integrata cegos

Spring Boot

Vorstellungsrunde

- Warum nehmen Sie an diesem Training teil?
- Gibt es besondere Wünsche?

Disclaimer!

Ein Trainer weiß einiges, aber nicht alles!

Das braucht man für gute Software-Entwicklung auch nicht - wichtig ist, schnell die Lösung für Probleme finden zu können:

- Richtig googlen
- Schwierige Probleme schriftlich festhalten - was versuche ich zu lösen, was funktioniert nicht?
- Hilfreiche Kollegen, Freunde, Netzwerk
- Nicht in Probleme verrennen, rechtzeitig aufhören, Abstand gewinnen

Didaktik

Exercise makes the master :)

Jede Lektion

- beginnt mit einem Theorieteil
- endet mit Übungen in Eigenarbeit
- schließt ab mit einer gemeinsamen Betrachtung einer (meiner?) Lösung

Nutzen Sie jetzt schon die Verweise auf öffentliche Dokumentationsquellen -- später wird Ihnen Vertrautheit mit diesen hilfreich sein.

Inhalt

Dieses Training besteht aus 8 Blöcken. Auf jeden Tag entfallen 4 Blöcke.

Tag 1:

1. Willkommen, Einführung, Spring Boot Grundlagen
2. Bean Definitionen
3. Migration
4. Testing

Tag 2:

1. Spring-Data JPA
2. Konfiguration
3. RESTful API
4. Logging, Profile, Monitoring

Tagesplan
09:00 - 11:00 Block #1
11:00 - 11:15 Pause
11:15 - 12:45 Block #2
12:45 - 13:30 Mittagspause
13:30 - 15:15 Block #3
15:15 - 15:30 Pause
15:30 - 17:00 Block #4

Warm up

Benötigte Tools -- bitte via Kommandozeile folgende Befehle ausführen. Die genauen Versionsnummern sind nicht relevant, die Tools sollten nur alle vorhanden sein.

- `java -version`
- `git --version`
- `mvn -version`

Gewünschte Versionen sind:

- Java 11 oder höher
- Maven 3

Kapitel 010

Einführung

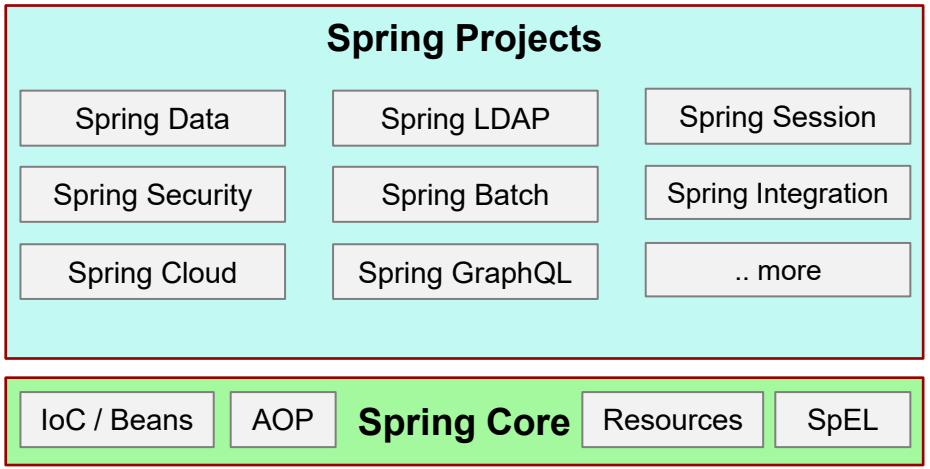
Was ist überhaupt das Spring Framework?*

Ein Open-Source Framework zur Erstellung von lose gekoppelten Java Anwendungen.

Es bietet ein einheitliches Entwicklungsparadigma, welches uns von möglichst viel unnötiger Arbeit befreit. Unter dieser Abstraktionsschicht werden oft vorhandene Bibliotheken und Spezifikationen (insbesondere Java EE) genutzt.

Im Kern bietet das Framework primär

- einen IoC Container (Dependency Injection)
- aspektorientierte Programmierung (AOP)
- Ressource-Handling (Zugriff auf Konfiguration)
- .. aber es hat noch über 20 Sub-Projekte



Vorteile von Spring Boot*

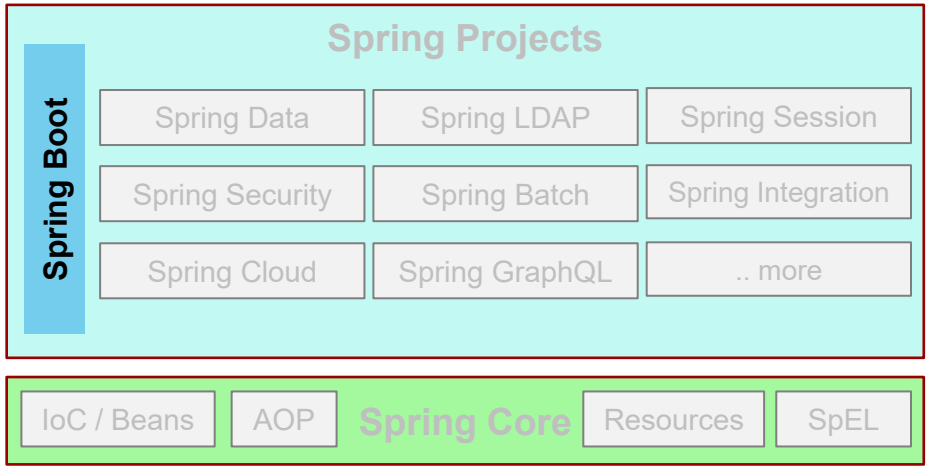
Spring Boot vereinfacht die Konfiguration und Ausführung von Spring Anwendungen und beschleunigt somit die Entwicklung.

- `application.properties` Datei
- eingebundene Bibliotheken (insbesondere Spring Module) werden vorkonfiguriert
- Start der Anwendung via `main()` Methode statt Deployment in Web-Container
- keine XML Konfiguration notwendig

Also was ist Spring Boot?

Spring Boot besteht im Kern aus:

- Hilfs- und Autoconfiguration-Klassen
- Maven Parent-POM Datei
- Maven "Starter" Dependencies



Technische Anforderungen

Wir arbeiten in diesem Training mit Spring Boot 2.6.8, welches folgende Anforderungen hat:

- Java 8 bis 18
- Spring Framework 5.3.20+
- Maven 3.5+

Folgende embedded Servlet Containers werden unterstützt:

- Tomcat 9.0 (default)
- Jetty 9.4 oder 10
- Undertow 2.0

Link zur Startseite der 2.6.8 Dokumentation

Setup und eine erste Übung

Basisprojekt auschecken und ausführen

Auschecken:

- Kommandozeile öffnen
- Wechseln Sie in Ihr bevorzugtes Projekteverzeichnis, dann:
- **`git clone --branch 020_beans_exercise https://github.com/tauinger-de/training.spring-boot.math.git`**
- Bitte darauf achten, dass der Branch "020_beans_exercise" ausgecheckt wird

Ausführen:

- Wechseln Sie in das Verzeichnis "training.spring-boot.math" und führen Sie folgenden Befehl aus:
- **`mvnw clean spring-boot:run`**

Hinweis: **`mvnw`** ist der Maven-Wrapper, der bei dem erstem Aufruf Maven lokal installiert

Projekt in IDE öffnen

- Öffnen Sie das soeben ausgecheckte Projekt in Ihrer bevorzugten Entwicklungsumgebung
  - Intellij: File > New > Project from Existing Sources …
  - Eclipse: File > Import … > Maven \ Existing Maven Projects

Was wird ausgeführt?

- Schauen Sie sich den Source Code an.
- Was passiert wohl beim Starten der Anwendung?
- Warum erscheint die Ausgabe "Los geht's" so früh und die Ausgabe "Welcome" so spät?

integrata cegos

Kapitel 015

Spring Boot Grundlagen

integrata cegos

Die Parent-POM

Sinn*

Vereinfacht den Build-Prozess und die Ausführung von Spring Boot Anwendungen.

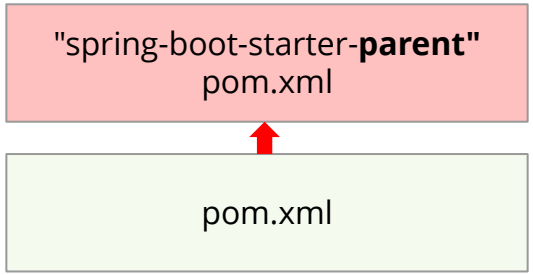
Wie?

- Konfiguration gängiger Maven Properties
- Plugin Management (z.B. Kotlin Support, Paketierung, Main-Class definieren)
- Dependency Management -- Vorgabe von empfohlenen Versionen abgesegneter Drittbibliotheken (erfolgt via `spring-boot-dependencies` Parent-POM)

Nutzung*

Die Spring Boot Parent-POM wird von spring.io veröffentlicht.

Analog einer JAR-Dependency wird sie über ein Maven Repository gefunden und heruntergeladen.



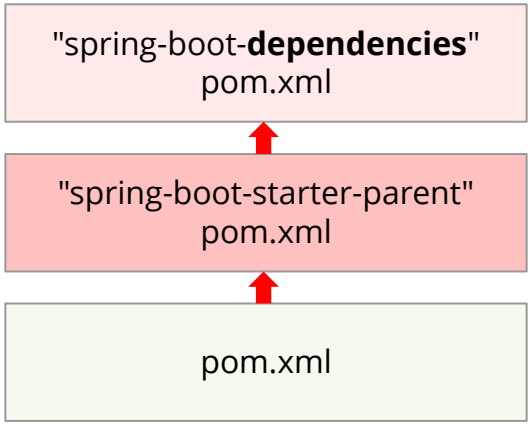

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.8</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Bill-Of-Materials*

Ebenfalls existiert eine Bill-Of-Materials POM.

Diese wird von der Parent-POM referenziert und genutzt.

Über diese Datei erfolgt die Vorkonfiguration von Dependency-Versionen.



integrata cegos

Starter Dependencies

Übersicht

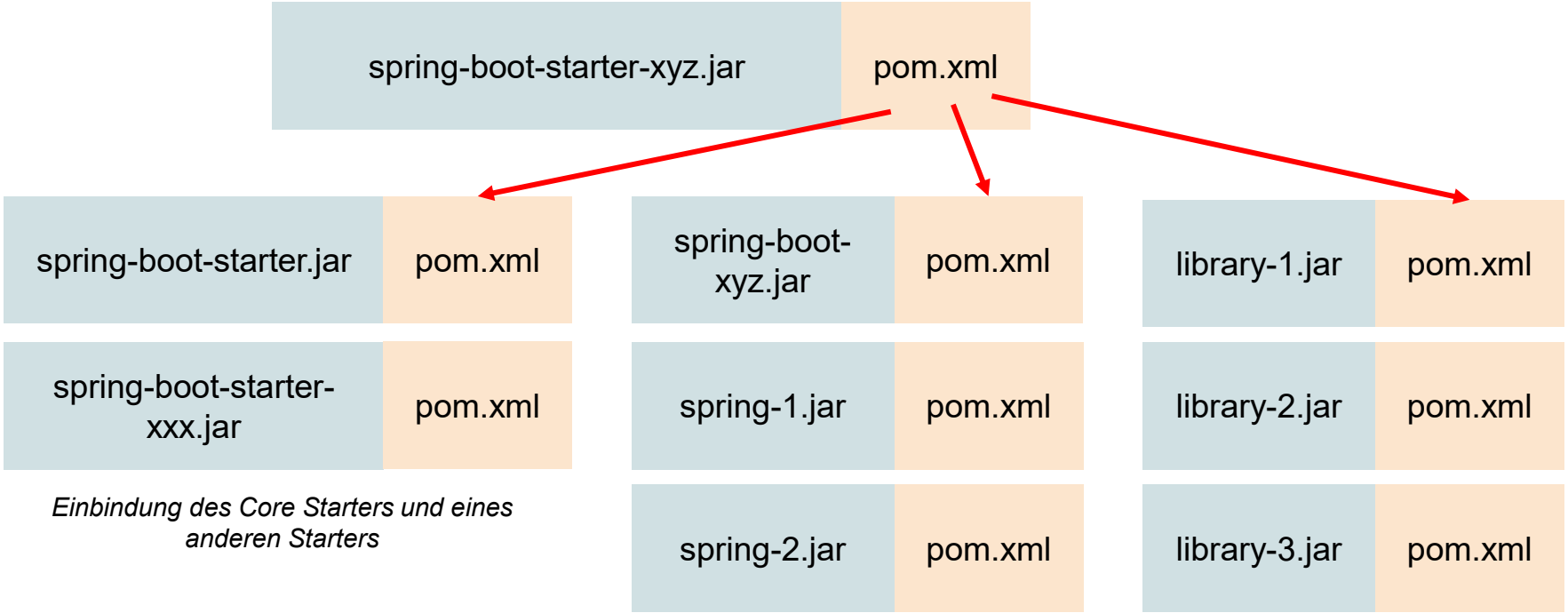
Ein "Starter" ist eine vorgefertigte Sammlung von relevanten Bibliotheken.

Häufig genutzte Starter sind:

- **spring-boot-starter**: der Core Starter
- **spring-boot-starter-data-jpa**: Datenbankzugriff mittels Spring-Data und JPA
- **spring-boot-starter-test**: Test Support mit JUnit, Hamcrest, Mockito
- **spring-boot-starter-web**: (REST) Controller Support via Spring MVC

Link: Starters Dokumentation, Starters Source-Code

Struktur eines Starters*



Einbindung des Core Starters und eines anderen Starters

Einbindung von neuen Spring Boot Klassen und normalen Spring Bibliotheken

Einbindung von Drittbibliotheken

Dependency Analyse

Anzeige des Dependency Baums mit `mvnw dependency:tree`

Hinweis: Sie können die Ausgabe auch in eine Datei pipen ("… > deps.txt") und dann in einem Editor betrachten und durchsuchen

```
[INFO] --- maven-dependency-plugin:3.1.2:tree (default-cli) @ spring-boot ---
[INFO] com.example:spring-boot:jar:1-SNAPSHOT
[INFO] \- org.springframework.boot:spring-boot-starter:jar:2.3.0.RELEASE:compile
[INFO]    +- org.springframework.boot:spring-boot:jar:2.3.0.RELEASE:compile
[INFO]    |  \- org.springframework:spring-context:jar:5.2.6.RELEASE:compile
[INFO]    |           +- org.springframework:spring-aop:jar:5.2.6.RELEASE:compile
[INFO]    |           +- org.springframework:spring-beans:jar:5.2.6.RELEASE:compile
[INFO]    |           \- org.springframework:spring-expression:jar:5.2.6.RELEASE:compile
[INFO]    +- org.springframework.boot:spring-boot-autoconfigure:jar:2.3.0.RELEASE:compile
[INFO]    +- org.springframework.boot:spring-boot-starter-logging:jar:2.3.0.RELEASE:compile
[INFO]    |  +- ch.qos.logback:logback-classic:jar:1.2.3:compile
```

integrata cegos

Die Application Klasse

Die Application Klasse

Herzstück jeder Spring Boot Anwendung ist eine mit `@SpringBootApplication` annotierte Klasse, die eine `main()` Methode enthält.

```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        System.out.println("Los geht's!");
        SpringApplication.run(DemoApplication.class, args);
    }

}
```

CommandLineRunner

Zusätzliche Funktionalität, die beim Starten einer Anwendung ausgeführt werden soll, kann durch Implementierung einer `CommandLineRunner` oder `ApplicationRunner` Bean erfolgen.

Die Reihenfolge der Ausführung kann mittels `@Order` definiert werden.

```
@Component
public class WelcomingRunner implements CommandLineRunner {

    @Override
    public void run(String... args) {
        System.out.println("Welcome");
    }

}
```

Paketierung / App ausführen

Paketierung als JAR*

Typischerweise werden Spring-Boot Anwendung von Maven als “fette” JAR Datei gebaut:

```
C:\my-project> mvnw clean package
```

Diese sind eigenständig ausführbar:

```
C:\my-project> java -jar target/<artifact-name>-<version>.jar
```

Ausführung in IDE

Die Main-Klasse der Spring-Boot Anwendung kann direkt in der IDE ohne weiteres (kein Deployment in Server) gestartet werden.

Ausführung per Kommandozeile*

Über das Spring-Boot Plugin kann die Anwendung von der Kommandozeile aus gestartet werden.

Das “clean” ist optional und sorgt dafür, dass auch wirklich der neuste Stand läuft.

```
C:\my-project> mvnw clean spring-boot:run
```

Deployment als WAR in Tomcat

In der pom.xml kann die Paketierung auf “war” geändert werden. Zusammen mit kleineren Anpassungen entsteht so eine WAR Datei, die in einen Servlet-Container wie z.B. Tomcat deployt werden kann.

- Link zu einem Beispiel

Deployment in die Cloud / Docker Container

Spring-Boots Paketierung als JAR Datei macht das Deployment in Cloud oder (Docker) Container leicht.

- Link zu Cloud Deployment Optionen
- Link zu einem Docker Beispiel

Developer Tools*

Spring-Boot liefert zusätzliche Werkzeuge (die "Developer Tools") aus, die das Entwickeln von Server-Anwendungen erleichtern.

Diese unterstützen insbesondere bei

- Kein Caching (immer neusten Stand sehen)
- Automatische Neustarts (Server läuft immer in neuester Version)
- Globale Einstellungen (nützlich bei Entwicklung verschiedener Systeme)

Link zur Dokumentation.

Aktiviert werden die Developer Tools durch Einbindung einer Dependency in der POM.

Tipp: Bei Intellij den Neustart über ein Trigger File auslösen lassen, um mehrfach Starts zu verhindern (siehe auch hier)

integrata cegos

Internals

Phasen des Programmstarts*

- `MyApplication.main()`
- `SpringApplication.run()`
- `ApplicationContext` wird erzeugt (konkreter Typ je nach webApplicationType)
- Bean Definitionen werden gesucht (u.a. package scan)
  - sowohl selbst definierte als auch von Spring deklarierte (insbes. Spring Boot)
- Beans werden erzeugt
  - unter Beachtung von `@Conditional` Annotationen (Stichwort Autoconfiguration)
  - konfliktfreie Reihenfolge (Stichwort Abhängigkeiten)
  - inkl. Ausführung der `@PostConstruct` annotierten Init-Methoden
- `ApplicationContext` ist nun vollständig vorhanden
- `CommandLineRunner` und `ApplicationRunner` laufen

integrata cegos

Kapitel 020

Bean Definition

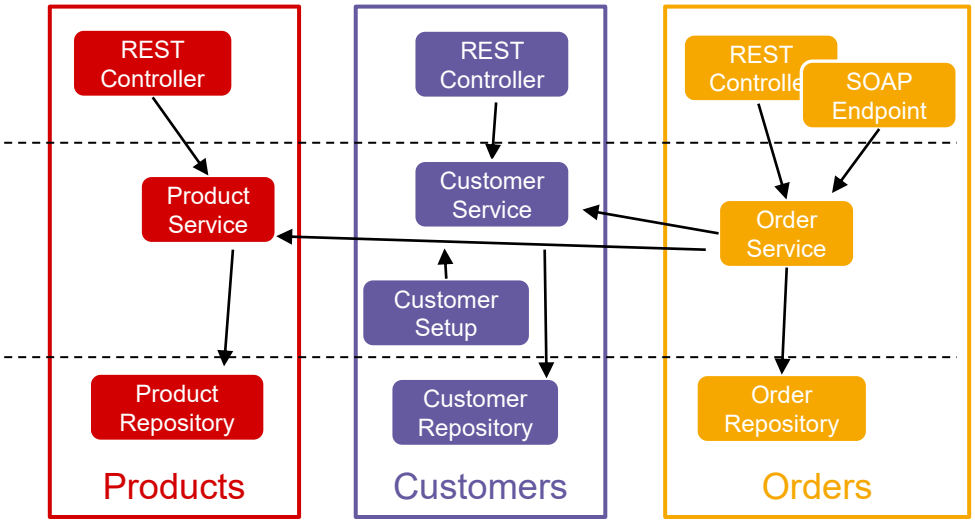
Beans – warum, wofür?*

Spring Beans sind Instanzen, welche die Infrastruktur und Geschäftslogik einer Anwendung abbilden.

Oft existieren nur wenige bzw. eine einzige (Singleton) Instanz je Klasse.

Welche Beans existieren – und welche Abhängigkeiten sie haben - legt der Entwickler fest.

Beans werden in dem Dependency-Injection Container verwaltet und über diesen gefunden.



Bean Definitionen per XML

“Damals” wurden Beans per XML Datei definiert und damit der Application-Context initialisiert:

```
<beans>
   <bean id="myBean" class="my.company.MyBean">
      <property name="dependency" ref="otherBean"/>
      <property name="someValue" value="42"/>
   </bean>
</beans>
```

Empfohlen wird dies heutzutage nicht mehr, ist jedoch mit Spring Boot leicht machbar und auch mit der empfohlenen Alternative kombinierbar. Siehe hier.

Bean Definitionen per @Service oder @Component*

Möchte man eine Instanz einer Klasse als Bean verfügbar machen, so annotiert man die Klasse mit `@Service` oder `@Component`.

```
@Service
public class SomeService {
    public double calculateSomething(int input) { // ... }
}



@Component
public class SomeHelper {
    public String helpWithSomething(String input) { // ... }
}
```

Bean Definitionen per @Configuration und @Bean*

Möchte man die Bean-Erzeugung selber ausprogrammieren so wird eine Factory Klasse genutzt.

Diese ist mit `@Configuration` annotiert. Die Rückgabewerte von `@Bean` Methoden sind Beans, die in den Application-Context aufgenommen werden:

```
@Configuration
public class SomeBeanProvider {

    @Bean
    public MyBean createSomeBean(OtherBean otherBean) {
        MyBean myBean = new MyBean(otherBean, 42);
        myBean.configureThisAndThat(1, 2, 3);
        return myBean;
    }
}
```

Die Methoden können Parameter definieren, die mit bereits bestehenden Beans bestückt werden.

Verdrahtung von Beans*

Bean Instanzen benötigen oft andere Beans, um arbeiten zu können. Diese Art der Verbindung nennt man "Dependency Injection".

Oder auch "wiring" (Verdrahtung).

Spring bietet hierfür mehrere Optionen:

1. per Konstruktor (empfohlen)
2. Aufruf einer Methode (z.B. eine Setter-Methode)
3. schreibender Zugriff auf Instanzvariablen per Reflection

Constructor Injection*

Die empfohlende Art der Dependency Injection erfolgt über den Konstruktor.

Vorteile:

- Instanzvariablen sind final
- Alle Abhängigkeiten sind bei Erzeugung der Instanz vorhanden
- Weniger Fehler, da keine Abhängigkeiten vergessen werden können

```
@Service
public class MyService {
    private final MyBean myBean;

    public MyService(MyBean myBean) {
        this.myBean = myBean;
    }

    public int someMethod() {
        return this.myBean.foo();
    }
}
```

Constructor Injection - Besonderheiten

Optionale Injections können mittels `java.util.Optional` Hilfsklasse ausgedrückt werden:

```
public class MyService {
    private final MyBean myBean;

    public MyService(Optional<MyBean> myBean) {
        this.myBean = myBean.orElse(null);
    }
}
```

Zirkuläre Abhängigkeiten zwischen Beans können mit `@Lazy` aufgelöst werden:

```
@Service
public class ServiceA {
    public ServiceA(ServiceB serviceB) { .. }
}

@Service
public class ServiceB {
    public ServiceB(@Lazy ServiceA serviceA) { .. }
}
```

Method Injection*

Eine mit `@Autowired` annotierte Methode kann eine oder mehrere Beans empfangen.

Dies kann eine einfache Setter-Methode sein (sogenannte Setter-Injection), aber auch eine Methode, die mehrere Beans empfängt.

```
@Autowired
public void injectDependencies(
        CustomerService customerService,
        ProductService productService
) {
    this.customerService = customerService;
    this.productService = productService;
}


@Autowired
public void setAddressSetup(AddressSetup addressSetup) {
    this.addressSetup = addressSetup;
}
```

Field Injection*

Eine mit `@Autowired` annotierte Feld-Deklaration (auch private) kann eine Bean empfangen.

Dies ist insbesondere bei hierarchischen Test-Klassen praktisch. Und wir können deklarieren, ob die Bean vorhanden sein muss!

```
class ProductServiceTest extends BaseTest {

    @Autowired
    ProductService productService;

    @Autowired(required=false)
    OtherService otherService;
}
```

PostConstruct*

Eine Methode in einer Bean kann mit dem PostConstruct Lifecycle Hook annotiert werden, so dass diese Methode direkt nach Verdrahtung aller Beans (dieser Klasse) aufgerufen wird.

Sprich, sobald die Instanz mit allen erforderlichen Dependencies versorgt wurde.

```
@Service
public class MyService {

    @PostConstruct
    public void doSomeInitialization() {
                // ...
    }
}
```

Beans benennen und referenzieren*

Jede Bean trägt einen Namen, der zur gezielten Verdrahtung (z.B. wenn es mehrere vom gleichen Typ gibt) genutzt werden kann.

Wird kein Name angegeben, so heißt die Bean wie die erzeugende Methode bzw. wird aus dem Namen der Klasse abgeleitet.

PS man kann auch `@Qualifier` verwenden, um den Namen einer Bean zu *definieren*

```
@Configuration
public BeanFactory{
    @Bean("myBean42") // default name "createSomeBean"
    public MyBean createSomeBean(){
        return //...;
    }
}

@Service("i-have-a-name-too")  // default name "myService"
public class MyService {
    public MyService(@Qualifier("myBean42") MyBean myBean) {
        this.myBean = myBean;
    }
}
```

```
@Service
@Qualifier("name-for-service-bean")
public class MyService {

...
}
```

Primäre Beans

Gibt es mehrere Beans vom gleichen Typ, so kann eine Bean mit `@Primary` annotiert werden, um diese bei Mehrdeutigkeiten zu verwenden.

```
@Configuration
public BeanFactory{
    @Bean
    @Primary
    public MyBean createSomeBean() {  ... }

    @Bean
    public MyBean otherBean() { ... }
}

@Service
public class MyService {
    public MyService(MyBean myBean) {
        this.myBean = myBean;
    }
}
```

Listen von Beans*

Es kann eine Liste aller Beans injiziert werden, die von einem gewünschten Typ (oder Sub-Typ) sind.

Der gewünschte Typ kann natürlich auch ein Interface statt einer Klasse sein.

```
@Service
public class MyService {

    public MyService(List<MyBean> myBeans) {
        // ...
    }
}
```

Map von Beans*

Es kann eine Map an Beans injiziert werden, die vom gewünschten Typ (oder Sub-Typ) sind.

Der Key der Map ist dann der Name jeder Bean, der Value natürlich die Bean selbst.

```
@Service
public class MyService {

    public MyService(Map<String, MyBean> myBeansByName) {
        // ...
    }
}
```

Bedingte Bean Definitionen*

Beans können so annotiert werden, dass sie nur unter bestimmten Bedingungen erstellt werden. Dies ist einer der grundlegenden Konfigurationsmechanismen von Spring-Boot.

Link zur Dokumentation der `@Conditional…` Annotationen

```
@Configuration
public class MyBeanFactory {
    @Bean
    @ConditionalOnMissingBean
    public MyBean createSomeBean(OtherBean otherBean) {
        // wird nur ausgeführt, wenn es noch keine Bean vom Typ “MyBean” gibt
        return new MyBean(otherBean, 42);
    }
}
```

Scopes

Spring Beans können mittels eines "Scopes" eine Lebensdauer zugewiesen bekommen:

Scope	Bedeutung	Verfügbarkeit
`ConfigurableBeanFactory.` *`SCOPE_SINGLETON`*	Exakt eine Instanz (**default**)	immer
`ConfigurableBeanFactory.` *`SCOPE_PROTOTYPE`*	Je Injection eine Instanz	Immer
`WebApplicationContext.` *`SCOPE_REQUEST`*	Pro Anfrage, d.h. Thread	Web-Anwendung
`WebApplicationContext.` *`SCOPE_SESSION`*	Pro Benutzer-Sitzung	Web-Anwendung
`WebApplicationContext.` *`SCOPE_APPLICATION`*	Pro Servlet-Context (Deployment)	Web-Anwendung

```
@Configuration
public class MyBeanFactory {
    @Bean
    @Scope(WebApplicationContext.SCOPE_SESSION)
    public Object someBean() {
        return // ...
    }
}


@Service
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class ServicePrototype {

}
```

Übungen

- Führen Sie die in der Datei EXERCISES.md beschriebenen Übungen aus
- Eine mögliche Lösung finden Sie im Branch "020_beans"

Kapitel 030

Migration

Motivation

Klassische Spring Anwendungen lassen sich auf den Spring Boot Ansatz migrieren.

Dazu bedarf es einer Reihe an Änderungen, die teilweise mit einem Wechsel auf moderne Technologien gekoppelt sind.

Von XML zu Annotationen

Bestehende Kontext-XML Dateien können durch Annotationen ersetzt werden.

- @Component bzw. @Service oder @Bean nutzen
- XML importieren (wenn sinnvoll)
- Namespace-basierte Konfiguration durch neue Annotationen ersetzen (nicht trivial

web.xml

Eine Spring Boot Anwendung nutzt keine web.xml Datei.

- DispatcherServlet wird von Boot automatisch konfiguriert
- Filter können mit @Component annotiert werden
- Servlets können mit @WebServlet annotiert werden
  (@ServletComponentScan nicht vergessen)
- Servlets und Listener können mittels @Bean konstruiert werden

Links:

- web.xml Migration generell
- Notwendigkeit für @ServletComponentScan

DataSource

Spring Boot konfiguriert automatisch eine DataSource Bean, wenn entsprechende Datenbank-Treiber im Classpath sind (z.B. H2 oder Postgres).

- manuelle Konfiguration ist mit Boot auch möglich, aber oft nicht nötig
- generell beschränkt sich die Konfiguration auf Setzen der notwendigen Werte in der `application.properties`

Links:

- Spring Boot DataSource Configuration

Properties migrieren

Spring Boot sucht Konfigurationswerte in application.properties oder application.yml Dateien.

- ggf. müssen bestehende Konfigurationen dorthin kopiert werden
- oder Property-Namen an Spring Boot Konventionen angepasst werden

Mehr Details zum Thema Konfiguration in einem späteren Kapitel.

Tests migrieren

Spring Boot nutzt die Annotation @SpringBootTest auf Klassenebene, um Testfälle im Rahmen eines Application-Context auszuführen.

- soweit nicht anders definiert, wird der reguläre Anwendungskontext hochgefahren
- Details im folgenden Kapitel "Testing"

Autoconfiguration deaktivieren

Die Autoconfiguration von Spring Boot kann gezielt deaktiviert werden, wenn diese im Konflikt mit selbst erzeugten Beans steht.

Links:

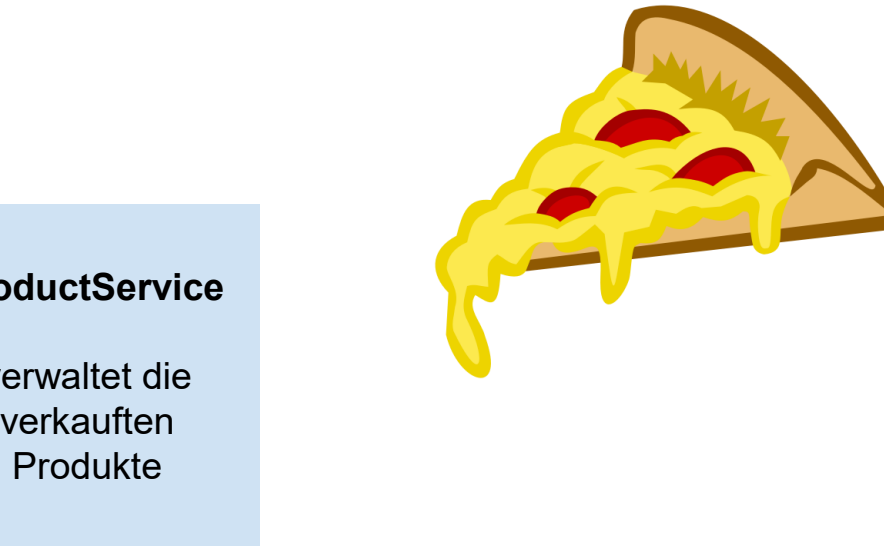
- Spring Boot docs zum Thema

Die "Pizza API" Anwendungen zum Üben

Übungsprojekt

Der rote Faden des Trainings ist ab hier das "Pizza API" Projekt.

Wir arbeiten nun mit einer von Lektion zu Lektion wachsenden Anwendung.



Was kann unsere Anwendung?

Als Kunde möchte ich eine Bestellung telefonisch aufgeben.

Als Kunde möchte ich bei erfolgter Bestellung über den Lieferzeitpunkt informiert werden.

Als Lieferdienst möchte ich Kundendaten verwalten, um für eine Telefonnummer einen Namen und Lieferadresse ableiten zu können.

Als Vermarkter möchte ich den Kunden Ermäßigungen je nach Wochentag Rabatte anbieten.

Migration

Die Übung zu diesem Kapitel ist die Migration der bestehenden Pizza API Anwendung von "klassischem" Spring hin zu einer Spring-Boot Anwendung.

Repository

Der Basis-Code und die Aufgaben sind in einem dedizierten Git Repository:
https://github.com/tauinger-de/training.spring-boot.pizza.git

Je Kapitel existieren dort zwei Branches:

- *nnn_thema*_**exercise** -- dies ist die Ausgangslage für die Übungen des Kapitels
- *nnn_thema*_**solution** -- und hier die dazugehörige Lösung

Los geht es nun mit Branch "030_migration_exercise".

Kapitel 040

Testing

Spring-Boot Starter Test

Mit dem Starter `spring-boot-starter-test` steigt man schnell in die Entwicklung von Testfällen ein.

Durch Aufnahme des Starters in die `pom.xml` verfügt das Projekt automatisch über folgende Bibliotheken:

- JUnit 5
- Spring Test & Spring Boot Test (Utilities)
- AssertJ (eine assertion library)
- Hamcrest (matcher a.k.a. constraints / predicates)
- Mockito (mocking framework)
- JSONassert (assertion für JSON)
- JsonPath (XPath für JSON)

JUnit 5*

Das Beispielprojekt nutzt JUnit 5 (Jupiter) als Test-Runner - sowohl für Unit als auch Integrationstests.

Unit-Test:

- benötigt eigentlich keinen Application-Context
- Klassen werden selbst mit "`new`" instanziiert
- sehr schnell in der Ausführung

Integrations-Test:

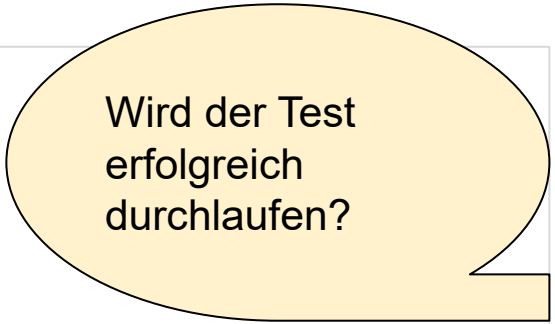
- beinhaltet mehrere Klassen und deren Zusammenspiel
- Instanzen kommen aus dem Application-Context oder wurden gemockt
- langsamer in der Ausführung durch Aufsetzen der Umgebung

Unit-Test

```java
public class AddressTest {

    private final String street = "Test-Allee 1";
    private final String postalCode = "12345";
    private final String city = "Erbshausen";

    @Test
    public void instantiating() {
        Address address = new Address(street, postalCode, city);
        Assertions.assertEquals(street, address.getStreet());
        Assertions.assertEquals(postalCode, address.getPostalCode());
        Assertions.assertEquals(city, address.getPostalCode());
    }
}
```



Integrations-Test

wichtig

für Testfälle field-injection nutzen

```
@SpringBootTest
class ProductServiceTest {

   @Autowired
   ProductService productService;

   String productId = "the-product-id";
   String productName = "blah";
   Double productPrice = 1.23;


   @Test
   void createProduct_failsForDuplicateProductId() {
      // given - create product
      productService.createProduct(new Product(productId, productName, productPrice));

      // when / then - call service as lambda and check instance of expected exception
      Assertions.assertThatThrownBy(
            () -> productService.createProduct(new Product(productId, productName, productPrice))
      ).isInstanceOf(IllegalStateException.class);
   }
}
```

Mocking

Wenn das Verhalten von Bean-Klassen verändert werden soll - oder wenn für Interfaces eine individuelle Implementierung benötigt wird, so kann die Klasse oder das Interface “gemockt” werden (Tutorial).

```
@SpringBootTest
class ProductServiceTest_WithMocks {

   @MockBean
   ProductRepository productRepository;

   @Test
   void getTotalPrice() {
       // mock test fixture
       Product someDish = new Product(productId, productName, productPrice);
       Mockito.when(this.productRepository.findById(productId))
              .thenReturn(Optional.of(someDish));
            ...
   }
```

Mocking Details

Im Rahmen von Spring Testing gilt für Mocks:

- `@SpringBootTest` aktiviert Mocking-Support
- Mit `@MockBean` können neue Beans erzeugt oder bestehende ersetzt werden -- diese gelten für alle Tests der jeweiligen Testklasse, aber nicht darüber hinaus
- Gemockte Beans werden nach jeder Testmethode zurückgesetzt

Mock Verifizierung

Ein Test kann bei der Mock Instanz abfragen, ob und wie oft und wie Methoden aufgerufen wurden (Tutorial):

- Wurde findAll() einmal aufgerufen?
  ```
  Mockito.verify(mockedRepository).findAll();
  ```

- Wurde findAll() mindestens zweimal aufgerufen?
  ```
  Mockito.verify(mockedRepository, VerificationModeFactory.atLeast(3)).findAll();
  ```

- Wurde gar nicht mit dem Mock interagiert (keine Aufrufe)?
  ```
  Mockito.verifyNoInteractions(mockedRepository);
  ```

- Wurde Methode mit konkretem Argument aufgerufen?
  ```
  Mockito.verify(mockedRepository).findById("some-id");
  ```

Spying

Eine `@MockBean` mockt alle Methoden der Klasse/des Interfaces – und dies ist standardmäßig ein „tue nichts“ Implementierung.

Eine `@SpyBean` ist ein **partieller** Mock einer Klasse, der das mocken („überschreiben“) einzelner Methoden erlaubt, aber die restliche Funktionalität beibehält.

```
@SpringBootTest
class ProductServiceTest_WithMocks {

   @SpyBean
   ProductService productService;

   @Test
   void someTest() {
       // given
       Product someDish = new Product(productId, productName, productPrice);
       Mockito.when(this.productService.getProduct(productId))
              .thenReturn(someDish);
       // when

       // then
       Mockito.verify(this.productService).getProduct(productId);
   }
```

Context Slicing

Spring Boot bietet eine Reihe von `@...Test` Annotationen, die es erlauben, nur Teile des Application-Contexts hochzufahren.

Dadurch können Tests schneller ausgeführt werden (Dokumentation). Es können auch eigene Slice-Annotationen programmiert werden (Tutorial).

- `@JsonTest` -- instanziiert nur den ObjectMapper
- `@WebMvcTest` -- instanziiert nur Controller, Filter und andere Web-Komponenten
- `@DataJpaTest` -- instanziiert nur Entities, Repositories und die Datenbank

```
@JsonTest
class ProductJsonTest {

...

}
```

Übungen

- Bitte aktualisieren Sie Ihr Projekt auf Branch " 040_testing_exercise"
- z.B. mit der Kommandozeile:
  `git checkout --force 040_testing_exercise`
- ACHTUNG, dies verwirft all Ihre Änderungen (aufgrund `--force`)!

integrata cegos

Kapitel 050

Spring Data JPA

Motivation*

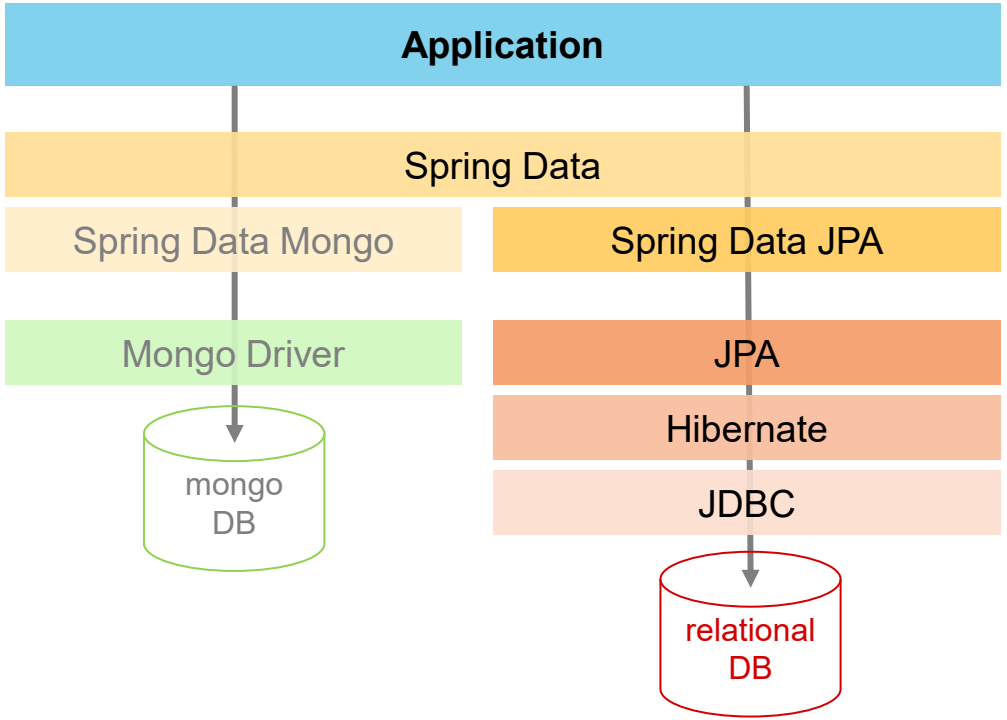
In vielen Anwendungen werden einfache und wiederkehrende Datenbankabfragen - insbesondere die einfachen Cread-Read-Update-Delete Operationen - immer wieder von Hand implementiert.

Dies ist zeitaufwändig und fehleranfällig.

Mit Spring-Data kann diese Arbeit dem Entwickler abgenommen werden.

Übersicht*

Spring Data JPA nutzt die Jakarta Persistence Architecture (JPA) im Stil der Spring Data Philosophie.



Spring-Boot-Data-Jpa Starter

Mit dem Starter “`spring-boot-starter-data-jpa`” steigt man schnell in die Persistenz auf Basis von JPA ein (Dokumentation).

Als einfache Datenbank für Entwicklungszwecke bietet sich H2 als In-Memory Datenbank an, die von Spring-Boot autokonfiguriert wird.

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

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Annotieren der Entitäten*

Model-Klassen, die als Entitäten zum Austausch von Daten genutzt werden sollen, nutzen folgende Annotationen:

Für **Klassen**:

- `@Entity`
- `@Table` -- optional
- `@Embeddable` -- für Unterobjekte ohne Id

Für **Felder**:

- `@Id`
- `@GeneratedValue` -- wenn die Id generiert werden soll
- `@Column` -- für Details wie Spaltenname, Null-Werte, Länge
- `@Embedded` -- für Unterobjekte ohne eigene Id
- `@ManyToOne` -- für eine n:1 Relation
- `@Transient` -- für auszuschließende Felder

```
@Entity
@Table(name="ORDERS")
public class Order {

   @Id
   @GeneratedValue
   Long id;

   @ManyToOne
   Customer customer;

   @Column(name = "total", nullable = false)
   Double totalPrice;

   ...
```

Automatische Schema Generierung*

Standardmäßig erzeugt JPA/Hibernate automatisch ein Datenbank-Schema basierend auf den Entitäts-Annotationen.

- Dies kann deaktiviert werden:
  `spring.jpa.hibernate.ddl-auto=none`
- und alternativ dann das Schema via Ablage einer `schema.sql` Datei erzeugt werden
  - siehe auch hier

Repositories*

Der Zugriff auf die Datenbank wird sehr komfortabel über das einfache Deklarieren von Repository Interfaces gestaltet.

Diese werden von Spring automatisch mit einer Implementierung versehen.

```
import org.springframework.data.repository.CrudRepository;

public interface ProductRepository extends CrudRepository<Product, String> {
}
```

PS Eine Annotation mit @Component oder @Repository ist nicht zwingend notwendig (siehe hier)

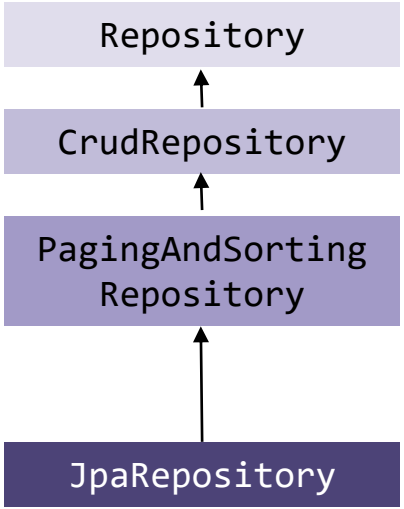
Repository Typen*

Spring stellt eine Reihe von allgemeinen Repository Typen zur Verfügung:

- `Repository` -- ein leeres Markierungs-Interface
- `CrudRepository` -- definiert viele typische CRUD (Create Read Update Delete) Methoden
- `PagingAndSortingRepository` – bietet Methoden zur Abfrage von Teilmengen und komplexer Sortierung

Zusätzlich gibt es noch Datenbank-spezifische Repository-Typen:

- `JpaRepository` – bietet u.a. query-by-example, besser nutzbare Rückgabewerte (Listen) und erlaubt das sofortige “flushen” von Entitäten



Individuelle Repository Methoden*

Um individuelle Datenbankzugriffe zu ermögliche, können neue Methoden in das Interface aufgenommen werden.

So lange diese einer gewissen Struktur folgen, wird auch hierfür die Implementation generiert (Dokumentation).

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {

    Optional<Customer> findByPhoneNumber(String phoneNumber);

}
```

Eigene Queries*

Für komplexe Queries kann die Query selbst entweder als JPQL oder natives SQL hinterlegt werden.

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {

  @Query("SELECT c FROM Customer c WHERE LOCATE(:prefix, c.phoneNumber) = 1")
  List<Customer> queryCustomersByPhoneNumberPrefix(@Param("prefix") String phoneNumberPrefix);

  @Query(value = "SELECT * FROM Customer c WHERE LOCATE(:prefix, c.phone) = 1", nativeQuery = true)
  List<Customer> queryCustomersByPhoneNumberPrefixNative(@Param("prefix") String phoneNumberPrefix);

}
```

Siehe auch Baeldung Tutorial hier oder Erläuterung hier.

Übungen

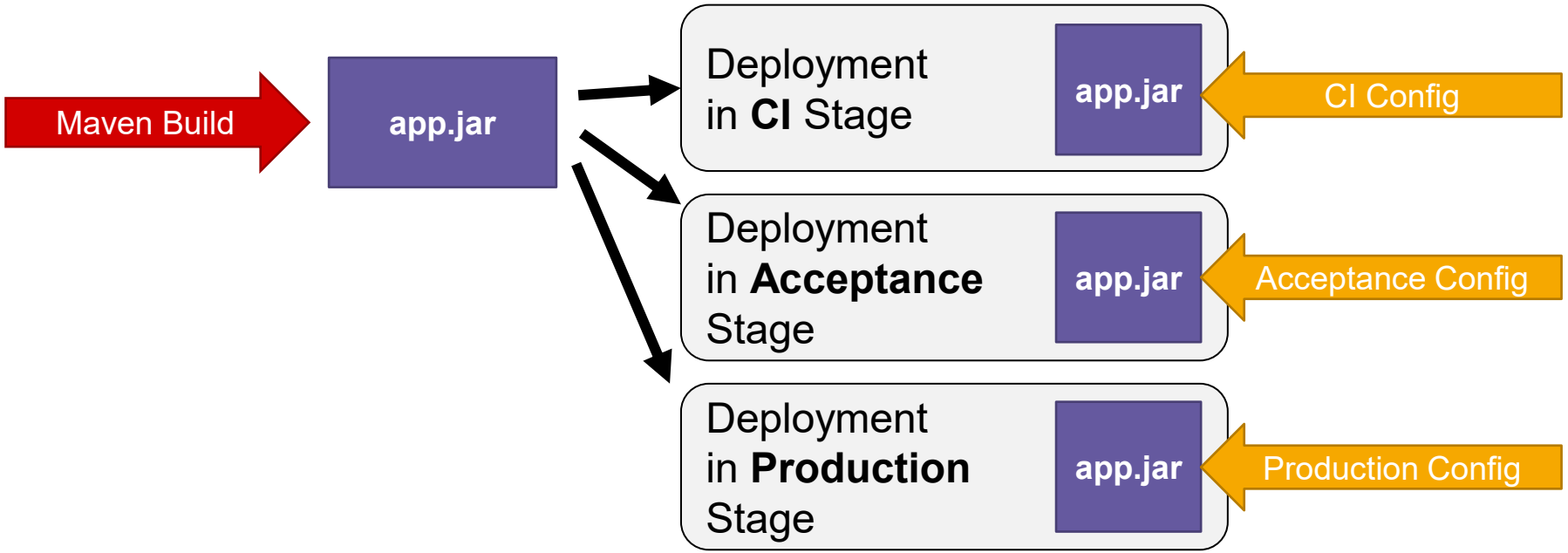
- Bitte aktualisieren Sie Ihr Projekt auf Branch "050_data-jpa_exercise"
- z.B. mit der Kommandozeile:
  `git checkout --force 050_data-jpa_exercise`
- ACHTUNG, dies verwirft all Ihre Änderungen (aufgrund `--force`)!

Kapitel 060

Konfiguration

Motivation*

Um einen im Build-Prozess erstellten Artefakt (also z.B. eine JAR Datei) in verschiedenen Umgebungen ohne Veränderung ausführen zu können



Einzelne Konfigurationswerte*

Um die eigene Geschäftslogik konfigurierbar zu machen, können Werte aus der Anwendungskonfiguration (`application.properties`) in Beans injiziert werden.

Dabei erfolgt eine automatische Konvertierung von String in den deklarierten Typ.

```
app.order.daily-discounts={MONDAY:'10'}
app.order.delivery-time-in-minutes=25
```

application.properties

```
@Service
public class OrderService {

    // use kebab-case!
    @Value("${app.order.delivery-time-in-minutes}")
    Integer deliveryTimeInMinutes;

    @Value("#{${app.order.daily-discounts}}")
    Map<String, Double> dailyDiscounts;

    // ...

}
```

OrderService.java

Konfigurationsklassen*

Um eine Vielzahl an Properties (thematisch zusammenhängend) mit wenig Aufwand zu nutzen kann man sich eine Bean dafür erstellen lassen (Dokumentation).

Achtung:

- Beispiel rechts funktioniert so nicht - braucht noch Converter für die Map (siehe hier)
- ConfigurationProperties Beans müssen explizit aktiviert werden (aufgrund Constructor-Injection in `OrderProperties`)

```
@SpringBootApplication
@EnableConfigurationProperties(OrderProperties.class)
public class PizzaApplication {
    // ...
}
```

```
@ConstructorBinding
@ConfigurationProperties("app.order")
public class OrderProperties {

    private final Integer deliveryTimeInMinutes;
    private final Map<String, Double> dailyDiscounts;

    public OrderProperties(
       Integer deliveryTimeInMinutes,
       Map<String, Double> dailyDiscounts)
    {
        this.deliveryTimeInMinutes = deliveryTimeInMinutes;
        this.dailyDiscounts = dailyDiscounts;
    }

    public Integer getDeliveryTimeInMinutes() {
        return deliveryTimeInMinutes;
    }

    public Map<String, Double> getDailyDiscounts() {
        return dailyDiscounts;
    }
}
```

Wie bzw. wo konfigurieren?

Bei Spring Boot gibt weitere Möglichkeiten, um die Konfiguration der Anwendung außerhalb des Quell-Codes zu hinterlegen.

Die wichtigsten sind:

Umgebungsvariable:

```
> set APP_ORDER_DELIVERYTIMEINMINUTES=45
> java -jar pizza.jar
```

System Property:

```
> java -Dapp.order.delivery-time-in-minutes=45 -jar pizza.jar
```

Programmargument:

```
> java -jar pizza.jar --app.order.delivery-time-in-minutes=45
```

Wer gewinnt?

Die Reihenfolge der wichtigsten Konfigurationsquellen ist wie folgt (von niedrigster zu höchster Priorität):

1. @PropertySource Annotationen
2. application.properties
3. OS Umgebungsvariablen
4. Java System/VM Properties
5. Kommandozeilenargumente
6. Testklassen Properties
7. Devtools Globale Properties

Die Spring Boot Dokumentation listet alle Möglichkeiten und deren Priorität (wer überschreibt wen) genau auf.

Konfigurations-Tipps

- Statt einer .properties Datei kann auch .yaml genutzt werden (`application.yaml`)
- In einer Konfigurationsdatei können Platzhalter genutzt werden
- Achtung, bei richtiger Schreibweise werden Umgebungs- oder Systemvariablen automatisch zum Überschreiben von Werten herangezogen, siehe Erläuterung zur Namenskonvertierung

```
app.delivery-time-in-minutes=25
```

application.properties

```
> set APP_DELIVERYTIMEINMINUTES=30
> java –jar spring-app.jar
```

shell

Tests konfigurieren

Der ApplicationContext einer Testklasse kann um eine individuelle Konfiguration ergänzt werden.
Diese Konfigurationsänderung gilt nur für die Ausführung der Testfälle in dieser Klasse.

```
@SpringBootTest
@TestPropertySource(properties = {"app.setup.customers = false"})
public class CustomerSetupTest {

   @Autowired
   CustomerService customerService;

   @Test
   public void noCustomersExist() {
       Assertions.assertFalse(customerService.getAllCustomers().iterator().hasNext());
   }
}
```

Anpassungen an Auto-Configuration Beans

Beans, die durch die Spring-Boot Auto-Configuration erzeugt wurden, konkurrieren nicht mit selbst konfigurierten Beans - sondern werden ersetzt (z.B. DataSource).

Außerdem können Teile der Auto-Configuration gezielt deaktiviert werden (Auflistung hier):

```
@SpringBootApplication(
        exclude={DataSourceAutoConfiguration.class},
        excludeName = {"not.on.classpath.SomethingAutoConfiguration"})
```

Änderung von Auto-Configuration Properties

Spring Boot benutzt Default-Werte für die Auto-Configuration. Diese können in der `application.properties` überschrieben werden (Liste).

```
spring.datasource.url=jdbc:h2:file:~/test
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Übungen

- Bitte aktualisieren Sie Ihr Projekt auf Branch "060_configuration_exercise"
- z.B. mit der Kommandozeile:
  `git checkout --force 060_configuration_exercise`
- ACHTUNG, dies verwirft all Ihre Änderungen (aufgrund `--force`)!

integrata cegos

Kapitel 080

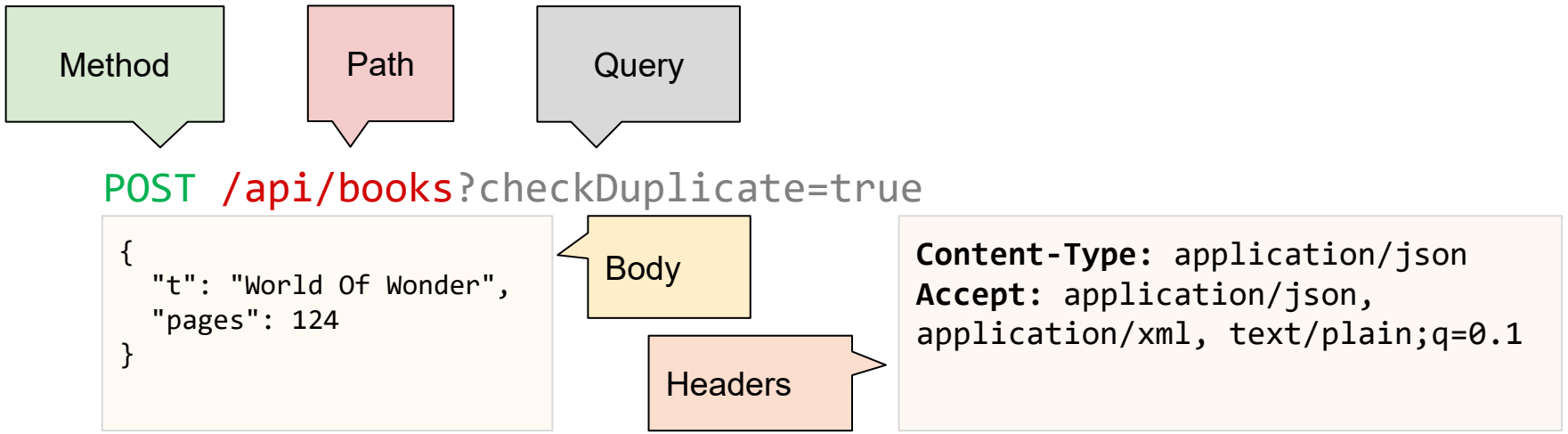
RESTful API

RESTful APIs

- REST (Representational State Transfer) ist der de-facto Standard für Datenschnittstellen im Web und basiert auf HTTP Aufrufen, wie sie auch von einem Browser an einen Webserver gesendet werden.
- Browser machen GET Aufrufe, um Inhalte zu laden.
  Und senden Daten zum Server (z.B. Formulare) über POST Aufrufe.
- Aber es gibt noch weitere HTTP-Methoden, wie z.B. PUT, DELETE und PATCH.
- All diese werde bei REST zur Beschreibung der gewünschten Aktion genutzt.

Bestandteile eines HTTP Aufrufs

Wichtig zu verstehen ist, dass jeder HTTP Aufruf an einen Server folgende Bestandteile enthält, die auch bei einer REST API genutzt werden:



REST Endpunkte

Methode	Aktion	Path	Query	Body
GET	Ein Objekt lesen	/products/123	keine	keiner
GET	Mehrere Objekte lesen	/products	Filterung, Pagination	keiner
POST	Ein neues Objekt anlegen	/products	keine	vollständige Beschreibung
PUT	Ein Objekt vollständig aktualisieren	/products/123	keine	vollständige Beschreibung
DELETE	Ein Objekt löschen	/products/123	keine	keiner
DELETE	Mehrere Objekte löschen	/products	Filterung	keiner
PATCH	Ein Objekt partiell aktualisieren	/products/123	keine	Änderungen

REST Controller

- Um mit Spring Boot REST Controller zu entwickeln, wird der “spring-boot-starter-web” Starter benötigt.
- Jeder Controller ist eine eigenständige Klasse mit annotierten Methoden.

```
@RestController
public class CustomerRestController {

   private static final String ROOT = "/customers";
   public static final String GET_ALL_ENDPOINT = ROOT;
   public static final String CREATE_ENDPOINT = ROOT;

   @GetMapping(GET_ALL_ENDPOINT)
   public Iterable<Customer> getAllCustomers() {
       // return ...
   }

   @PostMapping(CREATE_ENDPOINT)
   @ResponseStatus(HttpStatus.CREATED)
   public Customer createCustomer(@RequestBody Customer c) {
       // return ...
   }
}
```

JSON

JSON ist der de-facto Standard für die Serialisierung/Deserialisierung von Objekten zum Versand via HTTP.

Spring konvertiert automatisch Objekte und Listen von Objekten von und zu JSON (auch marshalling/unmarshalling genannt).

Hier findet sich eine Übersicht hilfreicher Artikel zu verschiedenen Aspekten und Sonderfällen.

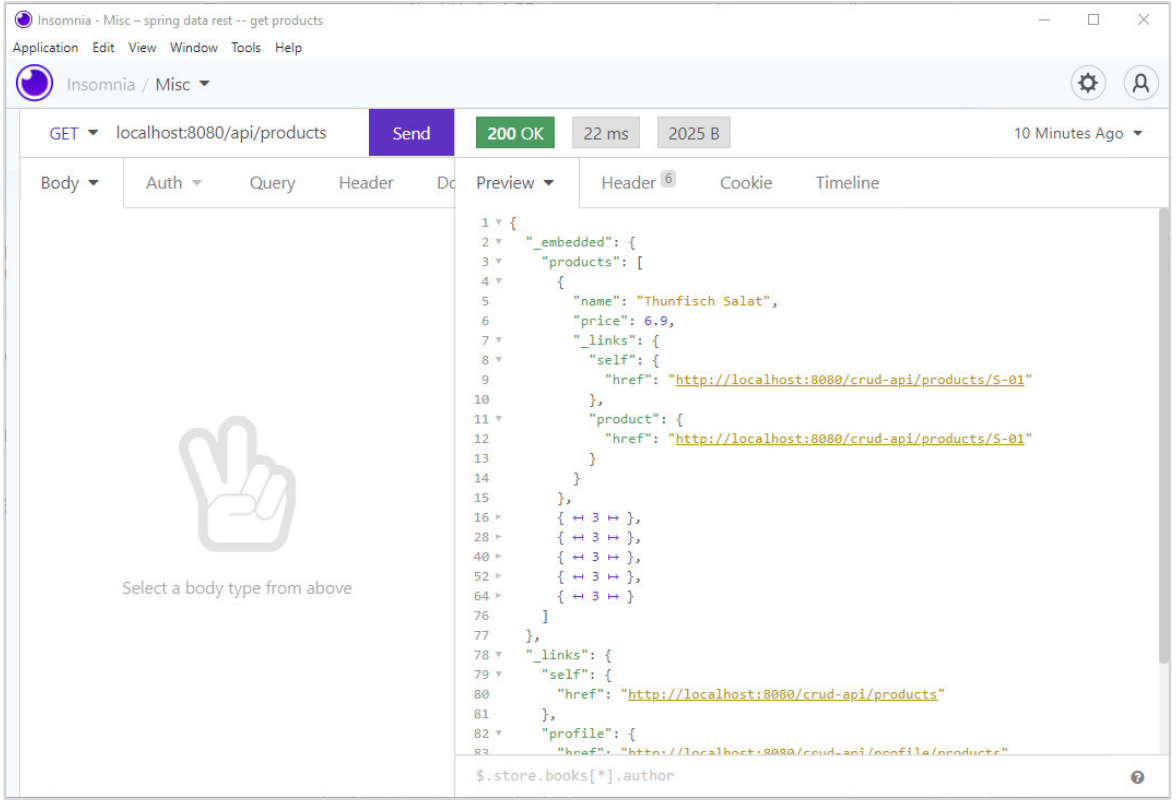
Spring Data REST

Das Spring Data REST Projekt generiert eine dynamische REST API (im HAL Format).

Dies erfolgt auf Basis des mit Spring Data Repositories definierten Domain Models.

Somit kann hier einiges an Arbeit gespart werden, um die gängigen CRUD Operationen anzubieten.

Und gelernt werden, wie die Spring Entwickler eine REST API entwerfen!



Übungen

- Bitte aktualisieren Sie Ihr Projekt auf Branch "080_configuration_exercise"
- z.B. mit der Kommandozeile:
  `git checkout --force 080_configuration_exercise`
- ACHTUNG, dies verwirft all Ihre Änderungen (aufgrund `--force`)!

integrata cegos

Kapitel 100

Profile, Logging und Monitoring

Profile

Motivation*

Spring-Profile bieten eine Möglichkeit, Teile der Anwendung zu separieren und somit je nach gewünschtem Anwendungsfall (z.B. Deployment-Umgebung) verfügbar zu machen.

- sie sind somit eine Art Ausführungsmodus -- bzw. Modi, denn es können mehrere aktiv sein
- dadurch kann die Anwendung verschlankt bzw. auch inhaltlich dynamisch angepasst werden
- ebenso können Profile zur Konfigurationssteuerung genutzt werden

Deklaration*

Jede `@Component`, `@Configuration` oder `@ConfigurationProperties` kann mit `@Profile` markiert werden, um diese beim Erstellen des Application-Contexts ein- oder auszuschließen.

Hinweis: `@Service` oder `@Repository` sind Erweiterungen von `@Component`

```
@Service
@Profile("customer")
public class CustomerService {
}
```

Deklaration*

Ebenso können `@Bean` annotierte Methoden um eine Profil-Info angereichert werden:

```
@SpringBootApplication
public class PizzaApplication {

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

   @Bean("greeting")
   @Profile("prod")
   public String resourceString() {
            // ...
   }
```

Profile Expressions

Für Ausdrücke in einer `@Profile` Annotation gelten folgende Regeln:

- `"p1, p2, p3"`: Wenn mehrere Profile angegeben werden, so muss eines von diesen aktiv sein
- `"p1 | p2 | p3"`: Gleiche Ausdruck wie zuvor, nur unter Benutzung des OR operators
- `"p1, !p2"`: Ein Profil kann negiert werden, dann darf es nicht aktiv sein
- `"p1 & p2"`: Beide Profile müssen aktiv sein
- `"p1 | (p2 & p3)"`: Wenn OR und AND gemischt wird, müssen Klammern genutzt werden

Konfiguration je Profil

Die aktiven Profile werden auch zum Laden weiterer Konfigurationsdateien herangezogen:

- `"application-{profile}.properties"` im gleichen Ordner

Beispiel: Aktive Profile sind p1 und p2, also z.B. `-Dspring.profiles.active=p1,p2`

Aktive Profil(e) setzen

Es können ein oder mehrere Profile gleichzeitig aktiv sein.

Das oder die aktiven Profile können über den `"spring.profiles.active"` Konfigurationswert gesetzt werden. Ohne eigene Profilaktivierung ist automatisch das Profil `"default"` aktiv.

Dabei gelten die gleichen Prioritätsregeln und Definitionsmöglichkeiten wie bei sonstigen Werten, also:

- in der `application.properties` Datei (nicht empfohlen)
- per Umgebungsvariable
- per Systemvariable
- per Argument zur Main-Klasse

```
D:\Temp\integrata-33089-spring-aufbau>java -Dspring.profiles.active=dies,das -jar target\pizza-10-SNAPSHOT.jar

2021-11-30 15:34:57.398 DEBUG 16420 --- [           main] com.example.pizza.PizzaApplication       : Running with
Spring Boot v2.3.1.RELEASE, Spring v5.2.7.RELEASE
2021-11-30 15:34:57.398  INFO 16420 --- [           main] com.example.pizza.PizzaApplication       : The following
profiles are active: dies,das
```

Aktives Profil für Tests

Zur Ausführung von Tests können ebenfalls aktive Profile gesetzt werden:

```
@SpringBootTest
@ActiveProfiles({"fast", "prod"})
public class CustomerSetupTest {
          // ...
}
```

integrata cegos

Logging

Logging im Code nutzen

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Service
public class OrderService {

   private static Logger LOG = LoggerFactory.getLogger(OrderService.class);

   public void foo() {
       LOG.info("Beginning execution at {} ms", System.currentTimeMillis());
       try {
           // ...
       }
       catch(SomeException e) {
           LOG.error("Something went wrong here", e);
       }
   }
```

Logging konfigurieren

Über die Anwendungskonfiguration kann das Logging konfiguriert werden (Dokumentation).

Wichtige Einstellungsmöglichkeiten sind:

- `logging.file.name` oder `logging.file.path` -- zusätzliches Datei-Logging
- `logging.file.max-size` -- definiert max. Größe einer Logdatei (roll-over)
- `logging.pattern.console` -- definiert das Ausgabeformat für die Konsole
- `logging.pattern.file` -- definiert das Ausgabeformat für die Dateiausgabe
- `logging.level.root=<level>` -- definiert Level des Root-Loggers
- `logging.level.my.package.goes.here=DEBUG` -- definiert individuelle Level

Fehlersuche mittels “debug” Property

Bei gesetztem Debug-Flag werden für eine Auswahl an elementaren Loggern (insbes. Servlet Container, Hibernate, Spring Boot) vermehrt Ausgaben erzeugt. Es ist nicht ein generelles Loggen auf DEBUG Level.

Gesetzt werden kann dies über die üblichen Konfigurationswege, also z.B.

- `debug=true` in `application.properties`
- oder Aufruf mit `"--debug"`

Fehlersuche mittels “trace” Property

Alternativ kann über das Setzen des Trace-Flags noch mehr Ausgaben erzeugt werden.

Gesetzt werden kann dies über die üblichen Konfigurationswege, also z.B.

- `trace=true` in `application.properties`
- oder Aufruf mit `"--trace"`

integrata cegos

Monitoring / Actuator

Aktivierung

Durch Hinzufügen des „actuator-starters“ wird das Monitoring der Anwendung aktiviert.

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

Abfrage

Die per HTTP veröffentlichten Endpunkte können einfach im Browser abgefragt werden:

- http://localhost:8080/actuator – listet alle Endpunkte
- http://localhost:8080/actuator/health -- Abfrage eines konkreten Endpunkts

Konfiguration

Die Dokumentation listet, welche Daten-Endpunkte per Standard aktiviert und auf welchen Kanälen (JMX oder HTTP) diese veröffentlicht sind.

Dies kann in der `application.properties` angepasst werden:

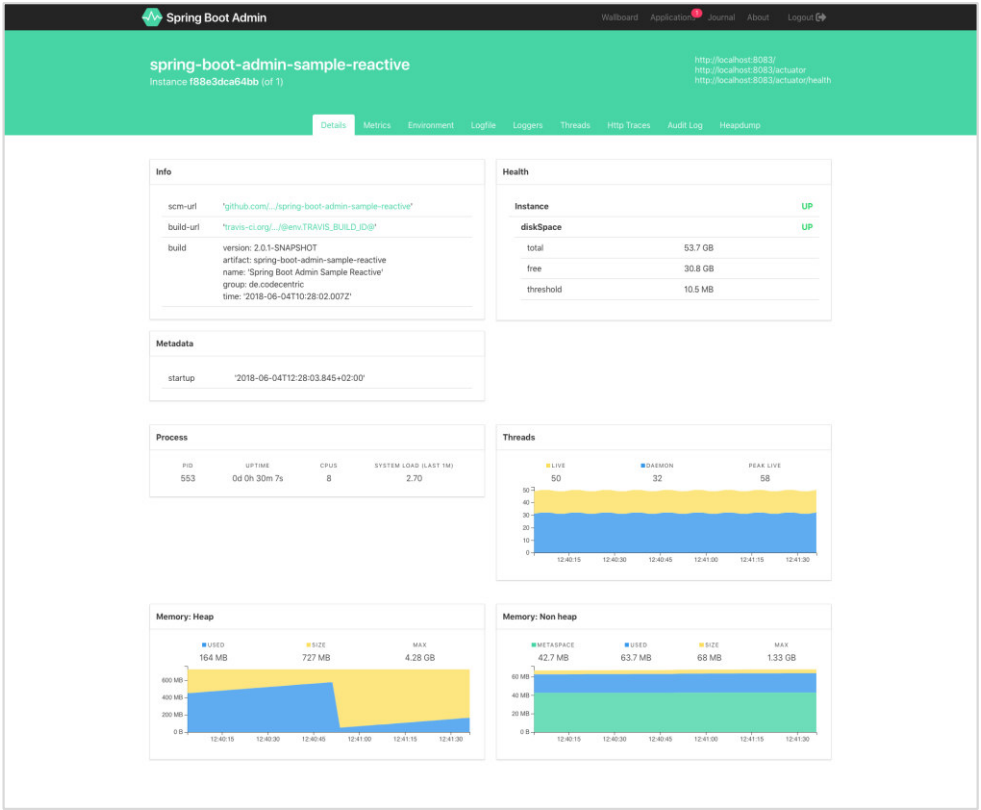
```
management.endpoints.enabled-by-default=false

management.endpoint.health.enabled=true
management.endpoint.info.enabled=true
management.endpoint.env.enabled=true

management.endpoints.web.exposure.include=health,info,env
```

Spring Boot Admin Server

Das Community Project Spring Boot Admin Server ermöglicht die Ausführung einer Web-Oberfläche, bei der sich ein oder mehrere Spring Anwendungen registrieren und von dort per Web UI administriert werden können.



Copyright und Impressum

Cegos Integrata GmbH
Zettachring 4
70567 Stuttgart