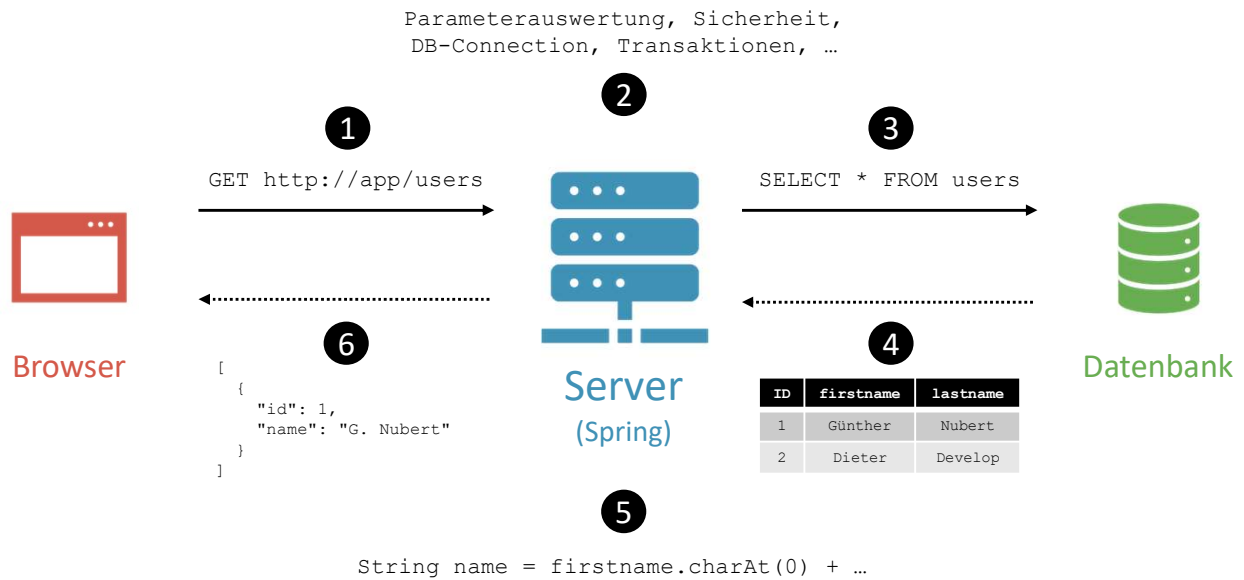


Einführung in das Spring Framework

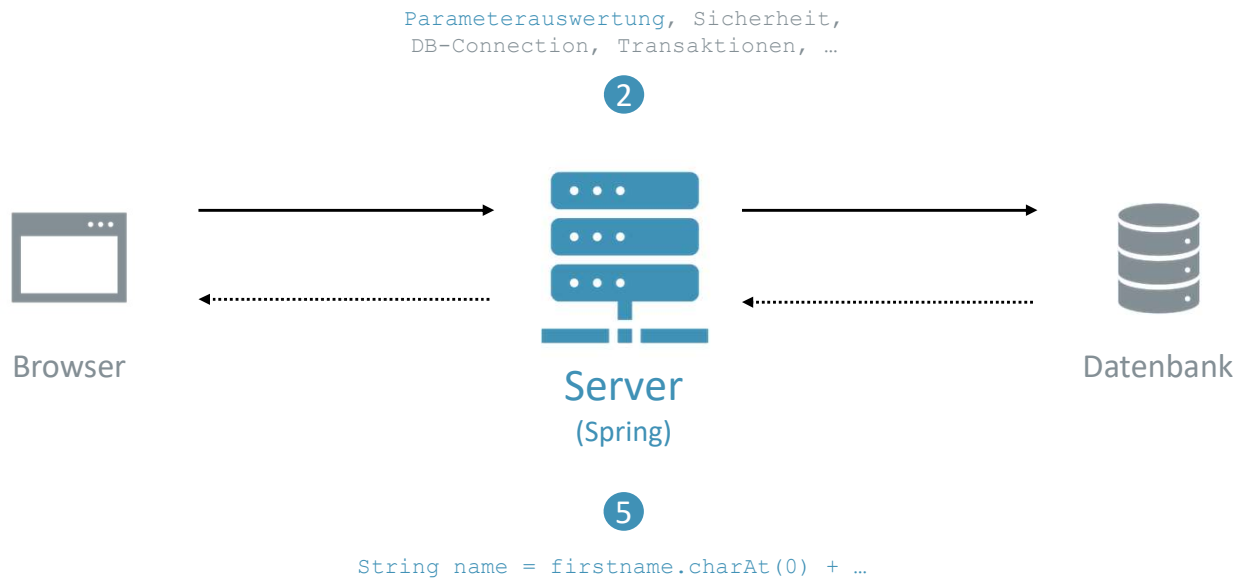
Gliederung

- Spring Grundlagen
- @Bean
- Testen
- Profile
- Properties
- AOP
- Datenbankzugriff
- Webanwendungen
- REST
- Build- und Deployment
- Betrieb – Spring Boot Actuator
- Security
- Sonstiges

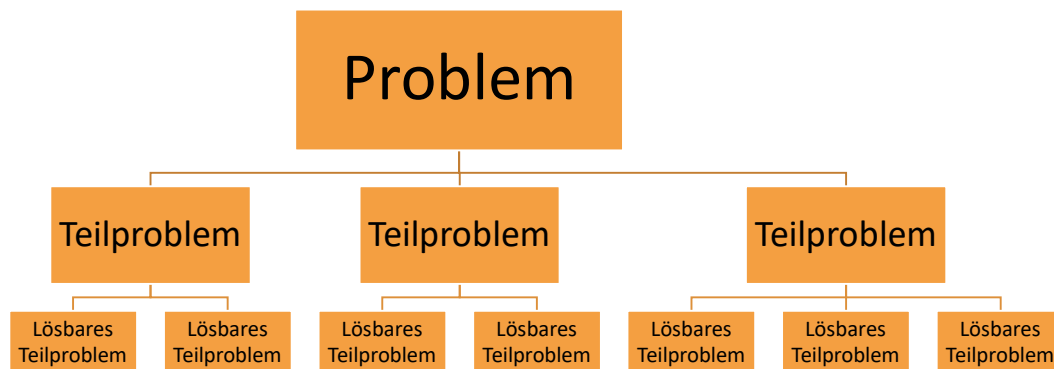
„Klassische“ Architektur



„Klassische“ Architektur



Schritt 1: Teile und Herrsche



5

Einführung in das Spring Framework

Das grundsätzliche Prinzip der Komplexitätsbeherrschung und Organisation lautet: Teile und herrsche.

Dies gilt auch für die Softwareentwicklung: Software besteht aus Teilen

Software wird in Softwaremodule aufgeteilt

Jedes Modul hat hierbei genau eine einzige Verantwortung

Ein Modul für sich alleinstehend betrachtet sollte in sich gekoppelt sein. Gibt es Teile, die nur lose angebunden sind, sind dies Kandidaten für eigene Module.

Abhängigkeiten nach außen minimieren: Ein Modul sollte nur minimale Kenntnisse über verwendete Module haben.



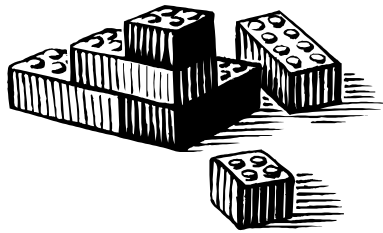
Single Responsible

Ziel: Jedes Modul erfüllt genau eine spezifische Aufgabe.

Ein Modul hat die alleinige Verantwortung über die Aufgabe.

Die Verantwortung ist ein Grund für Änderungen.

Komponenten



Komponenten

Ein zentraler Begriff bei Spring ist der Begriff der Komponente. Damit wird das Modularisierungsprinzip „Teile und Herrsche“ umgesetzt. Bei gleichzeitig durchgehender Beachtung des Prinzips „Programmieren gegen Schnittstellen“ entstehen klar strukturierte und somit gut wartbare Anwendungen.

Grafik = Microsoft Clipart

Eine erste „Teilproblemlösung“

```
public class UserService {  
    public List<User> getAllUsers() {  
        return Arrays.asList(new User("Guenther", "Nubert"));  
    }  
}
```


Spring übernimmt die Verwaltung

```
@Component  
public class UserService {  
  
    public List<User> getAllUsers() {  
        return Arrays.asList(new User("Guenther", "Nubert"));  
    }  
}
```

Schritt 2: Entkopplung



„Don't call us, we call you“

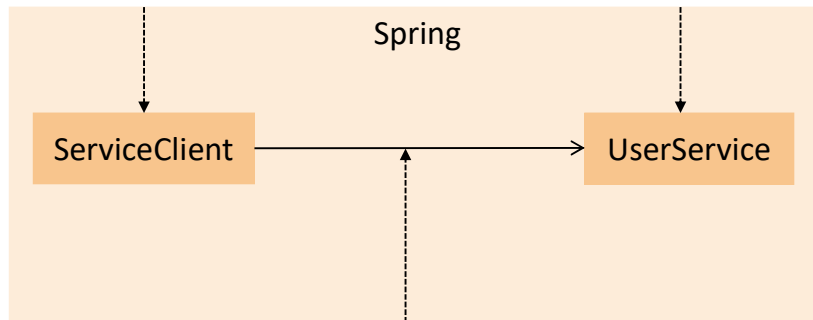
10

Einführung in das Spring Framework

Bild von Sasin.Tipchai auf Pixabay

Grafik = Microsoft Clipart

Dependency Injection



Dependency Injection

- Eine andere Komponente kann sich nun jede im Spring-Container bekannte Komponente im Konstruktor geben lassen
- Es muss **keine** Instanz manuell erzeugt werden.

```
@Component
public class ServiceClient {

    public ServiceClient(UserService userService) {
        userService.getAllUsers().forEach(System.out::println);
    }
}
```

Jede Komponente, der z.B. ebenfalls mit `@Component` markiert wird (wir werden noch andere Möglichkeiten hinzu kennenlernen), kann sich nun jede andere im Spring-Container bekannte Komponente per Konstruktor geben lassen. Die Folie zeigt ein Beispiel hierzu.

Spring Boot – Erste Schritte

Projekt-Wizard in der IDE

The screenshot shows the 'New Spring Starter Project' dialog box. It includes fields for 'Service URL' (set to <https://start.spring.io>), 'Name' (JdbcTemplate-1), 'Location' (D:\academy\FSB00T\Material\KursDisk\Loesungen\JdbcTemplat), 'Type' (Gradle (Buildship 3.x)), 'Packaging' (Jar), 'Java Version' (8), 'Language' (Java), 'Group' (de.ois.fsb00t.jdbc), 'Artifact' (JdbcTemplate-1), 'Version' (1.0), 'Description' (Demo project for Spring Boot), and 'Package' (de.ois.fsb00t.jdbc). There are also checkboxes for 'Use default location' and 'Add project to working sets', and buttons for 'New...', 'Select...', 'Back', 'Next >', 'Finish', and 'Cancel'.

Projekt über <http://start.spring.io> generieren

The screenshot shows the 'spring initializr' web interface. It has sections for 'Project' (Maven Project, Gradle Project), 'Language' (Java, Kotlin, Groovy), 'Spring Boot' (2.4.0 (SNAPSHOT), 2.4.0 (RC1), 2.3.6 (SNAPSHOT), 2.3.5, 2.2.12 (SNAPSHOT), 2.2.11, 2.1.18), 'Project Metadata' (Group: com.trivadis.spring, Artifact: user-management-test, Name: user-management-test, Description: Demo project for Spring Boot, Package name: com.trivadis.spring), 'Packaging' (Jar, War), and 'Java' (15, 11, 8). The 'Next' button is highlighted.

13

Einführung in das Spring Framework

Eine neues Spring Boot Projekt kann mit einem einfach zu bedienenden Wizard angelegt werden. Dieser Wizard ist verfügbar unter <http://start.spring.io> oder als Dialog in zahlreichen IDEs wie z.B. Spring Tool Suite (STS).

Spring Boot – Hello World

- „Main“-Klasse der Anwendung mit der Annotation **@SpringBootApplication** und der Hilfsklasse **SpringApplication**.

```
@SpringBootApplication
public class Application {

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

In der main-Methode der Anwendung wird mit Hilfe der Klasse SpringApplication ein Spring-Container gestartet. Der erste Parameter „Application.class“ gibt die Konfiguration des Containers (siehe Folie „Konfigurator“) an. Die Konfiguration ist dabei die Klasse in der sich die main-Methode befindet selbst. Die wird über die Annotation @SpringBootApplication ausgedrückt. Diese Annotation ist eigentlich eine Abkürzung für drei andere Annotation:

@Configuration: markiert die Konfiguration der Anwendung

@ComponentScan: sorgt dafür, dass Klasse mit @Component gefunden werden

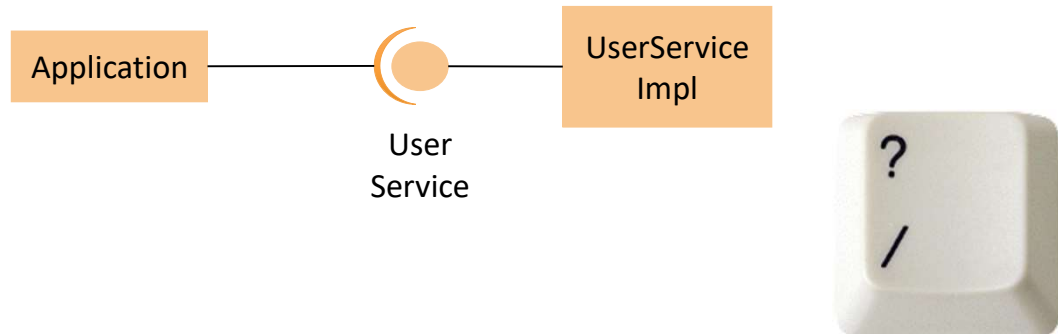
@EnableAutoConfiguration: sorgt dafür, dass der Klassenpfad nach weiteren Bibliotheken gescannt wird und startet diese ggf. in einer Standardkonfiguration (Convention over Configuration)

Warum Spring?

- **Millionenfach** über alle Branchen hinweg **im Einsatz**
- **Flexibilität** und **Erweiterbarkeit** als Grundprinzipien
- Integriert die besten Java-Tools und Framework - **Partnerschaft statt Konkurrenz**
- Adaptiert **modernste Ansätze** und **integriert Bestehendes**
- hohe **Produktivität**
 - Convention over Configuration für nahezu alle Standardaufgaben einer Enterprise-App
- **Sicher, schnell** und eine riesige **Community**

Übung

- „UserManagementBasic“
- Übungsszenario erstellen



16

Einführung in das Spring Framework

In dieser ersten Übung erstellen wir unser Übungsszenario. Dabei wird zunächst auf die Verwendung von Spring verzichtet.

- Erstellen Sie ein neues Java Projekt in der IDE mit dem Namen „UserManagementBasic“
- Legen Sie ein neues Package „com.accenture.spring.user.domain“ an.
- Legen Sie eine neue Klasse „User“ mit zwei String Feldern „firstname“ und „lastname“ an.

```
public class User {  
    private String firstname;  
    private String lastname;  
  
    public User() {  
    }  
  
    public User(String firstname, String lastname) {  
        this.firstname = firstname;  
        this.lastname = lastname;  
    }  
    // ...  
}
```

- Legen Sie ein neues Package „com.accenture.spring.user.service“ an.
- Legen Sie ein neues Interface „UserService“ in diesem Paket an.
- Fügen Sie eine neue Methode „getAllUsers“ in das Interface hinzu, die eine Liste von Benutzern zurückliefert.
- Erstellen Sie eine Klasse „UserServiceImpl“, die das Interface „UserService“ implementiert.
- Die Implementierung der Methode „getAllUsers“ liefert eine Liste mit drei Demo-Usern zurück. Geben Sie jedem User einen Vor- und einen Nachnamen.


```

public class UserServiceImpl implements UserService {

    @Override
    public List<User> getAllUsers() {
        return Arrays.asList(new User("Guenther", "Nubert"), new User("Bud",
"Spencer"), new User("Dieter", "Develop"));
    }
}

```

- Erstellen Sie eine neue Java-Klasse mit dem Namen „Application“ im Paket „com.accenture.spring.user“.
- Fügen Sie dieser Klasse einer Main-Methode hinzu und instanziiieren Sie in dieser Methode die Klasse "UserServiceImpl". Geben Sie anschließend die Liste aller User auf der Konsole aus, die Sie mit Hilfe des Service abrufen können:

```

public class Application {
    public static void main(String[] args) {
        UserService userService = new UserServiceImpl();
        userService.getAllUsers().forEach(System.out::println);
    }
}

```

Starten Sie die Anwendung als „Java Application“.

In den weiteren Übungen werden wir diese Basis zu einer Spring Anwendung umbauen.

Übung

- „UserManagementConstructor“

Project
☐ Maven Project ☒ Gradle Project

Language
☒ Java ☐ Kotlin ☐ Groovy

Spring Boot
☐ 2.4.0 (SNAPSHOT) ☐ 2.4.0 (RC1) ☐ 2.3.6 (SNAPSHOT) ☒ 2.3.5
☐ 2.2.12 (SNAPSHOT) ☐ 2.2.11 ☐ 2.1.18

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 15 ☒ 11 ☐ 8



17

Einführung in das Spring Framework

In dieser Übung schauen wir uns das Zusammenspiel von Klassen, Interfaces, Komponenten und einem Spring Boot Container an:

Legen Sie ein neues Spring mit den folgenden Daten an:

Name: UserManagementConstructor

Group: com.accenture.spring.user

Artifact: usermanagement

Package: com.accenture.spring.user (muss angepasst werden!)

Kopieren Sie die Klassen „User“, „UserServiceImpl“ und das Interface „UserService“ aus der ersten Übung „UserManagementBasic“

Machen Sie die Klasse „UserServiceImpl“ zu einer Komponente im Spring Container, indem Sie die Klasse mit der Annotation „@Component“ versehen.

```
@Component
public class UserServiceImpl implements UserService {
}
```

Erstellen Sie eine neue Klasse mit dem Namen „ServiceClient“ und implementieren Sie diese wie folgt:

```
@Component
public class ServiceClient {
    public ServiceClient(UserService userService) {
        userService.getAllUsers().forEach(System.out::println);
    }
}
```

Starten Sie die Klasse UserManagementConstructorApplication als „Java Application“.

Scope

- Der "Scope" bestimmt wie viele Instanzen einer Komponente durch den Container erstellt werden

Scope	Beschreibung
singleton	Eine Instanz der Klasse pro Java-Prozess (default!)
prototype	Eine Instanz pro Verwendung der Klasse (z.B. bei @Autowired)
session	Eine Instanz pro HTTP-Session (z.B. Warenkorb)
request	Eine Instanz pro HTTP-Request

```
@Component
@Scope(scopeName = "singleton")
public class UserServiceImpl implements UserService {
```

```
@Component
@Scope(scopeName = "prototype")
public class UserServiceImpl implements UserService {
```

Übung

- Scope



19

Einführung in das Spring Framework

Nutzen Sie als Basis für diese Übung die Übung „UserManagementConstructor“ und passen diese Übung leicht an.

Fügen Sie die Annotation „@Scope“ an der Klasse UserServiceImpl hinzu:

```
@Scope(scopeName = "prototype")
```

Damit wir unterschiedliche Scopes testen können, benötigen wir eine zweite Stelle, die den UserService verwendet und wir müssen zusätzlich erkennen welche Instanz verwendet werden. Fügen Sie hierzu zunächst eine Ausgabezeile für den Hashcode des UserService in die Klasse „ServiceClient“ ein:

```
public ServiceClient(UserService userService) {  
    System.out.println(userService.hashCode());  
    userService.getAllUsers().forEach(System.out::println);  
}
```

Kopieren Sie anschließend die Klasse ServiceClient und nennen die Kopie z.B. „ServiceClient2“

Starten Sie nun die Anwendung mehrfach mit unterschiedlichen Werten als Scope

```
prototype  
singleton
```

Dependency Injection - Field Injection

- Eine andere Komponente kann auch mit Hilfe von **@Autowired** verwendet werden.
- Es muss **keine** Instanz manuell erzeugt werden.

```
@Component
public class ServiceClient {

    @Autowired
    private UserService userService;

    public void doSomething() {
        List<User> allUsers = userService.getAllUsers();
        // ...
    }
}
```

Möchte man nun eine Komponente über ihr Interface nutzen, so kann man dies am einfachsten mit Hilfe der Annotation **@Autowired** tun. Der **ServiceClient** kennt somit nur die Schnittstelle **UserService** und nicht die auf der Folie zuvor dargestellte Implementierung. Dies ermöglicht ein einfaches Austauschen der Implementierung ohne Anpassung der Klasse „**ServiceClient**“.


Initialisierung von Komponenten 1 / 3

- Greift man im Konstruktor auf Felder zu, die per Autowiring gesetzt werden, kommt es zu einem Fehler

```
@Component
public class ServiceClient {

    @Autowired
    private UserService userService;

    public ServiceClient() {
        List<User> allUsers = userService.getAllUsers();
        // ...
    }
}
```



NullPointerException!

Aus der Reihenfolge der Initialisierung der Komponenten ergibt es bei unbedachter Verwendung ein Schwierigkeit: der Container erzeugt die Komponente „ServiceClient“ und ruft dazu den Konstruktor der Klasse auf. Falls nun im Konstruktor auf eine abhängige Komponente zugegriffen wird (in diesem Fall userService), so kann diese noch nicht gesetzt sein, da sie erst nach der Erzeugung über den Setter bzw. per Reflection gesetzt werden kann. Daher führt der dargestellte Code zu einer NullPointerException.

Die Lösungsmöglichkeiten dieser Schwierigkeit ist auf den folgenden beiden Folien dargestellt.

Initialisierung von Komponenten 2 / 3

- Statt einem Konstruktor kann man sich mit der Annotation **@PostConstruct** über die erfolgreiche Erzeugung der Komponente informieren lassen:

```
@Component
public class ServiceClient {

    @Autowired
    private UserService userService;

    @PostConstruct
    public void init() {
        List<User> allUsers = userService.getAllUsers();
        // ...
    }
}
```

Der Spring-Container erkennt bei jeder erzeugten Komponente automatisch die Annotation **@PostConstruct** und ruft die damit gekennzeichnete Methode nach der vollständigen Initialisierung und somit auch nach dem Setzen der Abhängigkeiten auf. Im dargestellten Beispiel kann man somit in der Methode „init“ davon ausgehen, dass der Container die Instanz des **UserService** bereits gesetzt hat und ein Zugriff somit möglich ist.

Initialisierung von Komponenten 3 / 3

- Alternativ zu `@PostConstruct` kann das Interface **CommandLineRunner** implementiert werden:

```
@Component
public class ServiceClient implements CommandLineRunner {

    @Autowired
    private UserService userService;

    @Override
    public void run(String... args) throws Exception {
        List<User> allUsers = userService.getAllUsers();
        // ...
    }
}
```

Die Folie zeigt eine Alternative zu `@PostConstruct`:

“Interface used to indicate that a bean should run when it is contained within a SpringApplication. Multiple CommandLineRunner beans can be defined within the same application context and can be ordered using the Ordered interface or `@Order` annotation.”

<http://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/CommandLineRunner.html>

Übung

- Komponenten mit Field Injection



Ändern Sie die bestehende Anwendung so, dass der „ServiceClient“ per Field Injection und nicht mehr per Constructor Injection auf die Komponente UserService zugreift.

Beachten Sie dabei die Reihenfolge bei der Initialisierung einer Komponente, um keine NullPointerException auszulösen.

Autowiring im Detail

- **Autowiring sucht die Komponente anhand des Typs**
- in diesem Fall „UserService“

```
@Component
public class ServiceClient {
    @Autowired
    private UserService userService;
}
```

- In nicht eindeutigen Situationen wirft Spring eine Exception
 - Spring versucht nicht zu erraten welche Komponente gemeint war

Autowiring im Detail – Namen der Komponenten

- Jede Komponente hat einen **Namen**
 - Standardmäßig der Klassenname mit kleinem Anfangsbuchstaben
 - „UserServiceImpl“ → „userServiceImpl“
- Mit @Component kann dieser Name geändert werden

```
@Component("myUserService")  
public class UserServiceImpl implements UserService {  
}
```

Autowiring im Detail – Instanz auswählen

- Variablenname = Komponentename

The diagram illustrates the autowiring process. A red dashed arrow originates from the variable name `myUserService` in the `ServiceClient` constructor and points to the `@Component("myUserService")` annotation on the `UserServiceImpl` class. Another red dashed arrow points from the equals sign (=) in the text "Variablenname = Komponentename" to the same `@Component("myUserService")` annotation. The code snippet is as follows:

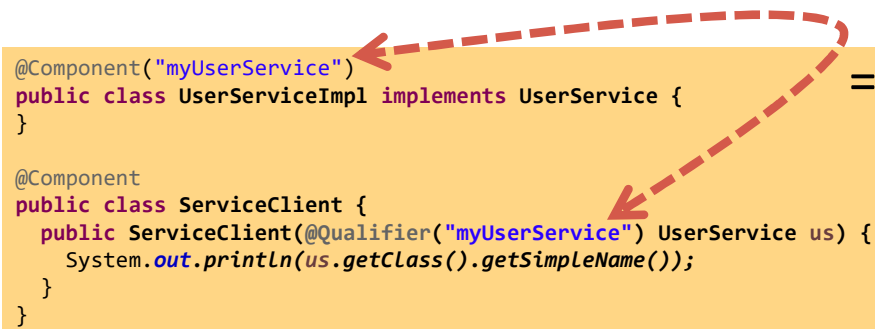
```
@Component("myUserService")
public class UserServiceImpl implements UserService {
}

@Component
public class ServiceClient {
    public ServiceClient(UserService myUserService) {
        System.out.println(myUserService.getClass().getSimpleName());
    }
}
```

Es gibt verschiedene Wege, wie eine Komponente aus einer Liste von Komponenten gleichen Typs die richtige Instanz auswählen kann. Die Folie zeigt wie diese Eindeutigkeit über die Namensgleichheit der Komponente und der Variable erreicht werden kann.

Autowiring im Detail – Instanz auswählen

- @Qualifier



```
@Component("myUserService")
public class UserServiceImpl implements UserService {
}

@Component
public class ServiceClient {
    public ServiceClient(@Qualifier("myUserService") UserService us) {
        System.out.println(us.getClass().getSimpleName());
    }
}
```

Es gibt verschiedene Wege, wie eine Komponente aus einer Liste von Komponenten gleichen Typs die richtige Instanz auswählen kann. Die Folie zeigt wie man eine Komponente anhand ihres Namens mit der Annotation @Qualifier auswählen kann.

Übung

- Autowiring eindeutig machen



Erstellen Sie eine zweite Instanz des Interfaces „UserService“.

Suchen Sie sich im „ServiceClient“ mit den gezeigten Mitteln eine der beiden Instanzen aus.

Gliederung

- Spring Grundlagen
- **@Bean**
- Testen
- Profile
- Properties
- AOP
- Datenbankzugriff
- Webanwendungen
- REST
- Build- und Deployment
- Betrieb – Spring Boot Actuator
- Security
- Sonstiges

Komponenten bekannt machen

- **Component-Scan** und **@Component**

- Ist automatisch aktiv in Spring Boot
- Base Package für den Scan ist das Package in dem die „Main“-Klasse liegt
- Alternativen: @Service, @Controller, @Repository, @Configuration, ...

- **@Configuration** und **@Bean**

- @Configuration kennzeichnet eine Konfigurationsklasse, in der Komponenten definiert werden können
- @Bean kennzeichnet eine Methode, die eine Komponente erzeugt

Im letzten Kapitel haben wir uns einen einfachen Spring-Container angeschaut, bei dem die Komponenten automatisch gesucht und konfiguriert werden. Diese automatische Suche kann man mit einer manuellen Konfiguration überschreiben.

Spring Java Config - Beispiel

- Eine einfache Java-Config-Klasse:

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

- Hinweise:
 - @Configuration wird durch den Component-Scan gefunden
 - @SpringBootApplication beinhaltet @Configuration

Component-Scan und @Component ist nicht die einzige Möglichkeit, um Komponenten in Spring bekannt zu machen. Daneben kann man auch Konfigurationsklassen mit @Configuration anlegen und daran Methoden definieren, die einzelne Komponenten per Java-Code erzeugen. Diese Methoden werden mit der Annotation @Bean gekennzeichnet.

@Scope

- Der Scope kann auch bei dieser Art der Konfiguration festgelegt werden

```
@Bean
@Scope(value = "prototype")
public A a() { return new A(); }
```

```
@Component
@Scope(value = "prototype")
class A { ... }
```

Standardmäßig gibt es in Spring von jeder Klasse, die als Komponente bekannt gemacht wird, genau eine Instanz im gesamten Container. Dies nennt man Singleton. Dieses Standardverhalten kann über die Annotation `@Scope` angepasst werden. Die dargestellte Tabelle zeigt die möglichen Scopes und deren Bedeutung.

Übung

- Konfiguration mit @Bean



34

Einführung in das Spring Framework

Öffnen Sie die Klasse „UserServiceImpl“ und entfernen Sie die Annotation @Component. Die Klasse ist somit zunächst keine Komponente mehr im Spring-Container und beim Starten der Anwendung erscheint eine Fehlermeldung:

```
Field userService in com.accenture.spring.user.UserManagementApplication
required a bean of type 'com.accenture.spring.user.service.UserService' that
could not be found.
```

Erstellen Sie eine neue Klasse „com.accenture.spring.user.UserManagementConfiguration“

Annotieren Sie diese Klasse mit @Configuration. Die Klasse wird dadurch automatisch von Spring Boot gefunden.

Erstellen Sie eine neue Methode userService, die eine Instanz von UserServiceImpl zurückliefert und mit der Annotation @Bean versehen ist:

```
@Bean
public UserService userService() {
    return new UserServiceImpl();
}
```

Starten Sie die Anwendung.

Demo

- Wie häufig wird der Konstruktor von LogServiceImpl aufgerufen?

```
@Configuration
public class UserManagementConfiguration {

    @Bean @Primary
    public UserService userService1() {
        return new UserServiceImpl(logService());
    }

    @Bean
    public UserService userService2() {
        return new UserServiceImpl(logService());
    }

    @Bean
    public LogService logService() {
        return new LogServiceImpl();
    }
}
```



35

Einführung in das Spring Framework

Standardmäßig generiert Spring Boot für jede @Configuration-Klasse mit Hilfe der CgLib eine Subklasse. Dies erlaubt sowohl 'inter-bean references' innerhalb der Konfigurationsklasse als auch externe Aufrufe zu den @Bean-Methoden dieser Konfiguration, z.B. von einer anderen Konfigurationsklasse. Wenn dies nicht erforderlich ist, da jede der @Bean-Methoden dieser speziellen Konfiguration in sich geschlossen und als reine Fabrikmethode für die Verwendung in Containern konzipiert ist, kann dieses Verhalten per Konfigurationsflag an der Annotation @Configuration oder @SpringBootApplication ausgeschaltet werden

@Configuration(proxyBeanMethods = **false**)

Das Beispiel kann im Projekt „UserManagementConfigurationExtended“ nachvollzogen werden.

Gliederung

- Spring Grundlagen
- @Bean
- **Testen**
- Profile
- Properties
- AOP
- Datenbankzugriff
- Webanwendungen
- REST
- Build- und Deployment
- Betrieb – Spring Boot Actuator
- Security
- Sonstiges

JUnit - Beispiel

- Ein einfacher JUnit-Test ohne Spring

```
public class MyTestClass {  
  
    @Before  
    public void init() { ... }  
  
    @Test  
    public void myTestMethod() {  
        int result = 1 + 1;  
        assertEquals("result of addition", 2, result);  
    }  
  
    @After  
    public void terminate() { ... }  
}
```

Das Beispiel zeigt einen einfachen JUnit-Test ohne den Einsatz von Spring

Es ist keine Ableitung von einer Basisklasse notwendig

Methoden mit der Annotation `@Before` werden vor jeder Testmethode ausgeführt

Methoden mit der Annotation `@After` werden nach jeder Testmethode ausgeführt

Die Namen der Testmethoden sind frei wählbar

Spring Testunterstützung

- Spring soll vor dem Starten der Tests die Komponenten und deren Abhängigkeiten erstellen (**@RunWith** bzw. **@ExtendWith**)
- Verwendung von **@Autowired** für Komponenten, die getestet werden sollen
- Unit-Test soll automatisch die Standard-Spring-Konfiguration verwenden (**@SpringBootTest**)

Testen mit Spring und JUnit 5

- Ein einfacher JUnit-Test mit Spring-Container

```
@ExtendWith(SpringExtension.class) // kann entfallen, da in @SpringBootTest enthalten
@SpringBootTest
public class UserServiceTest {

    @Autowired
    private UserService userService;

    @Test
    public void testGetAllUsers() {
        List<User> users = userService.getAllUsers();
        assertEquals(3, users.size());
    }
}
```

@RunWith wurde in JUnit 5 ersetzt durch @ExtendWith. Der SpringRunner muss in diesem Fall ersetzt werden durch die SpringExtension.

Übung

- Testen mit Spring Boot



40

Einführung in das Spring Framework

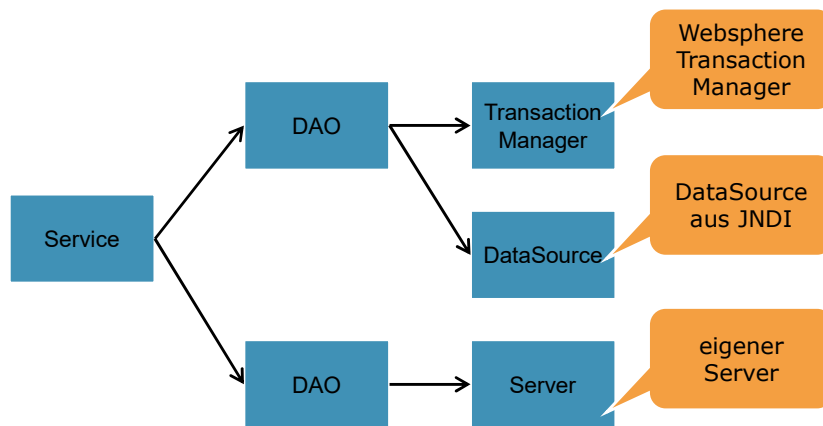
Erstellen Sie einen Test, der unsere Implementierung des UserService testet:

```
@SpringBootTest
public class UserServiceTest {
    @Autowired
    private UserService userService;
    @Test
    public void testGetAllUsers() {
        List<User> users = userService.getAllUsers();
        assertEquals(3, users.size());
    }
}
```

Fügen Sie weitere Testmethoden ein und starten Sie den Test als „JUnit-Test“.

Spezifische Konfigurationen

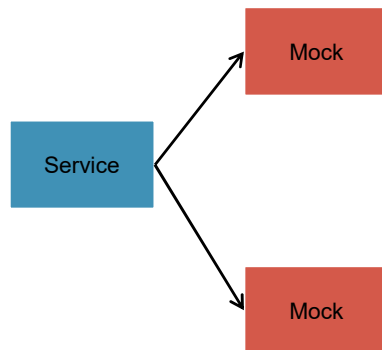
- Szenario – produktive Umgebung



Der große Vorteil dieser Art der Konfiguration ist, dass verschiedene Teile der Anwendung für verschiedene Szenarien ausgetauscht werden können. Hier ist die vollständige Konfiguration für den Produktionsbetrieb zu sehen. Auf den folgenden Seiten sind verschiedene Testszenarien abgebildet.

Spezifische Konfigurationen

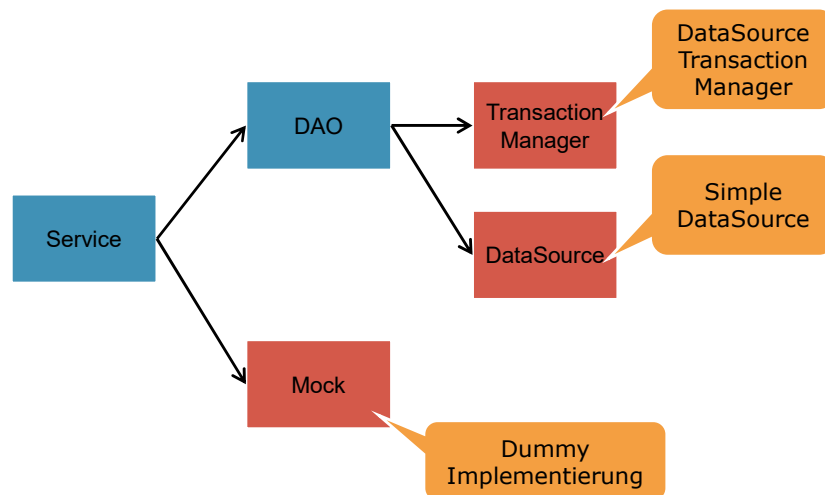
- Szenario - Unittest



Der große Vorteil dieser Art der Konfiguration ist, dass verschiedene Teile der Anwendung für verschiedene Szenarien ausgetauscht werden können. Hier ist die vollständige Konfiguration für den Produktionsbetrieb zu sehen. Auf den folgenden Seiten sind verschiedene Testszenarien abgebildet.

Spezifische Konfigurationen

- Szenario - Integrationstest



Der große Vorteil dieser Art der Konfiguration ist, dass verschiedene Teile der Anwendung für verschiedene Szenarien ausgetauscht werden können. Hier ist die vollständige Konfiguration für den Produktionsbetrieb zu sehen. Auf den folgenden Seiten sind verschiedene Testszenarien abgebildet.

Übung

- Testen mit spezifischer Konfiguration



Erstellen Sie einen Spring Boot Test mit individueller Konfiguration.

Gliederung

- Spring Grundlagen
- @Bean
- Testen
- **Profile**
- Properties
- AOP
- Datenbankzugriff
- Webanwendungen
- REST
- Build- und Deployment
- Betrieb – Spring Boot Actuator
- Security
- Sonstiges

Profiles

- Bean Definitionen in Abhängigkeit von der Laufzeitumgebung (Environment). z.B.: dev, production, test, cloud

```
@Profile("dev")
@Component
public class DataSourceDev implements DataSource {...}
```

```
@Profile("production")
@Component
public class DataSourceProduction implements DataSource {...}
```

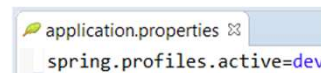
Komponenten können frei definierbaren Profilen zugewiesen werden. Diese Komponenten werden dann nur erzeugt und im Spring Container abgelegt, falls die Anwendung mit dem angegebenen Profil gestartet wird. Im Beispiel werden die beiden Profile „dev“ und „production“ eingesetzt. Das Starten mit Profilen folgt auf den nächsten Folien.

Welches Profile soll gestartet werden?

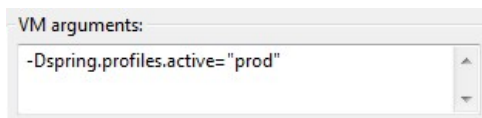
1. Umgebungsvariable



3. application.properties



2. JVM Parameter



4. JUnit-Test

```
@SpringBootTest
@ActiveProfiles(profiles = "dev")
public class DevTest {

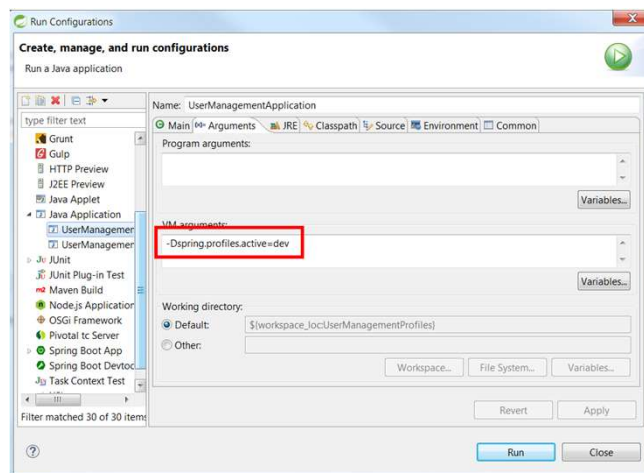
}
```

Die Folie zeigt die unterschiedlichen Varianten zum Starten einer Anwendung mit einem oder mehreren aktiven Profilen.

Alternativ zu „spring.profiles.active“ kann auch „spring_profiles_active“ als Name der Variable verwendet werden.

Übung

- Profile



48

Einführung in das Spring Framework

Fügen Sie die Annotation „@Profile“ mit dem Profilnamen „dev“ an die aktuelle Implementierung des `UserService`:

```
@Component
@Profile("dev")
public class UserServiceImpl implements UserService {
}
```

Erstellen Sie eine zweite Implementierung des Interfaces „`UserService`“ und versehen Sie diese mit dem Profil „`integrationtest`“. Die Methode „`getAllUsers`“ liefert in dieser Implementierung eine leicht abgewandelte Liste zurück.

Starten Sie die Anwendung einmal mit dem Profil „`dev`“ und einmal mit dem Profil „`integrationtest`“. Verwenden Sie hierzu den VM-Parameter „`spring.profiles.active`“ in der Run-Configuration der Anwendung.

Gliederung

- Spring Grundlagen
- @Bean
- Testen
- Profile
- **Properties**
- AOP
- Datenbankzugriff
- Webanwendungen
- REST
- Build- und Deployment
- Betrieb – Spring Boot Actuator
- Security
- Sonstiges

Properties – application.properties

- Spring Boot besitzt eine Default-Konfigurationsdatei, die bei Bedarf erweitert werden kann
- `application.properties`

```
# Spring Boot Parameter
server.port=8080

# Eigene Parameter
app.name=MyApp
app.description=${app.name} is a Spring Boot application
```

Spring Boot liefert bereits eine externe Konfigurationsdatei mit, in der die Default-Einstellungen der Anwendung überschrieben werden können. Ebenso können in dieser Datei („`application.properties`“) eigene Parameter festgelegt werden. Die Eigenschaften der Konfigurationsdatei können einfach durch eine externe Datei überschrieben werden. Dazu muss lediglich eine gleich benannte Datei neben das fertige Artefakt („`application.jar`“) gelegt werden. Alle Parameter, die in dieser externen Datei hinterlegt werden, überschreiben die Eigenschaften, die im JAR hinterlegt sind.

Properties – application.properties

- Nutzung der Konfigurationsparameter

```
@Component
public class ComponentUsingTheConfiguration {

    @Value("${app.description}")
    private String appDescription;

    @PostConstruct
    public void init() {
        System.out.println("App-Description: " + appDescription);
    }
}
```

Mit der Annotation `@Value` können extern festgelegte Konfigurationswerte (z.B. „app.description“) in Komponenten verwendet werden. Diese Werte sind – wie alle von Spring gesetzten Abhängigkeiten – im Konstruktor noch nicht verwendbar, können allerdings per Lifecycle-Callback (`@PostConstruct`) auch zur Initialisierung verwendet werden.

Properties – Listen

- Es können ebenfalls komma-separierte Listen in der Properties-Datei abgelegt werden:

```
# List of firstnames  
user.firstnames=Ben,Thomas
```

- Diese werden als Array [] in einer Komponente verwendet:

```
@Component  
public class UserServiceImpl implements UserService {  
  
    @Value("${user.firstnames}")  
    private String[] firstnames;  
  
}
```

Properties – @ConfigurationProperties und Records

- Ein Record, der auf die beiden Properties „user.firstnames“ und „user.lastnames“ zugreift:

```
@ConfigurationProperties(prefix = "user")
public record UserProperties(String[] firstnames, String[] lastnames) {
}
```

- Das Einlesen der Properties muss per Konfiguration aktiviert werden:

```
@SpringBootApplication
@EnableConfigurationProperties(UserProperties.class)
public class UserManagementApplication {
}
```

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

Properties – application.yaml

- application.yaml

```
app:
  name: My YAML App
  description: ${app.name} is a Spring Boot application
```

- Listen in YAML

```
my:
  servers:
    - dev.bar.com
    - foo.bar.com
```

Neben der Properties-Syntax kann auch eine YAML-Datei zur externen Konfiguration der Anwendung verwendet werden. Durch die erzwungene Einrückung (exakt zwei Leerzeichen) ist die Struktur der Konfigurationseinstellungen leichter erkennbar.

YAML ist ein rekursives Akronym für „YAML Ain’t Markup Language“ (ursprünglich „Yet Another Markup Language“).

Profilspezifische Properties

- Für jedes Profil kann eine eigene Konfigurationsdatei angelegt werden:
 - `application-{profile}.properties`
 - `application-{profile}.yaml`
- Beispiele:
 - `application-dev.properties`
 - `application-production.yaml`

Sowohl für Property- als auch für YAML-Dateien können profilspezifische Einstellungen hinterlegt werden. Eine solche Datei trägt dann beispielsweise den Namen „`application-dev.properties`“.

Profilspezifische Properties

- Properties - Multi-Document File

```
app.name=MyApp
#---
spring.config.activate.on-profile=test
app.name=MyTESTApp
```

- YAML - Multi-Document File

```
app:
  name: My YAML App
  description: ${app.name} is a Spring Boot application

---
spring:
  config:
    activate:
      on-profile: test
app:
  name: My YAML App FOR DEV IN ONE FILE
```

In beiden Formaten ist es mit Hilfe es sog. Multi-Document-Files auch möglich Einstellungen für mehrere Profile in einer einzigen Datei zu hinterlegen. Die Folie zeigt die notwendige Syntax.

Properties Reihenfolge

1. Devtools global settings properties on your home directory (~/.spring-boot-devtools.properties when devtools is active).
2. @TestPropertySource annotations on your tests.
3. properties attribute on your tests. Available on @SpringBootTest and the test annotations
4. Command line arguments.
5. Properties from SPRING_APPLICATION_JSON (inline JSON embedded in an environment variable or system property).
6. ServletConfig init parameters.
7. ServletContext init parameters.
8. JNDI attributes from java:comp/env.
9. Java System properties (System.getProperties()).
10. OS environment variables.
11. A RandomValuePropertySource that has properties only in random.*.
12. **Profile-specific application properties outside of your packaged jar (application-{profile}.properties and YAML variants).**
13. **Profile-specific application properties packaged inside your jar (application-{profile}.properties and YAML variants).**
14. **Application properties outside of your packaged jar (application.properties and YAML variants).**
15. **Application properties packaged inside your jar (application.properties and YAML variants).**
16. @PropertySource annotations on your @Configuration classes.
17. Default properties (specified by setting SpringApplication.setDefaultProperties).

Spring Boot verwendet eine sehr spezielle PropertySource-Reihenfolge, die so konzipiert ist, dass sie eine sinnvolle Überschreibung von Werten ermöglicht. Die Eigenschaften werden in der auf der Folie dargestellten Reihenfolge berücksichtigt.

Übung

- Properties



58

Einführung in das Spring Framework

Bearbeiten Sie die Datei „application.properties“ und fügen die folgenden zwei Properties ein:

```
user.firstnames=Ben,Thomas  
user.lastnames=Stiller,Mueller
```

Bearbeiten Sie die Klasse „UserServiceImpl“ so dass die Namen aus der Properties-Datei verwendet werden, um die User zu initialisieren:

```
@Value("${user.firstnames}")  
private String[] firstnames;  
  
@Value("${user.lastnames}")  
private String[] lastnames;  
  
@Override  
public List<User> getAllUsers() {  
    List<User> users = new ArrayList<>();  
    for (int i = 0; i < firstnames.length; i++) {  
        users.add(new User(firstnames[i], lastnames[i]));  
    }  
    return users;  
}
```

Starten und testen Sie die Anwendung.

Zusatzaufgabe 1:

Hinterlegen Sie getrennte Konfigurationsdateien für zwei unterschiedliche Profile (zb. „dev“ und „prod“) und testen Sie die Anwendung mit diesen unterschiedlichen Einstellungen. Beobachten Sie dabei besonders das

Verhalten von mehrfach hinterlegte Eigenschaften in unterschiedlichen Dateien.

Zusatzaufgabe 2:

Bauen Sie ein JAR und konfigurieren Sie für das Profile „dev“ externe Eigenschaften. Legen Sie dazu eine Konfigurationsdatei „application-dev.properties“ neben das JAR und starten Sie das JAR per Kommandozeile

Zusatzaufgabe 3:

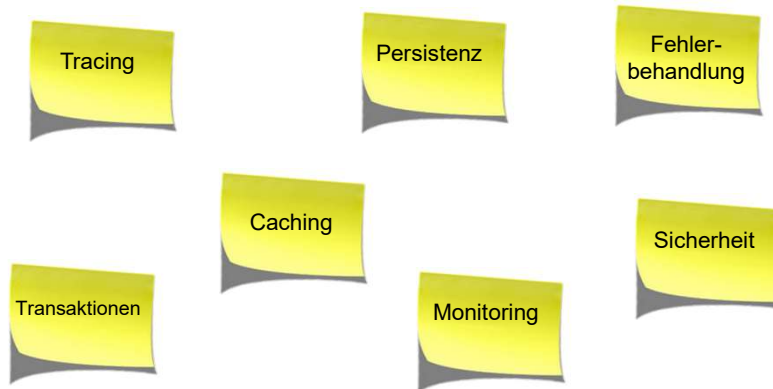
Konfigurieren Sie die Anwendung mit einer YAML-Datei anstatt der bisherigen Properties-Datei

Gliederung

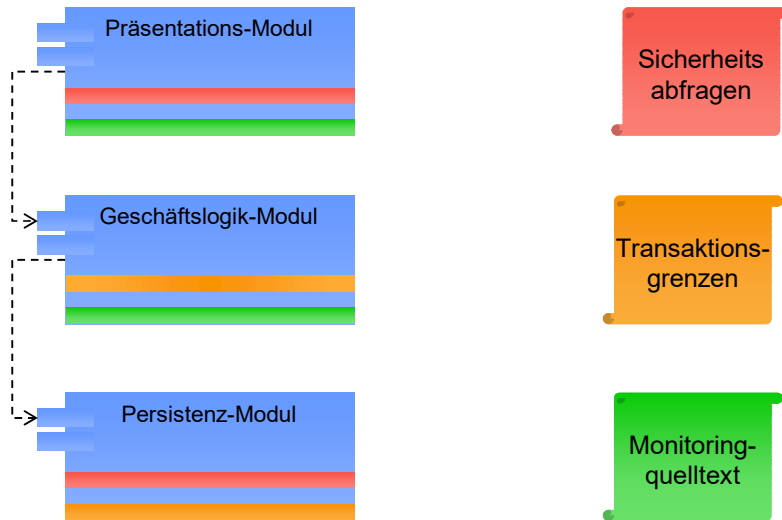
- Spring Grundlagen
- @Bean
- Testen
- Profile
- Properties
- **AOP**
- Datenbankzugriff
- Webanwendungen
- REST
- Build- und Deployment
- Betrieb – Spring Boot Actuator
- Security
- Sonstiges

AOP Begriffe

Aspekte - Querschnittsbelange



Modularisierung ohne AOP



61

Einführung in das Spring Framework

Bestandsaufnahme:

Funktionale Anforderungen sind in Module aufgeteilt

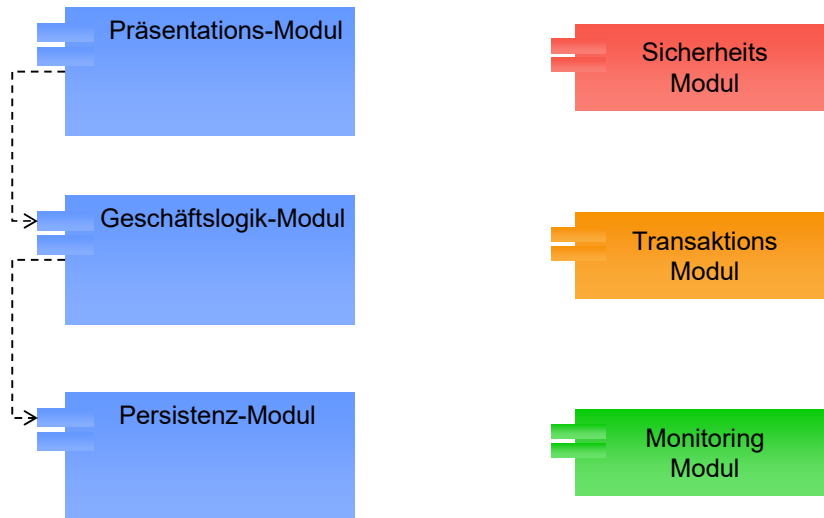
Module besitzen gerichtete Abhängigkeiten

Module enthalten Code von Querschnittsbelangen

Verletzung des DRY Prinzips

Änderungen ziehen sich durch das gesamte System

Modularisierung mit AOP

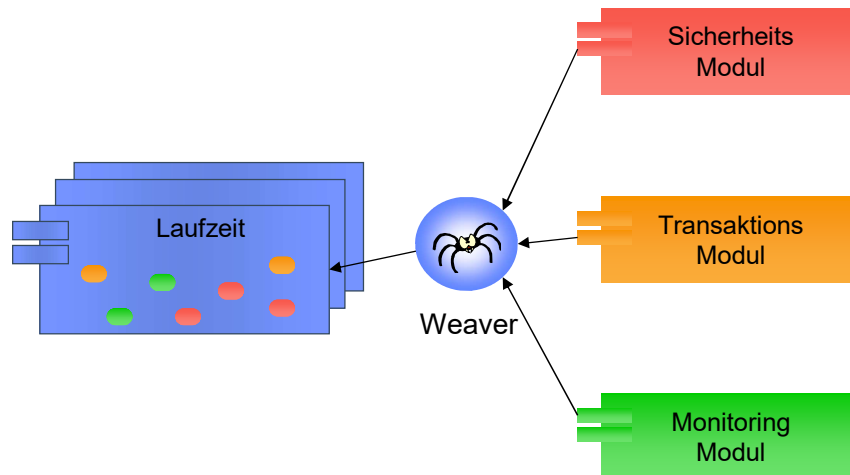


62

Einführung in das Spring Framework

Die Abhängigkeiten der Module werden aufgelöst indem die Aspekte vom Anwendungscode getrennte werden.

Modularisierung mit AOP



63

Einführung in das Spring Framework

Der Weaver / Weber webt Aspekte zur Laufzeit (zumindest bei Spring) in den Anwendungscode ein. Dazu benötigt er eine Konfiguration.

AOP Begriffe - Joinpoint

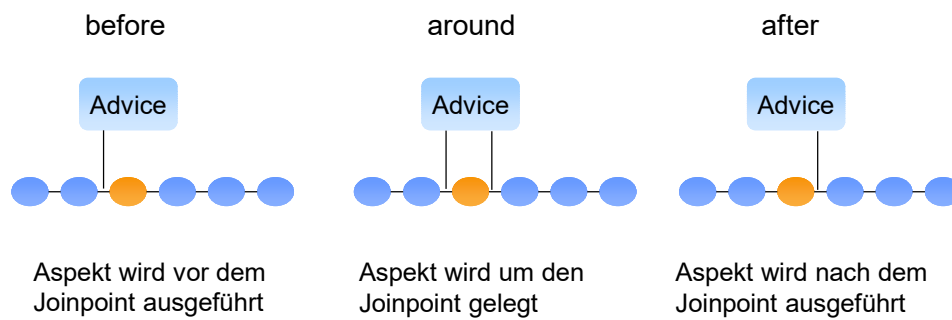
Punkt, an dem Aspekt eingeklinkt werden kann.

In Java: Bytecode Anweisungen

- Methodenaufruf
- Feldzugriffe (lesend, schreibend)
- Konstruktor Aufruf
- Behandlung einer Ausnahme

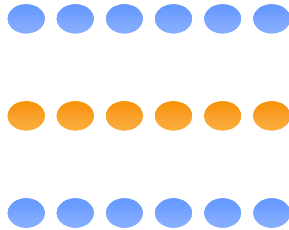
AOP Begriffe - Advice

Ein Advice ist die Implementierung eines Aspekts.

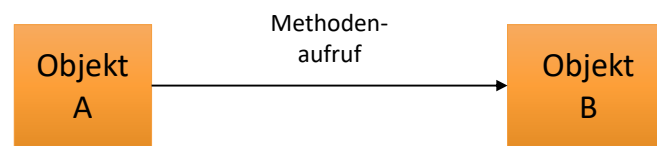


AOP Begriffe - Pointcut

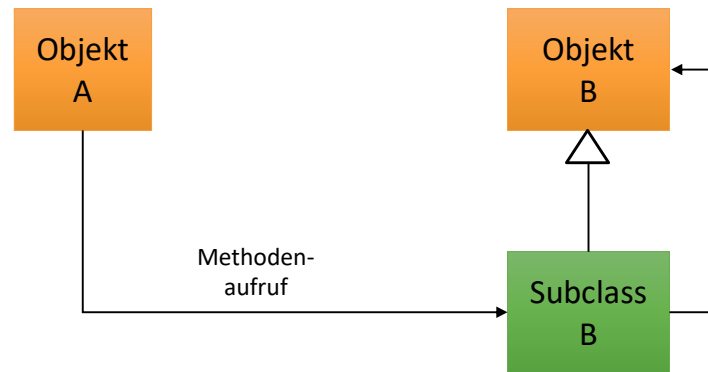
Die Adressierung eines oder mehrerer Join Points. Sie definieren an welchen Jointpoints Advices eingebunden werden sollen



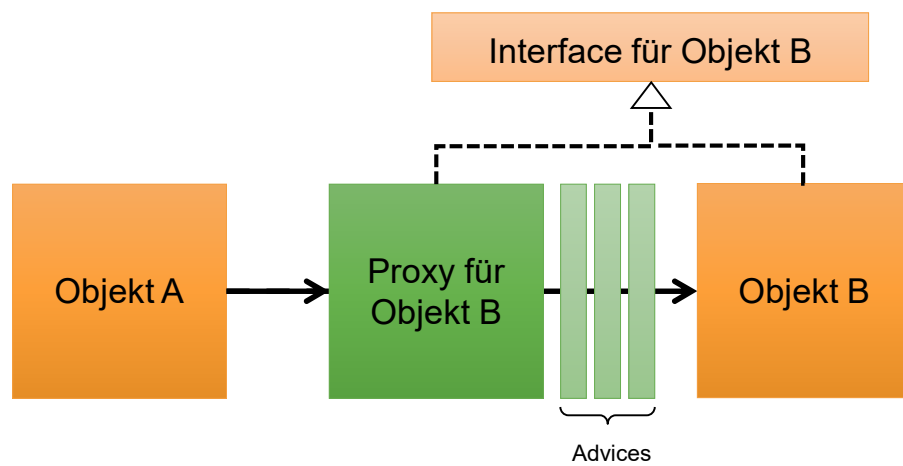
Spring AOP - CGLIB



Spring AOP - CGLIB



Spring AOP - Dynamic Proxies



Implementierung des Aspekts

```
public class PerformanceMonitor {

    public Object monitor (ProceedingJoinPoint pjp) throws Throwable {
        long start = System.currentTimeMillis();
        try {
            return pjp.proceed();
        } finally {
            long duration = System.currentTimeMillis() - start;

            System.out.println("The call of the method " +
                pjp.getTarget().getClass().getName() + "." +
                pjp.getSignature().getName() + " took " + duration + " ms");
        }
    }
}
```


AOP Konfiguration

Pointcut-Angabe direkt an Advice-Methode:

```
@Aspect
public class PerformanceMonitor {

    @Around("execution(* de.oio.fspring.services...*(..))")
    public Object monitor(ProceedingJoinPoint pjp) throws Throwable
    {
        long millis = System.currentTimeMillis();

        try {
            return pjp.proceed();
        }
        ...
    }
}
```

AOP Konfiguration mittels Annotations

Aktivierung mittels Annotation in der Konfigurationsklasse:

```
@Configuration  
@ComponentScan  
@EnableAspectJAutoProxy  
public class AppConfig {  
}
```

Pointcut Bezeichner „execution“

Syntax

`execution(modifiers-pattern? returntype-pattern declaring-type-pattern? name-pattern (param-pattern))`

- Methoden Modifier (z.B. private, public, protected)
- Rückgabewert (z.B. String, void)
- vollqualifizierter Klassenname (z.B. de.oio.fspring.MyClass)
- Methodenname (z.B. getValue)
- Methoden-Parameter (z.B. String)

Pointcut Bezeichner „execution“

Unterstützte Wildcards:

Wildcard	Beschreibung	Beispiel
*	beliebige Zeichenfolge innerhalb eines Namensraums	de.oio.*.services
..	beliebige Zeichenfolge über Namensräume hinweg	de..services
+	Implementierung des angegeben Typs	de..services.MyInterface+

Übung

Übung „Performance Monitor“



75

Einführung in das Spring Framework

Ergänzen Sie die Implementierung der Methode `UserServiceImpl.getAllUsers()` um eine zufällige Wartezeit, um den Effekt der späteren Performancemessung erkennen zu können:

```
try {  
    Thread.sleep((long) (Math.random() * 5000));  
} catch (InterruptedException e) {  
}
```

Fügen Sie eine neue Dependency für „Spring AOP“ in der Build-Datei (pom.xml oder build.gradle) hinzu:

```
Group: org.springframework.boot  
Artifact: spring-boot-starter-aop
```

Erstellen Sie ein neues Paket „com.accenture.spring.user.aop“.

Fügen Sie die Klasse „PerformanceMonitor“ in dieses neue Paket ein. Diese Klasse implementiert den Performance-Aspekt:

```
@Component  
@Aspect  
public class PerformanceMonitor {  
    @Around("execution(* com.accenture.spring.user.service..*.*(..))")  
    public Object monitor(ProceedingJoinPoint pjp) throws Throwable {  
        long start = System.currentTimeMillis();  
        try {  
            return pjp.proceed();  
        } finally {  
            long duration = System.currentTimeMillis() - start;  
            System.out.println("The call of the method " +  
pjp.getTarget().getClass().getName() + "." +  
+ pjp.getSignature().getName() + " took " + duration + " ms");  
        }  
    }  
}
```

```
}  
}
```

Starten und testen Sie die Anwendung.

Übung

Übung „Performance Monitor mit eigener Annotation“



76

Einführung in das Spring Framework

In dieser Übung wird die Performance-Messung so verändert, dass die entsprechenden JoinPoints nicht per Paket sondern stattdessen mit einer eigenen Annotation markiert werden.

Erstellen Sie hierfür zunächst eine neue Annotation mit dem Namen „Monitor“ im Paket „com.accenture.spring.user.aop“:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Monitor {
}
```

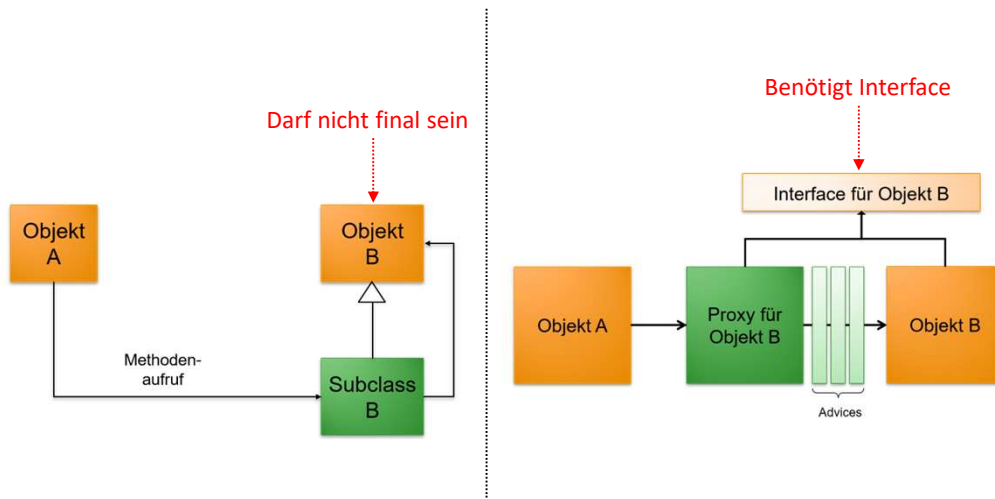
Versehen Sie nun die Methode „UserServiceImpl.getAllUsers()“ mit dieser Annotation.

Verändern Sie anschließend die Pointcut-Expression in der Klasse PerformanceMonitor wie folgt:

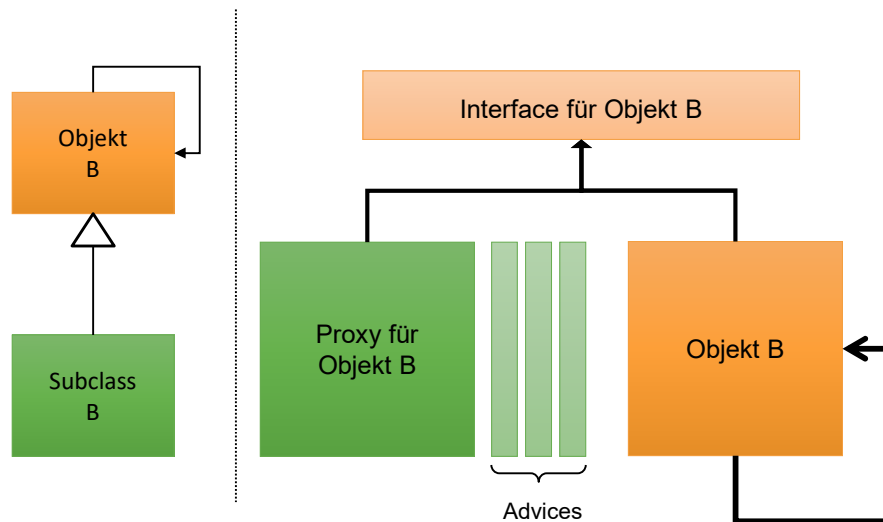
```
@Around("@annotation(com.accenture.spring.user.aop.Monitor)")
```

Starten und testen Sie die Anwendung erneut.

Spring AOP – Vergleich CGLIB mit Dynamic Proxy



Spring AOP – Eigenaufrufe



78

Was passiert wenn Objekt B eine interne Methode aufruft?
Objekt B umgeht den Proxy und somit auch die Advices.

Lösung:

Vom Spring Container den Proxy übergeben lassen und in der internen Methode über den Proxy arbeiten.

Spring AOP – Eigenaufrufe

```
@Component
public class UserServiceImpl implements UserService {

    public void doStuffSecretly() {
        getAllUsers();
    }

    @Override
    @Monitor
    public List<User> getAllUsers() {
        try {
            Thread.sleep((long) (Math.random() * 5000));
        } catch (InterruptedException e) {
        }
        return Arrays.asList(new User("Guenther", "Nubert"));
    }
}
```

Spring AOP – Eigenaufrufe - Lösung

```
@Component
public class UserServiceImpl implements UserService {

    @Autowired
    UserService selfInjectedInstance;

    public void doStuffSecretly() {
        selfInjectedInstance.getAllUsers();
    }

    @Override
    @Monitor
    public List<User> getAllUsers() {
        try {
            Thread.sleep((long) (Math.random() * 5000));
        } catch (InterruptedException e) {
        }
        return Arrays.asList(new User("Guenther", "Nubert"));
    }
}
```

Gliederung

- Spring Grundlagen
- @Bean
- Testen
- Profile
- Properties
- AOP
- **Datenbankzugriff**
- Webanwendungen
- REST
- Build- und Deployment
- Betrieb – Spring Boot Actuator
- Security
- Sonstiges

Datenzugriff ohne Spring

```
Connection con = null;
ResultSet rs;
try {
    con = ...;
    Statement stmt = con.createStatement();
    List users = new ArrayList ();
    rs = stmt.executeQuery("SELECT * FROM USERS");
    while (rs.next()) {
        users.add(new User(rs.getInt(1),...));
    }
    stmt.close();
} catch(SQLException e) {
    System.out.println("SQL-Exception:" + e);
} finally {
    con.close();
}
```

schwierig zu lesen

Ressourcen schließen?

Fehlerbehandlung?

NullPointerException?

Datenzugriff mit Spring

- Datenbankzugriff mit dem JdbcTemplate

```
public List<User> getUsers() {  
    return new JdbcTemplate(dataSource).query("SELECT * FROM USER",  
        new UserRowMapper());  
}
```

- RowMapper-Implementierung

```
public class UserRowMapper implements RowMapper<User> {  
    public User mapRow(ResultSet rs, int rowNum) {  
        return new User(rs.getInt(1)...);  
    }  
}
```

Vorteile der Spring Datenzugriffsschicht

- Einheitliche Exceptions bei unterschiedlichen Technologien
 - **DataAccessException** (RuntimeException)
 - Kein Verpflichtung zum Fangen der Exception
- Klassifizierung von Exceptions in Spring Exceptionhierarchie
 - Technologie und herstellerabhängige Exceptions werden umgesetzt
- Abstrakte Unterstützung für jede Technologie
 - Verwendung von Template Methoden

DataSource und JdbcTemplate erstellen

```
@Configuration
public class AppConfig {

    @Bean
    public DataSource dataSource() {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setDriverClassName("org.hsqldb.jdbcDriver");
        dataSource.setUrl("jdbc:hsqldb:hsqldb://localhost/springdb");
        dataSource.setUsername("SA");
        return dataSource;
    }

    @Bean
    public JdbcTemplate jdbcTemplate() {
        return new JdbcTemplate(dataSource());
    }
}
```

85

Einführung in das Spring Framework

Die gezeigte Konfiguration der DataSource und des JdbcTemplates ist in einer einfachen Spring Boot Anwendung bereits vorhanden und muss nicht manuell konfiguriert werden.

Übung

Übung „JdbcTemplate“



86

Einführung in das Spring Framework

Erstellen Sie ein neues Spring Boot Projekt mit dem Namen “JdbcTemplate” und den folgenden Abhängigkeiten

- JDBC API
- H2 Database

Öffnen Sie die Testklasse “JdbcTemplateApplicationTests” und fügen Sie den folgenden Testcode ein

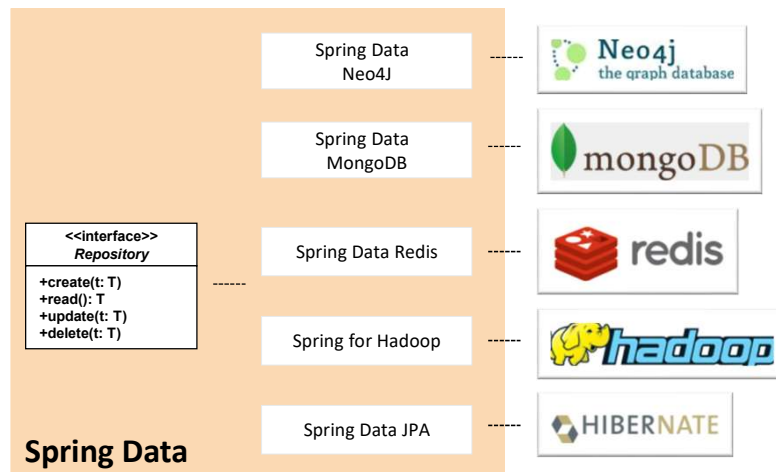
```
@SpringBootTest
public class JdbcTemplateApplicationTests {

    @Autowired
    JdbcTemplate jdbc;

    @Test
    public void testJdbcTemplate() {
        jdbc.execute("CREATE TABLE users (id int, lastname varchar(255), firstname
varchar(255))");
        jdbc.execute("INSERT INTO users (id, lastname, firstname) VALUES (1,
'Develop', 'Dieter')");
        jdbc.queryForMap("SELECT * FROM users").forEach((s, o) ->
System.out.println(s + ": " + o));
    }
}
```

Testen Sie weitere JdbcTemplate-Methoden.

Spring Data als Rahmenprojekt



Spring Data ist zunächst ein Rahmenprojekt für zahlreiche andere Projekte (z.B. Spring Data Neo4J usw.), in dem eine einheitliche Zugriffsschnittstelle auf verschiedenste Persistenzanbieter hinterlegt ist. Diese Schnittstelle nennt sich „Repository“.

Was ist Spring Data?

- Rahmenprojekt für zahlreiche Unterprojekte (z.B. Spring Data JPA)
- Generic DAOs für diverse Technologien
 - JDBC
 - JPA
 - NoSQL: Blob, Hbase, Cassandra, CouchDB, MongoDB, Neo4j, Riak, Redis, Membase und Hadoop
- Es werden ausschließlich Interfaces definiert.
- Die Implementierung wird anschließend beim Anwendungsstart dynamisch generiert.

```
public interface PersonRepository extends JpaRepository<Person, Long> {  
    List<Person> findByFirstnameAndLastname(String first, String last);  
}
```

Die Grundidee von Spring Data ist, dass lediglich eine Schnittstelle (Interface) für den Datenbankzugriff definiert werden muss und das Framework aus den darin enthaltenen Methode dynamisch eine Implementierung erstellt.

Im oben dargestellten Code-Beispiel stehen bereits mit der ersten Zeile alle CRUD-Operationen (Create, Read, Update, Delete) zur Verfügung ohne dass auch nur eine einzige Zeile Implementierungscode geschrieben werden muss.

Darüber hinaus können sog. Dynamic Finder in der Schnittstelle angegeben werden, die ebenfalls automatisch von Spring Data implementiert werden. Dies funktioniert selbstredend nur, falls man sich dabei an ein vordefiniertes Namensschema hält.

Wie funktioniert Spring Data?

- Eigenes DAO-Interface erweitert das „Repository“-Interface von Spring Data
- Beim Starten des ApplicationContexts werden alle Interfaces dieses Typs gesucht (Package-Scan) und entsprechende Implementierungen erzeugt.
- Der Dienstnutzer nutzt lediglich das Interface. Die Implementierungsgenerierung ist für ihn transparent.
- Verschiedene Strategien (query-lookup-strategy)
 - Suche nach expliziten @Query-Anweisungen
 - Namensschema (zb. findByUsername)

Basis-Interface „Repository“

```
Repository<T, ID> - org.springframework.data.repository
├── CrudRepository<T, ID> - org.springframework.data.repository
│   ├── PagingAndSortingRepository<T, ID> - org.springframework.data.repository
│   │   └── JpaRepository<T, ID> - org.springframework.data.jpa.repository
```

```
JpaRepository<T, ID> - org.springframework.data.jpa.repository
├── findAll(): List<T> - org.springframework.data.jpa.repository.JpaRepository
├── save(Iterable<? extends T>): List<T> - org.springframework.data.jpa.repository.JpaRepository
├── findAll(Sort): List<T> - org.springframework.data.jpa.repository.JpaRepository
├── flush(): void - org.springframework.data.jpa.repository.JpaRepository
├── saveAndFlush(T): T - org.springframework.data.jpa.repository.JpaRepository
├── deleteInBatch(Iterable<T>): void - org.springframework.data.jpa.repository.JpaRepository
├── findAll(Sort): Iterable<T> - org.springframework.data.repository.PagingAndSortingRepository
├── findAll(Pageable): Page<T> - org.springframework.data.repository.PagingAndSortingRepository
├── save(T): T - org.springframework.data.repository.CrudRepository
├── save(Iterable<? extends T>): Iterable<T> - org.springframework.data.repository.CrudRepository
├── findOne(ID): T - org.springframework.data.repository.CrudRepository
├── exists(ID): boolean - org.springframework.data.repository.CrudRepository
├── findAll(): Iterable<T> - org.springframework.data.repository.CrudRepository
├── count(): long - org.springframework.data.repository.CrudRepository
├── delete(ID): void - org.springframework.data.repository.CrudRepository
├── delete(T): void - org.springframework.data.repository.CrudRepository
├── delete(Iterable<? extends T>): void - org.springframework.data.repository.CrudRepository
├── deleteAll(): void - org.springframework.data.repository.CrudRepository
```

90

Einführung in das Spring Framework

Die dargestellte Basis-Schnittstelle „JpaRepository“ beinhaltet bereits alle CRUD-Methoden.

Konfiguration von Spring Data

In Spring Boot mit **@EnableAutoConfiguration** muss Spring Data überhaupt nicht konfiguriert werden.

Für eine manuelle Konfiguration ist eine Annotation ausreichend:

```
@Configuration
@EnableJpaRepositories(basePackages = "de.oio.fspring")
public class AppConfig {
    ...
}
```

Die Konfiguration von Spring Data ist denkbar einfach: bei Verwendung von **@SpringBootApplication** (und somit **@EnableAutoConfiguration**) ist keinerlei Konfiguration notwendig. Eine manuelle Konfiguration kann mit einer Annotation vorgenommen werden.

Dynamic finder - Supported keywords

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstname, findByFirstnames, findByFirstnameEquals	... where x.firstname = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	... where x.age <= ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	... where x.age >= ?1
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1
IsNull	findByAgeIsNull	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	... where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	... where x.age not in ?1
True	findByActiveTrue()	... where x.active = true
False	findByActiveFalse()	... where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	... where UPPER(x.firstname) = UPPER(?1)

92

Einführung in das Spring Framework

Quelle: <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>

Dynamic finder - Supported prefixes

- find...By
- read...By
- query...By
- count...By
- get...By

Property expressions

- Properties sind nicht immer eindeutig

```
List<Person> findByAddressZipCode (...);
```



- Mit Hilfe eines „_“ kann der Zugriff eindeutig gemacht werden:

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

Paging

- Direkte Unterstützung für Paging

```
Page<User> findByLastname(String lastname, Pageable pageable);  
  
Slice<User> findByLastname(String lastname, Pageable pageable);  
  
List<User> findByLastname(String lastname, Sort sort);  
  
List<User> findByLastname(String lastname, Pageable pageable);
```

Limitierung

- Die Ergebnismenge kann auf verschiedene Arten eingeschränkt werden:

```
User findFirstByOrderByLastnameAsc();  
  
User findTopByOrderByAgeDesc();  
  
Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);  
  
Slice<User> findTop3ByLastname(String lastname, Pageable pageable);  
  
List<User> findFirst10ByLastname(String lastname, Sort sort);  
  
List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

@Query

JPQL-Queries können mit der Annotation `@Query` als Implementierung für eine Methode angegeben werden:

```
@Query("select a from Account a where a.bankCode = :bankCode and  
a.accountNumber = :accountNumber")  
Account findByBankCodeAndAccountNumber(@Param("bankCode") int bankCode,  
@Param("accountNumber") int accountNumber);
```

Falls die Methodensignatur nicht der Syntax eines „Dynamic Finders“ entspricht, kann man mit der Annotation `@Query` die Implementierung der Methode mit Hilfe eines JPQL-Statements selbst angeben. Platzhalter können dabei mit der Annotation `@Param` festgelegt werden.

Modifying queries

Falls diese Queries schreibenden Zugriff auf die Datenbank haben, kommt zusätzlich die Annotation `@Modifying` zum Einsatz

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

Eigene Methoden in DAOs

Implementierung von eigenen Methoden in den DAOs über eine abstrakte Klasse ist NICHT möglich!

```
interface PersonRepositoryCustom {  
    public void someCustomMethod(Person person);  
}
```

```
class PersonRepositoryImpl implements PersonRepositoryCustom {  
    public void someCustomMethod(Person person) {  
        // ...  
    }  
}
```

```
public interface PersonRepository extends  
    JpaRepository<Person, Long>, PersonRepositoryCustom
```

Wenn die „Dynamic Finder“ und die Angabe von JPQL-Statements nicht ausreicht, kann eine Methoden implementieren, die von Spring Data an der entsprechenden Stelle eingebunden werden. Dazu ist die Erstellung eines neuen Interfaces (im Beispiel „PersonRepositoryCustom“) notwendig, in das die Methodensignatur der zusätzlichen Methode eingetragen wird. Alle Aufrufe an diese Methode werden anschließend über die zur Laufzeit erstellte Implementierung des Interfaces PersonRepository an die implementierende Klasse (in diesem Fall „PersonRepositoryImpl“ weitergeleitet.

Wichtig ist hierbei die Beachtung des dargestellten Namensschemas.

Was gibt es noch?

- RepositoryFactorySupport kann zum Einsatz außerhalb eines Spring-Containers eingesetzt werden.

```
RepositoryFactorySupport factory = ... // Instantiate factory here  
PersonRepository repository = factory.getRepository(PersonRepository.class);
```

Die Folie zeigt die Verwendung von Spring Data außerhalb eines Spring Containers.

Übung

Übung „Spring Data JPA“



101

Einführung in das Spring Framework

Fügen Sie eine Spring Boot Starter Dependency für „Spring Data JPA“ und für die Datenbank „H2 Database“ in der Build-Datei hinzu.

Machen Sie aus der Klasse „User“ eine JPA-Entität

- Fügen Sie die Annotationen „@Entity“ und „@Table(name = „users“)“ an die Klasse an.
- Fügen Sie ein neues Long-Feld „id“ ein und versehen Sie dieses mit den Annotationen „@Id“ und „@GeneratedValue“.
- Fügen Sie Getter- und Setter-Methoden für die neue Id hinzu.

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
```

```
@Entity
```

```
@Table(name = "Users")
```

```
public class User {
```

```
    @Id
    @GeneratedValue
    private Long id;
    private String firstname;
    private String lastname;
```

```
    public Long getId() {
        return id;
    }
```



```

    }

    public void setId(Long id) {
        this.id = id;
    }

    // ...
}

```

Erstellen Sie ein neues Paket „com.accenture.spring.user.repository“.

Erstellen Sie ein neues Interface „UserRepository“ in diesem Paket und lassen Sie es das Interface „JpaRepository<User, Long>“ erweitern.

Ändern Sie die Klasse „UserServiceImpl“ so, dass die User fortan aus der Datenbank geladen werden. Verwenden Sie hierzu das soeben erstellte Interface „UserRepository“:

```

@Component
public class UserServiceImpl implements UserService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

}

```

Falls wir die Anwendung nun starten, werden die User aus einer In-Memory-H2 geladen. Da diese allerdings leer ist, müssen wir zunächst noch dafür sorgen, dass User in dieser Datenbank gespeichert werden. Dies machen wir über eine neue Klasse namens „com.accenture.spring.user.repository.DatabaseInitializer“:

```

@Component
public class DatabaseInitializer {

    @Autowired
    private UserRepository userRepository;

    @PostConstruct
    public void fillDatabase() {
        userRepository.saveAll(Arrays.asList(
            new User("Guenther", "Nubert"),
            new User("Bud", "Spencer"),
            new User("Dieter", "Develop")));
    }

}

```

Übung

Übung „Spring Data JPA Extended“



102

Einführung in das Spring Framework

Fügen Sie die Methode „Set<User> findByLastname(String lastname);“ in das Interface „UserRepository“ ein. Schreiben Sie einen JUnit-Test, der diese neue Methode im Zusammenspiel mit den Testdaten aus der Übung Spring Data JPA testet:

```
@SpringBootTest
public class UserRepositoryTest {

    @Autowired
    private UserRepository userRepository;

    @Test
    public void testFindByLastname() {
        Set<User> users = userRepository.findByLastname("Spencer");
        assertEquals(1, users.size());
        Optional<User> firstUser = users.stream().findFirst();
        assertTrue(firstUser.isPresent());
        assertEquals("Bud", firstUser.get().getFirstname());
    }
}
```

Fügen Sie zwei weitere dynamische Finder in das Interface „UserRepository“ ein:

























```
public interface UserRepository extends JpaRepository<User, Long> {
    Set<User> findByLastname(String lastname);
    User findByFirstnameAndLastname(String firstname, String lastname);
    List<User> findByLastnameLikeOrderByLastnameDesc(String lastnameLike);
}
```

Schreiben Sie für die beiden neuen Methoden ebenfalls einen passenden JUnit-Test und testen Sie die Anwendung.

Spring Data Jdbc

- **JDBC-basierte (und nicht JPA-basierte) Repositorys**
 - Repositorys sind ein Konzept aus Domain Driven Design von Eric Evans
- Spring Data JDBC zielt darauf ab, konzeptionell **einfach** zu sein
 - KEIN Caching, Lazy Loading, Write-Back oder viele andere Funktionen von JPA
- **Aggregate Root**
 - ein Repository pro Aggregate Root
 - Ebenfalls aus Domain Driven Design
 - Aggregate Root beschreibt eine Entität, die den Lebenszyklus anderer Entitäten steuert

Spring Data Jdbc

	JdbcTemplate	Spring Data Jdbc	Spring Data JPA
Automatische Ressourcenverwaltung (Connection, Statements, ...)			
DataAccessException Hierarchie			
Einfachere CRUD-Operationen			
Interface-basierter Zugriff			
@Query-Methoden			
Verwaltung von Beziehungen („Automatische Joins“)			
Dynamic Finder			
Lazy Loading, Dirty Checking, Caching, ...			

Transaktionseigenschaften - ACID

- **A**tomicity / Atomarität
 - Ausführung vollständig oder gar nicht
- **C**onsistency / Konsistenz
 - Konsistenter Zustand wird in anderen konsistenten Zustand überführt
- **I**solation
 - Keine Transaktion merkt, dass andere Transaktionen parallel laufen.
- **D**urability / Dauerhaftigkeit
 - Die Ergebnisse einer erfolgreichen Transaktion werden gegen Versagen der Hard- und Software gesichert und können nur durch eine weitere Transaktion wieder geändert werden.

Transaktionen in Spring

- **@Transactional** macht eine Methode transaktional
 - Transaktion wird beim Betreten gestartet
 - Transaktion wird beim Verlassen **committet**
 - Falls eine Exception aufgetreten ist, wird ein **Rollback** durchgeführt

```
@Transactional
public void createUser(User user) {
    logEntryRepository.save(new LogEntry("User created"));
    userRepository.save(user);
}
```

Alle Datenbank-Aktionen, die innerhalb einer Methode ausgelöst werden, können mit der Annotation **@Transactional** in einen gemeinsamen Transaktionskontext verbunden werden. Falls eine der Aktionen fehlschlägt, werden alle Datenbankaktionen, die seit dem Betreten der Methode ausgelöst wurden, rückgängig gemacht.

In diesem Beispiel verbleibt der Logeintrag nur dann in der Datenbank, falls das Anlegen des Users im Nachgang erfolgreich ausgeführt werden kann.

Transaction Timeout

- Mit dem Attribut „timeout“ kann das Transaktionstimeout in Sekunden angegeben werden

```
@Transactional(timeout = 60)
public void createUser(User user) {
    // ...
}
```


Read only

- Bei ausschließlich lesenden Methoden kann ein entsprechender Hint gesetzt werden:

```
@Transactional(readOnly = true)
public List<User> getAllUsers() {
    // ...
}
```

Dieser Hint wird durch den Transactionmanager an die Datenbank weitergereicht. Ob dies letztlich zu einer Verbesserung der Performance führt oder nicht, liegt in den Händen der Datenbank und kann auf Seiten von Spring nicht entschieden werden.

Phenomena of concurrent transactions

- Dirty Reads
 - Lesen von nicht commiteten Daten
- Non-repeatable Reads
 - Zweimaliges Lesen in einer Transaktion liefert unterschiedliche Datenmenge
- Phantom Read
 - Spezialfall von Non-repeatable Reads
 - Beim zweiten Lesen kommen neue Datensätze hinzu (Phantome)

Dirty Reads

Transaktion B liest geänderte Daten einer Transaktion A, die noch nicht per Commit abgeschlossen wurde. Nach einem Rollback der Transaktion A hat Transaktion B Daten erhalten, die so nie in die Datenbank geschrieben wurden.













Non-repeatable Reads

Transaktion A erhält beim wiederholten Lesen Daten, die durch eine Transaktion B seit dem initialen Lesezugriff verändert und durch einen Commit abgeschlossen wurde. Transaktion A sieht unterschiedliche Daten bei dem exakt gleichen Lesezugriff.

Phantom Read

Transaktion A erhält eine Ergebnismenge, die für einen spezifizierten Filter zutreffen. Transaktion B ändert Daten (oder fügt Datensätze hinzu) derart, dass diese den Konditionen von Transaktion A entsprechen. Bei einem wiederholten Lesen von Transaktion A wurde die Ergebnismenge erweitert. Die hinzugekommenen Datensätze werden als „Phantoms“ bezeichnet.

Isolation Level

	Dirty Read	Non-repeatable Read	Phantom Read
Read uncommitted			
Read committed			
Repeatable Read			
Serializable			

```
@Transactional(isolation = Isolation.READ_UNCOMMITTED)
```

110

Einführung in das Spring Framework

Read uncommitted

Lesen von Daten einer anderen Transaktion möglich, die noch nicht durch einen Commit abgeschlossen wurden.

Read committed

Lesen von Daten einer anderen Transaktion möglich, die durch einen Commit abgeschlossen wurde.

Repeatable Read

Wiederholtes Lesen mit den exakt gleichen Daten einer Ergebnismenge möglich.

Serializable

Exklusiver Zugriff auf Daten.

Transaction Propagation

Transaktionsattribut	Vorher...	Nachher...
Required	none	T2
	T1	T1
RequiresNew	none	T2
	T1	T2
Mandatory	none	error
	T1	T1
NotSupported	none	none
	T1	none
Supports	none	none
	T1	T1
Never	none	none
	T1	error

Transaktionales Verhalten über mehrere Schichten

- Transaktion im Service

```
@Transactional(propagation=Propagation.REQUIRED)
public class CardServiceImpl implements ICardService {

    public List getCards() { ... }
}
```

- Transaktion im DAO

```
@Transactional(propagation=Propagation.MANDATORY)
public class CardDAOImpl implements IDAOImpl {

    @Transactional(readOnly=true)
    public List getCards() { ... }
}
```

Übung

- UserManagementDBTransaction
- Transaktionen



113

Einführung in das Spring Framework

Wir möchten in dieser Übung einen Fehler beim Zugriff auf die Datenbank auslösen, der in der Folge zum Rollback der Transaktion führt. Hierzu müssen wir zunächst eine mögliche Fehlerquelle konstruieren. Nehmen wir hierzu an, dass die Nachnamen unserer „User“ eindeutig sein müssen und hinterlegen wir dies entsprechend im Datenmodell

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private Long id;
    private String firstname;
    @Column(unique = true) // hinzufügen
    private String lastname;

    //...
```

Erstellen Sie eine neue Entität mit dem Namen „LogEntry“, die in der Lage ist, eine „message“ (String) zu speichern.

Erstellen Sie passend zur Entität LogEntry ein Spring Data JPA Repository „LogEntryRepository“.

Fügen Sie in die Klasse „UserServiceImpl“ die folgende neue Methode ein:

```
@Transactional
public void createUser(User user) {
    logEntryRepository.save(new LogEntry("User created"));
    userRepository.save(user);
}
```

Erstellen Sie anschließend eine neue Testklasse „com.accenture.spring.user.service.UserServiceTest“ und testen Sie darin die Transaktionalität der gerade erstellten Methode. Dies kann beispielsweise wie folgt aussehen:

```

@SpringBootTest
public class UserServiceTest {

    @Autowired private UserService userService;
    @Autowired private LogEntryRepository logEntryRepository;
    @Autowired private UserRepository userRepository;

    @Test
    public void testTransactional() {
        userRepository.deleteAll();
        assertEquals(0, userRepository.count());
        assertEquals(0, logEntryRepository.count());

        userService.createUser(createUserWithLastname("Maier"));
        assertEquals(1, userRepository.count());
        assertEquals(1, logEntryRepository.count());

        try {
            userService.createUser(createUserWithLastname("Maier"));
            fail("Exception should be thrown");
        } catch (DataAccessException e) {
            assertEquals(1, userRepository.count());
            assertEquals(1, logEntryRepository.count());
        }
    }

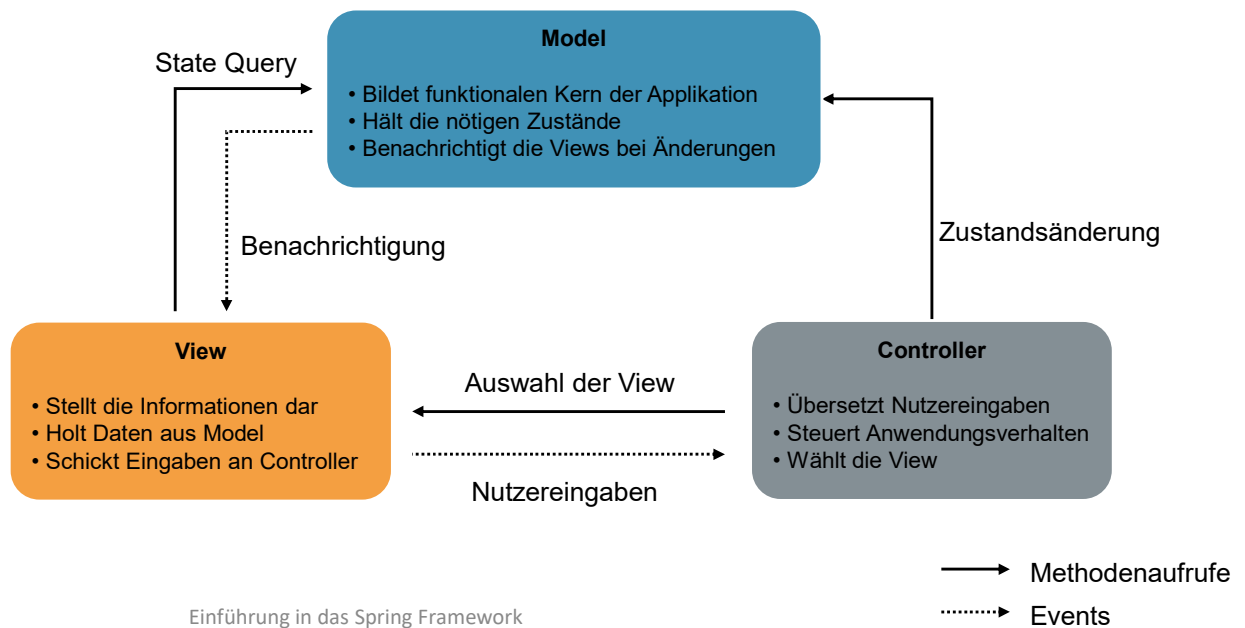
    private User createUserWithLastname(String lastname) {
        User user = new User(); user.setLastname(lastname); return user;
    }
}

```

Gliederung

- Spring Grundlagen
- @Bean
- Testen
- Profile
- Properties
- AOP
- Datenbankzugriff
- **Webanwendungen**
- REST
- Build- und Deployment
- Betrieb – Spring Boot Actuator
- Security
- Sonstiges

Model View Controller



115

Einführung in das Spring Framework

„Model View Controller (MVC, [englisch](#) für Modell-Präsentation-Steuerung) ist ein [Muster](#) zur Trennung von [Software](#) in die drei Komponenten Datenmodell (engl. model), Präsentation (engl. view) und Programmsteuerung (engl. controller). Das Muster kann sowohl als [Architekturmuster](#), als auch als [Entwurfsmuster](#) eingesetzt werden. Ziel des Musters ist ein flexibler Programmentwurf, der eine spätere Änderung oder Erweiterung erleichtert und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglicht. Es ist dann zum Beispiel möglich, eine Anwendung zu schreiben, die dasselbe Modell nutzt und es dann für Windows, Mac, Linux oder für das Internet zugänglich macht. Die Umsetzungen nutzen dasselbe Modell, nur Controller und View müssen dabei jeweils neu implementiert werden.“

Quelle: https://de.wikipedia.org/wiki/Model_View_Controller

Die wichtigsten MVC-Annotationen im Überblick

- **@Controller**
 - Kennzeichnung aller Controller
- **@ResponseBody**
 - Rückgabe einer Controller-Methode wird automatisch als Body interpretiert. Es findet explizit *keine* View-Auflösung statt. Dies wird beispielsweise benötigt, um JSON an den Client zurückzuliefern.
- **@RestController**
 - Ein Controller bei dem automatisch alle Container-Methoden mit @ResponseBody versehen sind.
- **@RequestMapping**
 - Zuordnung von URL zu Controller und Action

@Controller und @RestController

- Autoerkennung der Controller vgl. @Component
 - jeder Controller ist ein Bean im ApplicationContext
- Alle Methoden werden automatisch nach weiteren Annotationen durchsucht:
 - z.B. @RequestMapping

```
@Controller
public class UserController {
    // ...
}
```

```
@RestController
public class UserController {
    // ...
}
```

Die Controller beinhalten die Steuerungslogik für die View. Diese Controller müssen dem Spring Framework bekannt gegeben werden. Dies geschieht entweder über die Annotation @Controller oder @RestController.

@RequestMapping

- Erlaubt eine sehr flexible Zuordnung von URLs zu Klassen (Controller) und Methoden (Action)
- Kann an Klassen **und/oder** Methoden verwendet werden.

```
@RestController
public class UserController {

    @RequestMapping("/user")
    public List<User> getAllUsers() {
        // ...
    }
}
```

@RequestMapping

- @RequestMapping kann weitere Einschränkungen wie z.B. die erlaubte **http-Methode** (GET, POST, ...) beinhalten:

```
@RestController
public class UserController {
    @RequestMapping(value = "/user", method = RequestMethod.GET)
    public List<User> getAllUsers() {
        // ...
    }
}
```

- Hierfür gibt es mit **@GetMapping** eine kürzere Variante:

```
@RestController
public class UserController {
    @GetMapping("/user")
    public List<User> getAllUsers() {
        // ...
    }
}
```

Übung

Übung „Webanwendungen Hello World“



120

Einführung in das Spring Framework

Legen Sie ein neues Spring Project mit den folgenden Daten an:

Name: HelloWorldWebanwendung

Group: com.accenture.spring.helloworldweb

Artifact: HelloWorldWebanwendung

Package: com.accenture.spring.helloworldweb

Dependencies: Spring Web

Öffnen Sie die Klasse "HelloWorldWebanwendungApplication" und fügen Sie die Annotation "@RestController" an die Klasse an.

Schreiben Sie eine neue Methode "helloWorld" ohne Methodenparameter und mit einem String als Rückgabewert.

Annotieren Sie diese Methode mit "@RequestMapping("/helloWorld")"

Geben Sie „Hello World“ in der Methode zurück

Starten Sie die Anwendung als Java Application und rufen in einem Browser Ihrer Wahl die URL „http://localhost:8080/helloWorld“

```
@SpringBootApplication
@RestController
public class HelloWorldWebanwendungApplication {
    public static void main(String[] args) {
        SpringApplication.run(HelloWorldWebanwendungApplication.class, args);
    }

    @RequestMapping("/helloWorld")
    public String helloWorld() {
        return "Hello World";
    }
}
```

Request-Parameter auswerten

- **@RequestParam**

- z.B.: `http://www.oio.de/app/hello?name=dieter`

```
@GetMapping("/hello")
public String helloRequestParam(@RequestParam(value = "name", required = false) String name) {
    return "Hello " + name;
}
```

- **@PathVariable**

- Zuordnung Request-Parameter zu Methoden-Parameter
- z.B.: `http://www.oio.de/app/hello/dieter`

```
@GetMapping("/hello/{name}")
public String helloPathVariable(@PathVariable("name") String name) {
    return "Hello " + name;
}
```

Mit den beiden Annotationen `@RequestParam` und `@PathVariable` können Aufrufparameter aus einem http-Aufruf ausgewertet werden. Die beiden Annotationen unterscheiden sich lediglich in der unterstützten URL-Syntax:

`http://www.oio.de/app/hello?name=dieter`

`http://www.oio.de/app/hello/dieter`

Übung

Übung „Webanwendungen Hello World“ Teil 2



122

Einführung in das Spring Framework

Öffnen Sie das Projekt „HelloWorldWebanwendung“ und darin die Klasse
“HelloWorldWebanwendungApplication”

Fügen Sie eine neue Controller Action hinzu, mit der Sie den folgenden Aufruf entgegennehmen können:
<http://localhost:8080/hello?name=Dieter>

```
@GetMapping("/hello")
public String helloRequestParam(@RequestParam(value = "name", required = false)
String name) {
    return "Hello " + name;
}
```

Fügen Sie eine weitere Controller Action hinzu, mit der Sie den folgenden Aufruf entgegennehmen können:
<http://localhost:8080/hello/Dieter>

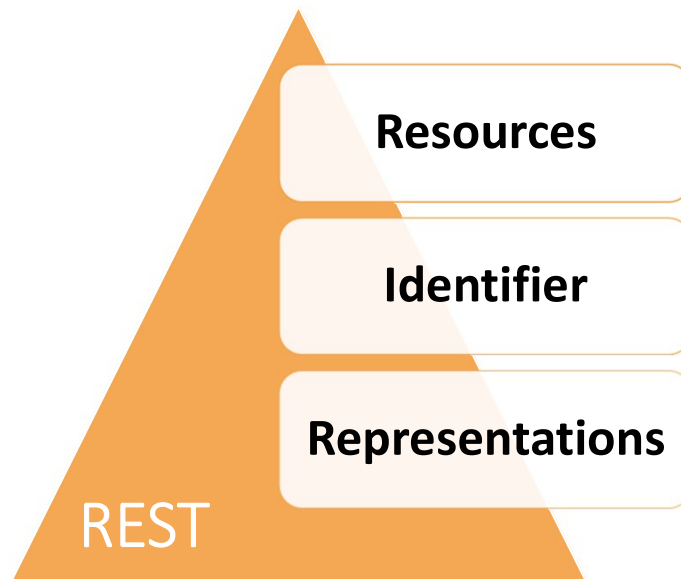
```
@GetMapping("/hello/{name}")
public String helloPathVariable(@PathVariable("name") String name) {
    return "Hello " + name;
}
```

Starten und testen Sie die Anwendung

Gliederung

- Spring Grundlagen
- @Bean
- Testen
- Profile
- Properties
- AOP
- Datenbankzugriff
- Webanwendungen
- **REST**
- Build- und Deployment
- Betrieb – Spring Boot Actuator
- Security
- Sonstiges

REST in a Nutshell



124

Einführung in das Spring Framework

Rest definiert drei Grundbegriffe Resources, Identifizier und Representations. Die Begriffe werden auf den nachfolgenden Folien näher erläutert.

Alles ist eine Ressource



Firstname	Dieter
Lastname	Develop
Employer	OIO
Hobbies	Coding, Biking
...	...

In einer Rest-Schnittstelle können ausschließlich mit Ressourcen interagiert werden. Dabei wird alles als Ressource angesehen. Die kann z.B. ein Person, aber auch eine Datenbankverbindung sein. Das Öffnen der Datenbankverbindung entspricht dann dem „Anlegen einer Ressource“ und das Schließen der Verbindung entspricht dem „Löschen der Ressource“.

Jede Ressource hat eine URI



<http://oio.de/people/dieter-develop>

Jede Ressource hat eine eindeutige URI und der sie angesprochen werden kann.

Resources - mehrere Repräsentationen



127

Einführung in das Spring Framework

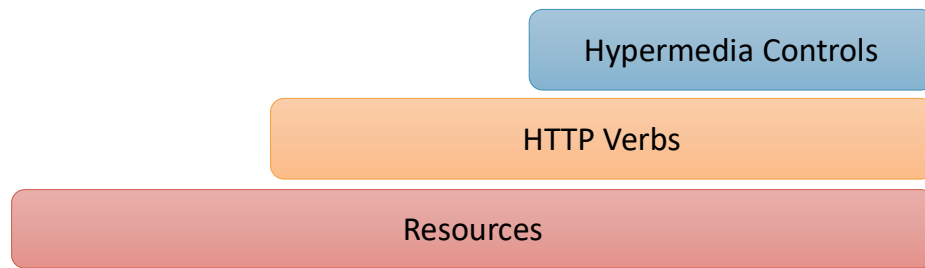
Resources können mehrere Repräsentationen besitzen. Als Beispiele für eine Person können dies sein:

Ein Bild

Ein JSON-String mit den Eigenschaften der Person

Eine HTML-Seite in der die Eigenschaften der Person in einer strukturierten Form dargestellt werden.

Richardson Maturity Model



Quelle: Fowler, Martin: Steps toward REST
<http://martinfowler.com/articles/richardsonMaturityModel.html> (2017-02-06)

Step 1: Individual Resources

- Viele URIs
- Hauptsächlich eine HTTP Methode
- Beispiele:

oio.de/people/
oio.de/people/dieter-develop/
oio.de/projects/
oio.de/projects/rest-example/



Der erste Schritt hin zu einer Rest-Ressource ist die Verwendung von verschiedenen URIs für verschiedene Ressourcen.

Step 2: Using HTTP Verbs

- Viele URIs
- Jede URI unterstützt verschiedene HTTP Methoden

- Beispiele:

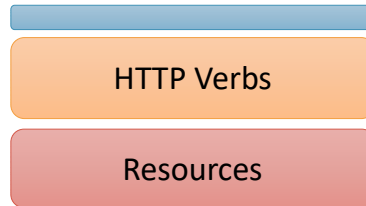
GET /projects/

POST /projects/

<project>

<name>REST Client</name>

</project>



Auf der zweiten Stufe ist die Verwendung von verschiedenen HTTP Methoden für unterschiedliche Aktionen vorgeschrieben. Die genaue Bedeutung der einzelnen HTTP Methoden ist auf der nächsten Folie zu finden.

Usage of HTTP Verbs

Resource	POST (create)	GET (read)	PUT (update)	DELETE (delete)
/projects	Create new Project	Get all projects	Update project list (bulk update)	Delete all projects
/projects/foo	error	Get Project "Foo"	Update "Foo" (if exists)	Delete Foo

Die Tabelle zeigt die Verwendung der verschiedenen HTTP Methoden einer Rest-Schnittstelle. Zu unterscheiden ist dabei, ob eine Collection aus Ressourcen (1. Zeile) oder eine einzelne Ressource (2. Zeile) angesprochen wird.

Hypermedia as the engine of Application state

- HATEOAS
- Ressourcen beschreiben ihre
 - Verbindungen zu anderen Ressourcen
 - möglichen Aktionen
- Die Schnittstelle wird damit selbst-beschreibend und selbst-dokumentierend

Hypermedia Controls

HTTP Verbs

Resources

HATEOAS is about Links

```
{
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/persons{&sort,page,size}",
      "templated" : true
    },
    "next" : {
      "href" : "http://localhost:8080/persons?page=1&size=5{&sort}",
      "templated" : true
    }
  },
  "_embedded" : {
    ... data ...
  },
  "page" : {
    "size" : 5,
    "totalElements" : 50,
    "totalPages" : 10,
    "number" : 0
  }
}
```

133 Einführung in das Spring Framework

Die Folie zeigt beispielhaft die Links in einer HATEOAS-Schnittstelle.

REST Service Client mit Spring MVC

- Mit `@PathVariable` können Platzhalter in der URL hinterlegt werden:

```
@RestController
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/user")
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @GetMapping("/user/{id}")
    public User getUser(@PathVariable("id") Long id) {
        return userService.findUserById(id);
    }
}
```

Mit Hilfe von Spring MVC kann in Spring sehr einfach eine REST Schnittstelle zur Verfügung gestellt werden. Dazu ist lediglich die Einhaltung der REST Vorgaben notwendig. Die technischen Voraussetzungen sind mit Spring MVC bereits vollständig vorhanden.

Übung

Übung „Webanwendungen REST Service“



135

Einführung in das Spring Framework

Fügen Sie die Abhängigkeit „org.springframework.boot:spring-boot-starter-web“ in die Datei Build-Datei ein.
Erstellen Sie ein neues Paket mit dem Namen „com.accenture.spring.user.controller“.

Erstellen Sie eine neue Klasse „UserController“.

Annotieren Sie diese Klasse mit den Annotation „@RestController“

Fügen Sie ein neues Feld vom Typ „UserService“ in den Controller ein und sorgen Sie per @Autowired dafür, dass Spring die Abhängigkeit von außen injiziert.

Schreiben Sie eine neue Methode „getAllUsers“, die die Liste aller User mit Hilfe des UserService zurückliefert und annotieren Sie diese Methode mit @GetMapping("/user").

Starten Sie die Anwendung und rufen Sie die URL „http://localhost:8080/user“ auf.

Schreiben Sie eine weitere Methode, die eine ID per @PathVariable entgegen nimmt und nur den User mit dieser ID im Browser anzeigt. Die URL soll z.B. wie folgt lauten: „http://localhost:8080/user/1“. Hierzu müssen Sie auch den UserService und eine entsprechende Funktion erweitern.

```
@RestController
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/user")
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @GetMapping("/user/{id}")
    public User getUser(@PathVariable("id") Long id) {
        return userService.findUserById(id);
    }
}
```

Zusatzaufgaben:

Implementieren Sie das Löschen eines Users anhand der ID (http DELETE)

Implementieren Sie das Anlegen eines neuen Users und testen Sie diese Schnittstelle

REST Client

- Der programmatische Zugriff auf eine REST-Schnittstelle ist mit Hilfe der Klasse **RestTemplate** möglich:

```
User user = new RestTemplate().getForObject("http://localhost:8080/user/{id}", User.class, 1);
```

HTTP	METHODE
DELETE	<code>delete(String, Object...)</code>
GET	<code>getForObject(String, Class, Object...)</code>
HEAD	<code>headForHeaders(String, Object...)</code>
OPTIONS	<code>optionsForAllow(String, Object...)</code>
POST	<code>postForLocation(String, Object, Object...)</code>
PUT	<code>put(String, Object, Object...)</code>

Das `RestTemplate` bietet eine einfache Möglichkeit, um auf bestehende REST-Schnittstelle zuzugreifen. Für die verschiedenen HTTP-Methoden, die von REST unterstützt werden, stehen jeweils mehrere teils überladene Methoden zur Verfügung.

Übung

- Übung „REST Client“



137

Einführung in das Spring Framework

Testen Sie unsere REST-Schnittstelle zum Abrufen eines Users mit Hilfe des RestTemplates:

```
new RestTemplate().getForObject("http://localhost:8080/user/{id}", User.class, id);
```


Reaktiver WebClient

- Seit Spring 5 existiert mit der Klasse WebClient die Möglichkeit, reaktiv auf http-Endpunkte zuzugreifen. Diese Variante ist mittlerweile zu bevorzugen:

```
WebClient.create("http://localhost:8080")
    .get()
    .uri("/user/{id}", 11)
    .exchangeToFlux(clientResponse -> clientResponse.bodyToFlux(User.class))
    .subscribe(user -> {
        //
    });
```

Das RestTemplate bietet eine einfache Möglichkeit, um auf bestehende REST-Schnittstelle zuzugreifen. Für die verschiedenen HTTP-Methoden, die von REST unterstützt werden, stehen jeweils mehrere teils überladene Methoden zur Verfügung.

Gliederung

- Spring Grundlagen
- @Bean
- Testen
- Profile
- Properties
- AOP
- Datenbankzugriff
- Webanwendungen
- REST
- **Build- und Deployment**
- Betrieb – Spring Boot Actuator
- Security
- Sonstiges

Produktionsartefakt erstellen

- Jede Spring Boot Anwendung bringt ein **Build-File** mit (Maven oder Gradle)
- Damit lässt sich einfach ein **Fat-JAR mit allen notwendigen Abhängigkeiten** erstellen

- Maven

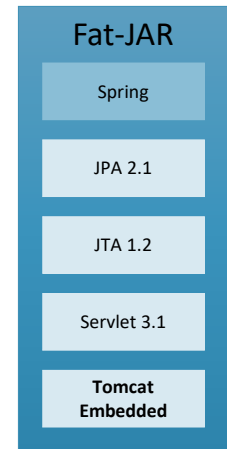
```
mvnw clean package
```

- Gradle

```
gradlew build
```

- Anwendung starten

```
java -jar app.jar
```



Docker Image erstellen

- Das Fat-JAR lässt sich einfach in **ein Docker Image** verpacken

```
FROM openjdk:17
EXPOSE 8080
ARG JAR_FILE=target/my-application.jar
ADD ${JAR_FILE} app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

- Allerdings hat diese Variante einige **Nachteile**
 - Fat-JAR ist **nicht ausgepackt** → Laufzeitoverhead
 - **Nicht effizient** bei häufigen Updates der Anwendung

→ **Lösung: Jar-Datei auspacken und in einen eigenen Layer packen**

Cloud Native Buildpacks

- “Gute” Dockerfiles sind aufwendig zu erstellen → Buildpacks liefern eine fertige Lösung
- “**Cloud Native Buildpacks** transform your application source code into images that can run on any cloud.” <https://buildpacks.io/>

- Docker Image erzeugen

```
mvn spring-boot:build-image
```

```
gradle bootBuildImage
```

- Image starten

```
docker run -it -p 8080:8080 demo:0.0.1-SNAPSHOT
```

Docker Layers - Dive

```
| Layers |
Cmp      Size  Command
63 MB    FROM 1a65221217b8d23
745 B
7 B
224 B
24 MB
398 kB
38 MB
2.4 MB
453 kB
0 B

| Layer Details |
Tags: (unavailable)
Id:    d682023bf5c4a142093ec9ecc57bc4612526a1cf559e5c
1b35a8a81f68ae0b21
Digest: sha256:ed0b1596f4234ba49885693359a8ab36af44bba
eac271cb696caef6abf55401b
Command:

| Image Details |

Total Image size: 128 MB
Potential wasted space: 3.3 MB
Image efficiency score: 98 %
```

Layered Jars

- Buildpack Images sind für einen Großteil der Anwendungsfälle ausreichend
- Für eigene Dockerfiles gibt es zusätzlich die Möglichkeit die **Layer eines Fat-JAR zu extrahieren**

```
java -Djarmode=layertools -jar app.jar
```

Usage:

```
java -Djarmode=layertools -jar app.jar
```

Available commands:

```
list      List layers from the jar that can be extracted
extract   Extracts layers from the jar for image creation
help      Help about any command
```

```
dependencies
spring-boot-loader
snapshot-dependencies
application
```

← Layer

GraalVM Native Image

- Graal =
 - „General Recursive Applicative and Algorithmic Language“
 - „Allgemeine rekursive Anwendungs- und Algorithmussprache“



**Schnellere
Startups**



Einführung in das Spring Framework



**Geringerer
Speicherbedarf**

spring-graalvm-native

- Maven Konfiguration anpassen

```
<dependency>
  <groupId>org.springframework.experimental</groupId>
  <artifactId>spring-native</artifactId>
  <version>0.9.0-SNAPSHOT</version>
</dependency>
```

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <image>
      <env>
        <BP_BOOT_NATIVE_IMAGE>true</BP_BOOT_NATIVE_IMAGE>
      </env>
    </image>
  </configuration>
</plugin>
```



```
mvn spring-boot:build-image
```

147

Einführung in das Spring Framework

Beeindruckend?

Started PetClinicApplication in **15.449 seconds** (JVM running for **17.339**)



Started PetClinicApplication in **0.212 seconds** (JVM running for **0.227**)

Gliederung

- Spring Grundlagen
- @Bean
- Testen
- Profile
- Properties
- AOP
- Datenbankzugriff
- Webanwendungen
- REST
- Build- und Deployment
- **Betrieb – Spring Boot Actuator**
- Security
- Sonstiges

Spring Boot Actuator

- Production-ready features
- HTTP oder JMX Endpoints für die Verwaltung und das Monitoring von Spring Boot Anwendungen
- Auditing, Health and Metrics

Spring Boot verspricht produktionsreife Anwendungen und dazu gehören auch die Möglichkeiten zur Verwaltung und zum Monitoring der Anwendung. Genau diese Möglichkeiten bekommen wir durch den Einsatz von Spring Boot Actuator. Neben Auditing liefert uns diese Bibliothek auch Health und Metrik-Informationen.

Actuator Endpunkte

/actuator

```
← → ↻ localhost:8080/actuator
1 // 20190111091535
2 // http://localhost:8080/actuator
3
4 {
5   "_links": {
6     "self": {
7       "href": "http://localhost:8080/actuator",
8       "templated": false
9     },
10    "health": {
11      "href": "http://localhost:8080/actuator/health",
12      "templated": false
13    },
14    "health-component-instance": {
15      "href": "http://localhost:8080/actuator/health/{component}/{instance}",
16      "templated": true
17    },
18    "health-component": {
19      "href": "http://localhost:8080/actuator/health/{component}",
20      "templated": true
21    },
22    "info": {
23      "href": "http://localhost:8080/actuator/info",
24      "templated": false
25    }
26  }
27 }
```

151

Einführung in das Spring Framework

Actuator Endpunkte

/actuator/health

```
1 // 20190111091543
2 // http://localhost:8080/actuator/health
3
4 {
5   "status": "UP"
6 }
```

management.endpoint.health.show-details=WHEN_AUTHORIZED

```
1 // 20190111091942
2 // http://localhost:8080/actuator/health
3
4 {
5   "status": "UP",
6   "details": {
7     "diskSpace": {
8       "status": "UP",
9       "details": {
10        "total": 510577549312,
11        "free": 277582024704,
12        "threshold": 10485760
13      }
14    }
15  }
16 }
```

152

Einführung in das Spring Framework

Die Folie zeigt den HTTP-Endpunkt „/health“, der beim Einsatz von Spring Boot Actuator automatisch zur Verfügung steht:

Health: ist die Anwendung gerade in einem fehlerfreien Zustand oder gibt es irgendwelche Probleme, die einen korrekten Betrieb einschränken. Dies kann z.B. eine volle Festplatte oder eine nicht erreichbare Datenbank sein.

Weitere Endpoints einschalten

- Neben `/actuator` und `/actuator/health` gibt es zahlreiche weitere Endpunkte in Actuator.
- Diese müssen zunächst aktiviert werden

```
management.endpoints.web.exposure.include=*
```


Metriken

```
localhost:8080/actuator/metrics
1 // 20190111092629
2 // http://localhost:8080/actuator/metrics
3
4 {
5   "names": [
6     "jvm.memory.max",
7     "jvm.threads.states",
8     "http.server.requests",
9     "jvm.gc.memory.promoted",
10    "tomcat.cache.hit",
11    "tomcat.servlet.error",
12    "tomcat.cache.access",
13    "jvm.memory.used",
14    "jvm.gc.max.data.size",
15    "jvm.gc.pause",
16    "jvm.memory.committed",
17    "system.cpu.count",
18    "logback.events",
19    "tomcat.global.sent",
20    "jvm.buffer.memory.used",
```

```
localhost:8080/actuator/metrics/jvm.memory.max
1 // 20190111092717
2 // http://localhost:8080/actuator/metrics/jvm.memory.max
3
4 {
5   "name": "jvm.memory.max",
6   "description": "The maximum amount of memory in bytes that can
7   be used for memory management",
8   "baseUnit": "bytes",
9   "measurements": [
10     {
11       "statistic": "VALUE",
12       "value": 5573705727
13     }
14   ]
15 }
```

Endpoints

Name	Beschreibung
auditevents	Exposes audit events information for the current application.
beans	Displays a complete list of all the Spring beans in your application.
caches	Exposes available caches.
conditions	Shows the conditions that were evaluated on configuration and auto-configuration classes and the reasons why they did or did not match.
configprops	Displays a collated list of all @ConfigurationProperties.
env	Exposes properties from Spring's ConfigurableEnvironment.
flyway	Shows any Flyway database migrations that have been applied.
health	Shows application health information.
httptrace	Displays HTTP trace information (by default, the last 100 HTTP request-response exchanges).

Liste aller HTTP-Endpunkte, die Actuator zur Verfügung stellt.

Endpoints

Name	Beschreibung
info	Displays arbitrary application info.
integrationgraph	Shows the Spring Integration graph.
loggers	Shows and modifies the configuration of loggers in the application.
liquibase	Shows any Liquibase database migrations that have been applied.
metrics	Shows 'metrics' information for the current application.
mappings	Displays a collated list of all @RequestMapping paths.
scheduledtasks	Displays the scheduled tasks in your application.
sessions	Allows retrieval and deletion of user sessions from a Spring Session-backed session store. Not available when using Spring Session's support for reactive web applications.
shutdown	Lets the application be gracefully shutdown.
threaddump	Performs a thread dump.

156

Einführung in das Spring Framework

Liste aller HTTP-Endpunkte, die Actuator zur Verfügung stellt.

Endpoints anpassen -application.properties

- Endpoints ein- bzw. ausschalten:

```
management.endpoint.shutdown.enabled=true
```

Actuator bietet zwei Sicherheitsmechanismen:

Deaktivieren von Endpunkten: einzelne Endpunkte können deaktiviert werden, so dass sie nicht mehr angesprochen werden können. Der besonders sicherheitskritische Endpunkt „shutdown“ ist defaultmäßig bereits deaktiviert und kann wie auf der Folie dargestellt aktiviert werden.

Passwortschutz für Endpunkte (sensitive): die meisten Endpunkte sind defaultmäßig mit einem Passwortschutz versehen, der sich allerdings ebenfalls deaktivieren lässt (siehe nächste Folie).

Eigene Endpoints hinzufügen 1 / 3

- HealthIndicator (/health)

```
@Component
public class MyHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        int errorCode = check(); // perform some specific health check
        if (errorCode != 0) {
            return Health.down().withDetail("Error Code", errorCode).build();
        }
        return Health.up().build();
    }
}
```

Die meisten Endpunkte lassen sich anpassen und durch eigene Informationen erweitern. Im Beispiel oben ist die Erweiterung des Endpunktes „Health“ zu sehen.

Eigene Endpoints hinzufügen 2 / 3

- InfoContributor (/info)

```
@Component
public class ExampleInfoContributor implements InfoContributor {

    @Override
    public void contribute(Info.Builder builder) {
        builder.withDetail("example", Collections.singletonMap("key", "value"));
    }
}
```

Eigene Endpoints hinzufügen 3 / 3

- Counter (/metrics)

```
@Component
public class SampleBean {

    private final Counter counter;

    public SampleBean(MeterRegistry registry) {
        this.counter = registry.counter("received.messages");
    }

    public void handleMessage(String message) {
        this.counter.increment();
        // handle message implementation
    }
}
```

Übung

- Erweitern Sie die Anwendung mit **Spring Boot Actuator**.
- Konfigurieren Sie die Anwendung so, dass alle Actuator Endpunkte zur Verfügung stehen.
- Starten Sie die Anwendung und testen Sie die verfügbaren Endpunkte (/env, /health, ...)
- Erweitern Sie zwei der bestehenden Endpunkte mit eigenen Informationen.



Gliederung

- Spring Grundlagen
- @Bean
- Testen
- Profile
- Properties
- AOP
- Datenbankzugriff
- Webanwendungen
- REST
- Build- und Deployment
- Betrieb – Spring Boot Actuator
- **Security**
- Sonstiges

Spring Security

- „Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications.”
 - <http://projects.spring.io/spring-security/>
- Integriert verschiedenste Authentifizierungsmechanismen
- Sicherheit auf URL-, Domainobjekt- und Methodenebene



Grundbegriffe von Security

- Authentifizierung (Wer bin ich?)
- Autorisierung (Was darf ich?)
- Benutzerkennung (Principal)
- Berechtigungsnachweis (Credential)
- Berechtigungsart (Rechte, Rollen)

Authentisierung

Die Authentisierung stellt den Nachweis einer Person dar, dass sie tatsächlich diejenige Person ist, die sie vorgibt zu sein. Eine Person legt also Nachweise vor, die ihre Identität bestätigen sollen.

Autorisierung

Die Autorisierung ist die Einräumung von speziellen Rechten. War die Identifizierung einer Person erfolgreich, heißt es noch nicht automatisch, dass diese Person bereitgestellte Dienste und Leistungen nutzen darf. Darüber entscheidet die Autorisierung.

Principal

Principal ist eine Abstraktion der Benutzerkennung. Typischerweise ist dies ein Benutzername.

Credential

Abstraktion für den Berechtigungsnachweis. Typischerweise ist dies ein Passwort.

Spring Security – Getting Started

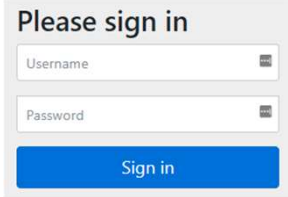
- Maven Dependency hinzufügen

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

- Convention over Configuration
 - Alle **http-Endpunkte abgesichert**
 - Standard-User: **user**
 - **Generiertes Passwort** wird beim Anwendungsstart auf die Konsole geschrieben

Using generated security password: 11eb1afc-dcd6-4d52-b902-e8337a7b534b

- Standard-HTML-Formular für den Login
- **Automatische Umleitung** auf die Login-Seite



Please sign in

Username

Password

Demo

- Default Spring Security Absicherung



Default-User anpassen

- application.properties

```
spring.security.user.name=user  
spring.security.user.password=password
```

In-Memory Authentication

- Benutzer programmatisch festlegen (für Testzwecke)

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration {

    @Bean
    public InMemoryUserDetailsManager userDetailsService() {
        UserDetails admin = User.withDefaultPasswordEncoder()
            .username("admin")
            .password("password")
            .roles("ADMIN")
            .build();

        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("password")
            .roles("USER")
            .build();

        return new InMemoryUserDetailsManager(admin, user);
    }
}
```

Für Testzwecke kann eine In-Memory Benutzerdatenbasis hinterlegt werden. In dieser Datenbasis werden die Benutzer mit Benutzernamen, Passwort und zugewiesenen Rollen direkt im Code angegeben. Diese Vorgehensweise ist nur bedingt für einen produktiven Einsatz sinnvoll.

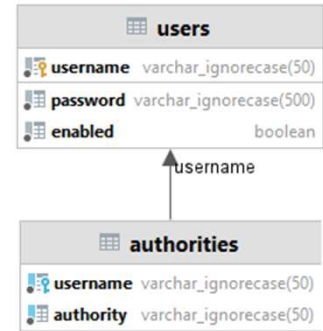
JDBC Authentication

- Benutzer in einer Datenbank ablegen

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .addScript(JdbcDaoImpl.DEFAULT_USER_SCHEMA_DDL_LOCATION)
            .build();
    }

    @Bean
    public UserDetailsManager users(DataSource dataSource) {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user").password("password").roles("USER").build();
        JdbcUserDetailsManager users = new JdbcUserDetailsManager(dataSource);
        users.createUser(user);
        return users;
    }
}
```



169

Einführung in das Spring Framework

In der Produktion gängiger als eine In-Memory Benutzerdatenbasis ist die Anbindung einer relationalen Datenbank. Hierfür stellt Spring Security ein Standard-Datenbankschema zur Verfügung.

LDAP Authentication

- Z.B. Microsoft Active Directory

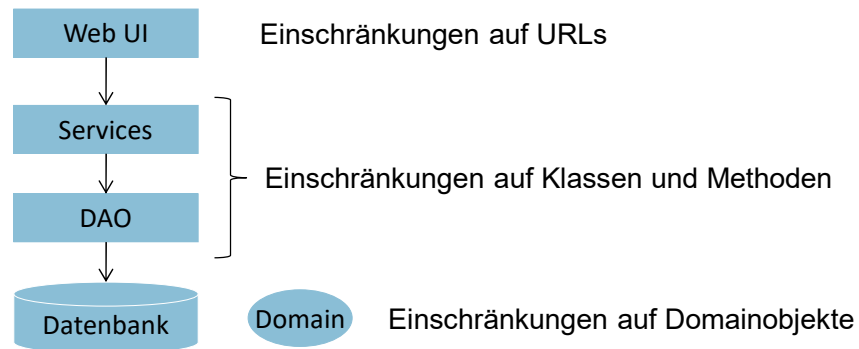
```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration {

    @Bean
    public EmbeddedLdapServerContextSourceFactoryBean contextSourceFactoryBean() {
        EmbeddedLdapServerContextSourceFactoryBean contextSourceFactoryBean =
            EmbeddedLdapServerContextSourceFactoryBean.fromEmbeddedLdapServer();
        contextSourceFactoryBean.setPort(0);
        return contextSourceFactoryBean;
    }

    @Bean
    AuthenticationManager ldapAuthenticationManager(BaseLdapPathContextSource contextSource) {
        LdapBindAuthenticationManagerFactory factory =
            new LdapBindAuthenticationManagerFactory(contextSource);
        factory.setUserDnPatterns("uid={0},ou=people");
        factory.setUserDetailsContextMapper(new PersonContextMapper());
        return factory.createAuthenticationManager();
    }
}
```

Autorisierung mit Spring Security

Spring-Security unterstützt Autorisierung auf verschiedenen Ebenen



171

Einführung in das Spring Framework

Spring Security unterstützt Autorisierung auf Web-, Methoden- und Domainobjekt-Ebene.

Web-Ebene

Eine Benutzer darf eine bestimmte URL nur aufrufen, falls er eine bekannt ist und einer der notwendigen Rollen zugewiesen wurde.

Methoden-Ebene

Die Einschränkung findet hier auf Basis eines Methodenaufrufs statt.

Domain-Ebene

Dies ist die speziellste und gleichzeitig aufwendigste Absicherung. Geprüft werden die Rückgabewerte einer Methode. Darf der eingeloggte Benutzer zur Rückgabewerte sehen?

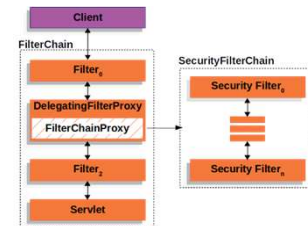
Die Principals des angemeldeten Benutzers sich immer mit der Hilfsklasse „SecurityContextHolder“ im Thread verfügbar.

Autorisierung – Einschränkung auf URLs

- Absicherung aller http-Endpunkte mit Default-Login-Formular

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration {

    @Bean
    public SecurityFilterChain filter(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests((authz) -> {
            authz.anyRequest().authenticated();
        }).formLogin();
        return http.build();
    }
}
```



Autorisierung – Einschränkung auf URLs

- Rollenbasierte Absicherung mit benutzerdefiniertem Login-Formular

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration {

    @Bean
    public SecurityFilterChain filter(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests((authz) -> {
            authz.mvcMatchers("/actuator").hasRole("ADMIN");
            authz.mvcMatchers("/user").hasRole("USER");
        }).formLogin().loginPage("/my-login-form").failureUrl("/my-error-page");
        return http.build();
    }
}
```

Autorisierung - Sicherheit auf Methodenebene

- Verwendung von Standard JSR-250 Annotationen

```
@RolesAllowed({"USER", "ADMIN"})  
public List<User> getAllUsers() {  
    return userRepository.findAll();  
}
```

- Konfiguration:

```
@EnableGlobalMethodSecurity(jsr250Enabled = true, securedEnabled = true)
```

Zur Absicherung einer Anwendung können bekanntlich verschiedene Mechanismen eingesetzt werden. Die Folie zeigt die Absicherung auf Methodenebene mit Hilfe der Annotation „@RolesAllowed“. Zur Aktivierung dieser Annotation kann die Konfiguration-Annotation „@EnableGlobalMethodSecurity“ eingesetzt werden.

Übung

- Spring Security



175

Einführung in das Spring Framework

Fügen Sie die folgende Dependency in der Build-Datei ein: `org.springframework.boot:spring-boot-starter-security`

Erstellen Sie eine neue Klasse „`com.accenture.spring.user.SecurityConfiguration`“

Fügen Sie die Annotation „`@EnableWebSecurity`“ in der Klasse „`SecurityConfiguration`“ hinzu.

Fügen Sie eine In-Memory-Authentication hinzu:

```
@Bean
public InMemoryUserDetailsManager userDetailsService() {
    UserDetails admin = User.withDefaultPasswordEncoder()
        .username("admin")
        .password("password")
        .roles("ADMIN")
        .build();
    UserDetails user = User.withDefaultPasswordEncoder()
        .username("user")
        .password("password")
        .roles("USER")
        .build();
    return new InMemoryUserDetailsManager(admin, user);
}
```

Starten Sie die Anwendung um versuchen Sie über die URL „<http://localhost:8080/user>“ auf die User-Liste zuzugreifen.

Zusatzaufgabe 1:

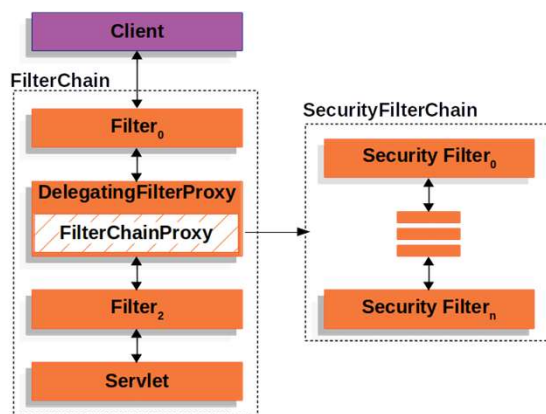
Fügen Sie eine Absicherung auf URL-Ebene in die `SecurityConfiguration` ein:

```
@Bean
public SecurityFilterChain filter(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests((authz) -> {
        authz.mvcMatchers("/actuator").hasRole("ADMIN");
    }).formLogin();
    return http.build();
}
```

Zusatzaufgabe 2:

Sichern Sie die Methoden in der Klasse „UserServiceImpl“ per Annotation ab und verwenden Sie hierzu eine zweite Rolle. Hinterlegen Sie mehrere Benutzer und weisen jeden Benutzer eine andere Kombination an Rollen zu. Testen Sie anschließend die Anwendung und analysieren Sie die jeweiligen Fehlermeldungen. Daraus sollte sich die Ursache der Zugriffsverletzungen ablesen lassen.

SecurityFilterChain



Default Spring Security Starter	OAuth2 Resource Server
DisableEncodeUrlFilter	DisableEncodeUrlFilter
WebAsyncManagerIntegrationFilter	WebAsyncManagerIntegrationFilter
SecurityContextHolderFilter	SecurityContextHolderFilter
HeaderWriterFilter	HeaderWriterFilter
CsrfFilter	CsrfFilter
LogoutFilter	LogoutFilter
UsernamePasswordAuthenticationFilter	BearerTokenAuthenticationFilter
DefaultLoginPageGeneratingFilter	
DefaultLogoutPageGeneratingFilter	
BasicAuthenticationFilter	
RequestCacheAwareFilter	RequestCacheAwareFilter
SecurityContextHolderAwareRequestFilter	SecurityContextHolderAwareRequestFilter
AnonymousAuthenticationFilter	AnonymousAuthenticationFilter
ExceptionTranslationFilter	ExceptionTranslationFilter
AuthorizationFilter	AuthorizationFilter

OAuth2

- OAuth2 ist ein **offenes Standardprotokoll** für sicheres Zugriffsmanagement auf **Ressourcen von Drittanbietern**.
- Benutzer können **Zugriff** auf bestimmte Daten **gewähren** und diesen Zugriff jederzeit **widerrufen**.
- OAuth2 ist ein Delegationsprotokoll, das auf **Tokens** basiert.
- Große Unternehmen wie Google, Facebook und Microsoft nutzen OAuth2 zur sicheren Kontrolle von **API-Zugriffen**.
- OAuth2 bietet eine einfachere und sicherere **Alternative zur traditionellen Benutzername-Passwort-Authentifizierung** und ermöglicht feinere Steuerung des Zugriffs.

OAuth2 != OAuth2

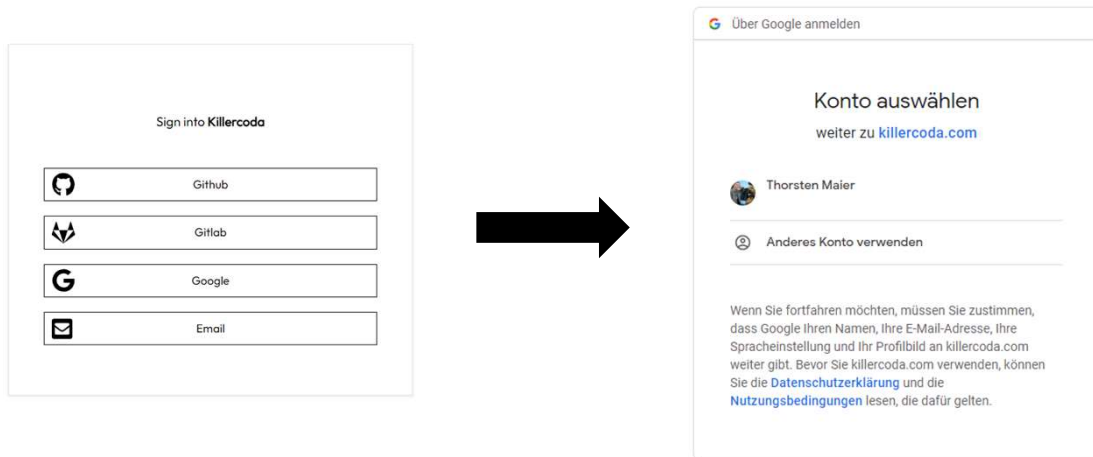
- Es gibt zahlreiche Ausprägungen und Varianten von OAuth2, was es in der Praxis unübersichtlich macht
- Betrachten wir zunächst den ursprünglichen Use-Case

Warum OAuth2?

„Ich bin ein total nette App und möchte alle deine Freunde zu deinem Geburtstag einladen. Dazu benötige ich deine Kontakte. Gib mir doch bitte einfach deine Gmail-Login-Daten. Den Rest mache ich dann schon“



Das geht besser!



OAuth2 Begriffe

Client

- Anwendung oder Dienst, der auf geschützte Ressourcen zugreifen möchte
- Stellt Anfrage an Authorization Server

Resource Owner

- Benutzer, dem die geschützten Ressourcen gehören

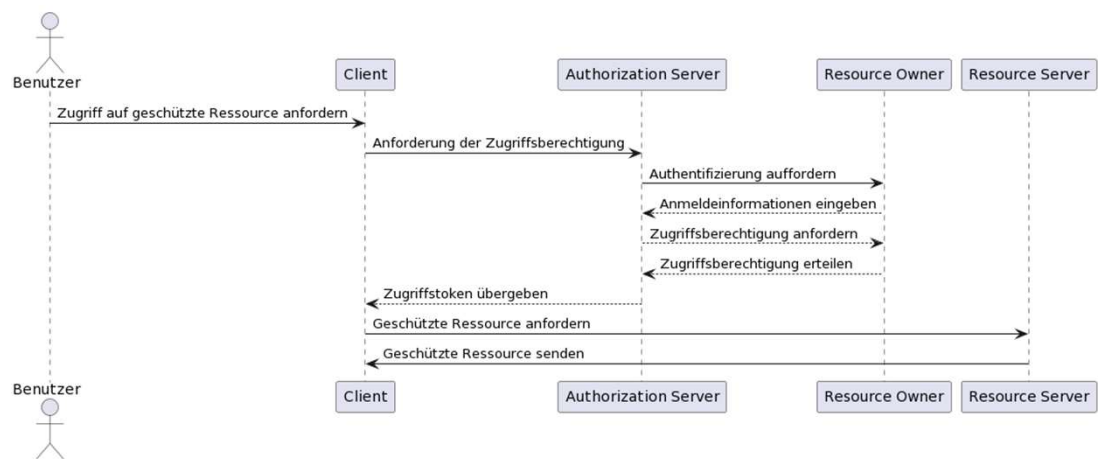
Resource Server

- Server, der geschützte Ressourcen hostet und verwaltet
- Stellt Ressourcen nur an berechtigte Clients zur Verfügung
- Authentifiziert und autorisiert Anfragen mittels Zugriffstoken

Authentication Server

- Verantwortlich für die Authentifizierung des Resource Owners
- Validiert die Identität des Benutzers
- Ausstellung eines Zugriffstokens für den Client

Interaktion in OAuth2



Spring Security mit OAuth2 und Github

- Dependency einfügen

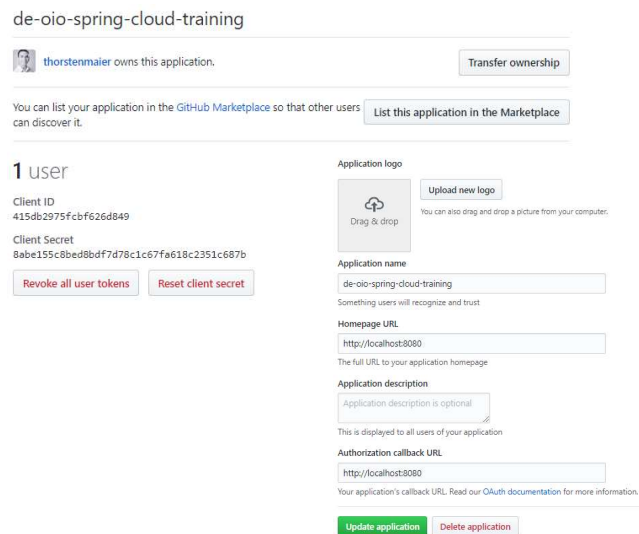
```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-oauth2-client</artifactId>  
</dependency>
```

- Konfiguration

```
spring:  
  security:  
    oauth2:  
      client:  
        registration:  
          github:  
            clientId: 415db2975fcbf626d849  
            clientSecret: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Übung

- Spring Cloud Security mit GitHub OAuth2 Server



The screenshot shows the GitHub OAuth2 application configuration interface. At the top, it displays the application name 'de-oio-spring-cloud-training' and the owner 'thorstenmaier'. Below this, there are buttons for 'Transfer ownership' and 'List this application in the Marketplace'. The '1 user' section shows the client ID '415db2975fcbf626d849' and the client secret '8abe155c8bed8bdf7d78c1c67fa618c2351c687b', with buttons for 'Revoke all user tokens' and 'Reset client secret'. The 'Application logo' section has an 'Upload new logo' button and a 'Drag & drop' area. The 'Application name' field is filled with 'de-oio-spring-cloud-training'. The 'Homepage URL' field is filled with 'http://localhost:8080'. The 'Application description' field is empty. The 'Authorization callback URL' field is filled with 'http://localhost:8080'. At the bottom, there are 'Update application' and 'Delete application' buttons.



184

Einführung in das Spring Framework

Erstellen Sie eine neue Spring Boot Anwendung mit den Abhängigkeiten „Web“ und „OAuth2 Client“. Erstellen Sie darin einen Rest Controller mit dem folgenden Inhalt:

```
@SpringBootApplication
@RestController
public class CloudSecurityApplication {
    public static void main(String[] args) {
        SpringApplication.run(CloudSecurityApplication.class, args);
    }

    @GetMapping("/")
    public String home() {
        return "Hello World";
    }
}
```

Dieser Service verwendet bereits Spring Cloud Security. Für eine korrekte Funktionsweise muss der OAuth2-Provider konfiguriert werden. Erstellen Sie hierzu eine Datei `application.yml` unter „src/main/resources“ und löschen Sie die dort bisher abgelegte „application.properties“. Der Inhalt der YAML-Datei muss für eine Authentifizierung über Github wie folgt aussehen:

```
spring:
  security:
    oauth2:
      client:
        registration:
          github:
            clientId: 415db2975fcbf626d849
            clientSecret: 8abe155c8bed8bdf7d78c1c67fa618c2351c687b
```

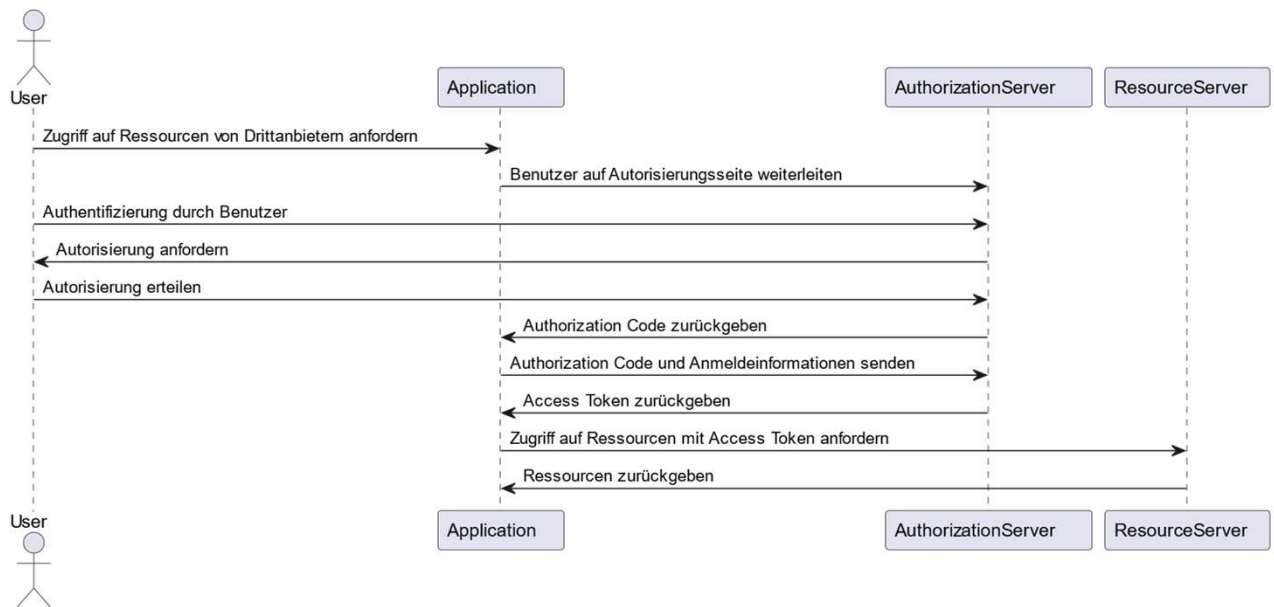

Selbstverständlich können die ClientId und das ClientSecret angepasst werden. Loggen Sie sich hierzu in Ihren Github-Account ein. Unter „Settings → Developer settings → OAuth Apps“ können Sie eine neue OAuth2 App registrieren.

Starten Sie die Anwendung und rufen Sie im Browser die URL <http://localhost:8080> auf. Sie sollten auf Github umgeleitet werden. Dort müssen Sie sich einloggen, falls dies nicht bereits zuvor geschehen ist. Anschließend werden Sie gefragt, ob Sie Ihre Profilinformationen an die anfragenden Anwendung weitergeben möchten.

OAuth2-Flows

- OAuth2 bietet verschiedene Flows zur Autorisierung des Zugriffs auf Ressourcen von Drittanbietern.
- **Authorization Code Flow:** Standard-Flow für die meisten Implementierungen. Der Benutzer wird auf die Autorisierungsseite weitergeleitet, um einen Authorization Code zu erhalten.
- **Implicit Flow:** Geeignet für öffentliche Clients wie JavaScript-basierte Webanwendungen, da der Access Token direkt zurückgegeben wird.
- **Resource Owner Password Credentials Flow:** Ermöglicht Benutzern, sich direkt mit Benutzername und Passwort zu authentifizieren und ein Access Token zu erhalten.
- **Client Credentials Flow:** Wird von Clients verwendet, die keinen Zugriff auf Ressourcen im Namen eines Benutzers benötigen, sondern nur für sich selbst Zugriff auf Ressourcen von Drittanbietern benötigen.

OAuth2 Authorization Code Flow



186

Einführung in das Spring Framework

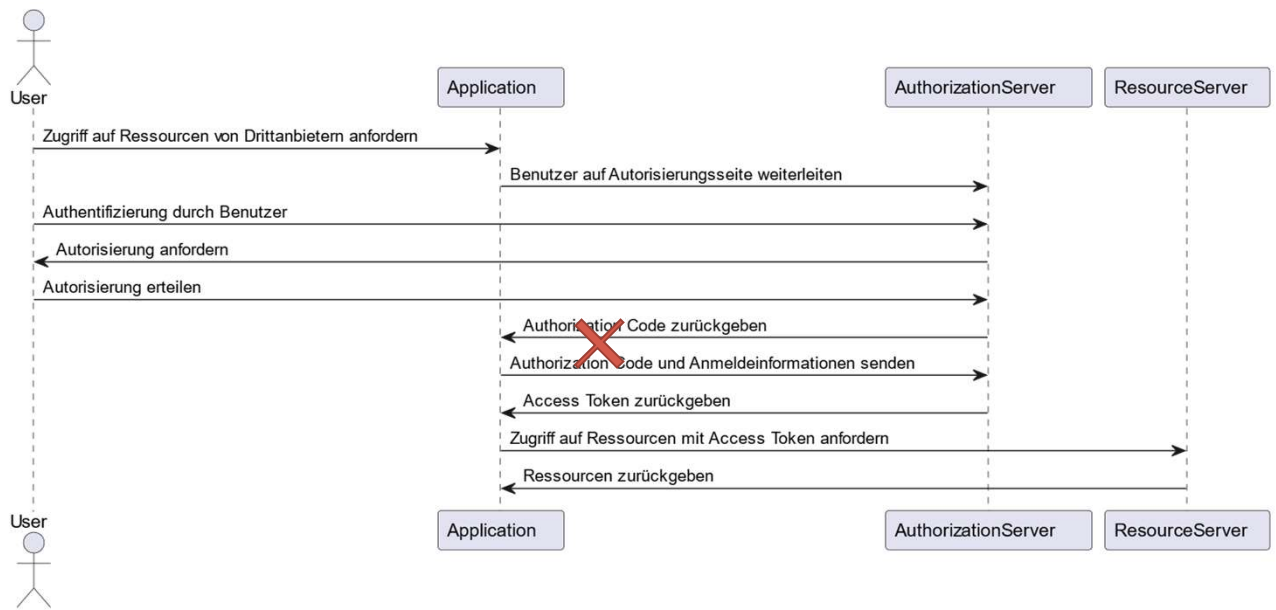
Der Authorization Code Flow ist der am häufigsten verwendete Flow in OAuth2. Er wird verwendet, wenn eine Anwendung den Zugriff auf Ressourcen von Drittanbietern auf sichere und kontrollierte Weise autorisieren muss.

Der Flow besteht aus folgenden Schritten:

1. Die Anwendung fordert vom Benutzer die Autorisierung zum Zugriff auf eine bestimmte Ressource an, indem sie den Benutzer auf die Autorisierungsseite des Authorization Servers weiterleitet.
2. Der Benutzer authentifiziert sich beim Authorization Server und erteilt der Anwendung die angeforderten Autorisierungsberechtigungen.
3. Der Authorization Server gibt einen Authorization Code zurück, der für den Erhalt eines Access Tokens verwendet wird.
4. Die Anwendung sendet den Authorization Code zusammen mit Anmeldeinformationen an den Authorization Server, um ein Access Token zu erhalten.
5. Der Authorization Server prüft die Gültigkeit des Authorization Codes und gibt ein Access Token zurück, wenn der Code gültig ist.
6. Die Anwendung kann nun das Access Token verwenden, um auf die angeforderten Ressourcen zuzugreifen.

Der Authorization Code Flow ist ein sicherer und effektiver Flow, da er die Anwendung vom Zugriff auf das Benutzerkennwort trennt und eine sichere Übertragung von Anmeldeinformationen gewährleistet. Es ist jedoch wichtig, sicherzustellen, dass das Access Token sicher gespeichert wird und nicht an unbefugte Personen weitergegeben wird.

OAuth2 Implicit Flow



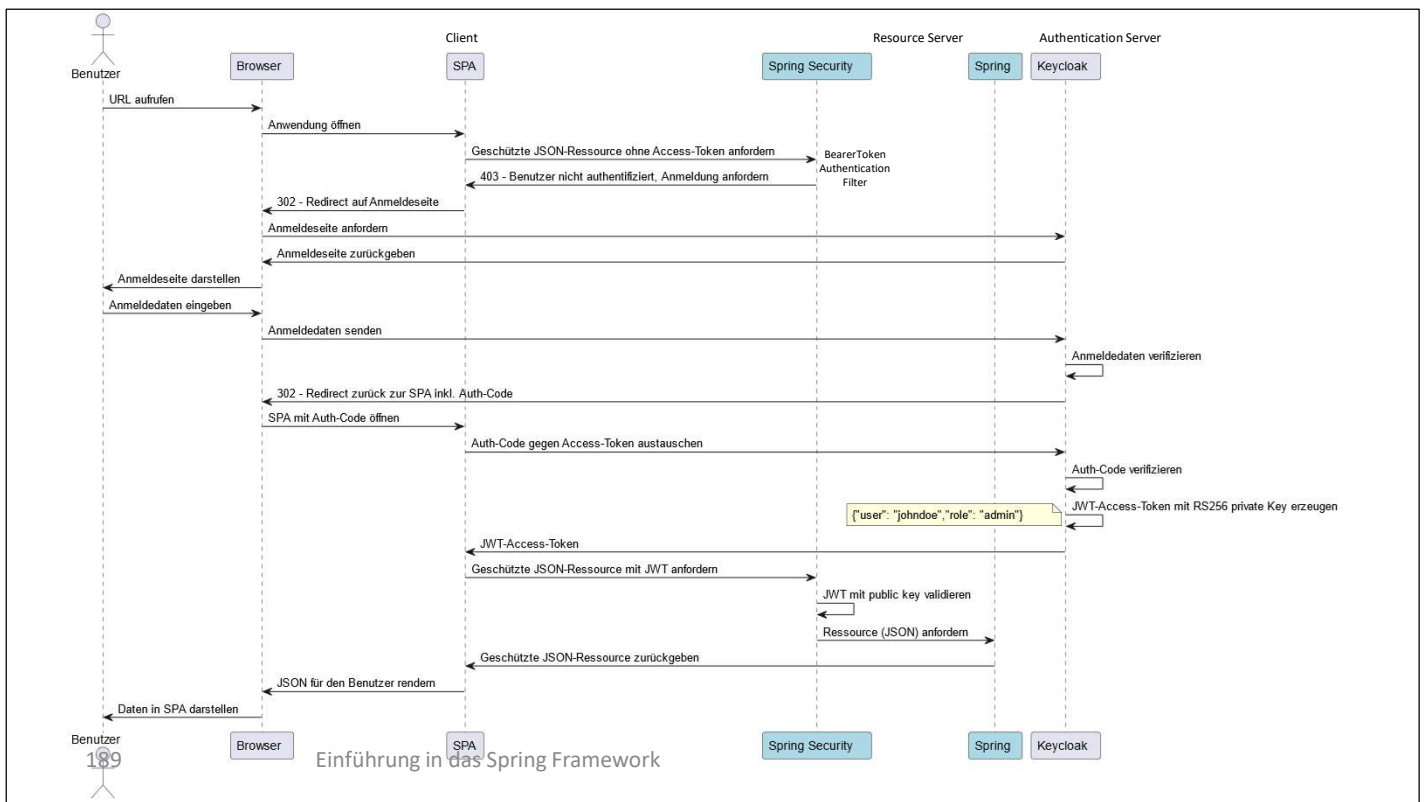
187

Einführung in das Spring Framework

OpenID Connect (OIDC)

- Basiert auf OAuth2 und erweitert es

	OAuth 2.0	OpenID Connect (OIDC)
Zweck	Autorisierung von Zugriffen	Authentifizierung von Benutzern und Autorisierung von Zugriffen
Fokus	Autorisierung	Identität und Authentifizierung
Token-Typ	Zugriffstoken	Zugriffstoken und ID-Token
Ausstellender Server	Authorization Server	Authorization Server
Verwendete Protokolle	OAuth 2.0	OAuth 2.0, ID-Token
Nutzung von Claims	Nicht spezifiziert in OAuth 2.0	Ja, zur Übermittlung von Benutzerinformationen
Benutzeridentität	Nicht explizit unterstützt	Ja, mit Hilfe von ID-Token
Anwendungsbereich	Gewährt Zugriff auf Ressourcen	Gewährt Zugriff auf Ressourcen und liefert Benutzerinformationen
Beispiel	Eine Anwendung, die Zugriff auf Kontakte benötigt	Single-Sign-On



Das Sequenzdiagramm zeigt den Ablauf einer Authentifizierung und Autorisierung in einer Single-Page-Anwendung (SPA) mit Spring Security und Keycloak. Ein Benutzer ruft zunächst eine URL auf, um die SPA zu öffnen. Die SPA fordert dann eine geschützte JSON-Ressource ohne JWT-Token an, was dazu führt, dass "Spring Security" eine 403-Fehlermeldung zurückgibt, da der Benutzer nicht authentifiziert ist. Daraufhin wird der Benutzer auf die Anmeldeseite weitergeleitet.

Der Browser fordert dann die Anmeldeseite vom Keycloak-Server an und gibt sie an den Benutzer weiter, der seine Anmeldedaten eingibt. Der Browser sendet diese Daten dann an Keycloak, welches die Anmeldedaten überprüft und anschließend ein JWT-Zugriffstoken mit einem RS256 private Key für den Benutzer erstellt. Das JWT-Token enthält Informationen über den Benutzer (hier: "johndoe") und seine Rolle (hier: "admin").

Der Keycloak-Server sendet dann eine 302-Weiterleitung zurück an den Browser, der daraufhin die SPA mit dem JWT-Token öffnet. Die SPA kann nun die geschützte JSON-Ressource mit dem JWT-Token anfordern. "Spring Security" validiert das JWT-Token mit dem public Key und gibt dann die Ressource (JSON) an die SPA zurück, die sie dann im Browser für den Benutzer rendert und darstellt.

@startuml

```

actor Benutzer
participant Browser
participant SPA
participant "Spring Security" #lightblue
participant Spring #lightblue
participant Keycloak
  
```

Benutzer -> Browser: URL aufrufen

Browser -> SPA: Anwendung öffnen

SPA-> "Spring Security": Geschützte JSON-Ressource ohne JWT anfordern

"Spring Security" -> SPA: 403 - Benutzer nicht authentifiziert, Anmeldung anfordern

SPA -> Browser: 302 - Redirect auf Anmeldeseite

Browser -> Keycloak: Anmeldeseite anfordern
Keycloak -> Browser: Anmeldeseite zurückgeben
Browser -> Benutzer: Anmeldeseite darstellen
Benutzer -> Browser: Anmeldedaten eingeben
Browser -> Keycloak: Anmeldedaten senden
Keycloak -> Keycloak: Anmeldedaten verifizieren
Keycloak -> Keycloak: JWT-Zugriffstoken mit RS256 private Key erzeugen
note left: {"user": "johndoe","role": "admin"}
Keycloak -> Browser: 302 - Redirect zurück zur SPA inkl. JWT-Zugriffstoken
Browser -> SPA: SPA mit JWT-Zugriffstoken öffnen
SPA -> "Spring Security": Geschützte JSON-Ressource mit JWT anfordern
"Spring Security" -> "Spring Security": JWT mit public key validieren
"Spring Security" -> Spring: Ressource (JSON) anfordern
Spring -> SPA: Geschützte JSON-Ressource zurückgeben
SPA -> Browser: JSON für den Benutzer rendern
Browser -> Benutzer: Daten in SPA darstellen
@endum1

JSON Web Tokens (JWT)

Encoded

PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gR691IiwiaWF0Ij0yMn0.NHVaYe26Mbt0YhSKkoKYdFVomg4i8ZJd8_-RU8Vnbftc4TSMb4bXP3l3Y1NWACwyXPGffz5aXHc6lty1Y2t4SWRqGteragsVdZufDn5BlnJl9pdR_kdVFUra2rWKEofkZeIC4yWytE58sMIihvo9H1ScmmVwBcQP6XETqYd0aSHp1g0a9RdUPDvoXQ5oqygTqVtxaDr6wUFKrKIItgBMzWIdNZ6y709E0DhEPTbE9rfBo6KTFsHAZnMg4k68CDp2woYIaXbmYTWcvbzIuH07_37GT79XdIwkm95QJ7hYC9RiwrV7mesbY4PAahERJawntho8my942XheVLmGwLMBkQ

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  -----BEGIN PUBLIC KEY-----
  MII81jANBgkqhkiG9w0BAQFAAOC
  AQ8AMII8CgKCAQEaUISuLLFVLPHC
  ozMxH2Mo
  -----BEGIN PRIVATE KEY-----
  MII81jANBgkqhkiG9w0BAQFA
  AASCBKkkggS1AgEAAoIBAQC7VJTU
  t9Us8cKj
  MzEFyyj3WA4R4/M2bS1GB4t7NXp9
)
```

190

Einführung in das Spring Framework

- Ein Access Token im JWT-Format ist ein verschlüsseltes Token zur Autorisierung von Zugriffen auf eine Ressource.
- Das Token enthält Ansprüche (Claims) mit Informationen über den Benutzer oder die Entität.
- Mögliche Anwendungsbereiche des Access Tokens im JWT-Format sind:
 - Authentifizierung des Benutzers
 - Autorisierung von Zugriffsberechtigungen
 - Single Sign-On bei verschiedenen Diensten und Anwendungen
 - Sicherer Datenaustausch zwischen Anwendungen und Diensten
- Ein Access Token im JWT-Format erhöht die Sicherheit von Anwendungen und Diensten, indem es eine sichere und zuverlässige Methode für die Authentifizierung und Autorisierung von Benutzern bereitstellt.

Warum JWT?

- Ein JWT kann Benutzerdaten wie z.B. den Benutzernamen, eine Rolle oder Berechtigungen beinhalten
- Das JWT ist über die Signatur vor Manipulation geschützt und kann daher über den Browser übertragen werden
- Spring Security kann per Public-Key schnell verifizieren, ob das Token valide ist

Spring Security Konfiguration

- Der oauth2-resource-server Starter übernimmt die Konfiguration von Spring Security

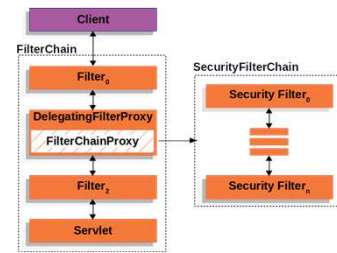
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

- JSON Web Key (JWK) liefert eine Liste der verfügbaren Public Keys für die Verifizierung des Access-Tokens

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          jwk-set-uri:http://idp/auth/realms/myrealm/protocol/openid-connect/certs
```

BearerTokenAuthenticationFilter

- Authentication Header mit Bearer Token vorhanden?
- JWT syntaktisch korrekt?
- Verify JWT
 - Public Key des Authentication Servers notwendig
 - Über JWK-Url des Authentication Servers abrufen
 - JWT mit Public Key verifizieren



- Alternative: Opaque Tokens
 - Token ist zufällige Zeichenkette
 - Token kann über einen zusätzlichen Aufruf beim Auth Server verifiziert werden

```
spring:
  security:
    oauth2:
      resourceserver:
        opaque-token:
          introspection-uri: https://idp.example.com/introspect
          client-id: client
          client-secret: secret
```

Gliederung

- Spring Grundlagen
- @Bean
- Testen
- Profile
- Properties
- AOP
- Datenbankzugriff
- Webanwendungen
- REST
- Build- und Deployment
- Betrieb – Spring Boot Actuator
- Security
- **Sonstiges**

@Async und @Scheduled

- **@Scheduled**

```
@Scheduled(fixedRate=5000)
public void doSomething() {
    // something that should execute periodically
}
```

```
@Scheduled(cron="*/5 * * * * MON-FRI")
public void doSomething() {
    // something that should execute on weekdays only
}
```

- **@Async**

```
@Async
void doSomething(String s) {
    // this will be executed asynchronously
}
```

```
@Async
Future<String> returnSomething(int i) {
    // this will be executed asynchronously
}
```

- **Konfiguration**

```
@Configuration
@EnableAsync
@EnableScheduling
public class AppConfig {
}
```

195

Einführung in das Spring Framework

Mit Spring können sehr einfach nebenläufige Aktionen ausgelöst werden. Dazu stehen zwei verschiedene Annotationen zur Verfügung:

@Scheduled

Mit dieser Annotation können Aktionen (Methoden) zu einem bestimmten Zeitpunkt ausgelöst werden. Es stehen die folgenden Varianten zur Verfügung:

fixedRate: zb. alle 5 Sekunden

fixedDelay: zb. einmalig eine Minute nach dem Start der Anwendung

initialDelay und fixedRate: zb. alle 5 Sekunden eine Minute nach dem Start der Anwendung

cron: zb. Montag bis Freitag um 3 Uhr nachts

@Async

Die @Async-Annotation kann auf einer Methode bereitgestellt werden, so dass der Aufruf dieser Methode asynchron erfolgt. Mit anderen Worten, der Anrufer wird sofort nach dem Aufruf zurückkehren und die tatsächliche Ausführung der Methode findet in einem separaten Thread statt.

Auch Methoden, die einen Wert zurückgeben, können asynchron aufgerufen werden. Solche Methoden benötigen einen Rückgabewert vom Typ „Future“. Dies bietet immer noch den Vorteil der asynchronen Ausführung, so dass der Anrufer andere Aufgaben vor dem Aufruf von get () auf diesem Future Objekt ausführen kann.

Cache Abstraktion

Caching von Methodenaufrufen per Annotation aktivieren

```
@Cacheable("cache")
public int add(int a, int b) {
    // Komplexe Berechnung
    return a + b;
}
```

Wie der Name schon sagt, wird `@Cacheable` verwendet, um Methoden zu markieren, die cachefähig sind, dh. Methoden, für die das Ergebnis in den Cache gespeichert wird, so dass bei nachfolgenden Aufrufen (mit den gleichen Argumenten) der Wert im Cache zurückgegeben wird, ohne dass die Methode tatsächlich ausgeführt werden muss. In ihrer einfachsten Form erfordert die Annotationsdeklaration den Namen des Caches, der mit der annotierten Methode verknüpft ist.

Cache Abstraktion

Konfiguration des JDK ConcurrentMap-based Cache. Als Alternative kann auch der Ehcache verwendet werden.

```
@Configuration
@EnableCaching

public class CacheConfig {

    @Bean
    public CacheManager cacheManager() {
        SimpleCacheManager simpleCacheManager = new SimpleCacheManager();
        simpleCacheManager.setCaches(Collections.singleton(new
            ConcurrentMapCache("cache")));
        return simpleCacheManager;
    }
}
```

197

Einführung in das Spring Framework

Die Konfiguration des Caches erfolgt über die Klasse „CacheManager“. Es stehen verschiedene Cache-Implementierungen von In-Memory-Caches bis hin zu großen persistenten und verteiltes Caches zur Verfügung.

Cache Abstraktion

Bedingtes Caching

```
@Cacheable(value="book", condition="#name.length < 32")  
  
public Book findBook(String name);
```

Benutzerdefinierter Cache-Schlüssel

```
@Cacheable(value="books", key="#isbn.rawNumber")  
  
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed);
```

Bedingtes Caching:

Caching wird nur verwendet, falls die angegebene Bedingung erfüllt ist. Ansonsten wird der Methodenaufruf jedesmal durchgeführt.

Benutzerdefinierter Cache-Schlüssel:

Nur spezielle Werte sollen für die Bestimmung des Schlüssels innerhalb der Cache-Map verwendet werden. Im Beispiel: „rawNumber“ aus der Klasse „ISBN“ da die beiden Flags keinen Einfluß auf das Ergebnis der Berechnung besitzen.

Normalerweise wird der Schlüssel aus den hashCodes der Parameter bestimmt

Cache Abstraktion

Verwerfen der Cache-Einträge

```
@CacheEvict(value = "books", allEntries = true)  
  
public void reBooks();
```

Methode befüllt den Cache, wird aber trotzdem bei jedem Aufruf durchlaufen

```
@CachePut("books")  
  
public Book reloadBook(String name);
```

Literaturhinweise



- **Spring Boot Reference Documentation**
 - <https://docs.spring.io/spring-boot/docs/current/reference/html/>

Links

- **Spring Boot Project**
 - <http://projects.spring.io/spring-boot/>
- **Spring Boot Reference Guide**
 - <https://docs.spring.io/spring-boot/docs/current/reference/html/>
- **Spring Blog**
 - <https://spring.io/blog>

Better together