

Agenda

- Einleitung
- Dependency Injection
- Spring Boot
- Externe Konfiguration
- MVC

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

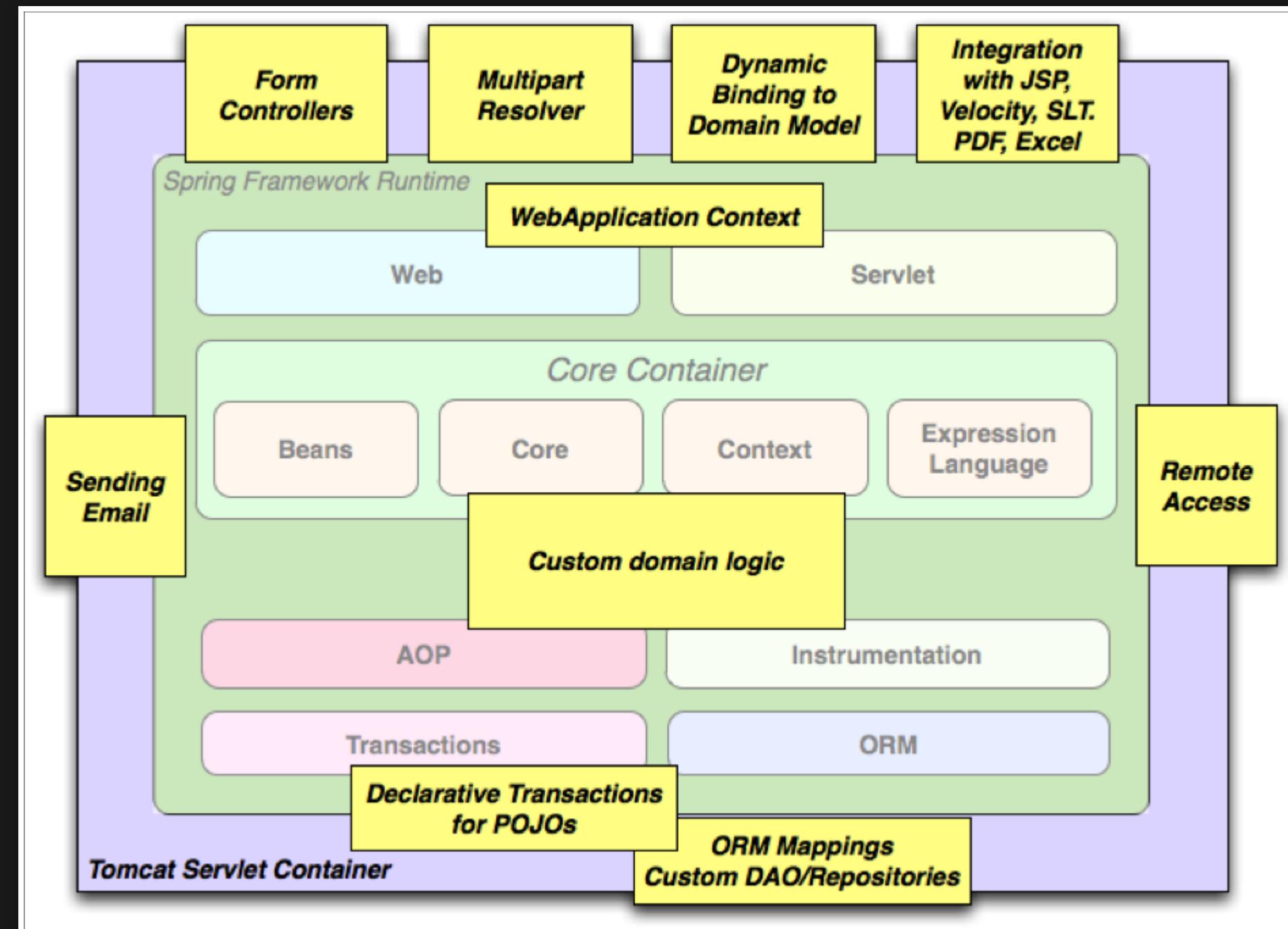
Spring Projekte

- 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'

- 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>2.3.5.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Einsatz von Spring Boot über das Dependency Management

- 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>2.3.5.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

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/2.3.5.RELEASE/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)

```
<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>
```

Lauffähiges Spring Boot JAR

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

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

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/> anschaust, oder
- in der Spring Tool Suite (STS) IDE, ein “New Spring Starter Project” anlegst

2. Mit der IDE:

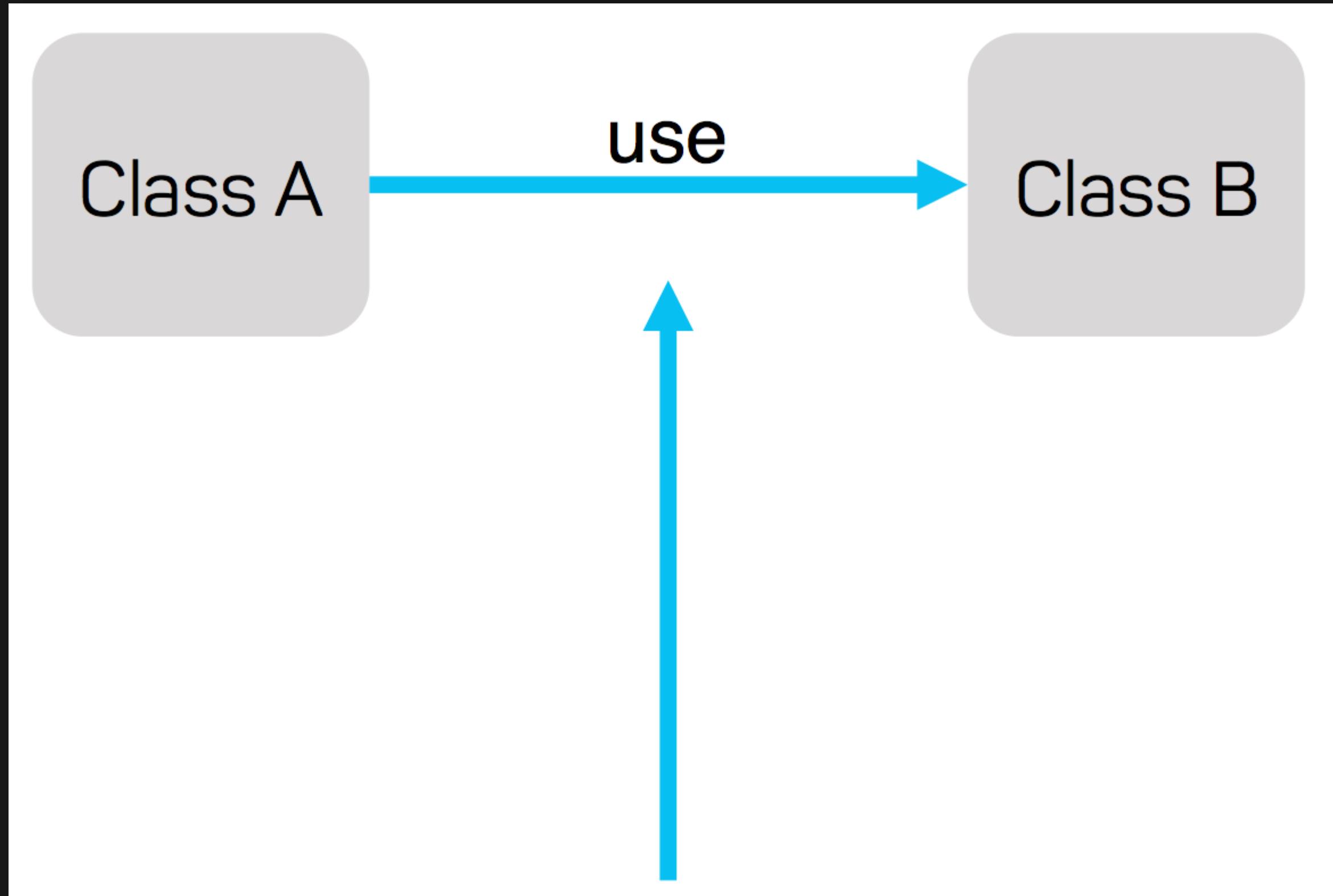
- Importieren, Compilieren und Starten der App

3. Auf der Kommandozeile:

- Compilieren und zusammenpacken: `./mvnw package`
- Starten: `java -jar target/hello-world-0.0.1-SNAPSHOT.jar`
- Alternativ mit maven: `./mvnw spring-boot:run`

Dependency Injection (DI)

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



dependency

- Sollte A von B wissen?
- Sollte A die Instanzierung von B vornehmen?
- Was ist, wenn C auch eine Instanz von B benötigt?

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

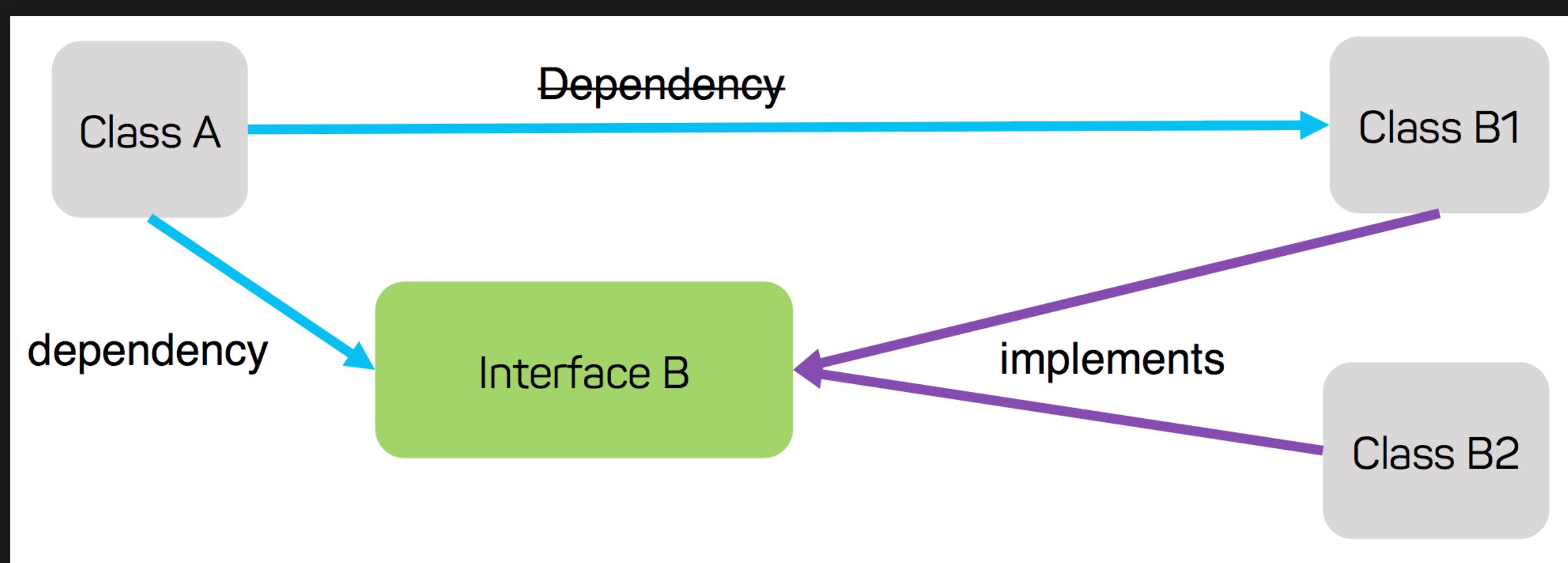
Beispiel für DI

```
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** UserRepository() 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)

- DI ist ein Beispiel für 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

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
- Sonst: methodenbasierte DI für optionale Abhängigkeiten
- Die Folien verwenden feldbasierte DI aus Platzgründen

Übung: DI ohne Container

1. Download via

```
git clone https://github.com/codecentric/spring-school
```

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")
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" Maven 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

- Das Spring Boot Maven Plugin erstellt ein lauffähiges JAR aus der Spring Boot Anwendung
- Das Plugin bringt schon die richtige Konfiguration mit, muss aber explizit in die POM eingefügt werden
- Wird in der Maven Phase “package” aktiv
- Es ist aber auch möglich, die Goals des Spring Boot Maven Plugins direkt aufzurufen:

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

Start einer Spring Boot Anwendung

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

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

oder man kann das JAR bauen und von Hand starten

```
$> mvn package  
$> 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 aktivierten 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]
```

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, ...
- 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

1. Default properties (specified by setting `SpringApplication.setDefaultProperties`).
2. `@PropertySource` annotations on your `@Configuration` classes. Please note that such property sources are not added to the `Environment` until the application context is being refreshed. This is too late to configure certain properties such as `logging.*` and `spring.main.*` which are read before refresh begins.
3. Config data (such as `application.properties` files)
4. A `RandomValuePropertySource` that has properties only in `random.*`.
5. OS environment variables.
6. Java System properties (`System.getProperties()`).
7. JNDI attributes from `java:comp/env` .
8. `ServletContext` init parameters.
9. `ServletConfig` init parameters.
10. Properties from `SPRING_APPLICATION_JSON` (inline JSON embedded in an environment variable or system property).
11. Command line arguments.
12. `properties` attribute on your tests. Available on `@SpringBootTest` and the [test annotations for testing a particular slice of your application](#).
13. `@TestPropertySource` annotations on your tests.
14. [Devtools global settings properties](#) in the `$HOME/.config/spring-boot` directory when devtools is active.

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 \${<Propertynname>} 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')}")

```

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 aktivierter 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

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

- Profilspezifische 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

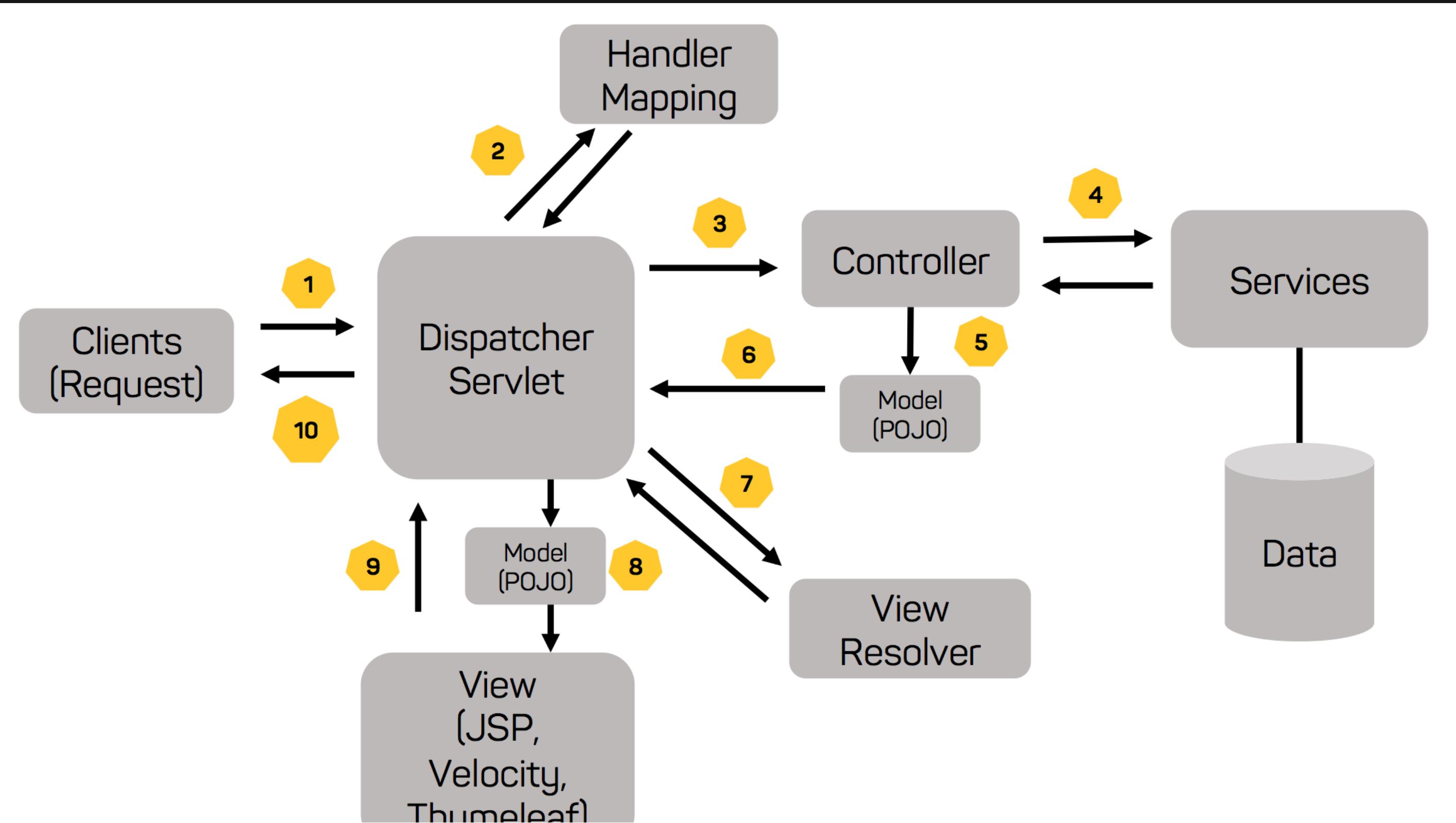
- Auch bekannt als "Spring MVC"
- Bietet ein umfangreiches "Model View Controller" Webframework an
- MVC ist ein verbreitetes Design Pattern für GUI und Web Applicationen
 - M = Model:
 - Verwaltet die Daten, Geschäftslogik und Regeln der Anwendung
 - V = View:
 - Die Darstellung der Informationen
 - 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

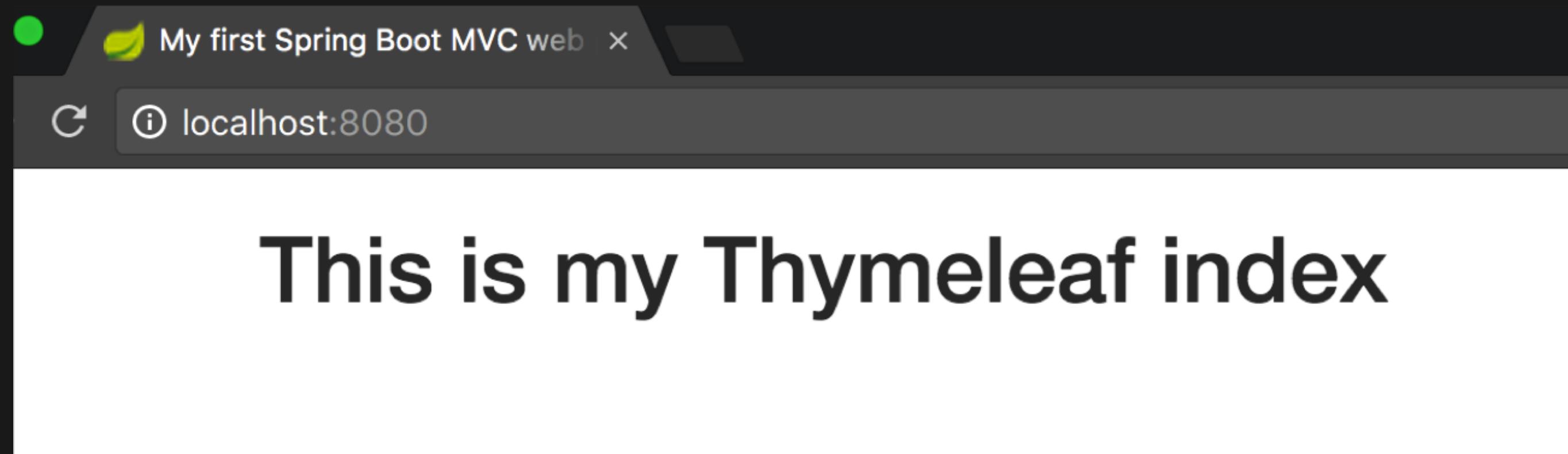
7) Finden des Views mit dem Namen “index”

```
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>
```

10) der Client stellt das Ergebnis dar



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
- HttpServlet{Request,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

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/>

Feedback

Wir würden uns über Feedback freuen -> Miro Board :)

Ende

Copyright 2018-2020

