird. Kann bspw. zum Aufräumen von benutzten Ressourcen genutzt werden.
- **@Primary**: Eine derartig annotierte Bean ist die bevorzugte. Verwendet, da unentscheidbar gleichwertige Beans (außer bei Collections) einen Fehler erzeugen.
- Außerdem: Bei Collections und Arrays sorgt **@Autowired** dafür, dass alle passenden Beans gesammelt werden. Dadurch wird das Design flexibler.

## Übung: Bau ein Auto mit Spring DI

1. ***git pull*** und schaue in das Verzeichnis ***03\_di\_with\_container***
2. Implementiere die Aufgaben aus der Readme

# Spring Boot

1. **Von Spring zu Spring Boot**
2. Die "Magie" (AutoConfiguration und Starter)
3. Erstellen und Starten
4. Events
5. Testen

## Von Spring zu Spring Boot

- Spring Boot ist Spring, aber mit einer großen Zahl vorgefertigter Konfigurationen für eine Reihe an Funktionen
- Spring Boot bietet für viele Funktionen dabei “convention over configuration”
- Zu Beginn: Es muss keine Anwendungsfunktionen **explizit** konfiguriert werden, es reicht die korrekte Maven Dependency hinzuzufügen
- Evtl. später: Konfiguration anpassen

# Spring Boot main()

- Spring Boot wird mittels einer einfachen main() Methode initialisiert
- SpringApplication wird als Launcher Klasse verwendet
- Die Anwendung wird mit @SpringBootApplication annotiert
- Die Methode SpringApplication.run():
  1. startet den Application Context
  2. optional: Startet einen Listener Thread (was die Beendung der JVM verhindert)
  3. Liefert den ApplicationContext zurück
- Der zurückgelieferte Kontext kann verwendet werden; **aber IoC beachten!**

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

@SpringBootApplication
public class App {
    public static void main(String[] args) {
        ConfigurableApplicationContext context =
            SpringApplication.run(App.class, args);
    }
}
```

# @SpringBootApplication

- ist eine sogenannte Composed-Annotation
- die 3 jeweiligen Annotationen sind Meta-Annotationen (d.h. sie können auf andere Annotationen angewendet werden)

```
@SpringBootConfiguration  
@EnableAutoConfiguration  
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),  
    @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })  
public @interface SpringBootApplication {  
    ...  
}
```

# @SpringBootConfiguration

- Reminder: Eine Klasse kann mit @Configuration annotiert werden, wenn programmatische Konfiguration von Spring benötigt wird
- @SpringBootConfiguration ist eine Spezialisierung von @Configuration

```
@SpringBootConfiguration
public class App {

    @Bean
    public UserRepository userRepository() {
        return new UserRepository();
    }

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

# @ComponentScan

- Sagt Spring, wo nach Klassen mit @Component-Annotationen gesucht werden soll
- Hint: @Configuration, @Service, @Repository, @Controller, ... sind mit @Component meta-annotiert
- Durchsucht ein Package - und alle darunterliegenden Packages
- ist kein Package angegeben (Attribut basePackages), erfolgt die Suche ausgehend vom Package der annotierten Klasse

Kann mit einem oder mehreren Package Namen verwendet werden

```
@ComponentScan("inc.monster.hello")
@EnableAutoConfiguration
public class App {
    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }
}
```

Oder mit Klassen (typsicher und daher präferiert)

```
@ComponentScan(basePackageClasses = {UserPackage.class, App.class})
@EnableAutoConfiguration
public class App {
    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }
}
```

# `@EnableAutoConfiguration`

- Diese Annotation weist Spring Boot an, zu "erraten", wie Spring konfiguriert werden soll
- Kann durch die Verwendung von "spring-boot-starters" Dependencies "konfiguriert" werden

# Spring Boot

1. Von Spring zu Spring Boot
2. **Die "Magie" (AutoConfiguration und Starter)**
3. Erstellen und Starten
4. Events
5. Testen

# AutoConfiguration Grundlagen: **@Import**

- **@Import** erlaubt es andere Konfigurationsklassen zu laden

```
@Configuration  
@Import({MyOtherConfiguration.class})  
public class MyConfiguration {  
    ...  
}
```

# AutoConfiguration Grundlagen: @Conditional

- @Conditional-Annotation erlauben es Komponenten und Konfigurationen nur unter bestimmten Bedingungen zu laden
- Es gibt vorgefertigte Annotationen
  - @ConditionalOnProperty
  - @ConditionalOnBean / @ConditionalOnMissingBean
  - ...
- Ermöglicht auch beliebige Bedingungen

```
@Configuration  
@ConditionalOnBean(OtherComponent.class)  
class ConditionalConfiguration {  
  
    @Bean  
    Bean bean(){  
        ...  
    };  
}
```

# AutoConfiguration Grundlagen: @Conditional

```
@Bean @ConditionalOnMissingBean(NotActive.class)
public SomeComponent a() {
    return new SomeComponent("NotActive");}

@Bean @Conditional(MyComplexCondition.class)
public SomeComponent b() {
    return new SomeComponent("my cond");}

public static class MyComplexCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context,
                          AnnotatedTypeMetadata metadata) {
        return false; // TODO
    }
}
```

# AutoConfiguration: `@EnableAutoConfiguration`

- `@EnableAutoConfiguration` ist u.a. mit `@Import(AutoConfigurationImportSelector.class)` annotiert
- `AutoConfigurationImportSelector` implementiert Interface `ImportSelector`
- `ImportSelector` kann programmatisch Konfigurationsklassen laden

```
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
    ...
}
```

# AutoConfiguration: AutoConfigurationImportSelector

- AutoConfigurationImportSelector lädt Konfigurationsklassen anhand aller Dateien META-INF/spring.factories, die er auf dem ClassPath findet
- die Einträge sind selbst wieder @Configuration-Klassen

AUSZUG:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
    org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\  
    org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\  
    org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\  
    ...
```

## AutoConfiguration: Alles kommt zusammen

- Der Clou: `AutoConfigurationImportSelector` lädt die Konfigurationen so spät wie möglich!
  - nach `@Import`
  - nach `@ComponentScan`
- d.h. eigene Beans sind bereits geladen bevor AutoConfigurations betrachtet werden
- Spring (genauer: die AutoConfigurations) kann jetzt mittels `@Conditional` entscheiden welche Beans noch geladen werden müssen

# Spring Boot Starter

- enthalten eine Datei META-INF/spring.factories und definieren unter dem Key `EnableAutoConfiguration` eigene Konfigurationsklassen, die mittels `@Conditional` prüfen welche Komponenten ihrer Anwendungslogik sie laden müssen

# AutoConfiguration anpassen

- je nach Starter schaltet das definieren eigener Beans entweder nur Teile oder die komplette AutoConfiguration eines Starters ab
- Die meisten Starter ermöglichen jedoch auch das Erweitern der AutoConfiguration durch Interfaces: ...Configurer oder ...ConfigurerAdapter

```
@Configuration  
public class LocaleConfig implements WebMvcConfigurer {  
  
    @Bean  
    public LocaleResolver localeResolver() {  
        ...  
    }  
}
```

# Spring Boot

1. Von Spring zu Spring Boot und Configuration
2. Die "Magie" (AutoConfiguration und Starter)
3. **Erstellen und Starten**
4. Events
5. Testen

# Building Spring Boot

- Die Spring Boot Maven und Gradle Plugins erstellen ein lauffähiges JAR aus der Spring Boot Anwendung
- Die Plugins bringt schon die richtige Konfiguration mit, muss aber explizit in die in der Buildkonfiguration eingefügt werden
  - Wird in der Maven Phase “package” bzw. in der Gradle Tasks “build” aktiv
  - Es ist aber auch möglich, die Goals bzw. Tasks der Spring Boot Plugins direkt aufzurufen:

## Maven

```
$> mvn spring-boot:help  
$> mvn spring-boot:repackage  
$> mvn spring-boot:<maven goal>
```

## Gradle

```
$> gradle bootJar  
$> gradle bootWar
```

# Start einer Spring Boot Anwendung

Es ist möglich Maven und Gradle zum Start einer Spring Boot Anwendung zu verwenden

```
$> mvn spring-boot:run  
$> gradle bootRun
```

oder man kann das JAR bauen und von Hand starten

```
$> mvn package oder gradle build  
$> java -jar target/hello-spring-boot-0.0.1-SNAPSHOT.jar
```

Achtung: Spring Boot benötigt ein paar Sekunden zum Starten und ist daher nicht für schnelle Kommandozeilenanwendungen geeignet

# Fehler beim Starten erkennen

- Spring Boot verwendet die automatisch aktivierte FailureAnalyzers
- fangen Exceptions beim Start der Anwendung und erstellen lesbare Fehlermeldungen
- z.B. Fehler bei Konfiguration

```
*****  
APPLICATION FAILED TO START  
*****
```

Description:

Failed to bind properties under 'my.number' to double:

```
Property: my.number  
Value: foo  
Origin: class path resource [application.yml]:5:11  
Reason: failed to convert java.lang.String to double
```

Action:

Update your application's configuration

# AutoConfiguration debuggen

- Anwendung starten mit --debug
- jar

```
java -jar target/helloworld.jar --debug
```

- maven

```
./mvnw spring-boot:run -Dspring-boot.run.arguments=--debug
```

- gradle

```
./gradlew bootRun -Dorg.gradle.debug=true
```

## Übung: AutoConfiguration

1. ***git pull*** und schaue in das Verzeichnis ***04\_autoconfiguration***
2. Implementiere die Aufgaben aus der Readme

# Spring Boot

1. Von Spring zu Spring Boot
2. Die "Magie" (AutoConfiguration und Starter)
3. Erstellen und Starten
4. **Events**
5. Testen

# Spring (Boot) Application Events

- Spring Framework benutzt Events
  - ContextStartedEvent
  - ContextRefreshedEvent
  - ...
- Spring Boot definiert zusätzliche Events
  - ApplicationStartingEvent
  - ApplicationEnvironmentPreparedEvent
  - ApplicationPreparedEvent
  - ApplicationReadyEvent
  - ApplicationFailedEvent
- Siehe Dokumentation

# Application Event Listener (1)

- Als @Component reicht es, das Interface (hier ApplicationListener) zu implementieren
- Wird dann im ApplicationContext instanziert und genutzt

```
@Component
public class MyApplicationListener implements ApplicationListener {

    @Override
    public void onApplicationEvent(ApplicationEvent event) {
        System.out.println("Using @Component: " + event);
    }
}
```

# Application Event Listener (2)

- Während des Aufsetzens der Anwendung

```
@SpringBootApplication
public class App {

    public static void main(String[] args) {
        ApplicationContext context = new SpringApplicationBuilder()
            .sources(App.class)
            .listeners(event -> System.out.println(
                "Using SpringApplicationBuilder.listeners(): " + event))
            .run(args);
    }
}
```

# Application Event Listener (3)

- Mit Hilfe von Konfigurationsdateien
- Benötigt dann keine @Component Annotation
- Benötigt die folgende Zeile in der Datei "src/main/resources/META-INF/spring.factories":

```
org.springframework.context.ApplicationListener=<FQN of class>
```

```
public class MyApplicationListener implements ApplicationListener {  
  
    @Override  
    public void onApplicationEvent(ApplicationEvent event) {  
        System.out.println("Added by config file: " + event);  
    }  
}
```

# Spring Boot

1. Von Spring zu Spring Boot
2. Die "Magie" (AutoConfiguration und Starter)
3. Erstellen und Starten
4. Events
5. **Testen**

# Unit Tests

- Unit Test = Spring spielt keine Rolle
- Mockito unterstützt verschiedene DI Varianten

```
@ExtendWith(MockitoExtension.class) // or use ...
// ... MockitoAnnotations.initMocks(this)
public class BasicUnitTest {
    @Mock private UserService service;
    @InjectMocks private UserController controller;

    @Test
    public void test() throws Exception {
        when(service....).thenReturn(...);

        assertThat(controller.isActive("1234")).isEqualTo(true);
    }
}
```

# Integrationstests

- Der Spring ApplicationContext wird gestartet:

```
@SpringBootTest
```

- `@SpringBootTest`: Durchsucht das eigene und übergeordnete Packages nach einer `@SpringBootApplication`
- Lädt die komplette Anwendung (kann dadurch dauern)
- Für Blackbox Testen kann man `REST Assured` oder `@TestRestTemplate` nutzen
- `@MockBean`: Mockito mockt die Bean
- Die Maven Dependency `spring-boot-starter-test` fügt `AssertJ` hinzu

```
@SpringBootTest
public class IntegrationWithMockTest {
    @MockBean private UserRepo userRepo;
    @Autowired private UserService service;

    @Test
    public void test() throws Exception {
        when(userRepo.count(false)).thenReturn(-1);

        assertThat(service.countActiveUsers()).isEqualTo(-1);
    }
}
```

# Nur Teile der Anwendung testen

- Idee: Nur einen Teil der Beans initialisieren, um die Startzeit des Tests zu reduzieren
- `@DataJpaTest`: Wenn nur JPA-relevante Teile getestet werden sollen
- `@WebMvcTest`: Wenn nur MVC-relevante Teile getestet werden sollen
- `@JsonTest`: Wenn nur JSON-relevante Teile getestet werden sollen
- `@SpringBootTest(classes=...)`: Wenn man sich die Klassen aussuchen will

```
@JsonTest
public class UserJsonTest {
    @Autowired private JacksonTester<User> json;

    private User USER = new User(1234, "admin");

    @Test public void testSerialize() throws Exception {
        JsonContent<User> asJson = json.write(USER);
        assertThat(asJson).hasJsonPathStringValue("@.username");
        assertThat(asJson).extractingJsonPathStringValue("@.username")
            .isEqualTo("admin");
    }
}
```

# Empfehlungen

- Die Erfahrung zeigt:
  - Beim Test von Teilen ist das Setup manchmal schwierig/komplex
  - Murphy: Der Teil, den man im Test mockt macht am Ende die Probleme
- Unit Tests sind vorzuziehen
  - Die Business Logik sollte möglichst mit Unit Tests verifiziert werden
- Ein paar Spring Integrationstests helfen, die DI und spezielle Spring-Themen zu testen
- Benutze Systemtests (z.B. mit REST Assured) um die komplette Anwendung mit externen Abhängigkeiten zu testen (Der Code funktioniert mit dem aktuellen DB Schema, ...)

# Externe Konfiguration

1. **Interne/Externe Konfiguration**
2. Konfigurationsdateien
3. Einsatz von @Value
4. Einsatz von @ConfigurationProperties
5. Profile
6. Verschiedenes

# Externe Konfiguration (1)

- Zur Konfiguration gehören bspw: Connection Parameter, Timeouts, ...
- Es gilt: Konfiguration variiert zwischen Umgebungen, Code nicht!
- Intern:
  - Die Konfiguration ist Teil des Source-Codes ⇒ Das ist ein Sicherheitsrisiko
  - Evtl. gibt es auch Konfiguration zu den verschiedenen Stages im Quellcode ⇒ Erhöht die Komplexität
- Extern: Die Konfiguration kommt von außen zur Anwendung
  - Konfigurationsdateien
  - Umgebungsvariablen
  - Kommandozeilenparameter
  - Konfigurationsserver
  - ...

## Externe Konfiguration (2)

- Wozu eine externe Konfiguration?
  - Die Anwendung soll sich auf die Domäne konzentrieren ⇒ Konfiguration als übergreifendes Element wird ausgelagert
  - Das gleiche Anwendungsartefakt kann in verschiedenen Stages o.ä. eingesetzt werden
  - Das Verhalten kann ohne neues Compilieren - und manchmal sogar ohne Neustart - angepasst werden
- Spring Boot nennt es "Externalized Configuration"
- Es wird eine spezielle **PropertySource** verwendet, um eine sinnvolle Hierarchie bei der Bestimmung von überschriebenen Werten zu erreichen



# Externe Konfiguration

1. Interne/Externe Konfiguration
2. **Konfigurationsdateien**
3. Einsatz von @Value
4. Einsatz von @ConfigurationProperties
5. Profile
6. Verschiedenes

# Konfigurationsdateien

- Die Konfiguration wird in der Datei `application.properties` oder `application.yml` gesucht
- Suchreihenfolge (Werte aus einem weiter unten aufgelisteten Fundort überschreiben die darüber)
  1. Im Classpath Root
  2. Im Classpath im `/config` Package
  3. Im aktuellen Verzeichnis
  4. Im Verzeichnis `./config` (OS process)
  5. In direkten Unterverzeichnissen des `/config` Verzeichnisses
- Es ist möglich `.properties` und `.yml` Dateien zu vermischen
  - YML eignet sich besser zum Spezifizieren hierarchischer Daten

# Konfigurationsdateien: Empfehlungen

- Spring bietet sehr viel Flexibilität
- Zunächst eine `application.{yml, properties}` in `src/main/resources` benutzen
  - Die Konfiguration ist nur für die lokale Entwicklung
  - **Die Konfiguration verweist für umgebungsabhängige Parameter auf Umgebungsvariablen oder Kommandozeilenparameter**
- Unterstützung für andere Umgebungen/Stages hängt dann vom verwendeten Deployment System ab
- YML oder Properties zu verwenden ist Geschmackssache
- Hilfreich: IntelliJ springt von einem Wert zur Deklaration in der property/yml Datei

# Externe Konfiguration

1. Interne/Externe Konfiguration
2. Konfigurationsdateien
3. **Einsatz von @Value**
4. Einsatz von @ConfigurationProperties
5. Profile
6. Verschiedenes

## @Value

- Die Konfiguration besteht aus Properties
- Jede Property hat einen Namen und einen Wert
- Der Name ist dabei hierarchisch aufgebaut, wobei Punkte die Ebenen trennen
- Um einen einzelnen Wert einer Property auszulesen, wird dieser mit @Value und dem Namen der Property \${<Propertyname>} referenziert
- Dabei werden bestimmte Typkonvertierungen unterstützt

## application.properties:

```
my.base.value=World
my.first.property=Hello ${my.base.value}!
my.first.integer=1111
my.first.long=2222
my.first.double=3333.3333
```

## SomeComponent.java:

```
@Component
public class SomeComponent {
    @Value("${my.first.property}")
    private String firstProperty;

    @Value("${my.first.integer}")
    private Integer firstInteger;

    @Value("${my.first.long}")
    private Long firstLong;

    @Value("${my.first.double}")
    private Double firstDouble;
}
```

# Fortgeschrittene Anwendungsfälle

- `@Value`: kann auch bei Parametern verwendet werden (z.B. bei Konstruktoren)
- Es ist möglich einen Defaultwert anzugeben, falls der Wert nicht gesetzt ist

```
@Value("${my.first.property:'my default value'}")
private String firstProperty;
```

- Es ist möglich mit `#{}`  die Spring Expression Language (SpEL) einzusetzen

```
public SomeComponent(
    @Value("#{'${my.first.property}'.length()}")
    Integer stringSizeOfPropertyValue) {
    ...
}
```

- Mit SpEL können andere Beans über `@` referenziert werden

```
@Value("#{@environment.getProperty('my.first.property')})")
```

Achtung! SpEL ist mächtig, aber bitte vorsichtig einsetzen. Alternative Wege sind meist besser wartbar.

## Code Dive: @Value

Führt ***git pull*** aus und schaut euch das Verzeichnis ***05\_at\_value*** an.

# Externe Konfiguration

1. Interne/Externe Konfiguration
2. Konfigurationsdateien
3. Einsatz von @Value
4. **Einsatz von @ConfigurationProperties**
5. Profile
6. Verschiedenes

# @ConfigurationProperties (1)

- Ist eine Alternative zu @Value, wenn eine Reihe von Properties gesetzt werden sollen
- Dazu wird ein wählbares Präfix genutzt, um dann die Werte zu injizieren

```
@ConfigurationProperties("my.cool.prefix")
public class MyProperties {

    private String value;

    // will return value of "my.cool.prefix.value"
    public String getValue() { return value; }

    public void setValue(String value) { this.value = value; }
}
```

## @ConfigurationProperties (2)

- Es ist notwendig, dass der Standardkonstruktor verfügbar ist
- Alternativ kann `@ConstructorBinding` verwendet werden
- `@ConstructorBinding` erlaubt auch die Klasse `Immutable` zu machen
- Unterstützt (im Gegensatz zu `@Value`) keine SpEL

# @EnableConfigurationProperties

- Um @ConfigurationProperties zu verwenden ist es nötig:
  - a) diese auch noch mit @Component zu versehen
  - b) @EnableConfigurationProperties an einer Konfiguration zu annotieren
    - alle Klassen mit @ConfigurationProperties in der @EnableConfigurationProperties Annotation aufzulisten

```
@EnableConfigurationProperties({MyProperties.class})
@SpringBootApplication
public class App {
    ...
}
```

# Injecten

- Mit `@ConfigurationProperties` annotierte Klassen sind Beans
- können als Dependency injiziert werden
- Kann Tests vereinfachen

```
@Component
public class MyComponent {

    private MyProperties properties;

    @Autowired
    public MyComponent(MyProperties properties) {
        this.properties = properties;
    }

    public void printProperties() {
        System.out.println(properties);
    }
}
```

# Verschachtelte Konfigurationen

- Spring Boot bildet verschachtelte Properties auf verschachtelte Elemente ab
  - Müssen keine internen Klassen sein
  - Auflösung über den Namen des Feldes

```
other:  
  value: Hello again!  
  
nested:  
  value: Hello nested value!
```

```
@ConfigurationProperties("other")  
public class OtherProperties {  
    private String value; // omit setter/getter for space reasons  
    private Nested nested;  
  
    public Nested getNested() { return nested; }  
    public void setNested(Nested nested) { this.nested = nested; }  
  
    public static class Nested {  
        private String value;  
  
        public String getValue() { return value; }  
        public void setValue(String value) { this.value = value; }  
    }  
}
```

# Collections

- Spring Boot bildet Wertelisten auf Java Listen oder Arrays ab

```
foo:  
  roles:  
    - USER  
    - ADMIN
```

```
@ConfigurationProperties("foo")  
public class FooProperties {  
    private List<String> roles;  
  
    public List<String> getRoles() { return roles; }  
    public void setRoles(List<String> roles) { this.roles = roles; }  
}
```

# Validierung (1)

- Wird durch hinzufügen des Maven Moduls "spring-boot-starter-validation" aktiviert und die Annotation `@Validated`
- Erlaubt es, durch Annotationen Werte von Properties prüfen zu lassen

```
import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;

import org.springframework.validation.annotation.Validated;

@Validated
@ConfigurationProperties("valid")
public class ValidProperties {

    @NotNull
    @NotBlank
    @Email
    private String email;

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
}
```

## Validierung (2)

Mit aktiverter Validierung startet die Anwendung nicht, wenn es Fehler gibt

```
*****
APPLICATION FAILED TO START
*****  
  
Description:  
  
Binding to target inc.monster.app.ValidProperties@6a705cee[email=me@@you.de] failed:  
  
    Property: valid.email  
    Value: me@@you.de  
    Reason: keine gültige E-Mail-Adresse
```

# Flexibles Binding

- Spring Boot erlaubt ein wenig Flexibilität beim Einsatz von `@ConfigurationProperties`
- Das heißt: Namen von Properties müssen nicht 1:1 mit den Java Namen übereinstimmen
- Angenommen eine Klasse wurde mit `@ConfigurationProperties("person")` und enthält ein Feld "firstName"
  - In der `application.yml` und bei Kommandozeilenparameter sind neben "`person.firstName`" auch "`person.first-name`" und "`person.first_name`" erlaubt
  - Bei Umgebungsvariablen ist auch "`PERSON_FIRSTNAME`" erlaubt

## Code Dive: @ConfigurationProperties

Führt ***git pull*** aus und schaut euch das Verzeichnis  
***06\_typesafe\_configuration\_properties*** an

# Externe Konfiguration

1. Interne/Externe Konfiguration
2. Konfigurationsdateien
3. Einsatz von @Value
4. Einsatz von @ConfigurationProperties
5. **Profile**
6. Verschiedenes

# Profile

- Profile sind ein Weg zur Trennung von Konfigurationen
- Anwendungsfälle:
  - Steuerung von Beans in Tests
  - Stage- oder umgebungsabhängige Konfiguration (**bitte nicht**)
  - Verschiedene Verwendungsmodi (**bitte nicht**)
- Sollten extrem vorsichtig und sparsam eingesetzt werden
- Aktive Profile: Eine (möglicherweise leere) Menge an Profilnamen

# Profile Festlegen

- Mit Hilfe der Property `spring.profiles.active` lässt sich eine (kommaseparierte) Liste von aktiven Profilen angeben
- Typische Quellen: Umgebungsvariablen, Kommandozeilenparameter, Konfigurationsdateien
  - Auch hier gilt die übliche Reihenfolge bei der Suche nach Properties
- Mittels `spring.profiles.include` können zusätzliche Profile hinzugefügt werden
- Das ist auch noch während des Starts möglich

```
@SpringBootApplication
public class AppWithProfiles {
    public static void main(String[] args) {
        System.setProperty("spring.profiles.active", "p1,p2");
        System.setProperty("spring.profiles.include", "p3,p4");
        ApplicationContext context = new SpringApplicationBuilder()
            .sources(AppwithProfiles.class)
            .profiles("p5", "p6").run(args);
        Environment env = context.getEnvironment();
        System.out.println("active profiles = " +
            Arrays.toString(env.getActiveProfiles()));
        System.out.println("default profiles = " +
            Arrays.toString(env.getDefaultProfiles()));
    }
}
// active profiles = [p3, p4, p5, p6, p1, p2]
// default profiles = [default]
```

# Beans in Abhängigkeit vom Profil

- `@Profile({"profileName1", "profileName2"})` kann verwendet werden an
  - Klassen die mit `@Component` (d.h. auch `@Configuration`) annotiert sind
  - Methoden, die mit `@Bean` annotiert sind
- Das Element wird nur geladen, wenn **mindestens ein** Profil aktiv ist
- Die Aktivierung kann auch negiert werden: "`!profileName`"

```
@Bean  
@Profile({"p1", "!p2"})  
public SomeComponent getSome() {  
    return new SomeComponent("p1 OR not p2");  
}
```

## Andere Anwendungsmöglichkeiten von Profilen

- Profspezifische Konfigurationen können in der Datei `application-{profile}.properties` definiert werden
- Es ist möglich, mehrere Profile in einer YML Datei zu definieren
- Das Logging kann profilspezifisch konfiguriert werden
- Mit `@ActiveProfiles` können im Test gezielt Profile aktiviert werden

# Externe Konfiguration

1. Interne/Externe Konfiguration
2. Konfigurationsdateien
3. Einsatz von @Value
4. Einsatz von @ConfigurationProperties
5. Profile
6. **Verschiedenes**

# Spring Boot Developer Tools

- Funktionen:
  - Schaltet das Caching aus
  - Die Anwendung startet bei Änderungen automatisch neu
  - Properties können überschrieben werden
- Developer Tools sind NICHT für Produktivsysteme gedacht!
  - Wenn man die Applikation mittels `java -jar ...` startet, gilt das als "Produktivsystem"
- Wird als Maven Dependency hinzugefügt

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
</dependency>
```

# Spring Boot Developer Tools: Konfiguration

- Kann alle anderen Properties überschreiben
- Liegt in "\$HOME/.spring-boot-devtools.properties" (kein YML möglich)
- Wird daher für alle Spring Anwendungen des Benutzers angewendet, die aktivierte Developer Tools haben

# Die Environment Klasse

- Liest Umgebungsvariablen und Systemproperties und stellt sie bereit
- Ist vordefiniert und muss nur als Dependency injiziert werden

## Allgemeine Spring Properties

- Spring kennt sie von sich aus
- IntelliJ kennt sie auch
- Lasst uns zusammen die [Doku](#) anschauen

# Spring Web MVC

1. **Einführung**
2. Ein Detaillierter Blick auf Controller
3. Und jetzt zum Code

# Spring Web MVC

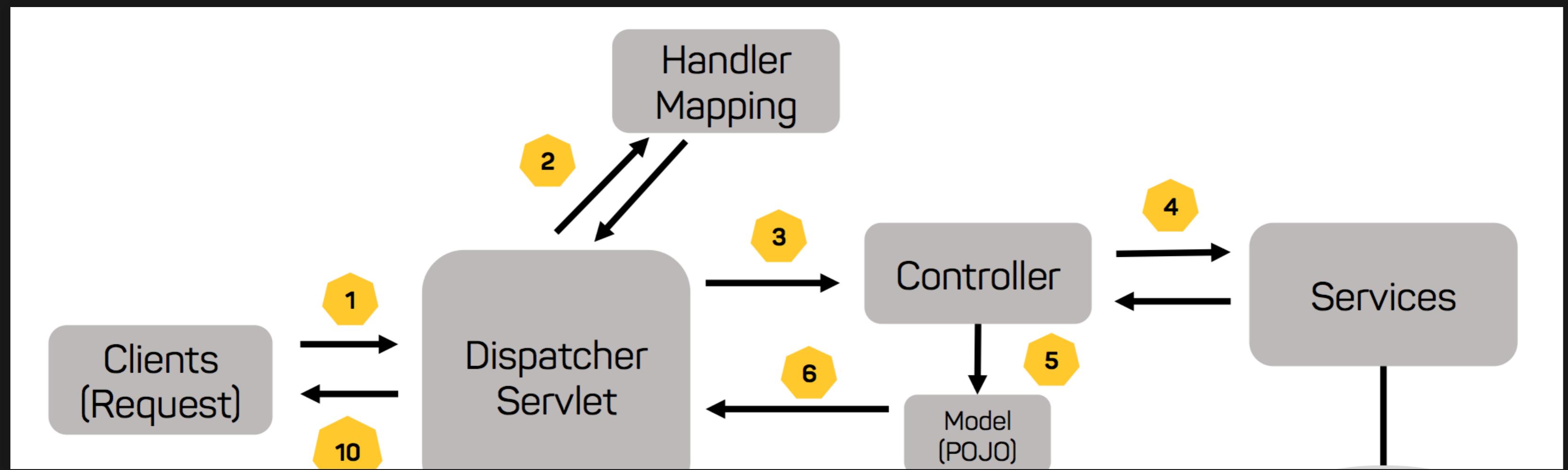
- Auch bekannt als "Spring MVC"
- Bietet ein umfangreiches "Model View Controller" Webframework an
- MVC ist ein verbreitetes Design Pattern für GUI und Web Applicationen
  - M = Model:
    - Verwaltet die Daten, Geschäftslogik und Regeln der Anwendung
  - V = View:
    - Die Darstellung der Informationen
  - C = Controller:
    - Nimmt Kommandos für die Daten entgegen und leitet sie an das Modell weiter
- Ziel: Separation of concerns

# MVC in einer "klassischen" Webanwendung

- Model: Serviceklassen und Datenhaltung (SQL, ... ) mit Zugriffsklassen (DAO, Repository)
- View: Templates (JSF, JSP, Velocity, Thymeleaf, ...) die HTML generieren
- Controller: Klassen, die ein Formular (`<form>`) verarbeiten
- Der Server verwaltet den Zustand

# MVC in einer Single Page Application (SPA)

- Single Page Application (SPA): Intelligenz und Logik liegt im JavaScript Frontend
- Spring Anwendung als Server muss kein HTML mehr erzeugen
- Model: *keine Änderung*
- View: POJOs werden dynamisch in JSON konvertiert
- Controller: Bekommt JSON als Eingabe, wird per AJAX getriggert
- Variante: keine serverseitige Zustandsverwaltung mehr



- 1) Ein Client fragt <http://localhost:8080> an
- 2) Es wird geprüft, wer den HTTP Request verarbeiten sollte

```
: DispatcherServlet with name 'dispatcherServlet' processing GET request for [/]
: Looking up handler method for path /
: Returning handler method [public java.lang.String inc.monster.controllers.IndexController.index()]
```

```
@Controller
public class IndexController {

    @RequestMapping("/")
    public String index(){
        return "index";
    }
}
```

- 3) Aufruf der Methode in der Controller Klasse
- 4 und 5) Wird nicht benötigt
- 6) "index" ist der Name der neuen View

## 7) Finden des Views mit dem Namen "index"

```
DEBUG o.s.w.s.v.ContentNegotiatingViewResolver : \
Returning \
[org.thymeleaf.spring4.view.ThymeleafView@b9a4d6a] \
based on requested media type 'text/html'
```

## 8-9) Thymeleaf liefert `src/main/resources/templates/index.html` aus

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
...
<body>
<div class="container">
    <h1>This is my Thymeleaf index</h1>
</div>
</body>
</html>
```



# Spring Web MVC

1. Einführung
2. **Ein Detaillierter Blick auf Controller**
3. Und jetzt zum Code

# @RequestMapping

- Fungiert als Filter bei der Suche nach Methoden zur Verarbeitung von Requests
- URL-Präfix/Muster als Filterkriterium
- HTTP Methode als Filterkriterium

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

- Man kann auch PostMapping & Co. zur Abkürzung verwenden
- Content Types als Filterkriterium (consumes und produces)

```
@RequestMapping(value = "/purchases",  
                 consumes = "application/json"  
                 produces = "application/json; charset=UTF-8")
```

- Kann auf Klassenebene zur Deklaration eines gemeinsamen URL Präfix benutzt werden

# Methodenparameter für Controller (1)

- Erlaubt ein hohes Maß an Flexibilität
- `HttpServletRequest, Response` für die volle Kontrolle
- `HttpSession`
- `InputStream / Reader` um den Inhalt der Anfrage selbst zu lesen
- `OutputStream / Writer` um das Ergebnis selbst zu schreiben
- Ein Parameter vom Typ `Model` kann benutzt werden, um Informationen an den View zu geben
- Vollständige Liste: <https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html#mvc-ann-arguments>

## Methodenparameter für Controller (2)

- Wenn ein Parameter mit @RequestBody annotiert ist, wird versucht, den Inhalt der Anfrage auf den Zieltyp zu mappen (JSON to POJO)
- Es kann dabei Validierung benutzt werden; dann gibt es einen zusätzlichen Errors Parameter

```
@RequestMapping("/list-purchases")
public String purchases(@RequestBody PurchaseSearchQuery query,
                      Model model) {
    List<Purchase> purchases = repo.search(query);
    model.addAttribute("bigList", purchases);
    return "purchases";
}
```

## Methodenparameter für Controller (3)

- Teile der Anfrage-URL können extrahiert werden:
  - wird im `@RequestMapping` deklariert
  - Zugriff über `@PathVariable`
  - Typkonvertierung ist möglich
  - Ab Java 8: Der Name kann weggelassen werden, da Spring ihn normalerweise über Reflection bekommt (Compiler Option: "-parameters")

```
// http://localhost:8080/order/12345/load
@RequestMapping("/order/{orderId}/load")
public String getSingleOrder(@PathVariable("orderId") Long id) {
    ...
}
```

## Methodenparameter für Controller (4)

- Zugriff auf URL-Parameter:
  - mittels `@RequestParam`
  - Typkonvertierung ist möglich

```
// http://localhost:8080/load-order?orderId=1234
@RequestMapping("/load-order")
public String getSingleOrder(@RequestParam("orderId") Long id) {
    ...
}
```

- Es ist noch mehr möglich - gehört aber nicht zu den Basics

# Rückgabewerte von Controllern

- Hier gibt es auch viel Flexibilität
- Schon gesehen: String zurückgeben, der den Namen des Views enthält
- Model und ModelAndView können für "klassische" MVC Anwendungen benutzt werden
- @ResponseBody oder ein POJO-Wert wird nach JSON konvertiert (wenn produces das so festlegt, was standardmäßig der Fall ist)
- ResponseEntity kann zurückgegeben werden, wenn mehr Kontrolle über HTTP Status Codes, Header und den Body notwendig ist
- Es ist auch asynchrones Antworten und mehr möglich - gehört aber nicht zu den Basics

```
@Controller
@RequestMapping("/order")
public class OrderController {
    // not shown: define and initialize repo

    @RequestMapping("load/{id}")
    public ResponseEntity<Order> load(@PathVariable("id") Long id) {
        Order order = repo.load(id);
        if (order == null) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(order);
    }
}
```

# Exception Handling

- Standardverhalten: Das Framework fängt RuntimeExceptions und liefert einen Fehler 500 zurück

```
$ curl 'http://localhost:8090/hello' -H 'Accept: text/html'  
<html><body><h1>Whitelabel Error Page</h1><p>This application has ...  
$ curl 'http://localhost:8090/hello'  
{"timestamp":1513007357136, "status":500,  
"error":"Internal Server Error",  
"exception":"java.lang.RuntimeException",  
"message":"My Exception", "path":"/hello"}  
$
```

- Ist als Standard hilfreich
- Durch die Stacktraces können aber technische oder andere Interna nach außen gelangen

# Angepasstes Exception Handling

- `@ExceptionHandler`: Erlaubt eine feine Anpassung der Behandlung
- Kann auch auf bestimmte Typen von Exceptions eingeschränkt werden
- Auch hier sind wieder verschiedene Methodenparameter und Rückgabewerte möglich

```
@ExceptionHandler(NumberFormatException.class)
public ResponseEntity<String> handleException
    (Exception exception, HttpServletRequest request) {
    System.out.println("exception = " + exception);
    System.out.println("request = " + request);
    return ResponseEntity.badRequest().body("The number was bad");
}
```

```
$ curl 'http://localhost:8090/hello2'
The number was bad
```

# @ControllerAdvice

- Dient der Festlegung des Verhaltens aller oder einer Teilmenge der Controller
- Eine derartig annotierte Klasse liefert sinnvolles Standardverhalten
- Ist empfohlen, um zu steuern, welche Informationen einen Server verlassen

```
@ControllerAdvice
public class GlobalExceptionHandler
    extends ResponseEntityExceptionHandler {

    @ExceptionHandler(IndexOutOfBoundsException.class)
    public ResponseEntity<String> handleException
        (Exception exception, HttpServletRequest request) {
        System.out.println("exception = " + exception);
        System.out.println("request = " + request);
        return ResponseEntity.badRequest().body("The index was bad");
    }
}
```

# @ResponseStatus

- Kann an Methoden oder Exceptions verwendet werden
- Legt den Status Code und einen "Grund" fest
- Risiko: Geschäftsbezogener Code muss wissen, dass es so etwas wie eine REST API gibt

```
@ResponseStatus(value = HttpStatus.NOT_FOUND,
               reason = "No object found")
public class EntityNotFoundException extends RuntimeException {
    ...
}
```

# Spring Web MVC

1. Einführung
2. Ein Detaillierter Blick auf Controller
3. **Und jetzt zum Code**

## Code Dive: Ein Komplexeres MVC Beispiel

Führt `git pull` aus und schaut euch das Verzeichnis `07_spring_mvc_deep_dive` an.

# Übung: Erweiterung des MVC CRUD Beispiels

1. *git pull* und schaue in das Verzeichnis ***08\_spring\_mvc\_crud\_customer***
2. Implementiere die Aufgaben aus der Readme

# Spring Boot Restful Webservices

1. **REST im Allgemeinen**
2. REST Server mit Spring
3. REST Client mit Spring

# Einführung in REST

- REpresentational State Transfer (REST)
- Roy Fielding:
  - Hauptautor der HTTP Spezifikation
  - Hat ebenfalls an der Spezifikation der URI gearbeitet
- REST wurde von Fielding zuerst 2000 beschrieben:
  - Beschreibt im Rahmen seiner Dissertation REST als Architekturstil, der durch die Web implementiert wird
- RESTful WebServices nutzen die Web Infrastruktur (Routing, Caching, HTTP Methoden)

# Kernprinzipien von REST (1)

- Klare Identifikation:
  - Ressourcen werden durch eine URI beschrieben
  - Eine bekannte Variante der URIs sind WWW URLs
- Hypermedia:
  - Formale Beschreibung: "Hypermedia as the engine of application state" (HATEOAS)
  - Ressourcen sind mit anderen Ressourcen "verknüpft"
  - Ein Client kann den Verknüpfungen in einer Antwort folgen (analog zu den Links in Webseiten)
  - Welche Links zurückgeliefert werden, hängt vom Zustand der Ressource ab

## Kernprinzipien von REST (2)

- Ressourcen und Repräsentationen
  - Der Client kann verschiedene Repräsentationen einer Ressource anfordern (content negotiation)
  - XML, JSON, TEXT, ...
- Die Kommunikation ist zustandslos
  - Kein temporärer Zustand des Clients wird auf dem Server gespeichert

## Kernprinzipien von REST (3)

- Standard Methoden (einheitliche Schnittstelle):
  - Jede Ressource muss dieselbe Schnittstelle implementieren
  - GET: Die Daten der Ressource holen
  - POST: Anlegen einer neuen Ressource
  - PUT: Eine Ressource anlegen oder aktualisieren
  - DELETE: Eine Ressource löschen
  - PATCH, HEAD, OPTIONS, TRACE

# HTTP Methoden

- Sicher: die Methode liest nur Daten
- Idempotent: Mehrfache identische Aufrufe verändern den Serverzustand nicht (mehr)

HTTP Methode	Idempotent	Sicher
GET	ja	ja
HEAD	ja	ja
OPTIONS	ja	ja
PUT	ja	nein
DELETE	ja	nein
POST	nein	nein
PATCH	nein	nein

# Spring Boot Restful Webservices

1. REST im Allgemeinen
2. **REST Server mit Spring**
3. REST Client mit Spring

# RESTful Web Service mit Spring (1)

- Verglichen mit Spring MVC: keine Erweiterung, sondern eine Reduktion
- HTTP Requests werden von einem Controller verarbeitet
- `@RestController` = `@Controller + @ResponseBody` für alle Methoden
- Achtung: `@ResponseBody` überspringt den View

```
@RestController
@RequestMapping(value = "/hello")
public class HelloController {

    @GetMapping
    public Hello getHello() {
        return new Hello("Hello World!");
    }

    @PostMapping
    public Hello updateHello(@RequestBody Hello hello) {
        return hello;
    }
}
```

# RESTful Web Service mit Spring (2)

- Die Konvertierung der Anfrage und der Antwort hängt von den HTTP Headern "Content-Type" und "Accept" ab

```
$ curl --request POST --url http://localhost:8080/hello \
--header 'accept: application/xml' \
--header 'content-type: application/json' \
--data '{"message":"hallo du"}'
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:hello xmlns:ns2="http://monster.inc/hello">
    <message>hallo du</message>
</ns2:hello>
```

# Konvertierung

- Spring MVC benutzt das `HttpMessageConverter` Interface für die Konvertierung von HTTP Anfragen und Antworten
- Objekte können automatisch konvertiert werden von/nach:
  - JSON mit Hilfe von Jackson
  - XML mit Hilfe der Jackson XML Erweiterung, falls verfügbar, sonst mit JAXB

# HttpMessageConverters

- Wird verwendet, um Konvertierung hinzuzufügen oder anzupassen

```
@Configuration
public class ConverterConfig {
    @Bean public HttpMessageConverters customConverters() {
        return new HttpMessageConverters(new MyPlainTextConverter());
    }

    public static class MyPlainTextConverter
        implements HttpMessageConverter<Object> {
        @Override
        public void write(Object o, MediaType contentType, HttpOutputMessage outputMessage) throws IOException, HttpMessageNotWritableException {
            // rather crude for demonstration
            outputMessage.getBody().write(o.toString().getBytes());
        }
        ...
    }
}
```

## Übung: Erweitern eines REST Server

1. ***git pull*** und schaue in das Verzeichnis ***09\_spring\_rest\_server***
2. Implementiere die Aufgabe aus der Readme

# Spring Boot Restful Webservices

1. REST im Allgemeinen
2. REST Server mit Spring
3. **REST Client mit Spring**

## Low-Level: RestTemplate

- Ist die Instantlösung von Spring für das Ausführen von REST Anfragen
- Kann auf verschiedenen HTTP Bibliotheken genutzt werden
- Mit Hilfe eines Interceptor kann die Anfrage/die Antwort verändert werden
- Unterstützt Typkonvertierung
- Kann in Tests mittels Mocking gekapselt werden

# RestTemplate

- Verschiedene Gruppen von Methoden
  - Low Level: `execute( ... )`
  - Mid Level: `exchange( ... )`
  - High Level: angepasst an die HTTP Methode, wie bspw. `delete( ... )` oder `getForObject( ... )`

```
@SpringBootApplication
public class App {

    // have to provide a rest template
    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        // Do any additional configuration here
        return builder.build();
    }

    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run(App.class, args);
        RestTemplate template = context.getBean(RestTemplate.class);
        String content = template.getForObject("http://www.google.de",
                                                String.class);
        System.out.println("content = " + content.substring(0, 50) + "...");
    }
    // content = <!doctype html><html itemscope="" itemtype="http://...
```

# JSON Antwort

```
String url = "https://api.github.com/orgs/github/repos";
List<Map<String, Object>> repos =
    template.getForObject(url, List.class);
List<String> names = repos
    .stream()
    .map(r -> (String) r.get("name"))
    .collect(Collectors.toList());
System.out.println("names = " + names);
// names = [media, albino, hubahuba, jquery-hotkeys, jquery-relatize_date, reques
```

# Daten senden

```
public static class User {  
    public final long id;  
    public final String username;  
  
    public User(long id, String username) {  
        this.id = id;  
        this.username = username;  
    }  
}  
  
Object body = new User(1234, "admin");  
ResponseEntity<String> response =  
    template.postForEntity("http://httpbin.org/post",  
    body, String.class);  
System.out.println("response = " + response.getBody());  
// ... "data": "{\"id\":1234, \"username\":\"admin\"}", ...
```

# URL Templates

- Sind in allen Methoden des RestTemplate verfügbar
- Platzhalter in der URL "{name}" werden wie bei @RequestMapping behandelt
- Entweder eine Liste von Werten oder eine Map

```
 ResponseEntity<String> withTemplate1 = template.getForEntity(  
    "http://httpbin.org/anything/delete-order/{userId}/{orderId}",  
    String.class, 1234, UUID.randomUUID());  
 System.out.println("response = " + withTemplate1.getBody());  
 // "url": "http://httpbin.org/anything/delete-order/1234/7e5ef64b-f014-4f4b-842c-06110078147d"
```

- Alternative UriComponentsBuilder

# RestTemplate und Tests

- MockRestServiceServer kann benutzt werden, um einen REST Server zu simulieren

```
@SpringBootTest(classes = App.class)
public class RestTemplateTest {
    @Autowired
    private UserClient userClient;

    @Autowired
    private MockRestServiceServer server;

    @Test
    public void test() throws Exception {
        server
            .expect(once(), requestTo("http://api.example.org/user/1234"))
            .andExpect(method(HttpMethod.DELETE))
            .andRespond(withSuccess());

        userClient.delete(1234);
        server.verify();
    }
}
```

# High Level: Empfehlung

- Ziel: Zugriff auf einen externen Service Foo
- Erstelle eine Klasse `FooClient`, die ein normales objektorientiertes Interface hat und die Implementierungsdetails kapselt
- Verbindungsparameter werden in einer externen Konfigurationsdatei hinterlegt
- Die Werte daraus werden entweder direkt per `@Value` injiziert, oder es werden `FooClientProperties` deklariert und per `@ConfigurationProperties` gesetzt
- Auch einen Blick wert: <https://github.com/OpenFeign/feign>

## Übung: Erweitern eines REST Client

1. ***git pull*** und schaue in das Verzeichnis ***10\_rest\_client***
2. Implementiere die Aufgabe aus der Readme
3. Optional: Implementiere die Bonusaufgabe aus der Readme

# Spring (Boot) und Betrieb

1. **Logging**
2. Actuator

# Logging

- Technisches Logging:
  - ERROR Meldungen
  - Stacktraces
  - Informationen zum Start und Stop der Anwendung
- Business Logging:
  - User Loginvorgänge
  - User Aktivitäten/Transaktionen

# Logging und Spring Boot

- Bringt nur wenig neues (im Vergleich zu Standards)
- Loglevel werden in der `application.{yml, properties}` definiert

```
logging:  
  level:  
    org.springframework.web: DEBUG
```

- Benutzt standardmäßig logback
- Logkonfiguration ist vorbelegt, um einen schnellen Einstieg zu ermöglichen
- Wiederverwendung von Dateien ist möglich
- In der Anwendung wird SLF4J benutzt

# logback-spring.xml

- Zunächst wird nach logback-spring.xml geschaut
- Spring benutzt einige Erweiterungen
- **springProfile** Macht eine Einstellung abhängig vom Profil

```
<springProfile name="p1, !p2">  
  ...  
</springProfile>
```

- **springProperty** ermöglicht die Verwendung von Environment Properties in der Konfiguration; Kann bspw. genutzt werden, um den Namen der Anwendung/Stage als Attribut zur Logmeldung hinzuzufügen

# application.properties

```
loggingAndMonitoring.application=myportal
loggingAndMonitoring.service=myportal-order
loggingAndMonitoring.environment=production
```

## logback-spring.xml

```
<configuration debug="true">
    <!-- Every property defined in the 'context' scope is added
        as a custom field to the result json. -->
    <springProperty scope="context" name="application"
        source="loggingAndMonitoring.application"/>
    <springProperty scope="context" name="service"
        source="loggingAndMonitoring.service"/>
    <springProperty scope="context" name="environment"
        source="loggingAndMonitoring.environment"/>

    <include resource="org/springframework/boot/logging/logback/defaults.xml"/>
    <property name="LOG_FILE" value="${user.dir}/logs/${service}.log"/>
    <include resource="org/springframework/boot/logging/logback/console-appender.xml"/>
    <include resource="org/springframework/boot/logging/logback/file-appender.xml"/>
    ...

```

# Logging: Big Picture

- Lombok bietet die Annotation `@Slf4j` zur Injektion eines Loggers
- JSON als Format macht die Verarbeitung einfacher (als Text zu analysieren) und schlanker (als XML Fragmente)
- ELK (Elasticsearch, Logstash, Kibana), Splunk oder ein vergleichbares Tool sollte zum Betrachten und Analyseren der Logs benutzt werden
- Falls ssh und grep (immer noch) die einzige Option sind: Jede Logmeldung sollte als einzelne Zeile in einer Textdatei **zusätzlich** zur JSON Datei ausgegeben werden
- Zusammen mit dem Betrieb entscheiden, wie Logs gespeichert werden: in eine Datei schreiben oder über das Netz schicken
- Zusammen mit dem Betrieb werden die einzelnen Level, die verwendete Sprache/Vokabular, Formate für die Daten etc festgelegt
- Das gleiche Standardlevel sollte für alle Stages verwendet werden
- Frühzeitig einen Prozess aufsetzen, der Logmeldungen überwacht

# Spring (Boot) und Betrieb

1. Logging
2. **Actuator**

# Problemstellung

- Monitoring und Alarmierung sind Schlüsselaufgaben im "DevOps" Konzept
- Monitoring: Messen und Speichern von allgemeinen Gesundheitswerten und wichtigen Metriken der Anwendung
- Alarmierung: Informieren/Alarmieren einer Person oder Gruppe, wenn die Anwendung nicht mehr "gesund" ist oder Metriken ungewöhnliche Muster aufweisen
- Microservices erhöhen den Druck, da mehr Services laufen, Deployments häufiger stattfindet etc.
- Wie kann ich also meine Anwendung monitoren?
- Und wie kann ich im Fehlerfall einen Alarm auslösen?

# Spring Boot Actuator

- Spring Boot Actuator ist eine Sammlung von Werkzeugen, um eine Spring Boot Anwendung zu monitoren
- Sammelt Metriken
- Erlaubt die Abfrage/das Versenden der Metriken
- Liefert allgemeine Gesundheitsinformationen über die Anwendung
- Erlaubt Einblicke in die laufende Anwendung

# Standard Endpoints

- Es gibt verschiedene Endpoints für unterschiedliche Zwecke:
- /actuator
- /auditevents
- /autoconfig
- /beans
- ...

Lasst uns die [Doku](#) anschauen

# Health Endpoint

- Liefert zurück, ob die Anwendung läuft und eine grundlegende Tests grün sind
- Eine der wichtigsten Informationen, wenn man produktiv laufen möchte
- Die Anwendung untersucht sich dabei selbst und die Services, die sie benutzt
- Spring liefert einige Standard Gesundheitswerte
- Es ist jedoch möglich, eigene Werte zu sammeln
- Dazu muss die eigene Implementierung eines `HealthIndicator` im `ApplicationContext` geladen werden

```
@Slf4j
@Component
public class UserCustomHealthIndicator implements HealthIndicator {
    @Autowired private final UserService service;

    @Override public Health health() {
        Health.Builder health = new Health.Builder();
        Optional<User> user = service.getUser(10L);
        if (user.isPresent()) {
            health.up();
        } else {
            log.error("user with id 10 not found!");
            health.down();
        }
        return health.build();
    }
}
```

# Application Information

- Liefert Informationen wie Release/Version, git Informationen oder das Deployment Datum
- Spring liefert einige Standard Informationen
- Es ist jedoch möglich, eigene Informationen zu sammeln
- Dazu muss die eigene Implementierung eines `InfoContributor` im `ApplicationContext` geladen werden
- Siehe auch <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-application-info>

# Zugriff auf git Information

- Spring hat bereits den `GitInfoContributor`, der die Datei "git.properties" einliest
- Um diese zu erzeugen, kann man die pom.xml wie folgt anpassen:

```
<build>
  <plugins>
    <plugin>
      <groupId>pl.project13.maven</groupId>
      <artifactId>git-commit-id-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

- Die Information ist dann unter dem REST Endpoint "/info" abrufbar

# Metriken

- Metriken werden über REST API ausgeliefert
- Sollten Metriken per Pull oder Push verarbeitet werden? Beides ist möglich
- Pull: Periodisch Daten von einem Metrik Endpoint abrufen
- Push: Periodischer Push an einen externen Dienst mit einem MetricWriter (Statsd, JMX, ...)
- Out of the box: Speicher, ClassLoader, ThreadPools, ...
- Wenn verfügbar, auch Metriken über DataSource, Cache und Tomcat
- Siehe auch <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-metrics>

## Eigene Metriken

- Unterstützt Metriken als Zähler oder "Tempo"
- Zähler: zählt nur hoch oder runter
- "Tempo": bestimmter Wert zu einem Zeitpunkt

```
@Service
public class MyServiceWithMetrics {
    private final MeterRegistry meterRegistry;

    @Autowired
    public MyServiceWithMetrics(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
    }

    public void exampleMethod() {
        meterRegistry.counter("my.service.with.metrics.counter").increment();
        meterRegistry.gauge("my.service.with.metrics.gauge", new Random().nextDouble());
    }
}
```

# Eigene Endpoints

- Es ist möglich, eigene Endpoints hinzuzufügen
- Eigene Endpoints dienen der Bereitstellung von Metriken oder Daten über "custom actuator endpoints"
- Manchmal muss ein eigener URL Pfad verwendet werden (wenn es einen Konflikt mit der Anwendung gibt)
- Vielleicht auch nur, um eigene Metriken zu liefern

# DevOps: Big Picture

- Entwickler und Betrieb (sprich: auch Support) sollten miteinander reden
- Wie die Anwendung am Laufen gehalten wird, sollte für alle relevant sein
- Alle müssen über Monitoring und Alarmierung sprechen
- Wünschenswert: You built it, you run it
- Betriebsaspekte sollten von Tag 1 berücksichtigt werden
- Actuator funktioniert out of the box, aber muss an die Anforderungen jedes Projektes angepasst werden

## Code Dive: Spring Actuator

- ***git pull*** und schaue in das Verzeichnis ***12\_custom\_endpoints***
  1. Schaut euch den Java Code an
  2. Schaut die Properties an
  3. Startet die Anwendung und greift auf die Endpoints per Browser zu
  4. Benutzt IntelliJ/STS um die Anwendung anzuschauen

Durchatmen

Viel gelernt und es gibt noch mehr

# Package Struktur

- Die Domäne sollte als oberste Strukturebene verwendet werden:
  - `company.{order,payment,search,...}`.
- Im besten Fall ist nur eine handvoll Klassen in jedem Package
- Ansonsten durch weitere Subpackages strukturieren:
  - Domäne (search query, search response, search execution plan, search cache, search performance monitoring, ...) oder
  - Technisch (service, controller, domain, dto, dao, ...)

## Was nun?

- Bislang wurde nur die Basisfunktionalität von Spring angeschaut
- Es gibt noch so viel mehr zu entdecken
- Zum Beispiel: AOP from Spring Framework
- Spring Data: Zugriff auf Datenspeicher (relationale und NoSQL)
- Spring Security: Authentifizierung und Autorisierung, Zugriffsschutz
- Spring Cloud: Unterstützung für verteilte Systeme (Konfiguration, Service Discovery, Resilienz, ...)

# Spring Data

1. **Spring Data Allgemein**
2. Spring Data Repositories
3. Spring Data JDBC
4. Spring Data JPA

# Einführung in Spring Data (1)

- Vertrautes, konsistentes, spring-basiertes Programmiermodell für Datenzugriffe für verschiedenste Arten von Speichern
  - trotzdem auch noch spezifische Funktionen verfügbar
- Abstrahiert Object-Relational-Mapping (ORM)
- Unterstützt Transaktionsmanagement
- Unterstützt Auditing
- Bietet die Möglichkeit, Abfragen basierend auf dem Methodennamen automatisch abzuleiten

```
long countByLastname(String lastname);  
User findByEmailAddress(EmailAddress emailAddress);
```

## Einführung in Spring Data (2)

- Breite Unterstützung von Datenspeicherframeworks:
  - JDBC
  - JPA
  - LDAP
  - MongoDB
  - KeyValue
  - ElasticSearch
  - ...

# Spring Data

1. Spring Data Allgemein
2. **Spring Data Repositories**
3. Spring Data JDBC
4. Spring Data JPA

# Repositories

- `@Repository<T, ID>` als Markerinterface
- `@CrudRepository<T, ID>` erweitert `@Repository` um Standard-CRUD Operationen
- `@PagingAndSortingRepository<T, ID>` erweitert `@CrudRepository` um Sortierung und Paganierung

# Eigene Repositories

- Bei der Verwendung der vorgefertigten Interfaces werden **alle** Methoden exponiert
  - Risiko, dass jemand sie nutzt, obwohl der direkte Zugriff nicht erwünscht ist
  - Risiko, dass man durch die Benutzung unerwünschte Seiteneffekte erhält
  - Lösung: eigene Interfaces bereitstellen, die mindestens von `@Repository` erben

```
@NoRepositoryBean
interface RestrictedRepository<T, ID> extends Repository<T, ID> {
    Optional<T> findById(ID id);
    <S extends T> S save(S entity);
}

interface UserRepository extends RestrictedRepository<User, Long> {
    User findByEmailAddress(String emailAddress);
}
```

# Kombinieren von Modulen

- Es ist möglich, dass man verschiedene Module (JPA, MongoDB, ElasticSearch, ...) gleichzeitig in einer Anwendung nutzen möchte
- Dazu muss eine eindeutige Zuordnung von Repository zu Modul möglich sein

```
@Entity  
class Person { ... }  
  
interface PersonRepository extends Repository<Person, Long> { ... }  
  
@Document  
class User { ... }  
  
interface UserRepository extends Repository<User, Long> { ... }
```

- Sonst: Fehler beim Start

# Queries

# Definition von Queries (1)

- manuelle Definition mit @Query

```
public interface UserRepository extends JpaRepository<User, Long> {  
    @Query("select u from User u where u.emailAddress = ?1")  
    User findByEmailAddress(String emailAddress);  
}
```

- implizite Definition über den Methodennamen

```
interface PersonRepository extends Repository<Person, Long> {  
    List<Person> findByEmailAddressAndLastname(String emailAddress, String lastname);  
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);  
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);  
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);  
}
```

## Definition von Queries (2)

- Erkennt verschiedene Methodennamen
  - find, read, query, count, get, ...
  - Between, LessThan, Like, ...
  - And, Or
  - OrderBy, Asc/Desc
  - Distinct
  - First, First10, Top, ...
- Kann mit verschachtelten Properties umgehen
  - Person -> address -> zipCode

```
List<Person> findByAddressZipCode(ZipCode zipCode);  
List<Person> findByAddress_ZipCode(ZipCode zipCode);
```

## Definition von Queries (3)

- Je nach Typ des Rückgabewerts, wird ein oder mehrere Ergebnisse geliefert
- Bestimmte Parameter werden automatisch verarbeitet: Sort, Pageable
  - Achtung: diese dürfen **NIE** null sein
  - Notfalls Sort.unsorted() oder Pageable.unpaged() übergeben
  - Paginierung braucht im Zweifel eine count-Abfrage: Slice statt Page als Rückgabewert, wenn Gesamtzahl nicht wichtig ist

## Definition von Queries (4)

- Rückgabetypen `Iterable`, `List` und `Set` sind für Collections möglich
- von Spring kommt noch `Streamable`
  - Concatenation, Filtern, Mappen von Ergebnissen
  - Zusätzliche Funktionen für nicht-parallele Streams
- muss sauber geschlossen werden, da es sonst zu Problemen kommt

# Validierung

- zusätzlich zu javax.validation.\*
- Saubere Prüfung, ob null als Ein- oder Ausgabe zugelassen ist
  - Bei Verstoß gibt es einen Fehler
- @NotNullApi am Interface/Klasse bedeutet, dass weder Ein- noch Ausgaben null sein dürfen
- @NotNull an einem Rückgabewert oder Parameter sagt, dass dieser niemals null sein darf
- @Nullable an einem Rückgabewert oder Parameter sagt, dass dieser auch null sein darf
- Collections, Iterable als Rückgabewerte etc werden niemals null sein, sondern maximal leer
- mit Optional als Rückgabewert, kann man elegant Null-Handling umsetzen

# Spring Data

1. Spring Data Allgemein
2. Spring Data Repositories
3. **Spring Data JDBC**
4. Spring Data JPA

# Spring Data JDBC

- Sehr Low-Level
- Wenn man eine Entity lädt, werden eine Reihe von Statements ausgeführt, danach ist alles geladen
  - Kein Caching
  - Kein Lazy-Loading
- Wenn man eine Entity speichert, wird sie auch gespeichert
  - Wenn nicht, dann nicht
  - Kein Dirty-Checking
- Das Modell, wie Entities auf Tabellen abgebildet werden ist einfach und hat nicht viel Spielraum für Anpassungen

# Aktivierung von Spring Data JDBC (1)

- Einbinden des Moduls

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

- `@EnableJdbcRepositories` an einer Konfiguration

# Aktivierung von Spring Data JDBC (2)

```
@Configuration  
@EnableJdbcRepositories  
class ApplicationConfig {  
  
    @Bean  
    public DataSource dataSource() {  
        ...  
    }  
}
```

# Spring Data JDBC Objekt Mapping

- Die Prinzipien sind einfach, können aber von konkreten Speichern angepasst werden
  1. Erzeugen einer Instanz, je nach verfügbaren Konstruktoren
    1. Konstruktor ohne Argumente
    2. Einziger Konstruktor
    3. Der mit `@PersistenceConstructor` annotierte
  2. Befüllen aller Attribute
    1. unveränderlich, aber `wither`: neue Instanz mit gesetztem Wert
    2. Setter existiert: Setter wird aufgerufen
    3. Feld direkt setzen

# Empfehlungen für das Objekt Mapping

- Unveränderliche Objekte benutzen
- All-Args Konstruktor anbieten
- Factory Methoden benutzen
- Verschiedene Varianten des Objekt Mapping sind schneller: deren Bedingungen einhalten
- Der Einsatz von Lombok reduziert den Boilerplate-Code

# Unterstützte Typen

- Primitive und ihre Klassenäquivalente
- Enum: wird auf Namen abgebildet
- String
- Date und java.time.\* Klassen
- Alles, was die Datenbank unterstützt
- Referenzen nur als Embedded, 1-1, 1-n, aber nicht n-1 oder n-m
- Eigene Konverter sind möglich

# Entity Mapping

- Es gibt eine Standard **NamingStrategy**, die aber angepasst werden kann
- **@Table** weist einer Entity einen Tabellennamen zu (an Entity)
- **@Column** weist einer Property einen Spaltennamen zu (an Property)
- **@MappedCollection** weist einer 1-n zu, welche Spalte der Relation die Referenz auf diese Entity hält
- Bei Maps gibt es neben der **idColumn** noch die **keyColumn**
- **@Embedded** für eingebettete Value-Objekte
  - mit spezieller Behandlung von leeren Werten: **@Embedded . Empty** vs. **@Embedded . Nullable**

# Queries

- Per Annotation `@Query`
- Keine impliziten Abfragen
- Nur benannte Parameter (oder Compiler-Flag `-parameters`)

```
@Query("select firstName, lastName from User u where u.emailAddress = :email")
User findByEmailAddress(@Param("email") String email);
```

# Transactions

- CRUD Methoden sind automatisch transaktional
  - Zum Ändern der Transaktionalität einfach selbst neu deklarieren
- Bei der Verwendung von Services, die mehrere Operationen in einer Transaktion zusammenfassen sollen: `@Transactional` an Methode
  - `@EnableTransactionManagement` an der Konfiguration
  - Achtung: Proxy-Falle!
- Wenn nur lesender Zugriff: `readOnly=true`
  - Verwendung von `@Modifying` überschreibt das
  - ist nur ein Hint an den Treiber, dass hier optimiert werden könnte

# Spring Data

1. Spring Data Allgemein
2. Spring Data Repositories
3. Spring Data JDBC
4. **Spring Data JPA**

# JPA

# Einführung in JPA

- Sehr starke Abstraktionsschicht für Object-Relational-Mapping (ORM)
- Standardisierung als JSR 220, aktuell Version 2.1 (2013)
- Definiert die API, Entities, Metadaten für Relationen und eine Abfragesprache (JPQL)

# Domänenmodell

- Entities
  - Eigenständige Objekte
  - Eindeutig identifizierbar
  - Haben einen Lebenszyklus
- Werteobjekte
  - Treten nur als Bestandteil eines anderen Modellelements auf
  - Haben keine ID
  - Haben auch keinen Lebenszyklus
  - Basistypen (int, Integer, ...), Embeddable Types (ValueTypes nach DDD), Collections

# Entities

- Basis sind einfache POJOs
- @Entity Annotation, ggfs. mit Name
- Klassen dürfen nicht final sein
- Angabe der relevanten Tabelle mit @Table möglich, sonst gilt Default

```
@Entity  
@Table(name="CAR")  
public class Car {  
...  
}
```

# Vererbung

- Es gibt verschiedene Strategien, Vererbung abzubilden

```
@Entity  
@Data  
public class Konto {  
    @Id  
    private Long id;  
    private String iban;  
    private BigDecimal balance;  
}  
  
@Entity  
@Data  
public class GiroKonto extends Konto {  
    private BigDecimal dispo;  
}
```

# SINGLE\_TABLE

- Alle Attribute aller Klassen der Hierarchie
- Zusätzlicher Diskriminator

```
create table Konto (
    DTYPE varchar(31) not null,
    id bigint not null,
    balance numeric(19,2), i
    ban varchar(255),
    dispo numeric(19,2),
    primary key (id))
```

# JOINED

- Tabelle der Basisklasse enthält ihre Attribute
- Pro Vererbung Tabelle mit neuen Attributen
- Gejoint über ID

```
create table GiroKonto (
    dispo numeric(19,2),
    id bigint not null,
    primary key (id))

create table Konto (
    id bigint not null,
    balance numeric(19,2),
    iban varchar(255),
    primary key (id))
```

# TABLE\_PER\_CLASS

- Jede Klasse hat eine eigene Tabelle
- Jede Tabelle enthält alle aufgelösten Attribute der Klasse

```
create table GiroKonto (
    id bigint not null,
    balance numeric(19,2),
    iban varchar(255),
    dispo numeric(19,2),
    primary key (id))
```

```
create table Konto (
    id bigint not null,
    balance numeric(19,2),
    iban varchar(255),
    primary key (id))
```

# Anders herum

```
@Data  
@MappedSuperclass  
public class CarPart {  
  
    @Id  
    private Long id;  
    private long serial;  
}  
  
@Entity  
@Data  
public class Engine extends CarPart {  
    private int ps;  
}  
  
@Entity  
@Data  
public class Wheel extends CarPart {  
    private int diameter;  
}
```

# SQL

```
create table Engine (
    id bigint not null,
    serial bigint not null,
    ps integer not null,
    primary key (id))
```

```
create table Wheel (
    id bigint not null,
    serial bigint not null,
    diameter integer not null,
    primary key (id))
```

# Identifier

- Attribut mit Annotation @Id
  - Kann an verschiedenen Attributen stehen - dann muss aber eine @IdClass angegeben werden
  - Bei Multi-Property IDs kann auch ein Embedded Value Type benutzt werden
- Kann gesetzt werden oder automatisch erzeugt

```
@GeneratedValue(strategy=GenerationType.AUTO)
```

- AUTO: Eine globale Sequenz für alle Objekte
- IDENTITY: Eine Sequenz pro Klassenhierarchie
- SEQUENCE: Definition und Verwendung einer nativen Sequenz
- TABLE: Verwendung einer Tabelle für die Sequenzen

# Spaltendefinition

- einfache Werte können per @Column Annotation definiert werden

```
@Column(name="BRAND", length=50, nullable=false, unique=false)
```

- für Optimierung kann insertable oder updatable angegeben werden
- Optional auch SQL Statement für Spaltendefinition, explizit die Tabelle und noch viel mehr

## Besondere Annotationen für Attribute

- Eine Entity kann Werte enthalten, die berechnet sind oder aus anderen Gründen nicht gespeichert werden sollen: `@Transient`
- Für die Speicherung von Datumswerten: `@Temporal`, mit Angabe der Granularität (z.B. `TemporalType.TIME`)
- Für Enumerationen: `@Enumerated`, mit Angabe ob nach Name `EnumType.STRING` (sonst ordinal, problematisch bei Veränderungen)
- Für alle nicht-relationalen Typen wird automatisch `@Basic` angenommen, muss also nicht explizit annotiert werden
  - explizit verwenden für Überschreiben von `nullable` und Ladeverhalten

# Relationen

- `@JoinColumn/@JoinTable`: Definiert die Spalte(n) (und ggfs. die Tabelle), über die ein Join erstellt wird
- `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`: Legt die Art der Relation fest
- `mappedBy`: Legt für bidirektionale Relationen fest, welches das Attribut der Gegenseite ist, das die Relation hält
- `fetch`: Sollen die relationalen Daten auf jeden Fall (`EAGER`, Standard), oder möglichst spät (`LAZY`) geladen werden?
- `cascade`: Welche Operationen sollen kaskadiert werden?
- `orphanRemoval`: sollen verwaiste Entitäten aufgeräumt werden?

# Einseitige 1:n Beziehung

```
@Data  
@Entity  
public class Bank {  
    @Id  
    private Long id;  
    private String name;  
    private String bic;  
  
    @OneToMany  
    private List<Konto> accounts;  
}
```

```
create table Bank (  
    id bigint not null,  
    bic varchar(255),  
    name varchar(255),  
    primary key (id))  
create table Bank_Konto (  
    Bank_id bigint not null,  
    accounts_id bigint not null)  
create table Konto (  
    DTTYPE varchar(31) not null,  
    id bigint not null,  
    balance numeric(19,2),  
    iban varchar(255),  
    dispo numeric(19,2),  
    primary key (id))  
alter table Bank_Konto add constraint UK_6h84kywh8eg6myt2u7i3s19nn unique (accounts_id)
```

# Bidirektionale 1:n Beziehung

- erzeugt dieselbe DDL wie n:1

```
@Entity  
@Data  
public class Konto {  
    ...  
  
    @ManyToOne  
    private Bank bank;  
}  
  
@Data  
@Entity  
public class Bank {  
    ...  
  
    @OneToMany(mappedBy = "bank")  
    private List<Konto> accounts;  
}
```

```
create table Konto (  
    DTYPYE varchar(31) not null,  
    id bigint not null,  
    balance numeric(19,2),  
    iban varchar(255),  
    dispo numeric(19,2),  
    bank_id bigint,  
    primary key (id))
```

# Lebenszyklus-Methoden

- Annotierte Methoden, die zu bestimmten Events aufgerufen werden
- @PrePersist/@PostPersist: vor/nach dem Persistieren eines Entity
- @PreRemove/@PostRemove: vor/nach dem Löschen einer Entity
- @PreUpdate/@PostUpdate: vor/nach dem Aktualisieren einer Entity
- @PostLoad: Nach dem Laden einer Entity

```
@PrePersist  
protected void onCreate() {  
    // z.B. ID erzeugen  
}  
  
@PreUpdate  
protected void onUpdate() {  
    // z.B. Änderungsdatum setzen  
}
```

# Embedded/Embeddable (1)

- Wird als Bestandteil der übergeordneten Entität gespeichert
- Elegant über Lombok @Value

```
@Value  
@Embeddable  
public class Mandant {  
    String name;  
    int year;  
}
```

```
@Entity  
@Data  
public class ProduktKonto {  
    @Id  
    private Long id;  
  
    @Embedded  
    private Mandant mandant;  
    private BigDecimal saldo;  
}
```

```
create table ProduktKonto (  
    id bigint not null,  
    name varchar(255),  
    year integer not null,  
    saldo numeric(19,2),  
    primary key (id))
```

# Embedded/Embeddable (2)

- Ggf. ist Disambiguierung von Spaltennamen notwendig

```
@Embeddable  
@Value  
public class Embedded {  
  
    private String lastname;  
    private String firstname;  
}
```

```
@Entity  
@Data  
public class Embedding {  
    @Id  
    private Long id;  
    private String firstname;  
  
    @AttributeOverrides(  
        @AttributeOverride(name = "firstname", column = @Column(name = "FIRST_FIRSTNAME")))  
    private Embedded first;  
}
```

```
create table Embedding (  
    id bigint not null,  
    FIRST_FIRSTNAME varchar(255),  
    lastname varchar(255),  
    firstname varchar(255),  
    primary key (id))
```

# Embedded/Embeddable (3)

- Kann als Identifier für Entitäten benutzt werden

```
@Embeddable  
@Value  
public class Mandant implements Serializable {  
    String mandant;  
    int year;  
}
```

```
@Entity  
@Data  
@IdClass(Mandant.class)  
public class ProduktKonto {  
  
    @Id  
    private String mandant;  
    @Id  
    private int year;  
    private BigDecimal saldo;  
}
```

```
@Entity  
@Data  
public class ProduktKonto {  
    @EmbeddedId  
    private Mandant mandant;  
    private BigDecimal saldo;  
}
```

## Embedded/Embeddable (4)

```
create table ProduktKonto (
    mandant varchar(255) not null,
    year integer not null,
    saldo numeric(19,2),
    primary key (mandant, year))
```

```
@Query("select p from ProduktKonto p where p.mandant = :mandant")
public List<ProduktKonto> getByMandant(Mandant mandant);
```

```
@Query("select p from ProduktKonto p where p.mandant.mandant = :mandant")
public List<ProduktKonto> getByMandant(String mandant);
```

## Übung: Entities bauen

1. ***git pull*** und schaue in das Verzeichnis ***13\_entities***
2. Implementiere die beiden Aufgaben aus der Readme

# JPA @ Spring Data

# Aktivierung von Spring Data JPA (1)

- Einbinden des Moduls

```
<dependencies>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-jpa</artifactId>
    </dependency>
<dependencies>
```

- `@EnableJpaRepositories` an einer Konfiguration
- `@EnableTransactionManagement` an einer Konfiguration

# Aktivierung von Spring Data JPA (2)

```
@Configuration
@EnableJpaRepositories
@EnableTransactionManagement
class ApplicationConfig {

    @Bean
    public DataSource dataSource() {
        ...
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource) {
        LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();
        factory.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
        factory.setPackagesToScan("inc.monster.app");
        factory.setDataSource(dataSource);
        return factory;
    }

    @Bean
    public PlatformTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {
        JpaTransactionManager txManager = new JpaTransactionManager();
        txManager.setEntityManagerFactory(entityManagerFactory);
        return txManager;
    }
}
```

# Entity Status Erkennung

- Frage: Ist die Entity schonpersistiert oder neu?
- @Version und @Id: erst Version prüfen (null = neu), ansonsten Id (null = neu), ansonsten nicht neu
  - Problem bei manuell vergebenen Ids
- Implementieren von **Persistable**
  - Transienter Wert, der die Information hält, ob neu
- Implementieren von **EntityInformation**

```
@Entity
class PersistableWorker implements Persistable {

    @Id
    private String personalId;
    private String name;

    @Transient
    private boolean isNew = true;

    public PersistableWorker(String personalId) {
        this.personalId = personalId;
    }

    @PrePersist
    @PostLoad
    protected void markNotNew() {
        this.isNew = false;
    }

    @Override
    public boolean isNew() {
        return isNew;
    }
}
```

# Queries

- Alle impliziten Queries, die auch allgemein funktionieren
  - Und einige mehr: siehe [Doku](#)
- Explizite Queries mit @Query in JPQL an Methoden
- Explizite Queries mit @Query in nativem SQL an Methoden
  - nativeQuery=true, dann aber keine automatische Sortierung/Paginierung
  - Möglich durch Angabe einer countQuery
- @NamedQuery in JPQL an Entities für wiederverwendbare Abfragen, an Methoden referenziert

# Sortierung

- JPA erlaubt standardmäßig keine Funktionsaufrufe in der Sortierung
  - Entweder Funktion im Select ausführen und nach Wert sortieren

```
@Query("select u.id, LENGTH(u.firstname) as fn_len from User u where u.lastname like ?1%")
...
repo.findByLastNameAndSort("edda", new Sort("fn_len"));
```

- Oder `JpaSort.unsafe()` verwenden

```
@Query("select u from User u where u.lastname like ?1%")
...
repo.findByLastNameAndSort("jones", JpaSort.unsafe("LENGTH(firstname)"));
```

# Parameter

- Parameter einer Query werden standardmäßig der Reihenfolge nach eingesetzt

```
@Query("update User u set u.firstname = ?1 where u.lastname = ?2")
int setFixedFirstnameFor(String firstname, String lastname);
```

- unübersichtlich
- macht das Refactoring schwierig
- @Param gibt einem Methodenparameter einen Parameternamen in der Query

```
@Query("update User u set u.firstname = :firstname where u.lastname = :lastname")
int setFixedFirstnameFor(@Param("firstname") String firstname, @Param("lastname") String lastname);
```

- seit Java 8 per Compilerflag -parameters kann @Param auch entfallen

```
@Query("update User u set u.firstname = :firstname where u.lastname = :lastname")
int setFixedFirstnameFor(String firstname, String lastname);
```

## @Modifying

- generell nur notwendig, wenn an einer Methode mit @Query
  - bei allen anderen weiß Spring, ob lesend oder nicht
- EntityManager hat danach ggfs. veraltete Daten, die (noch) nicht gespeichert werden sollen
  - EntityManager.clear() wird daher nicht automatisch aufgerufen
  - kann durch Angabe von clearAutomatically=true erzwungen werden
- Methoden, die modifizieren können void, int oder boolean liefern

# Implizite Deletes

```
void deleteByYear(int year);

@Modifying
@Query("delete from Accounts a where a.year = ?1")
void deleteByYear2(int year);
```

- `deleteByYear`
  - implizit
  - erst ein Select, dann einzelne Delete
  - `EntityManager` bekommt Events zum Lifecycle und weiß damit von der Lösung
- `deleteByYear2`
  - explizit
  - nur eine Delete Query
  - `EntityManager` bekommt keine Events und weiß nichts von der Löschung

# Zugriffsoptimierung

- Früher konnte man statisch festlegen, ob Relations sofort oder bei Bedarf geladen werden sollten (@FetchType)
- Heute gibt es den @EntityGraph
  - Verwendung an der Query-Methode
  - Angabe der zu ladenden Attributen
  - Angabe des Typ (Fetch oder Load)
- @NamedEntityGraph an Entities (analog @NamedQuery)
  - Referenz an Methode mit Name und Typ

## Übung: Speichere Autos

1. ***git pull*** und schaue in das Verzeichnis ***14\_jpa***
2. Implementiere die beiden Aufgaben aus der Readme

# Projections

# Einführung in Projections

- Sind dedizierte Typen für die selektive Rückgabe von Werten

```
interface PersonRepository extends Repository<Person, UUID> {  
    Collection<NamesOnly> findByLastname(String lastname);  
}
```

- Klassenbasiert
- Interface-basiert

# Klassenbasierte Projections

- Keine Rekursion möglich
- Direktes Setzen der Attribute
- Sehr elegant mit Lombok

```
@Value  
class NamesOnly {  
    String firstname, lastname;  
}
```

# Interface-basierte Projections

- Rekursion ist möglich, da lediglich ein Proxy auf die Werte verwendet wird
- "geschlossen": alle ausgewählten Attribute werden 1:1 übernommen
- "offen": kann Berechnungen enthalten

# "Geschlossene" Projections

```
interface NamesOnly {  
    String getFirstname();  
    String getLastname();  
}
```

# "Offene" Projections

```
interface NamesOnly {  
    @Value("#{target.firstname + ' ' + target.lastname}")  
    String getFullName();  
    ...  
}
```

- Komplexität steigt, liegt in Annotation
- Funktionalität ist eingeschränkt
- Etwas Abhilfe schaffen Java 8 Default Methoden
- Dank SpEL auch Zugriff auf Beans möglich

# Dynamische Projections

- Über Generics kann der Rückgabetyp verändert werden

```
interface PersonRepository extends Repository<Person, UUID> {  
    <T> Collection<T> findByLastname(String lastname, Class<T> type);  
}
```

# Specifications

# Specifications (1)

- dynamische Spezifikation einer Where-Klausel
- können über Fluent-API verknüpft werden
- zur Verwendung muss das Repository noch den `JpaSpecificationExecutor` erweitern
- Implementierung von `Specification.toPredicate()` liefert JPA Query (Criteria)
  - macht v.a. in Kombination/Wiederverwendbarkeit Sinn

# Specifications (2)

```
public static Specification<Customer> isLongTermCustomer() {  
    return new Specification<Customer>() {  
        public Predicate toPredicate(Root<Customer> root, CriteriaQuery<?> query,  
                                     CriteriaBuilder builder) {  
            LocalDate date = new LocalDate().minusYears(2);  
            return builder.lessThan(root.get(Customer_.createdAt), date);  
        }  
    };  
}
```

```
List<Customer> customers = customerRepository.findAll(isLongTermCustomer());  
List<Customer> customers = customerRepository.findAll(isLongTermCustomer().and(hasBirthday()));
```

# Query By Example

## Query by Example (1)

- Pro: funktioniert auch bei häufigem Refactoring
- Pro: funktioniert unabhängig vom Datenspeicher
- Con: wenig Flexibilität beim Matching
- Con: keine Gruppierung von Bedingungen möglich
- Con: funktioniert nur mit Einzelwerten
- Repository muss `QueryByExampleExecutor` erweitern

## Query by Example (2)

- Example enthält Beispielobjekt und optional Vergleichsvorschriften

```
Person person = new Person();
person.setFirstname("Dave");
```

```
Example example1 = Example.of(person);
```

```
ExampleMatcher matcher = ExampleMatcher.matching()
    .withIgnorePaths("lastname")
    .withIncludeNullValues()
    .withStringMatcherEnding();
```

```
Example example2 = Example.of(person, matcher);
```

## Query by Example (3)

- ExampleMatcher
  - "all" vs. "any"
  - Null-Handling
  - Groß-/Kleinschreibung
  - Property-Pfade

# Transaktionen

- Bei Verwendung von CrudRepository werden Transaktionen automatisch verwendet
  - Zum Ändern der Transaktionalität einfach selbst neu deklarieren
- Bei der Verwendung von Services, die mehrere Operationen in einer Transaktion zusammenfassen sollen: @Transactional an Methode
  - Achtung: Proxy-Falle!
- Wenn nur lesender Zugriff: readOnly=true
  - Verwendung von @Modifying überschreibt das
  - ist nur ein Hint an den JPA/Treiber, dass hier optimiert werden könnte

# Locking

- Verschiedene Locks
  - NONE, READ, WRITE
  - Optimistisches und pessimistisches Locking
- Optimistisches Locking benötigt Versions-Attribut
- @Lock an Query-Methode
  - für Standardmethoden, die Locking bekommen sollen, neu deklarieren

# Automatisches Auditing

- Unterstützung durch Annotation an Entity
  - `@CreatedDate`: Erstellungsdatum, wird beim ersten Speichern gesetzt
  - `@CreatedBy`: Ersteller der Entity
  - `@LastModifiedDate`: Datum der letzten Änderung
  - `@LastModifiedBy`: Letzter Bearbeiter
- Woher kommt der Benutzer?
  - Implementierung von `Auditable`
  - Bereitstellung von `AuditorAware`
- Aktivierung per Annotation und Konfigurationsanpassung

# Spring Boot Reactive Webservices

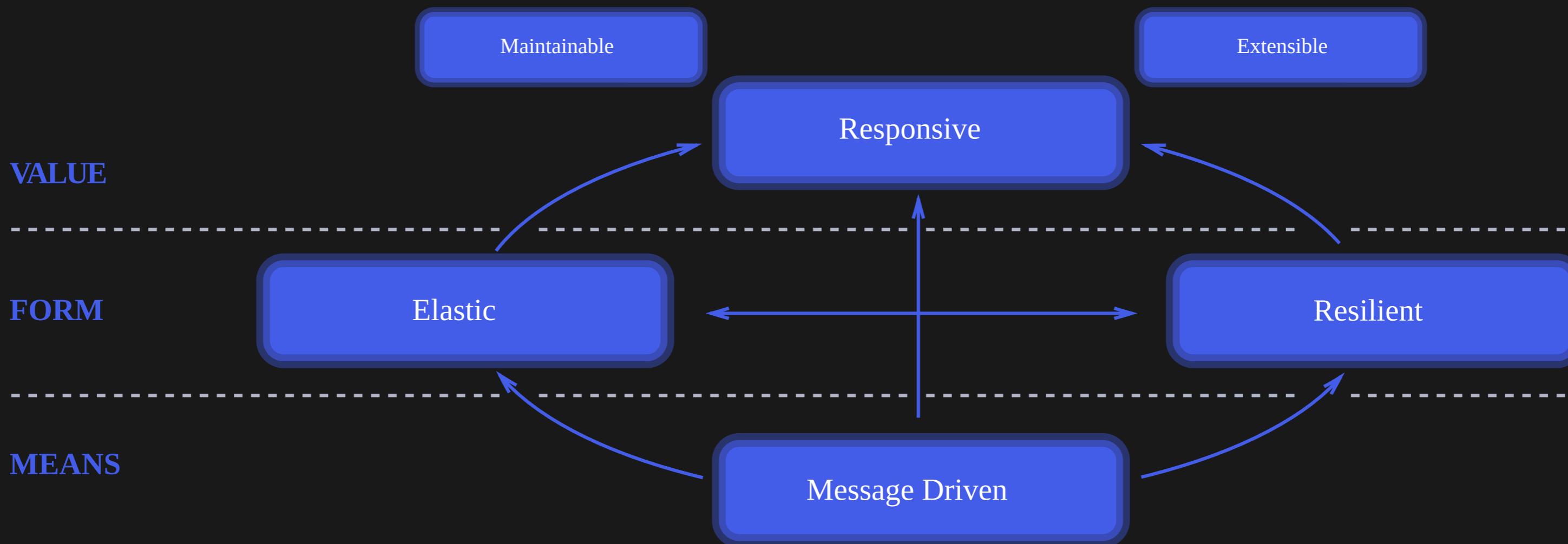
1. **Reactive im Allgemeinen**
2. Reactive in Spring Boot
3. Reactive Server mit Spring
4. Reactive Client mit Spring

# Einführung in Reactive

- Reactive Programmierung betrachtet Datenströme und reagiert auf Veränderungen
- Erste kommerzielle Nutzung durch spreadsheets! (VisCalc, 1979)
- Kaum akademische Beachtung bis in die späten 90er Jahre
- Erlangt Popularität in den 2010er durch
  - JavaScript UI frameworks: Knockout, Ember, Meteor
  - aktuell: React, Vue, Angular
- The Reactive Manifesto (2014)

# The Reactive Manifesto (1)

- Erstellt von J. Bonér, D. Farley, R. Kuhn und M. Thompson. (~31k Unterschriften)
- Definiert eine "reaktive"-Systemarchitektur mit folgenden Eigenschaften



# The Reactive Manifesto (2)

- **Responsive**
  - Responsive Systeme antworten rechtzeitig, wenn überhaupt möglich
    - Fokus auf schnellen und konsistenten Antwortzeiten
    - Etablieren von zuverlässigen Obergrenzen, um konsistente gute Qualität zu Gewährleisten
  - Zweck von Responsitität ist:  
erleichterte Fehlerbehandlung, schaffen von Benutzervertrauen und Ermutigung zu weitere Interaktionen
- **Resilient**
  - System bleibt im Falle eines Ausfalls responsive
  - Wird errichtet durch *Replikation, Containment und Isolierung*, sowie *Delegation*
  - Nicht nur systemkritische Systeme, sondern "Alle"

# The Reactive Manifesto (3)

- **Elastic**
  - System bleibt Responsive unter variierender Arbeitslast
  - Erweitern oder reduzieren allokierten Ressourcen, gemäß aktueller Arbeitslast
  - Elastizität wird stets kostensparend umgesetzt
- **Message Driven**
  - System ist auf asynchrone Nachrichtenzustellung angewiesen
  - Errichtet eine Abgrenzung zwischen den Komponenten
  - Erlaubt Lose Kopplung, Isolierung und Standorttransparenz

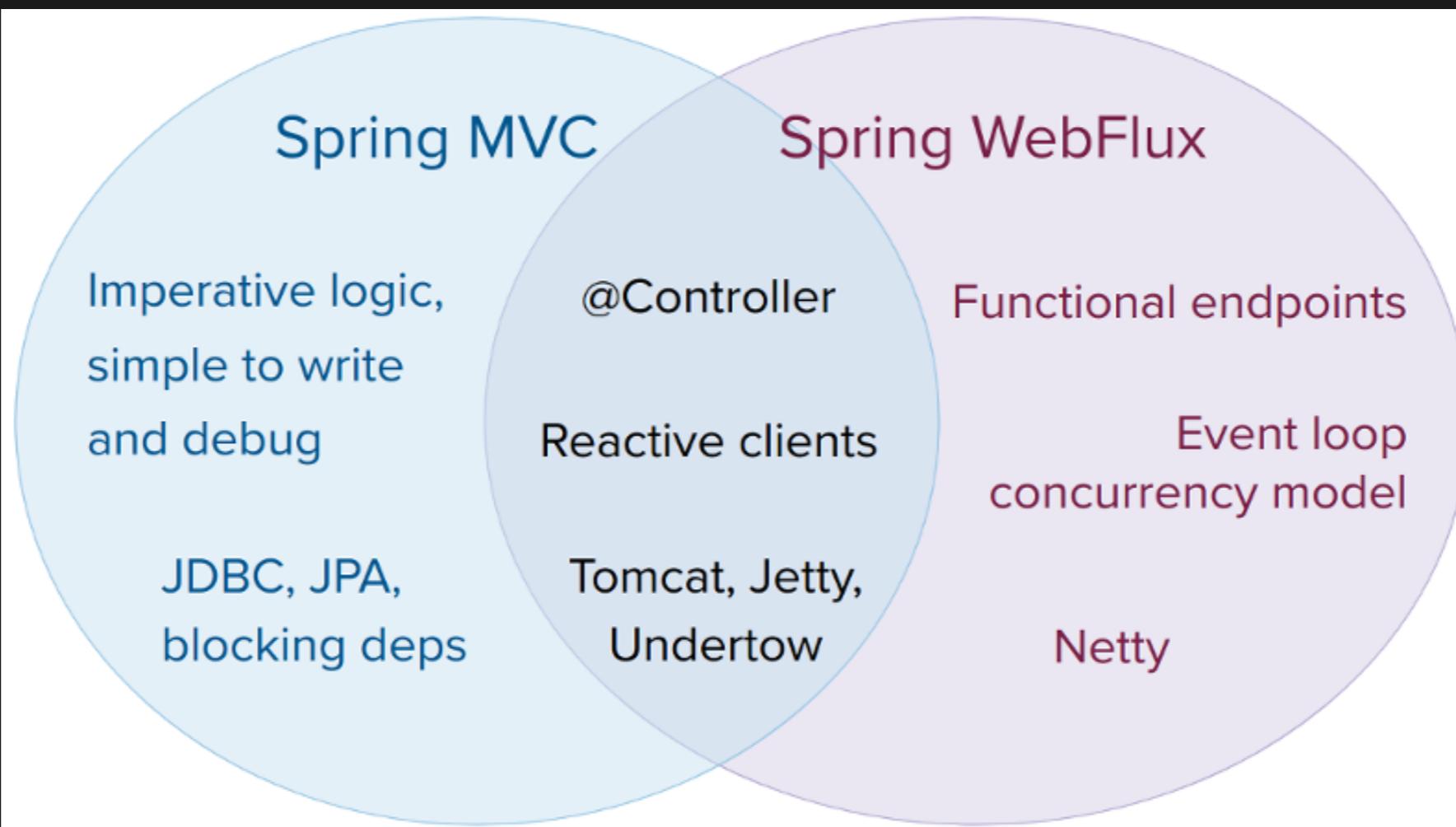
1. Reactive im Allgemeinen
2. **Reactive in Spring**
3. Reactive Server mit Spring
4. Reactive Client mit Spring

# Reative in Spring

- Spring Webflux erlaubt erstellen nicht-blockierender asynchroner Anwendungen
- Unterstützt resilente Applikationen durch die Inklusion von Back-Pressure
- Kann moderne Multicore-Prozessoren besser nutzen als klassische MVC Anwendungen
- Client und Server kommunizieren über
  - Ajax und HTTP via Streaming oder Long Polling
  - Websockets
- Nutzt die Bibliothek Project Reactor, welche das Publisher Pattern

# Vergleich Spring MVC vs Spring Webflux

- Spring MVC und Spring Webflux sind nicht miteinander kompatibel
- Spring Webflux ist größtenteils API kompatibel zu MVC



# Webflux Terminologie (1)

- **Mono**: Ein Publisher der 0 oder 1 Element emittiert

```
Mono<String> mono = Mono.just("John");
Mono<Object> monoEmpty = Mono.empty();
Mono<Object> monoError = Mono.error(new Exception());
```

## Webflux Terminologie (2)

- **Flux**: Ein Publisher, der 0 bis N Elemente aussendet, die für immer weiter ausgesendet werden können. Er gibt eine Folge von Elementen zurück und sendet eine Benachrichtigung, wenn er die Rückgabe aller seiner Elemente abgeschlossen hat.

```
Flux<Integer> flux = Flux.just(1, 2, 3, 4);
Flux<String> fluxString = Flux.fromArray(new String[]{"A", "B", "C"});
Flux<String> fluxIterable = Flux.fromIterable(Arrays.asList("A", "B", "C"));
Flux<Integer> fluxRange = Flux.range(2, 5);
Flux<Long> fluxLong = Flux.interval(Duration.ofSeconds(10));
```

```
List<String> dataStream = new ArrayList<>();
Flux.just("X", "Y", "Z")
    .log()
    .subscribe(dataStream::add);
```

# Webflux Operators

- **Map** - Wird verwendet, um von einem Element in ein anderes zu transformieren
- **FlatMap** - Transformiert die ausgegebenen Elemente asynchron in Publisher, und fasst diese inneren Publisher durch Zusammenführen zu einem einzigen Flux zusammen
- **FlatMapMany** - Mono-Operator, der verwendet wird, um ein Mono-Objekt in ein Flux-Objekt zu transformieren
- **DelayElements** - Verzögert die Veröffentlichung der einzelnen Elemente um eine bestimmte Zeit
- **Concat** - Wird verwendet, um die von Publishern ausgegebenen Elemente zu kombinieren, wobei die Reihenfolge der Publisher intakt bleibt.
- **Merge** - Wird verwendet, um die von Publishern ausgegebenen Elemente zu kombinieren, ohne ihre Reihenfolge beizubehalten.
- **Zip** - Wird verwendet, um zwei oder mehr Publisher zu kombinieren, indem man wartet, bis alle Quellen ein Element emittieren, und diese Elemente zu einem Ausgabewert kombiniert.

1. Reactive im Allgemeinen
2. Reactive in Spring
3. **Reactive Server mit Spring**
4. Reactive Client mit Spring

# Repositories

- `@Repository<T, ID>` als Markerinterface, aquivalent zu Spring Data
- `@ReactiveCrudRepository<T, ID>` erweitert `@Repository` um Standard-CRUD Operationen
- `@ReactiveSortingRepository<T, ID>` erweitert `@ReactiveCrudRepository` um Sortierung und Paginierung
- **Achtung!** Implementierung für Relationale Datenbanken noch nicht vollaugereift

# Definition von Queries

- manuelle Definition mit @Query

```
public interface UserRepository extends Repository<User, Long> {  
    @Query("select u from User u where u.emailAddress = ?1")  
    Mono<User> findByEmailAddress(String emailAddress);  
}
```

- implizite Definition über Methodennamen

```
interface PersonRepository extends Repository<Person, Long> {  
    Flux<Person> findByEmailAddressAndLastname(String emailAddress, String lastname);  
    Flux<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);  
    Flux<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);  
    Flux<Person> findByLastnameOrderByFirstnameAsc(String lastname);  
}
```

- **Achtung!** Mono und Flux Producer holen bzw. manipulieren Daten erst wenn eine aktive Subscription vorhanden ist.

# Reactive Controller (1)

- Implementierung durch Annotierte Controller (wie Spring MVC) oder funktionale Endpunkte
- Webflux konvertiert Daten in einen valide Antwort, abhängig des Accept-Headers des Clients
  - application/json - Blockierende Anfrage. Client erhält seine Antwort, wenn der Producer fertig ist
  - text/event-stream - Reactive Anfrage. Nutzt HTML SSE (Server-Sent Events). Client bekommt seine Antwort tröpfchenweise, sobald Daten sobald diese vorliegen.

## Reactive Controller (2)

```
@RestController
@RequestMapping(value = "/person")
public class PersonController {
    ...

    @GetMapping
    public Flux<Person> getAllPersons() {
        return personRepository.findAll();
    }

    @GetMapping("/{id}")
    public Mono<Person> getPerson(@PathVariable("id") Long id) {
        return personRepository.findById(id);
    }
}
```

**Achtung!** Im Lebenszyklus unserer API sollten keine blockierenden Aufrufe stattfinden.

# Reactive Controller (3)

```
HandlerFunction<ServerResponse> helloworld =  
    request -> ServerResponse.ok().bodyValue("Hello World");
```

```
public class PersonHandler {  
    ...  
  
    public Flux<ServerResponse> getAllPersons() {  
        Flux<Person> people = personRepository.findAll();  
        return ok().contentType(APPLICATION_JSON).body(people, Person.class);  
    }  
  
    public Mono<ServerResponse> getPerson(@PathVariable("id") Long id) {  
        int personId = Integer.valueOf(request.pathVariable("id"));  
        return personRepository.findById(personId)  
            .flatMap(person -> ok().contentType(APPLICATION_JSON).bodyValue(person))  
            .switchIfEmpty(ServerResponse.notFound().build());  
    }  
}
```

```
RouterFunction<ServerResponse> route = route()  
    .GET("/person", accept(APPLICATION_JSON), handler::getAllPersons)  
    .GET("/person/{id}", accept(APPLICATION_JSON), handler::getPerson)  
    .build();
```

1. Reactive im Allgemeinen
2. Reactive in Spring
3. Reactive Server mit Spring
4. **Reactive Client mit Spring**

# WebClient

- Webflux inkludiert WebClient, um reaktive HTTP Anfragen zu stellen
- WebClient bietet eine "fluent"-API basierend auf Project Reactor
- WebClient erlaubt den deklarative Aufbau von asynchroner Logik, ohne selbst mit Threads und Concurrency umzugehen
- WebClient ist komplett nicht-blockierend und unterstützt dieselben Codecs, wie die Serverseite
- WebClient benötigt eine HTTP client Bibliothek, um Anfragen zu stellen
  - Reactor Netty
  - Jetty Reactive HttpClient
  - Apache HttpComponents
- **Zur Erinnerung:** WebClient != Spring Webflux

# WebClient Erzeugung (1)

Der einfachste Weg einen WebClient zu Erzeugen, ist durch die statischen Methoden:

```
WebClient.create();  
  
WebClient.create(String baseUrl);
```

## WebClient Erzeugung (2)

Mittels `WebClient.builder()` können weitere Optionen angegeben werden:

- `uriBuilderFactory`: Anpassen der UriBuilderFactory, welche eine Base-Url bereitstellt
- `defaultHeader`: Headers für jede Anfrage
- `defaultCookie`: Cookies für jede Anfrage
- `filter`: Client-Filter für jede Anfrage, z.B. Logging oder Authentifizierung
- `exchangeStrategies`: HTTP-Message Reader/Writer Anpassung
- `clientConnector`: HTTP-Client Bibliothekeneinstellungen

# WebClient Anfragen (2)

`retrieve()` Method beschreibt, wie der Response extrahiert wird

- als `ResponseEntity`

```
WebClient client = WebClient.create("https://example.org");

Mono<ResponseEntity<Person>> result = client.get()
    .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
    .retrieve()
    .toEntity(Person.class);
```

- nur der Body

```
Mono<Person> result = client.get()
    .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
    .retrieve()
    .bodyToMono(Person.class);
```

- oder als Stream

```
Flux<Quote> result = client.get()
    .uri("/quotes").accept(MediaType.TEXT_EVENT_STREAM)
    .retrieve()
    .bodyToFlux(Quote.class);
```

## WebClient Anfragen (2)

`exchangeToMono()` und `exchangeToFlux()` bieten mehr Kontrolle über die Dekodierung

```
Mono<Person> entityMono = client.get()
    .uri("/persons/1")
    .accept(MediaType.APPLICATION_JSON)
    .exchangeToMono(response -> {
        if (response.statusCode().equals(HttpStatus.OK)) {
            return response.bodyToMono(Person.class);
        }
        else {
            // Turn to error
            return response.createException().flatMap(Mono::error);
        }
    });
});
```

# WebClient Tests (1)

```
Mono<String> result = this.webClient.get()
    .uri("/greeting")
    .retrieve()
    .bodyToMono(String.class);

StepVerifier.create(result)
    .expectNext("Hello Spring!")
    .expectComplete()
    .verify(Duration.ofSeconds(3));
```

## WebClient Tests (2)

```
Flux<Double> result = this.webClient.get()
    .uri("/temperatures")
    .retrieve()
    .bodyToFlux(Double.class);

StepVerifier.create(result)
    .expectNext(18.1)
    .expectNext(19.4)
    .expectNext(25.3)
    .expectNext(17.4)
    .expectComplete()
    .verify(Duration.ofSeconds(3));
```

# Übung: Erstellen einer Spring Webflux Anwendungen

1. ***git pull*** und schaue in das Verzeichnis ***15\_spring\_webflux***
2. Implementiere die Aufgabe aus der Readme

# WebFlux vs. Blocking (MVC):

## NUTZERERLEBNIS

- Für nicht paginierte und große Seiten bietet Reactive einen großen Vorteil, da Daten inkrementell angezeigt werden können

## PERFORMANZ

- Reactive Stack fügt einen Delay (gegenüber MCV) in der Bearbeitungszeit hinzu
- Die Stärke von Reactive liegt in der Performanz des gesamten Server
- Hoher Durchsatz geht auf Kosten der Latenz!

## ENTWICKLUNG

- Pattern muss ausreichend bei den Entwicklungsteams bekannt sein (Busfaktor)
- Schwieriger zu debuggen aufgrund der "fluent"-API (Methodenverkettung)

# Spring Security

1. Allgemein
2. Big Picture
3. Authentifizierung
4. Autorisierung

# Einführung in Spring Security

- Application Layer Security Framework
- Umfassende Unterstützung in den Bereichen
  - Login und Logout Funktionalität
  - Zugriffsregeln auf URLs & mit welcher Rolle dies erfolgt
  - Behandelt allgemeine Schwachstellen
  - Integration mit anderen Spring Libraries
- Ist sehr flexibel & konfigurierbar, da Security bei Anwendungen sehr unterschiedlich ist

# Terminologie (1)

- Authentisierung
  - Nachweis/Bestätigung einer Identität
- Authentifizierung (authentication)
  - Prüfung der Angaben auf Echtheit
- Autorisierung (authorization)
  - Prüfen von Berechtigungen (Erlauben / Verweigern von Zugriffen)

## Terminologie (2)

- Principal
  - Aktueller Benutzer, welcher identifiziert wurde
- Granted Authority vs Rolle
  - Authority: individuelle Privilegien (z.B. hasAuthority('READ\_PRIVILEGE'))
  - Rolle: Gruppe von Privilegien (z.B. hasRole('ADMIN'))

# Auto Konfiguration

- Erzeugt eine Servlet Filter (Bean **springSecurityFilterChain**)
  - Verantwortlich für die gesamte Security
- Registriert den Filter im Servlet Container für jeden Request
- Erzeugt ein **UserDetailsService** Bean (mit User, Password und Privilegien)

# Standard Funktionalitäten (1)

- Erfordert einen autorisierten Benutzer für jede Interaction/Request
- Liefert ein Standard Login Form (/login)
- Erzeugt automatisch einen 'user' mit Passwort (siehe console)
- Schützt den Password Store mit BCrypt (Einwegtransformation durch PasswordEncoder)
- Logout Funktionalität

## Standard Funktionalitäten (2)

- Vorbeugung gegen Cross Site Request Forgery (CSRF)
- Session Fixation Schutz (SessionId)
- Integration von Default Security HTTP Response Headern

# Standard Security HTTP Response Header

- Cache Control
- X-Content-Type-Options Integration damit Content-Type-Headern befolgt werden
- HTTP Strict Transport Security für Transportverschlüsselung
- X-Frame-Options Integration zur Vorbeugung von Clickjacking
- X-XSS-Protection Integration stoppt page loading wenn XSS erkannt wird

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
X-Frame-Options: DENY
X-XSS-Protection: 1; mode=block
```

# Aktivierung von Spring Security

- Einbinden des Moduls

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

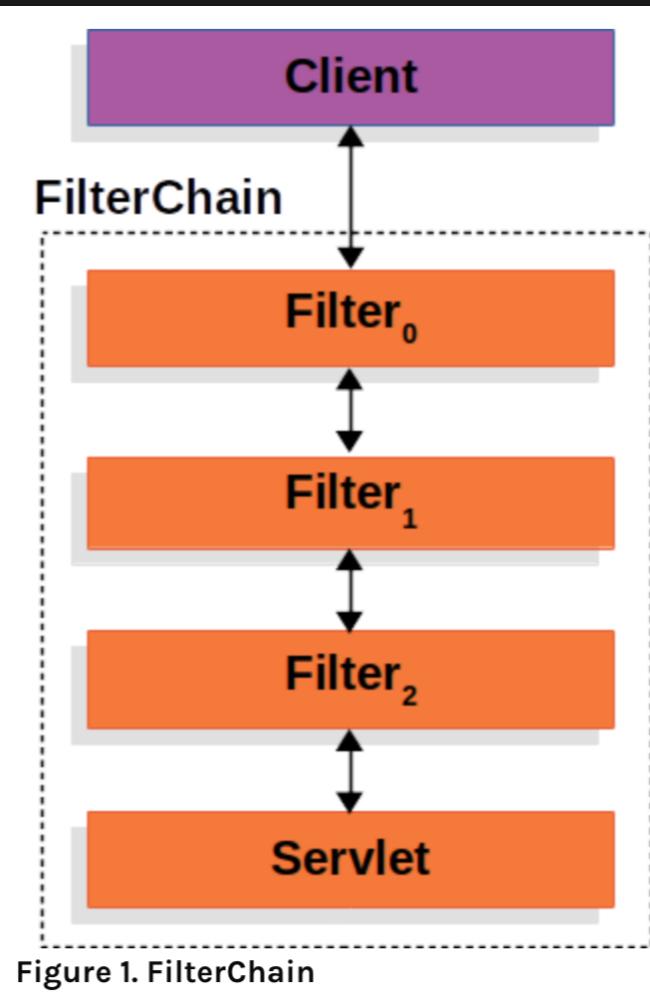
- Einschalten und konfigurieren

```
@EnableWebSecurity(debug = false)
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    // ...
}
```

# Spring Security

1. Allgemein
2. **Big Picture**
3. Authentifizierung
4. Autorisierung

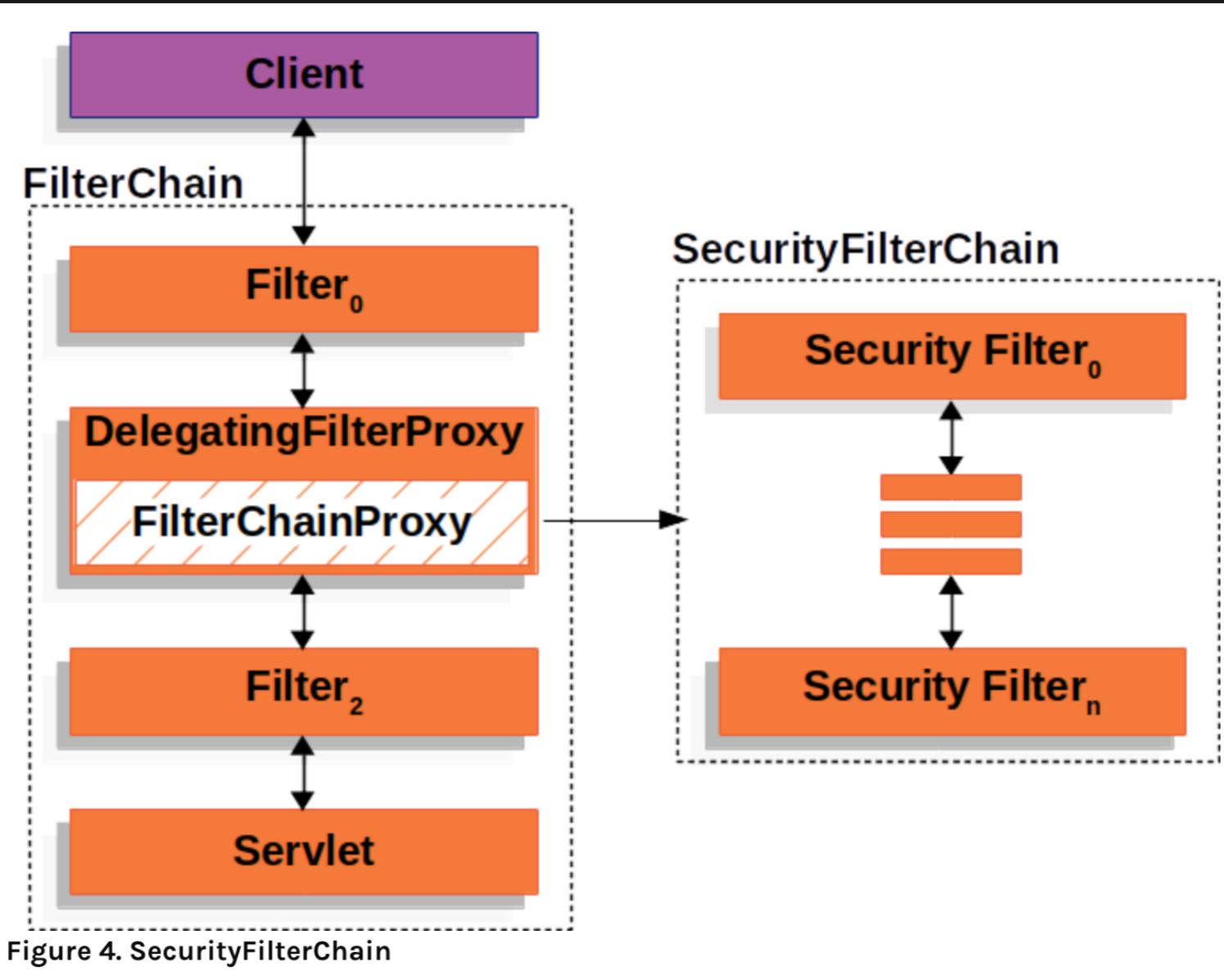
# FilterChain einer Servlet Application



# Dummy Filter

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {  
    // do something before the rest of the application  
    chain.doFilter(request, response); // invoke the rest of the application  
    // do something after the rest of the application  
}
```

# SecurityFilterChain



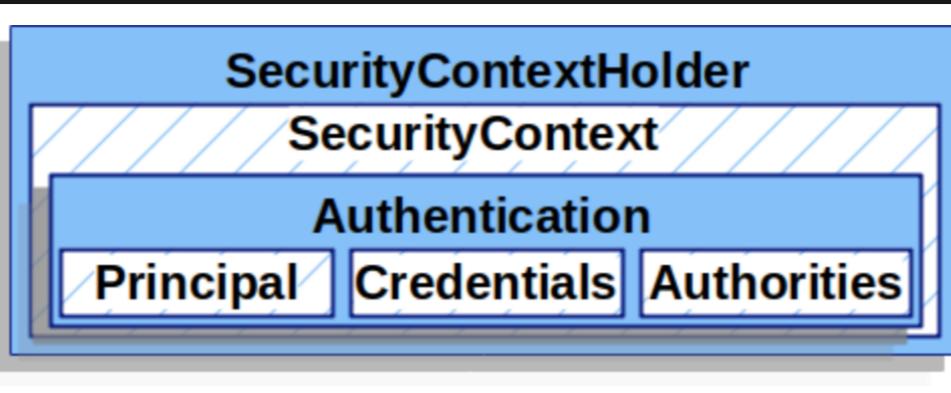
Dokumentation der 32 Security Filter

# Spring Security

1. Allgemein
2. Big Picture
3. **Authentifizierung**
4. Autorisierung

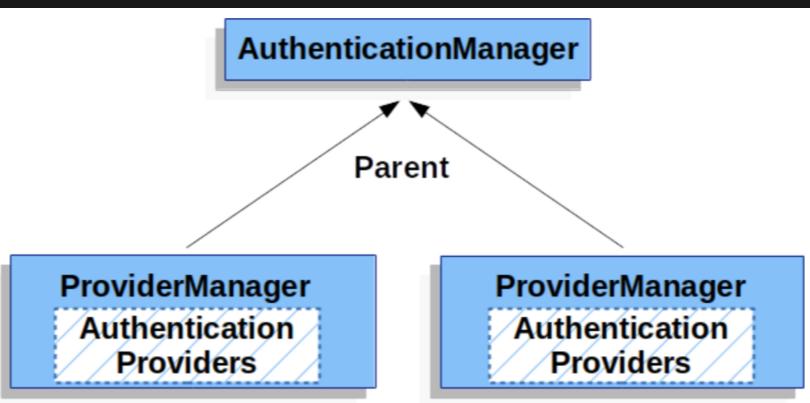
# Komponenten (1)

- SecurityContext/Holder
  - Herzstück von Spring Security (speichert Details im Authentication Object)
- Authentication enthält
  - Principal (identifiziert Benutzer, UserDetails)
  - Credentials (Passwort, häufig leer nach Login)
  - (Granted) Authorities (List Berechtigungen)



## Komponenten (2)

- AuthenticationManager
  - API, definiert wie die Filter die Authentifizierung durchführen
- ProviderManager
  - übliche Implementierung des AuthenticationManager
- AuthenticationProvider
  - z.B. DaoAuthenticationProvider, JwtAuthenticationProvider



# Authentifizierungsmechanismen

- Benutzer und Password
- OAuth 2.0 Framework (mit OpenID Connect)
- SAML 2.0
- Central Authentication Server (CAS)
- Remember Me
- Java Authentication and Authorization Service (JAAS)
- OpenID
- Pre-Authentication Scenarios
- X509 Authentifizierung

# JSON Web Token (JWT)

- Struktur besteht aus Header, Payload und Signatur
- Benötigte Komponenten sind UserDetailsService, RequestFilter, TokenHandler, AuthenticationEntryPoint und deren Konfiguration

# Spring Security

1. Allgemein
2. Big Picture
3. Authentifizierung
- 4. Autorisierung**

# Benutzer im Security Context

```
SecurityContext context = SecurityContextHolder.createEmptyContext();
context.setAuthentication(new TestingAuthenticationToken("username", "password", "ROLE_USER"));
SecurityContextHolder.setContext(context);
// ...
SecurityContextHolder.getContext().getAuthentication();
```

```
@Bean
public UserDetailsService userDetailsService() {
    User.UserBuilder users = User.withDefaultPasswordEncoder();
    UserDetails user = users
        .username("user")
        .password("password")
        .roles("ROLE1", "ROLE2")
        .authorities("READ", "WRITE")
        .build();
    return new InMemoryUserDetailsManager(user);
}
```

# Request Autorisierung

```
@EnableWebSecurity
public class MySecurityConfiguration extends WebSecurityConfigurerAdapter {
    // ...
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests(authorize -> authorize
                .antMatchers("/", "/resources/**", "/**.bundle.**.js").permitAll()
                .antMatchers("/user/**").hasRole("USER")
                .antMatchers("/konto/**").hasAnyAuthority("READ")
                .antMatchers("/network/**").hasIpAddress("192.168.1.0/24")
                .antMatchers("/admin/**").access("hasRole('ADMIN') and hasRole('DBA')")
                .anyRequest().authenticated()
            )
            .csrf().disable().cors().disable()
            .formLogin(formLogin -> formLogin
                .loginPage("/login")
                .successForwardUrl("/dashboard")
                .failureUrl("/error")
            )
            .logout(logout -> logout
                .invalidateHttpSession(true)
                .deleteCookies("JSESSIONID")
                .logoutSuccessUrl("/login")
            );
    }
    // ...
}
```

Weiterer Expressions Dokumentation.

# Methoden Autorisierung (1)

- Zum Aktivieren @EnableGlobalMethodSecurity
- Annotation Varianten
  - Pre / Post
  - Secured
  - JSR 250

# Methoden Autorisierung (2)

```
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true, jsr250Enabled = true)
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    // ...
}
```

```
@Component
public interface CarService {

    @PreAuthorize("hasRole('ROLE_VIEWER') or hasRole('ROLE_USER')")
    public List<Car> findAll();

    @Secured({ "ROLE_USER" })
    public Car getDetails(int id);

    @RolesAllowed({ "ROLE_VENDOR" })
    public Car edit(int id, String name);
    // ...
}
```

# CrossOrigin Konfiguration

```
@GetMapping("/api/cars")
@CrossOrigin(origins = "http://localhost:9000", allowedHeaders = "*", methods = "GET")
public List<Car> findCars() {
    // ...
}
```

```
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    // ...
    @Bean
    CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(asList("https://localhost:9000"));
        configuration.setAllowedMethods(asList("GET", "POST"));
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/api/**", configuration);
        return source;
    }
}
```

# Testing (1)

```
<dependencies>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

```
@Secured("ROLE_USER")
public String getUsername() {
    SecurityContext securityContext = SecurityContextHolder.getContext();
    return securityContext.getAuthentication().getName();
}
```

## Testing (2)

```
@Test
@WithMockUser(username = "foo", roles = { "ROLE_USER" }, authorities = { "READ" })
public void givenRoleUser_whenCallGetUsername_thenReturnUsername() {
    String userName = userRoleService.getUsername();

    assertThat(userName).equals("foo");
}

// ...

@Test(expected = AccessDeniedException.class)
@WithAnonymousUser
public void givenAnonymousUser_whenCallGetUsername_thenAccessDenied() {
    userRoleService.getUsername();
}
```

# Übung: Spring Security

1. ***git pull*** und schaue in das Verzeichnis ***16\_security***
2. Implementiere die Aufgabe aus der Readme

# Zusammenfassung

# Wrap-Up

Was haben wir gelernt?

# Ausblick

- Spring (Boot) kann noch so viel mehr!
- Aspektorientierte Programmierung (AOP) erlaubt z.B. Transaktionen (@Transactional)
- Spring Data JPA: ORM mit Hibernate

```
public interface CarRepository extends PagingAndSortingRepository<Car, Long> {  
  
    List<Car> findByBrand(String brand);  
  
    @Query("select c from Car c where c.front.left.brand = :brand"  
          + "or c.front.right.brand = :brand or c.rear.left.brand = :brand"  
          + "or c.rear.right.brand = :brand")  
    List<Car> findByWheelBrand(String brand);  
}
```

- Spring Data Rest: Autogenerierte REST-API mit HAL als Media-Type
- Spring Security
- ...

# Ausblick

- Spring Boot ist gut dokumentiert und sehr populär -> es gibt unendlich viele Quellen
- Empfehlung: Spring Boot Infografik von Michael Simons von jax.de: <https://jax.de/spring-boot-cheat-sheet/>
- <http://springbootbuch.de/>
  - <https://github.com/springbootbuch>
- codecentric Blog :)
  - <https://blog.codecentric.de/category/java/>
  - <https://blog.codecentric.de/en/category/java-en/>