

Kostenloses eBook

LERNEN spring-boot

Free unaffiliated eBook created from
Stack Overflow contributors.

#spring-boot

Inhaltsverzeichnis

Über ... 1

Kapitel 1: Erste Schritte mit Spring-Boot ... 2

Bemerkungen ... 2

Versionen ... 2

Examples ... 2

Installation oder Setup ... 2

Einfache Spring Boot-Webanwendung mit Gradle als Buildsystem ... 4

Kapitel 2: Bereitstellen einer Beispielanwendung mithilfe von Spring-Boot auf Amazon Elast ... 6

Examples ... 6

Bereitstellen einer Beispielanwendung mit Spring-Boot im Jar-Format unter AWS ... 6

Kapitel 3: Controller ... 13

Einführung ... 13

Examples ... 13

Spring Boot Rest Controller ... 13

Kapitel 4: Eine Spring-Boot-Anwendung an MySQL anschließen ... 16

Einführung ... 16

Bemerkungen ... 16

Examples ... 16

Spring-Boot-Beispiel mit MySQL ... 16

Kapitel 5: Erstellen und Verwenden mehrerer application.properties-Dateien ... 21

Examples ... 21

Dev- und Prod-Umgebung mit unterschiedlichen Datenquellen ... 21

Stellen Sie das richtige Federprofil ein, indem Sie die Anwendung automatisch erstellen (m ... 22

Kapitel 6: Federstiefel + Spring Data JPA ... 25

Einführung ... 25

Bemerkungen ... 25

Anmerkungen ... 25

Offizielle Dokumentation ... 25

Examples ... 26

Beispiel für die Integration von Spring Boot und Spring Data JPA ... 26

Hauptklasse 26
Entity-Klasse 26
Transiente Eigenschaften 27
DAO-Klasse 28
Serviceklasse 28
Service Bean 29
Controller-Klasse 30
Anwendungseigenschaftendatei für MySQL-Datenbank 31
SQL-Datei 31
pom.xml-Datei 31
Erstellen eines ausführbaren JAR 32
Kapitel 7: Installieren der Spring Boot-CLI 33
Einführung 33
Bemerkungen 33
Examples 34
Manuelle Installation 34
Installieren Sie unter Mac OSX mit HomeBrew 34
Installieren Sie unter Mac OSX mit MacPorts 34
Auf jedem Betriebssystem mit SDKMAN installieren! 34
Kapitel 8: REST-Dienste 35
Parameter 35
Examples 35
Erstellen eines REST-Service 35
Erstellen eines Ruhedienstes mit JERSEY und Spring Boot 38
1.Projekt-Setup 38
2. Controller erstellen 38
3.Wiring Jersey-Konfigurationen 39
4.Done 39
REST-API mit RestTemplate (GET) verwenden 40
Kapitel 9: Scannen von Paketen 42
Einführung 42

Parameter ... 42
Examples ... 43
@SpringBootApplication ... 43
@ComponentScan ... 44
Erstellen Sie Ihre eigene Autokonfiguration ... 45
Kapitel 10: Spring Boot + JPA + REST ... 47
Bemerkungen ... 47
Examples ... 47
Spring Boot Startup ... 47
Domain-Objekt ... 47
Repository-Schnittstelle ... 48
Maven-Konfiguration ... 49
Kapitel 11: Spring Boot + Ruhezustand + Web-Benutzeroberfläche (Thymeleaf) ... 51
Einführung ... 51
Bemerkungen ... 51
Examples ... 51
Abhängigkeiten von Maven ... 51
Ruhezustand der Konfiguration ... 52
Entitäten und Repositories ... 53
Thymeleaf-Ressourcen und Federsteuerung ... 53
Kapitel 12: Spring Boot + Spring Data Elasticsearch ... 56
Einführung ... 56
Examples ... 56
Integration von Spring Boot und Spring Data Elasticsearch ... 56
Spring Boot und Spring Data Elasticsearch Integration ... 56
Kapitel 13: Spring Boot-Hibernate-REST-Integration ... 65
Examples ... 65
Hibernate-Unterstützung hinzufügen ... 65
REST-Unterstützung hinzufügen ... 66
Kapitel 14: Spring-Boot + JDBC ... 68
Einführung ... 68
Bemerkungen ... 68

Examples ... 69
schema.sql-Datei ... 69
Erste JdbcTemplate-Startanwendung ... 70
data.sql ... 70
Kapitel 15: Springboot + JPA + MongoDB ... 71
Examples ... 71
CRUD-Vorgang in MongoDB mit JPA ... 71
Kundencontroller ... 72
Kunden-Repository ... 73
pom.xml ... 73
Fügen Sie Daten mit dem Restclient ein: POST-Methode ... 73
Anforderungs-URL abrufen ... 74
Anforderungsergebnis abfragen: ... 74
Kapitel 16: Spring-Boot Microservice mit JPA ... 76
Examples ... 76
Anwendungsklasse ... 76
Buchmodell ... 76
Buchspeicher ... 77
Validierung aktivieren ... 77
Laden von Testdaten ... 78
Validator hinzufügen ... 78
Gradle Build-Datei ... 79
Kapitel 17: Testen im Spring Boot ... 80
Examples ... 80
So testen Sie eine einfache Spring-Boot-Anwendung ... 80
Laden verschiedener yaml-Dateien [oder Eigenschaften] oder Überschreiben einiger Eigenscha ... 83
Laden einer anderen Yml-Datei ... 83
Alternativ Optionen ... 83
Kapitel 18: ThreadPoolTaskExecutor: Konfiguration und Verwendung ... 85
Examples ... 85
Anwendungskonfiguration ... 85
Kapitel 19: Vollständig responsive Spring Boot-Webanwendung mit JHipster ... 86

Examples 86
Erstellen Sie Spring Boot App mit jHipster unter Mac OS 86
Kapitel 20: Zwischenspeicherung mit Redis mit Spring Boot für MongoDB 90
Examples 90
Warum zwischenspeichern? 90
Das Basissystem 90
Credits 97

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: spring-boot

It is an unofficial and free spring-boot ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official spring-boot.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit Spring-Boot

Bemerkungen

In diesem Abschnitt erhalten Sie einen Überblick darüber, was Spring-Boot ist und warum ein Entwickler es verwenden möchte.

Es sollte auch alle großen Themen im Springboot erwähnen und auf die verwandten Themen verweisen. Da die Dokumentation für Spring-Boot neu ist, müssen Sie möglicherweise erste Versionen dieser verwandten Themen erstellen.

Versionen

Ausführung	Veröffentlichungsdatum
1,5	2017-01-30
1.4	2016-07-28
1.3	2015-11-16
1.2	2014-12-11
1.1	2014-06-10
1,0	2014-04-01

Examples

Installation oder Setup

Die erste Einrichtung von Spring Boot ist dank der harten Arbeit der Spring Community recht schnell.

Voraussetzungen:

1. Java installiert
2. Java IDE empfohlen (nicht erforderlich) (Intellij, Eclipse, Netbeans usw.)

Sie müssen Maven und / oder Gradle nicht installiert haben. Die vom Spring Initializr generierten Projekte werden mit einem Maven Wrapper (Befehl `mvnw`) oder Gradle Wrapper (Befehl `gradlew`) `gradlew` .

Öffnen Sie Ihren Webbrowser für https://start.spring.io. Dies ist ein Launchpad zum Erstellen neuer Spring Boot-Anwendungen. Jetzt werden wir das Nötigste tun.

Fühlen Sie sich frei, von Maven zu Gradle zu wechseln, wenn dies Ihr bevorzugtes Build-Tool ist.

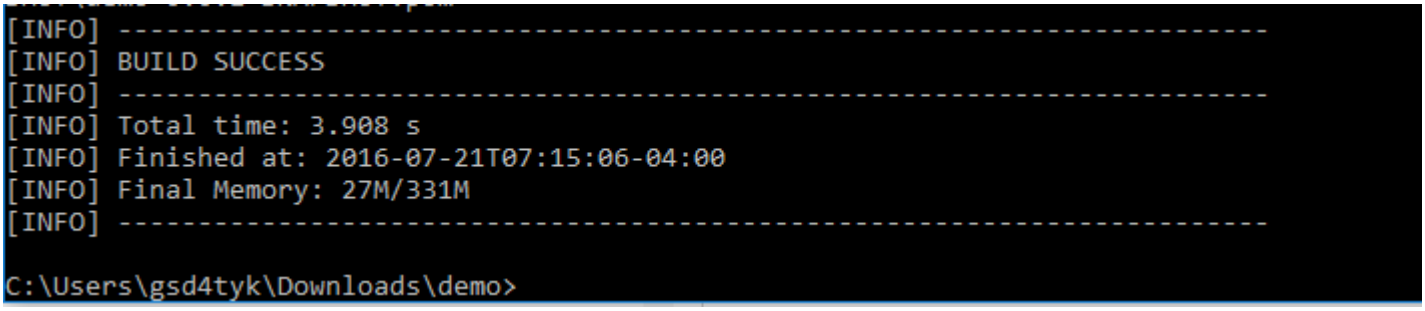
Suchen Sie unter "Suche nach Abhängigkeiten" nach "Web" und fügen Sie es hinzu.

Klicken Sie auf Projekt generieren!

Dadurch wird eine ZIP-Datei namens Demo heruntergeladen. Fühlen Sie sich frei, diese Datei an einem beliebigen Ort auf Ihrem Computer zu extrahieren.

Wenn Sie maven auswählen, navigieren Sie in einer Eingabeaufforderung zum Basisverzeichnis und geben Sie den Befehl `mvn clean install`

Sie sollten eine Build-Erfolgsausgabe erhalten:



Ausführen Ihrer Anwendung: `mvn spring-boot:run`

Jetzt startet Ihre Spring Boot-Anwendung. Navigieren Sie in Ihrem Webbrowser zu localhost: 8080

Glückwunsch! Sie haben gerade Ihre erste Spring Boot-Anwendung in Betrieb genommen. Jetzt können wir ein kleines bisschen Code hinzufügen, damit Sie sehen können, wie es funktioniert.

Verwenden Sie also `Strg` + `C` , um den aktuell laufenden Server zu `beenden` .

Navigieren Sie zu: `src/main/java/com/example/DemoApplication.java` Aktualisieren Sie diese Klasse, um einen Controller zu erhalten

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@SpringBootApplication
public class DemoApplication {

    @RequestMapping("/")
    String home() {
        return "Hello World!";
    }

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

Mit guten Sachen können Sie jetzt das Projekt mit `mvn clean install spring-boot:run` erneut erstellen und ausführen `mvn clean install spring-boot:run` !

Navigieren Sie jetzt in Ihrem Webbrowser zu localhost: 8080

Hallo Welt!

Glückwunsch! Wir haben gerade die Erstellung einer Spring Boot-Anwendung abgeschlossen und unseren ersten Controller so eingerichtet, dass "Hello World!" Willkommen in der Welt von Spring Boot!

Einfache Spring Boot-Webanwendung mit Gradle als Buildsystem

In diesem Beispiel wird davon ausgegangen, dass Sie Java und Gradle bereits installiert haben.

Verwenden Sie die folgende Projektstruktur:

```
src/
  main/
    java/
      com/
        example/
          Application.java
build.gradle
```

`build.gradle` ist Ihr `build.gradle` für Gradle Build System mit folgendem Inhalt:

```
buildscript {
  ext {
    //Always replace with latest version available at http://projects.spring.io/spring-
boot/#quick-start
    springBootVersion = '1.5.6.RELEASE'
  }
  repositories {
    jcenter()
  }
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
  }
}

apply plugin: 'java'
apply plugin: 'org.springframework.boot'

repositories {
  jcenter()
}

dependencies {
  compile('org.springframework.boot:spring-boot-starter-web')
}
```

`Application.java` ist die Hauptklasse der Spring Boot-Webanwendung:

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentScan
@RestController
public class Application {

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

  @RequestMapping("/hello")
  private String hello() {
    return "Hello World!";
  }
}
```

Jetzt können Sie die Spring Boot-Webanwendung mit ausführen

```
gradle bootRun
```

und greifen Sie mit `curl` auf den veröffentlichten HTTP-Endpunkt zu

```
curl http://localhost:8080/hello
```

oder Ihren Browser durch Öffnen von localhost: 8080 / hallo .

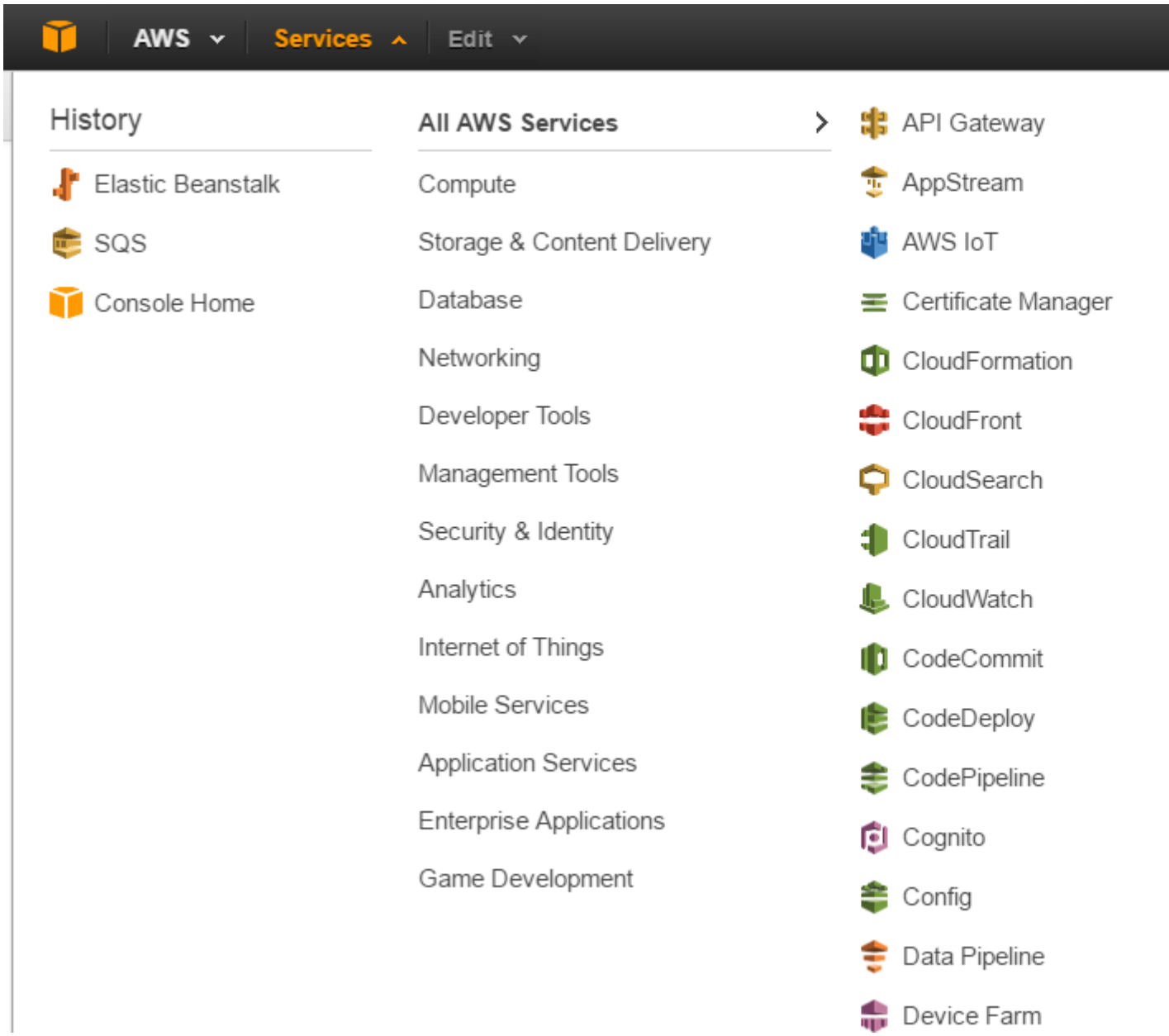
Erste Schritte mit Spring-Boot online lesen: https://riptutorial.com/de/spring-boot/topic/829/erste-schritte-mit-spring-boot

Kapitel 2: Bereitstellen einer Beispielanwendung mithilfe von Spring-Boot auf Amazon Elastic Beanstalk

Examples

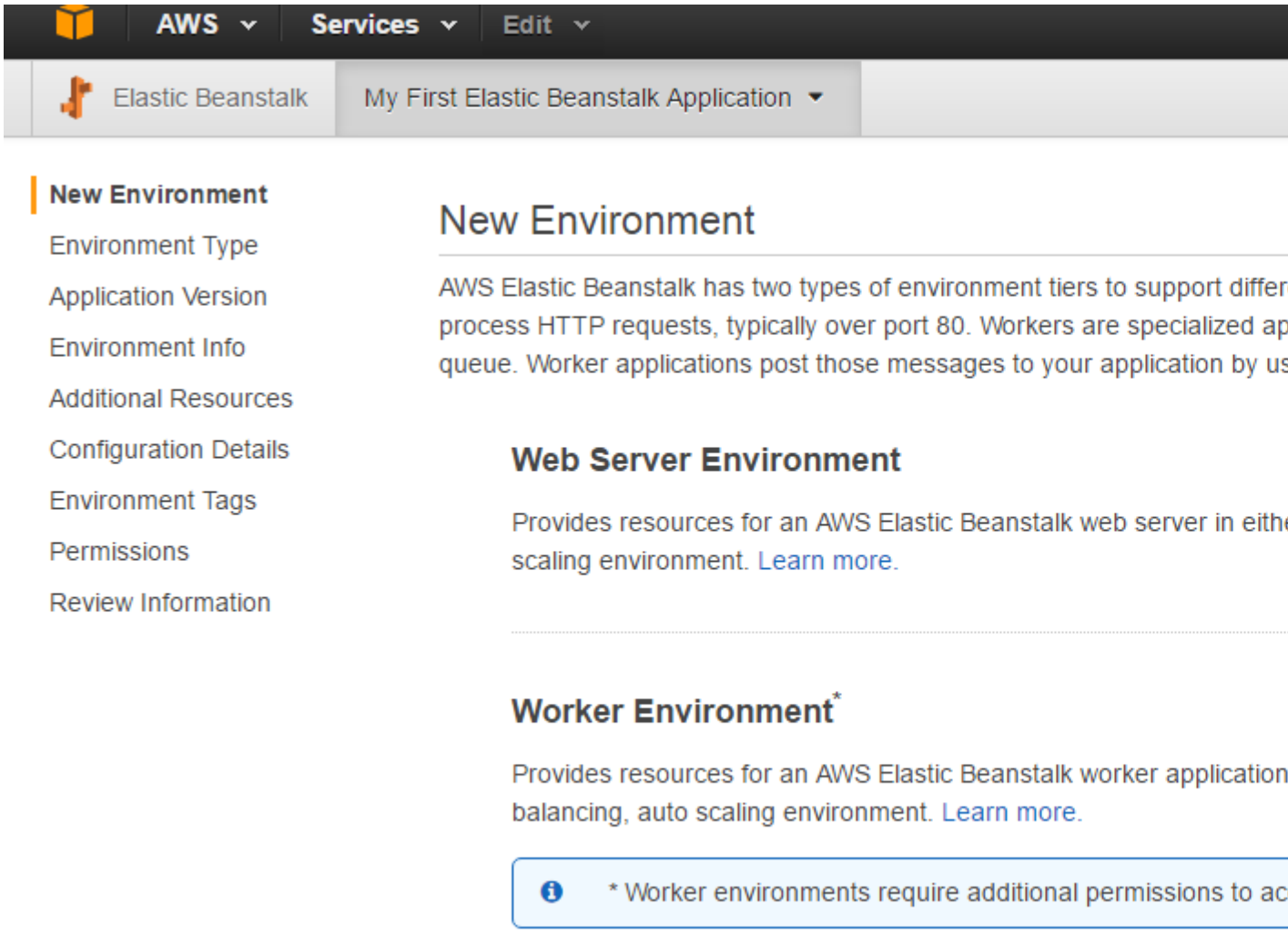
Bereitstellen einer Beispielanwendung mit Spring-Boot im Jar-Format unter AWS

1. Erstellen Sie eine Beispielanwendung mit Spring-Boot von der Spring-Boot-Initialisierungssite .
2. Importieren Sie den Code in Ihre lokale IDE und führen Sie das Ziel als **sauberes Installations-Spring-Boot aus: run -e**
3. Gehen Sie zum Zielordner und suchen Sie nach der JAR-Datei.
4. Öffnen Sie Ihr Amazon-Konto oder erstellen Sie ein neues Amazon-Konto und wählen Sie wie unten gezeigt für Elastic Beanstalk aus



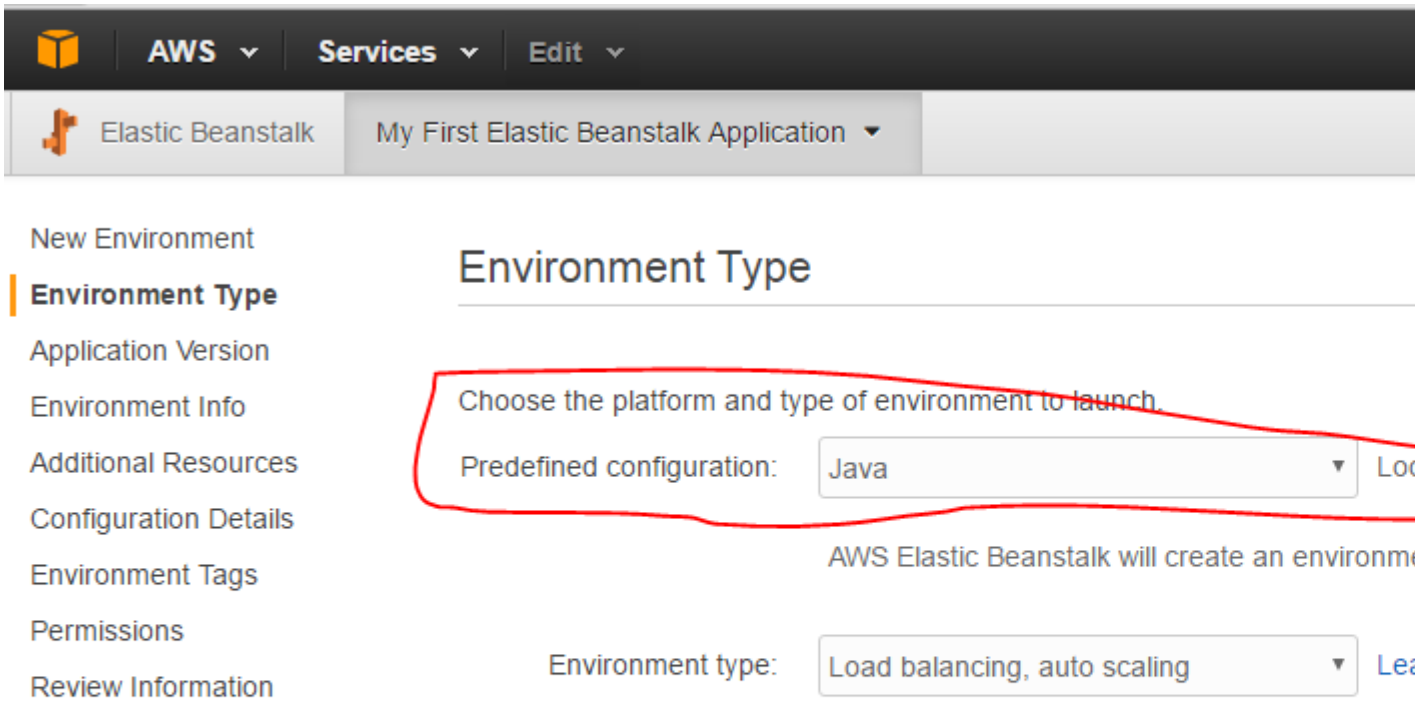
.

5. Erstellen Sie eine neue Webserverumgebung, wie unten gezeigt

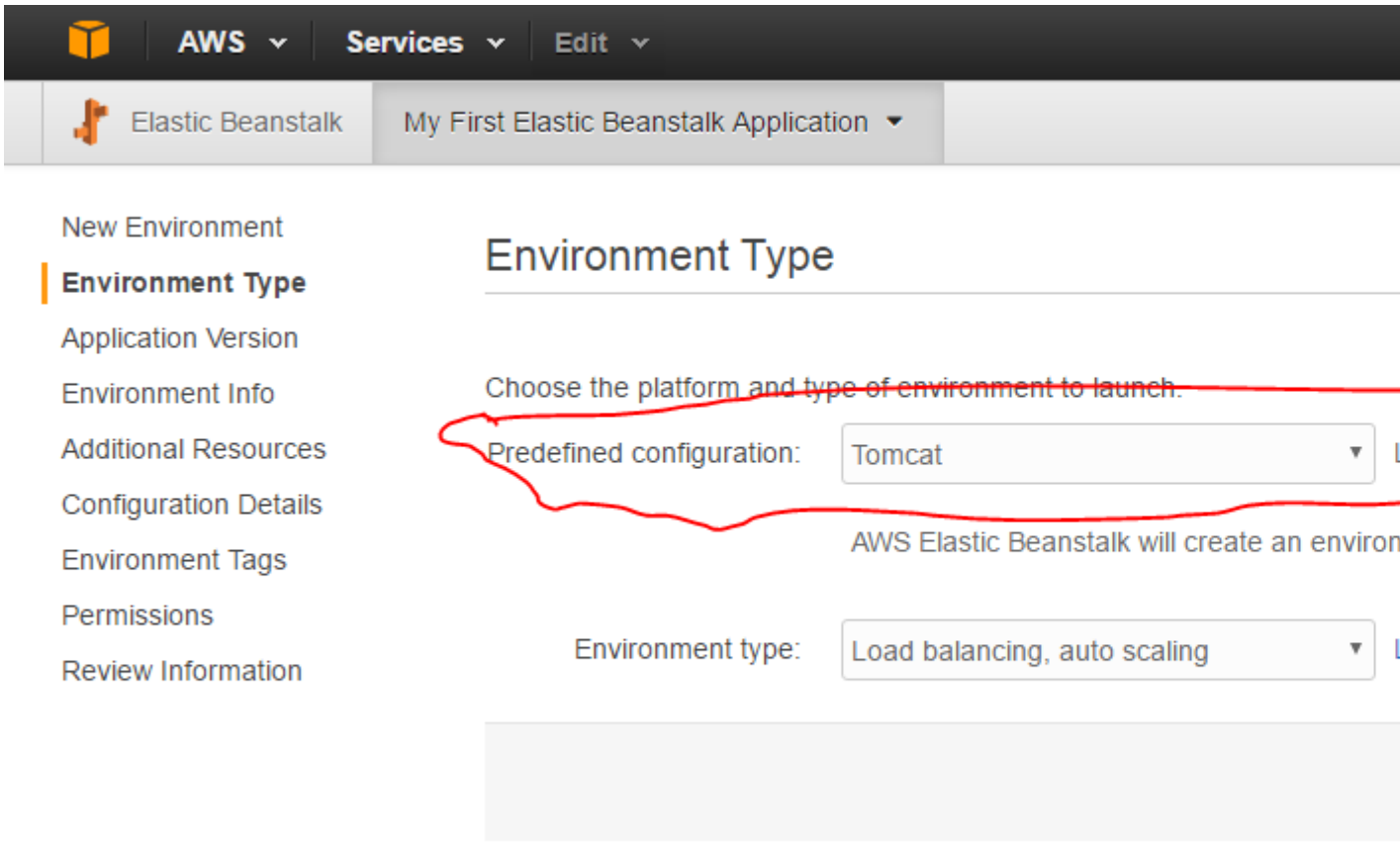


.

6. Wählen Sie den Umgebungstyp als Java für die Bereitstellung von **JAR-** Dateien für Spring-Boot aus. Wenn Sie die Bereitstellung als **WAR-** Datei planen, sollte diese als Tomcat ausgewählt werden (siehe unten)

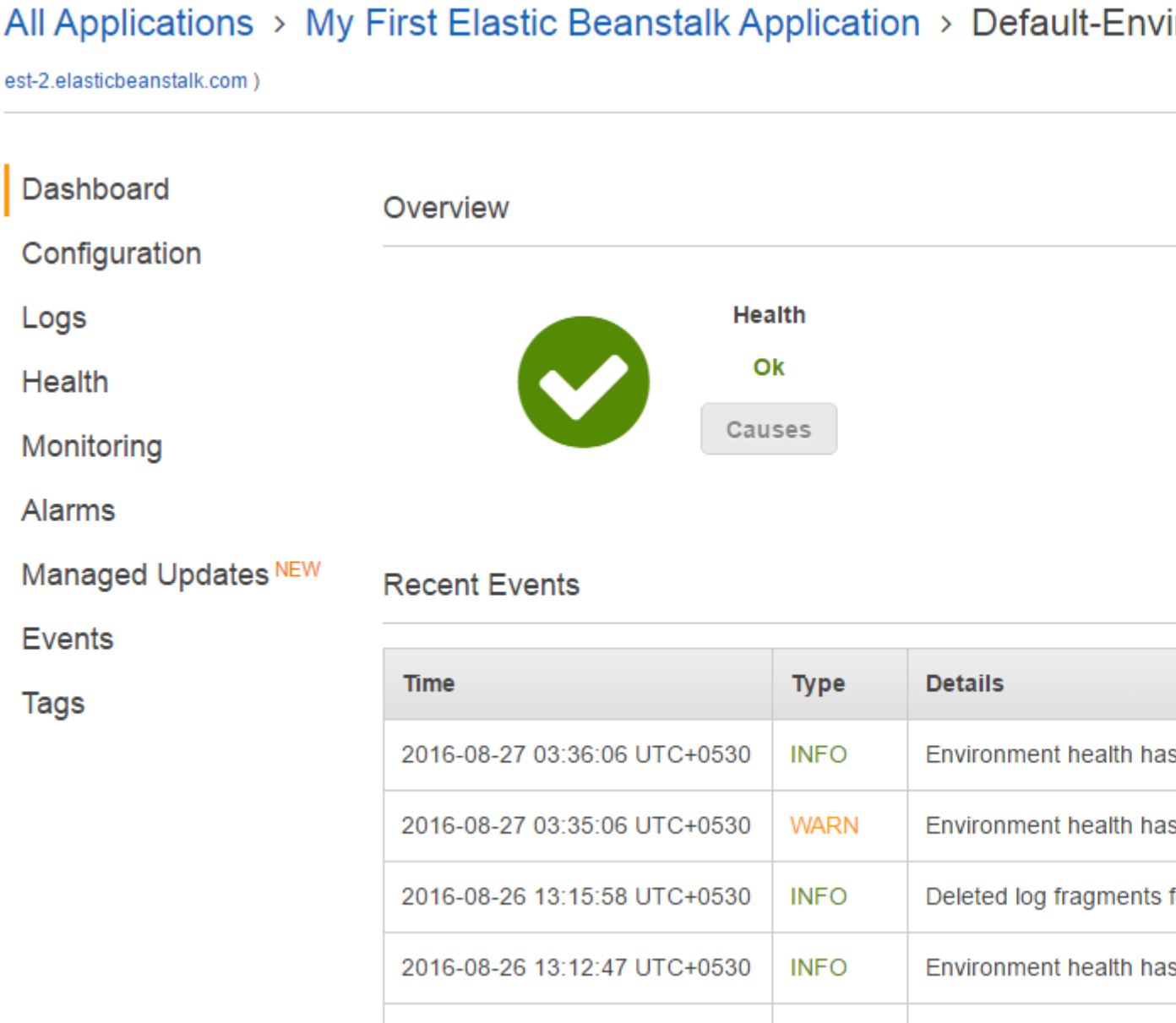

AWS
Services
Edit
Elastic Beanstalk
My First Elastic Beanstalk Application
New Environment
Environment Type
Application Version
Environment Info
Additional Resources
Configuration Details
Environment Tags
Permissions
Review Information
Environment Type
Choose the platform and type of environment to launch.
Predefined configuration:
Java
AWS Elastic Beanstalk will create an environme
Environment type:
Load balancing, auto scaling

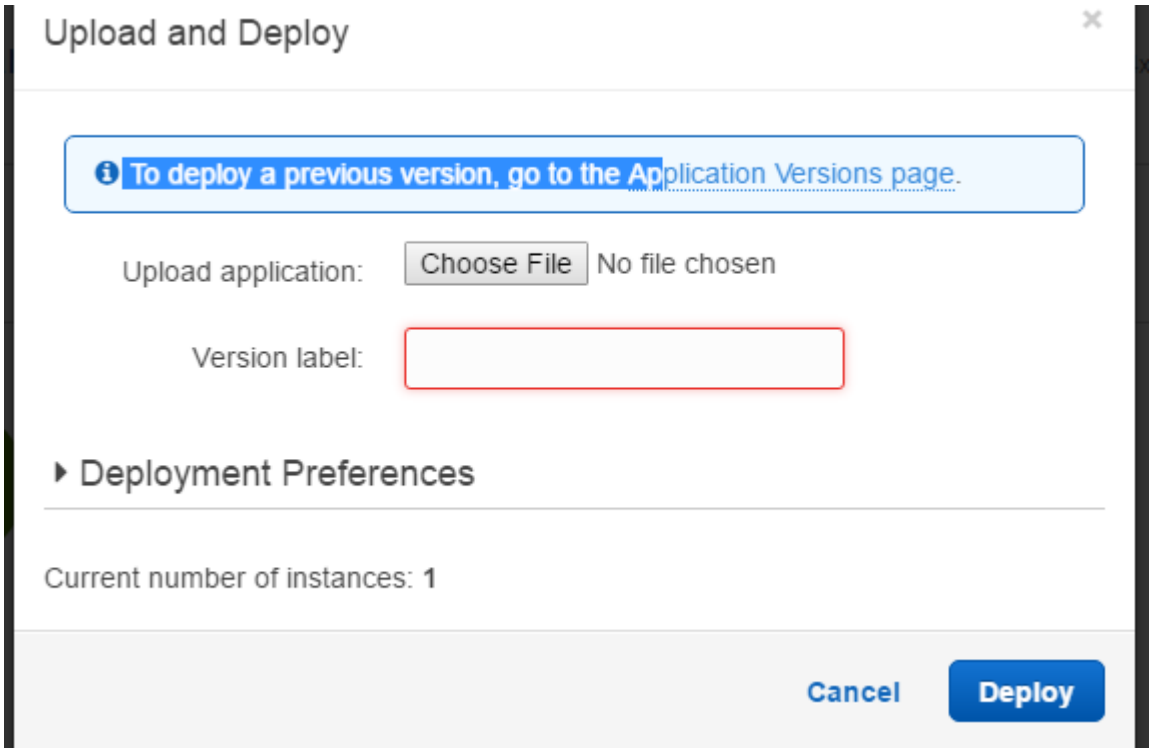
.


AWS
Services
Edit
Elastic Beanstalk
My First Elastic Beanstalk Application
New Environment
Environment Type
Application Version
Environment Info
Additional Resources
Configuration Details
Environment Tags
Permissions
Review Information
Environment Type
Choose the platform and type of environment to launch.
Predefined configuration:
Tomcat
AWS Elastic Beanstalk will create an environ
Environment type:
Load balancing, auto scaling

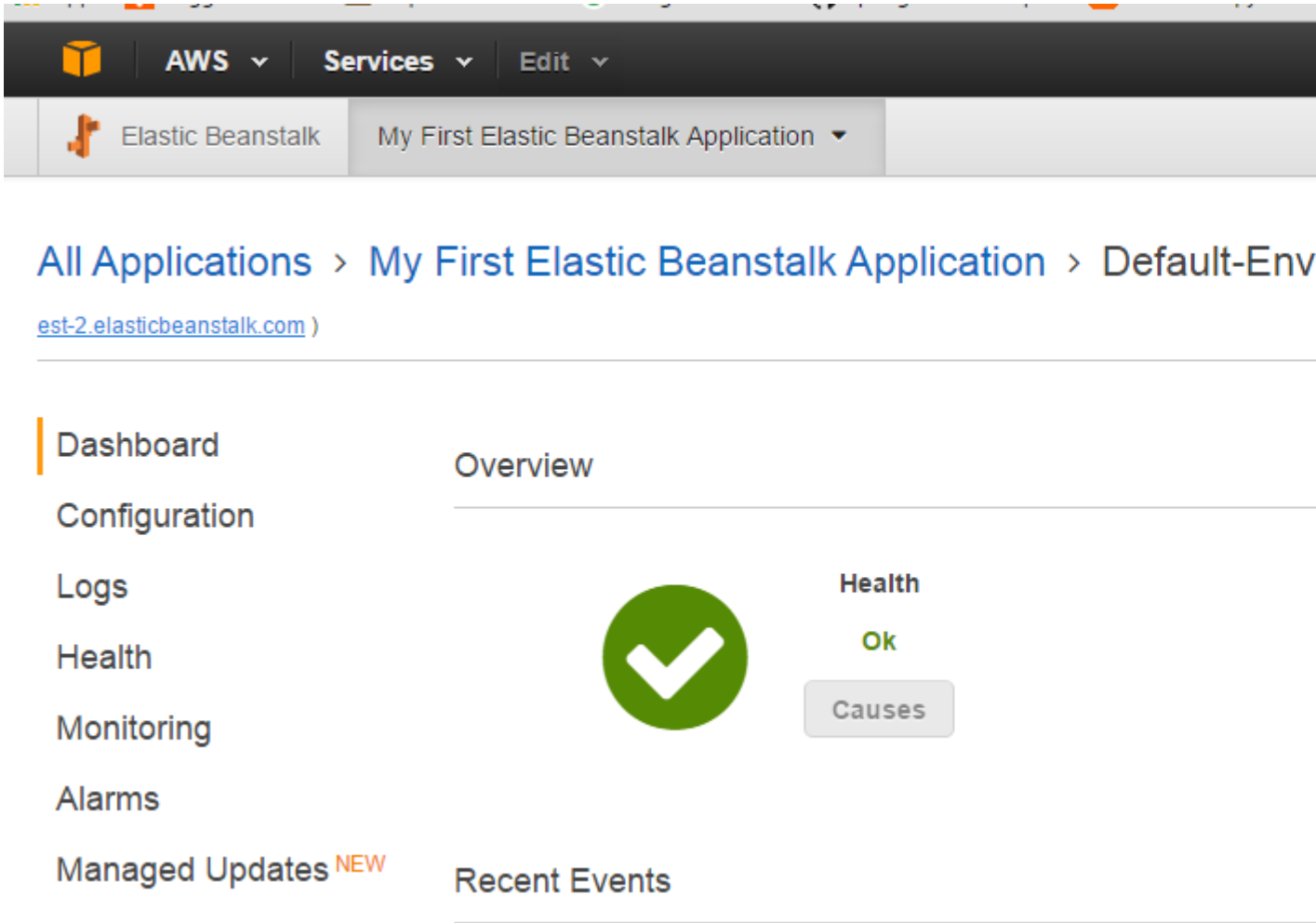
7. Klicken Sie auf Weiter, wenn Sie auf Weiter klicken.

8. Wenn Sie die Standardkonfiguration abgeschlossen haben, kann die JAR-Datei im Übersichtsbildschirm wie in den Abbildungen gezeigt hochgeladen und bereitgestellt werden.

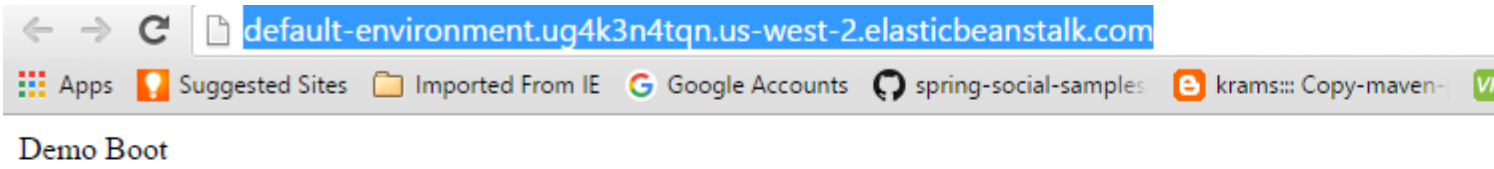




9. Sobald die Bereitstellung erfolgreich war (5 bis 10 Minuten zum ersten Mal), können Sie die Kontext-URL wie in der folgenden Abbildung gezeigt anklicken.



10. Das Ergebnis ist wie unten gezeigt, es sollte genauso funktionieren wie in Ihrer lokalen Umgebung.



11. Bitte finden Sie meine Github-URL

Bereitstellen einer Beispielanwendung mithilfe von Spring-Boot auf Amazon Elastic Beanstalk online lesen: https://riptutorial.com/de/spring-boot/topic/6117/bereitstellen-einer-beispielanwendung-mithilfe-von-spring-boot-auf-amazon-elastic-beanstalk

Kapitel 3: Controller

Einführung

In diesem Abschnitt füge ich ein Beispiel für den Spring Boot Rest-Controller mit Get- und Post-Request hinzu.

Examples

Spring Boot Rest Controller.

> In diesem Beispiel werde ich zeigen, wie man einen Rest-Controller formuliert, um Daten mit JPA mit der einfachsten und geringsten Code-Anforderung in die Datenbank zu laden und zu posten.

In diesem Beispiel beziehen wir uns auf die Datentabelle "customerRequirement".

BuyingRequirement.java

@Entity @Table (name = "BUYINGREQUIREMENTS") @NamedQueries ({@NamedQuery (name = "BuyingRequirement.findAll"), query = "SELECT b FROM BuyingRequirement b")}) öffentliche Klasse BuyingRequirement erweitert Domäne implementiert serializable {private static final long serialVersionUID = 1L;

```
@Column(name = "PRODUCT_NAME", nullable = false)
private String productname;

@Column(name = "NAME", nullable = false)
private String name;

@Column(name = "MOBILE", nullable = false)
private String mobile;

@Column(name = "EMAIL", nullable = false)
private String email;

@Column(name = "CITY")
private String city;


public BuyingRequirement() {
}


public String getProductname() {
    return productname;
}

public void setProductname(String productname) {
    this.productname = productname;
}
```

```
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getMobile() {
    return mobile;
}

public void setMobile(String mobile) {
    this.mobile = mobile;
}

public String getEmail() {
    return email;
}

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

public String getCity() {
    return city;
}

public void setCity(String city) {
    this.city = city;
}
```

}

Dies ist die Entitätsklasse, die den Parameter enthält, der sich auf Spalten in der byuingRequirement-Tabelle und deren Getter und Setter bezieht.

IBuyingRequirementsRepository.java (JPA-Schnittstelle)

```
@Repository
@RepositoryRestResource
public interface IBuyingRequirementsRepository extends JpaRepository<BuyingRequirement, UUID>
{
   // Page<BuyingRequirement> findAllByOrderByCreatedDesc(Pageable pageable);
    Page<BuyingRequirement> findAllByOrderByCreatedDesc(Pageable pageable);
    Page<BuyingRequirement> findByNameContainingIgnoreCase(@Param("name") String name,
Pageable pageable);
}
```

BuyingRequirementController.java

```
@RestController
@RequestMapping("/api/v1")
public class BuyingRequirementController {

    @Autowired
    IBuyingRequirementsRepository iBuyingRequirementsRepository;
```

```
    Email email = new Email();

    BuyerRequirementTemplate buyerRequirementTemplate = new BuyerRequirementTemplate();

    private String To = "support@pharmerz.com";
    // private String To = "amigujarathi@gmail.com";
    private String Subject = "Buyer Request From Pharmerz ";

    @PostMapping(value = "/buyingRequirement")
    public ResponseEntity<BuyingRequirement> CreateBuyingRequirement(@RequestBody
BuyingRequirement buyingRequirements) {

        String productname = buyingRequirements.getProductname();
        String name = buyingRequirements.getName();
        String mobile = buyingRequirements.getMobile();
        String emails = buyingRequirements.getEmail();
        String city = buyingRequirements.getCity();
        if (city == null) {
            city = "-";
        }

        String HTMLBODY = buyerRequirementTemplate.template(productname, name, emails, mobile,
city);

        email.SendMail(To, Subject, HTMLBODY);


        iBuyingRequirementsRepository.save(buyingRequirements);
        return new ResponseEntity<BuyingRequirement>(buyingRequirements, HttpStatus.CREATED);
    }


    @GetMapping(value = "/buyingRequirements")
    public Page<BuyingRequirement> getAllBuyingRequirements(Pageable pageable) {

        Page requirements =
iBuyingRequirementsRepository.findAllByOrderByCreatedDesc(pageable);
        return requirements;
    }

    @GetMapping(value = "/buyingRequirmentByName/{name}")
    public Page<BuyingRequirement> getByName(@PathVariable String name,Pageable pageable){
        Page buyersByName =
iBuyingRequirementsRepository.findByNameContainingIgnoreCase(name,pageable);

        return buyersByName;
    }
}
```

Es beinhaltet eine Methode

1. Post-Methode, die Daten in die Datenbank schreibt.
2. Holen Sie sich eine Methode, die alle Datensätze aus der buyRequirement-Tabelle abruft.
3. Dies ist auch eine Get-Methode, die die Kaufanforderung anhand des Namens der Person findet.

Controller online lesen: https://riptutorial.com/de/spring-boot/topic/10635/controller

Kapitel 4: Eine Spring-Boot-Anwendung an MySQL anschließen

Einführung

Wir wissen, dass Spring-Boot standardmäßig mit der H2-Datenbank läuft. In diesem Artikel erfahren Sie, wie Sie die Standardkonfiguration an die MySQL-Datenbank anpassen können.

Bemerkungen

Stellen Sie als Voraussetzung sicher, dass MySQL bereits auf Port 3306 ausgeführt wird und Ihre Datenbank erstellt wurde.

Examples

Spring-Boot-Beispiel mit MySQL

Wir folgen dem offiziellen Leitfaden für Spring-Boot und Spring-Data-jpa . Wir werden die Anwendung mit Gradle erstellen.

1. **Erstellen Sie die Gradle Build-Datei**

   build.gradle

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.4.3.RELEASE")
    }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'org.springframework.boot'

jar {
    baseName = 'gs-accessing-data-jpa'
    version =  '0.1.0'
}

repositories {
    mavenCentral()
    maven { url "https://repository.jboss.org/nexus/content/repositories/releases" }
}

sourceCompatibility = 1.8
```

```
targetCompatibility = 1.8

dependencies {
    compile("org.springframework.boot:spring-boot-starter-data-jpa")
    runtime('mysql:mysql-connector-java')
    testCompile("junit:junit")
}
```

2. **Erstellen Sie die Kundenentität**

   src / main / java / hallo / Customer.java

```
@Entity
public class Customer {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;
    private String firstName;
    private String lastName;

    protected Customer() {}

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return String.format(
                "Customer[id=%d, firstName='%s', lastName='%s']",
                id, firstName, lastName);
    }
}
```

3. **Erstellen Sie Repositorys**

   src / main / java / hallo / CustomerRepository.java

```
import java.util.List;
import org.springframework.data.repository.CrudRepository;

public interface CustomerRepository extends CrudRepository<Customer, Long> {
    List<Customer> findByLastName(String lastName);
}
```

4. **Erstellen Sie die Datei application.properties**

```
################### DataSource Configuration ##########################
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/your_database_name
jdbc.username=username
jdbc.password=password

init-db=false
```

```
################### Hibernate Configuration ##########################

hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.show_sql=true
hibernate.hbm2ddl.auto=update
```

5. **Erstellen Sie die PersistenceConfig.java-Datei**

   In Schritt 5 definieren wir, wie die Datenquelle geladen wird und wie sich unsere Anwendung mit MySQL verbindet. Das obige Snippet ist die minimale Konfiguration, die wir benötigen, um eine Verbindung zu MySQL herzustellen. Hier stellen wir zwei Bohnen zur Verfügung:

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages="hello")
public class PersistenceConfig
{
    @Autowired
    private Environment env;

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory()
    {
        LocalContainerEntityManagerFactoryBean factory = new
LocalContainerEntityManagerFactoryBean();

        HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        vendorAdapter.setGenerateDdl(Boolean.TRUE);
        vendorAdapter.setShowSql(Boolean.TRUE);

        factory.setDataSource(dataSource());
        factory.setJpaVendorAdapter(vendorAdapter);
        factory.setPackagesToScan("hello");

        Properties jpaProperties = new Properties();
        jpaProperties.put("hibernate.hbm2ddl.auto",
env.getProperty("hibernate.hbm2ddl.auto"));
        factory.setJpaProperties(jpaProperties);

        factory.afterPropertiesSet();
        factory.setLoadTimeWeaver(new InstrumentationLoadTimeWeaver());
        return factory;
    }

    @Bean
    public DataSource dataSource()
    {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(env.getProperty("jdbc.driverClassName"));
        dataSource.setUrl(env.getProperty("jdbc.url"));
        dataSource.setUsername(env.getProperty("jdbc.username"));
        dataSource.setPassword(env.getProperty("jdbc.password"));
        return dataSource;
    }

}
```

- ***LocalContainerEntityManagerFactoryBean*** Dies gibt uns einen ***Überblick*** über die EntityManagerFactory-Konfigurationen und ermöglicht Anpassungen. Es erlaubt uns auch, den PersistenceContext in unsere Komponenten wie folgt einzubringen:

```
@PersistenceContext
private EntityManager em;
```

- ***DataSource*** Hier geben wir eine Instanz der `DriverManagerDataSource` . Es ist eine einfache Implementierung der Standard-JDBC-DataSource-Schnittstelle, die einen einfachen alten JDBC-Treiber über Bean-Eigenschaften konfiguriert und für jeden getConnection-Aufruf eine neue Verbindung zurückgibt. Beachten Sie, dass ich empfehle, dies strikt zu Testzwecken zu verwenden, da es bessere Alternativen wie `BasicDataSource` gibt. Weitere Einzelheiten finden Sie hier

6. **Erstellen Sie eine Anwendungsklasse**

   src / main / java / hallo / Anwendung.java

```
@SpringBootApplication
public class Application {

    private static final Logger log = LoggerFactory.getLogger(Application.class);

    @Autowired
    private CustomerRepository repository;

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

    @Bean
    public CommandLineRunner demo() {
        return (args) -> {
            // save a couple of customers
            repository.save(new Customer("Jack", "Bauer"));
            repository.save(new Customer("Chloe", "O'Brian"));
            repository.save(new Customer("Kim", "Bauer"));
            repository.save(new Customer("David", "Palmer"));
            repository.save(new Customer("Michelle", "Dessler"));

            // fetch all customers
            log.info("Customers found with findAll():");
            log.info("-------------------------------");
            for (Customer customer : repository.findAll()) {
                log.info(customer.toString());
            }
            log.info("");

            // fetch an individual customer by ID
            Customer customer = repository.findOne(1L);
            log.info("Customer found with findOne(1L):");
            log.info("--------------------------------");
            log.info(customer.toString());
            log.info("");

            // fetch customers by last name
```

```
            log.info("Customer found with findByLastName('Bauer'):");
            log.info("--------------------------------------------");
            for (Customer bauer : repository.findByLastName("Bauer")) {
                log.info(bauer.toString());
            }
            log.info("");
        };
    }
```

}

7. **Anwendung ausführen**

Wenn Sie eine IDE wie STS verwenden , können Sie einfach mit der rechten Maustaste auf Ihr Projekt klicken -> Ausführen als -> Gradle (STS) Build ... Geben Sie in der Aufgabenliste bootRun und Ausführen ein.

Wenn Sie Gradle in der Befehlszeile verwenden , können Sie die Anwendung einfach wie folgt ausführen:

```
./gradlew bootRun
```

Sie sollten so etwas sehen:

```
== Customers found with findAll():
Customer[id=1, firstName='Jack', lastName='Bauer']
Customer[id=2, firstName='Chloe', lastName='O'Brian']
Customer[id=3, firstName='Kim', lastName='Bauer']
Customer[id=4, firstName='David', lastName='Palmer']
Customer[id=5, firstName='Michelle', lastName='Dessler']

== Customer found with findOne(1L):
Customer[id=1, firstName='Jack', lastName='Bauer']

== Customer found with findByLastName('Bauer'):
Customer[id=1, firstName='Jack', lastName='Bauer']
Customer[id=3, firstName='Kim', lastName='Bauer']
```

Eine Spring-Boot-Anwendung an MySQL anschließen online lesen: https://riptutorial.com/de/spring-boot/topic/8588/eine-spring-boot-anwendung-an-mysql-anschlie-en

Kapitel 5: Erstellen und Verwenden mehrerer application.properties-Dateien

Examples

Dev- und Prod-Umgebung mit unterschiedlichen Datenquellen

Nach dem erfolgreichen Setup der Spring-Boot-Anwendung wird die gesamte Konfiguration in einer Datei application.properties behandelt. Sie finden die Datei unter `src/main/resources/` .

Normalerweise muss hinter der Anwendung eine Datenbank vorhanden sein. Für die Entwicklung seines guten einen Aufbau haben `dev` und einen `prod` Umgebungen. Mit mehreren `application.properties` Dateien können Sie Spring-Boot mitteilen, in welcher Umgebung die Anwendung gestartet werden soll.

Ein gutes Beispiel ist die Konfiguration von zwei Datenbanken. Eine für `dev` und eine für `productive` .

Für die `dev` können Sie eine In-Memory-Datenbank wie `H2` . Erstellen Sie eine neue Datei im Verzeichnis `src/main/resources/` Namen `application-dev.properties` . In der Datei befindet sich die Konfiguration der In-Memory-Datenbank:

```
spring.datasource.url=jdbc:h2:mem:test
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```

Für die `prod` Umgebung stellen wir eine Verbindung zu einer "echten" Datenbank her, zum Beispiel `postgreSQL` . Erstellen Sie eine neue Datei im Verzeichnis `src/main/resources/` Namen `application-prod.properties` . In der Datei befindet sich die Konfiguration der `postgreSQL` Datenbank:

```
spring.datasource.url= jdbc:postgresql://localhost:5432/yourDB
spring.datasource.username=postgres
spring.datasource.password=secret
```

In Ihrer Standarddatei `application.properties` Sie jetzt festlegen, welches Profil von Spring-Boot aktiviert und verwendet wird. Setzen Sie einfach ein Attribut in:

```
spring.profiles.active=dev
```

oder

```
spring.profiles.active=prod
```

Wichtig ist, dass der Teil nach `-` in `application-dev.properties` der Bezeichner der Datei ist.

Jetzt können Sie die Spring-Boot-Anwendung im Entwicklungs- oder Produktionsmodus starten, indem Sie einfach die Kennung ändern. Eine In-Memory-Datenbank wird gestartet oder die Verbindung zu einer "echten" Datenbank hergestellt. Natürlich gibt es auch viel mehr Anwendungsfälle, um mehrere Eigenschaftsdateien zu haben.

Stellen Sie das richtige Federprofil ein, indem Sie die Anwendung automatisch erstellen (maven).

Durch das Erstellen mehrerer Eigenschaftendateien für die verschiedenen Umgebungen oder Anwendungsfälle ist es manchmal schwierig, den Wert von `active.profile` manuell auf den `active.profile` Wert zu ändern. Aber es gibt einen Weg , um das setzen `active.profile` in der `application.properties` Datei , während die Anwendung des Aufbau unter Verwendung von `maven-profiles` .

Nehmen wir an, in unserer Anwendung gibt es drei Umgebungs-Eigenschaftsdateien:

`application-dev.properties`:

```
spring.profiles.active=dev
server.port=8081
```

`application-test.properties`:

```
spring.profiles.active=test
server.port=8082
```

`application-prod.properties`.

```
spring.profiles.active=prod
server.port=8083
```

Diese drei Dateien unterscheiden sich lediglich im Port- und im aktiven Profilnamen.

In der Hauptdatei `application.properties` legen wir unser Federprofil mithilfe einer Maven-Variablen fest :

`application.properties`.

```
spring.profiles.active=@profileActive@
```

Danach müssen wir nur die Maven-Profile in unserer `pom.xml` hinzufügen. Wir werden Profile für alle drei Umgebungen festlegen:

```
 <profiles>
        <profile>
            <id>dev</id>
            <activation>
                <activeByDefault>true</activeByDefault>
            </activation>
```

```
            <properties>
                <build.profile.id>dev</build.profile.id>
                <profileActive>dev</profileActive>
            </properties>
        </profile>
        <profile>
            <id>test</id>
            <properties>
                <build.profile.id>test</build.profile.id>
                <profileActive>test</profileActive>
            </properties>
        </profile>
        <profile>
            <id>prod</id>
            <properties>
                <build.profile.id>prod</build.profile.id>
                <profileActive>prod</profileActive>
            </properties>
        </profile>
    </profiles>
```

Sie können jetzt die Anwendung mit Maven erstellen. Wenn Sie kein Maven-Profil festlegen, wird das Standardprofil erstellt (in diesem Beispiel ist es dev). Für eine Angabe müssen Sie ein Maven-Schlüsselwort verwenden. Das Schlüsselwort zum Einrichten eines Profils in maven lautet `-P` direkt gefolgt vom Namen des Profils: `mvn clean install -Ptest` .

Jetzt können Sie auch benutzerdefinierte Builds erstellen und diese für schnellere Builds in Ihrer `IDE` speichern.

Beispiele:

```
mvn clean install -Ptest
```

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.3.RELEASE)

2017-06-06 11:24:44.885  INFO 6328 --- [           main] com.demo.SpringBlobApplicationTests
: Starting SpringApplicationTests on KB242 with PID 6328 (started by me in
C:\DATA\Workspaces\spring-demo)
2017-06-06 11:24:44.886  INFO 6328 --- [           main] com.demo.SpringApplicationTests
: The following profiles are active: test
```

```
mvn clean install -Pprod
```

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.3.RELEASE)
```

```
2017-06-06 14:43:31.067  INFO 6932 --- [           main] com.demo.SpringBlobApplicationTests
: Starting SpringApplicationTests on KB242 with PID 6328 (started by me in
C:\DATA\Workspaces\spring-demo)
2017-06-06 14:43:31.069  INFO 6932 --- [           main] com.demo.SpringApplicationTests
: The following profiles are active: prod
```

Erstellen und Verwenden mehrerer application.properties-Dateien online lesen: https://riptutorial.com/de/spring-boot/topic/6334/erstellen-und-verwenden-mehrerer-application-properties-dateien

Kapitel 6: Federstiefel + Spring Data JPA

Einführung

Spring Boot erleichtert die Erstellung von Spring-betriebenen Anwendungen und Services in Produktionsqualität mit minimalem Aufwand. Es bevorzugt Konvention gegenüber Konfiguration.

Spring Data JPA , Teil der größeren Spring Data- Familie, vereinfacht die Implementierung von JPA-basierten Repositories. Es erleichtert das Erstellen von Apps, die Datenzugriffstechnologien verwenden.

Bemerkungen

Anmerkungen

`@Repository` : Gibt an, dass eine kommentierte Klasse ein "Repository" ist, ein Mechanismus zum Einkapseln von Speicher-, Abruf- und Suchverhalten, mit dem eine Sammlung von Objekten emuliert wird. Teams, die traditionelle J2EE-Muster wie "Data Access Object" implementieren, können diesen Stereotyp auch auf DAO-Klassen anwenden. Bevor Sie dies tun, sollten Sie jedoch den Unterschied zwischen Data Access Object- und DDD-Repositorys verstehen. Diese Anmerkung ist ein allgemeines Stereotyp, und einzelne Teams können ihre Semantik einschränken und entsprechend verwenden.

`@RestController` : Eine bequeme Anmerkung , die sich mit Anmerkungen versehen ist `@Controller` und `@ResponseBody.Types` , die diese Anmerkung tragen , werden als Controller behandelt , wo `@RequestMapping` Methoden übernehmen `@ResponseBody` Semantik standardmäßig.

`@Service` : Gibt an, dass eine kommentierte Klasse ein "Service" ist (z. B. eine Business-Service-Fassade). Diese Annotation dient als Spezialisierung von `@Component` , sodass Implementierungsklassen durch Klassenpfad-Scanning automatisch erkannt werden können.

`@SpringBootApplication` : Viele Spring Boot-Entwickler haben ihre Hauptklasse immer mit `@Configuration` , `@EnableAutoConfiguration` und `@ComponentScan` . Da diese Anmerkungen so häufig verwendet werden (insbesondere wenn Sie die oben genannten bewährten Methoden befolgen), bietet Spring Boot eine praktische `@SpringBootApplication` Alternative.

`@Entity` : Gibt an, dass die Klasse eine Entität ist. Diese Anmerkung wird auf die Entitätsklasse angewendet.

Offizielle Dokumentation

Pivotal Software hat eine ziemlich ausführliche Dokumentation zu Spring Framework zur Verfügung gestellt

- https://projects.spring.io/spring-boot/
- http://projects.spring.io/spring-data-jpa/
- https://spring.io/guides/gs/accessing-data-jpa/

Examples

Beispiel für die Integration von Spring Boot und Spring Data JPA

Wir werden eine Anwendung erstellen, die POJOs in einer Datenbank speichert. Die Anwendung verwendet Spring Data JPA zum Speichern und Abrufen von Daten in einer relationalen Datenbank. Die überzeugendste Funktion ist die Möglichkeit, Repository-Implementierungen automatisch zur Laufzeit über eine Repository-Schnittstelle zu erstellen.

Hauptklasse

```
package org.springboot;

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

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

Die `main()` -Methode verwendet die `SpringApplication.run()` -Methode von Spring Boot, um eine Anwendung zu starten. Bitte beachten Sie, dass es keine einzige XML-Zeile gibt. Auch keine web.xml-Datei. Diese Webanwendung besteht zu 100% aus reinem Java und Sie müssen sich nicht mit der Konfiguration von Rohrleitungen oder Infrastrukturen beschäftigen.

Entity-Klasse

```
package org.springboot.model;

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

@Entity
public class Greeting {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String text;

    public Greeting() {
        super();
```

```
    }

    public Greeting(String text) {
        super();
        this.text = text;
    }

    /* In this example, the typical getters and setters have been left out for brevity. */
}
```

Hier haben Sie eine `Greeting` mit zwei Attributen, der `id` und dem `text` . Sie haben auch zwei Konstruktoren. Der Standardkonstruktor existiert nur aus Gründen der JPA. Sie werden es nicht direkt verwenden, daher kann es als `protected` . Der andere Konstruktor ist derjenige, mit dem Sie `Greeting` erstellen, die in der Datenbank gespeichert werden.

Die `Greeting` Klasse wird mit `@Entity` annotiert, um `@Entity` , dass es sich um eine JPA-Entität handelt. `@Table` keine `@Table` Annotation vorhanden ist, wird davon ausgegangen, dass diese Entität einer Tabelle mit dem Namen 'Greeting' zugeordnet wird.

Die `id` `@Id` Begrüßung wird mit `@Id` kommentiert, sodass JPA sie als ID des Objekts erkennt. Die `id` `@GeneratedValue` wird auch mit `@GeneratedValue` kommentiert, um `@GeneratedValue` , dass die ID automatisch generiert werden soll.

Die andere Eigenschaft, `text` wird nicht kommentiert. Es wird davon ausgegangen, dass es einer Spalte zugeordnet wird, die denselben Namen wie die Eigenschaft selbst hat.

Transiente Eigenschaften

In einer Entitätsklasse, die der obigen ähnelt, können wir Eigenschaften haben, die wir nicht in der Datenbank persistieren oder als Spalten in unserer Datenbank erstellen möchten, vielleicht weil wir sie nur zur Laufzeit setzen und in unserer Anwendung verwenden möchten. Daher können wir diese Eigenschaft mit der Annotation @Transient versehen.

```
package org.springboot.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Transient;

@Entity
public class Greeting {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String text;
    @Transient
    private String textInSomeLanguage;

    public Greeting() {
        super();
```

```
    }

    public Greeting(String text) {
        super();
        this.text = text;
        this.textInSomeLanguage = getTextTranslationInSpecifiedLanguage(text);
    }

    /* In this example, the typical getters and setters have been left out for brevity. */
}
```

Hier haben Sie die gleiche Greeting-Klasse, die jetzt eine vorübergehende Eigenschaft `textInSomeLanguage` , die zur Laufzeit initialisiert und verwendet werden kann und nicht in der Datenbank gespeichert bleibt.

DAO-Klasse

```
package org.springboot.repository;

import org.springboot.model.Greeting;
import org.springframework.data.repository.CrudRepository;

public interface GreetingRepository extends CrudRepository<Greeting, Long> {

    List<Greeting> findByText(String text);
}
```

`GreetingRepository` erweitert die `CrudRepository` Schnittstelle. Die Art der Entität und die ID, mit der sie arbeitet, `Greeting` und `Long` , werden in den allgemeinen Parametern in `CrudRepository` . Durch die Erweiterung von `CrudRepository` erbt `GreetingRepository` verschiedene Methoden für die Arbeit mit der Beständigkeit der `Greeting` , einschließlich Methoden zum Speichern, Löschen und Suchen von `Greeting` .
Sehen Sie sich diese Diskussion zum Vergleich von `CrudRepository` , `PagingAndSortingRepository` und `JpaRepository` .

Mit Spring Data JPA können Sie auch andere Abfragemethoden definieren, indem Sie deren Methodensignatur einfach deklarieren. Im Fall von `GreetingRepository` wird dies mit einer `findByText()` -Methode `findByText()` .

In einer typischen Java-Anwendung würden Sie erwarten, eine Klasse zu schreiben, die `GreetingRepository` implementiert. Aber genau das macht Spring Data JPA so leistungsfähig: Sie müssen keine Implementierung der Repository-Schnittstelle schreiben. Spring Data JPA erstellt eine Implementierung, wenn Sie die Anwendung ausführen.

Serviceklasse

```
package org.springboot.service;

import java.util.Collection
import org.springboot.model.Greeting;
```

```
public interface GreetingService {

    Collection<Greeting> findAll();
    Greeting findOne(Long id);
    Greeting create(Greeting greeting);
    Greeting update(Greeting greeting);
    void delete(Long id);

}
```

Service Bean

```
package org.springboot.service;

import java.util.Collection;
import org.springboot.model.Greeting;
import org.springboot.repository.GreetingRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class GreetingServiceBean implements GreetingService {

    @Autowired
    private GreetingRepository greetingRepository;

    @Override
    public Collection<Greeting> findAll() {
        Collection<Greeting> greetings = greetingRepository.findAll();
        return greetings;
    }

    @Override
    public Greeting findOne(Long id) {
        Greeting greeting = greetingRepository.findOne(id);
        return greeting;
    }

    @Override
    public Greeting create(Greeting greeting) {
        if (greeting.getId() != null) {
            //cannot create Greeting with specified Id value
            return null;
        }
        Greeting savedGreeting = greetingRepository.save(greeting);
        return savedGreeting;
    }

    @Override
    public Greeting update(Greeting greeting) {
        Greeting greetingPersisted = findOne(greeting.getId());
        if (greetingPersisted == null) {
            //cannot find Greeting with specified Id value
            return null;
        }
        Greeting updatedGreeting = greetingRepository.save(greeting);
        return updatedGreeting;
    }
```

```
    @Override
    public void delete(Long id) {
         greetingRepository.delete(id);
    }

}
```

Controller-Klasse

```
package org.springboot.web.api;

import java.util.Collection;
import org.springboot.model.Greeting;
import org.springboot.service.GreetingService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(value = "/api")
public class GreetingController {

    @Autowired
    private GreetingService greetingService;

    // GET [method = RequestMethod.GET] is a default method for any request.
    // So we do not need to mention explicitly

    @RequestMapping(value = "/greetings", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Collection<Greeting>> getGreetings() {
        Collection<Greeting> greetings = greetingService.findAll();
        return new ResponseEntity<Collection<Greeting>>(greetings, HttpStatus.OK);
    }

    @RequestMapping(value = "/greetings/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Greeting> getGreeting(@PathVariable("id") Long id) {
        Greeting greeting = greetingService.findOne(id);
        if(greeting == null) {
            return new ResponseEntity<Greeting>(HttpStatus.NOT_FOUND);
        }
        return new ResponseEntity<Greeting>(greeting, HttpStatus.OK);
    }

    @RequestMapping(value = "/greetings", method = RequestMethod.POST, consumes =
MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Greeting> createGreeting(@RequestBody Greeting greeting) {
        Greeting savedGreeting = greetingService.create(greeting);
        return new ResponseEntity<Greeting>(savedGreeting, HttpStatus.CREATED);
    }

    @RequestMapping(value = "/greetings/{id}", method = RequestMethod.PUT, consumes =
MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<Greeting> updateGreeting(@PathVariable("id") Long id, @RequestBody
```

```
Greeting greeting) {
        Greeting updatedGreeting = null;
        if (greeting != null && id == greeting.getId()) {
            updatedGreeting = greetingService.update(greeting);
        }
        if(updatedGreeting == null) {
            return new ResponseEntity<Greeting>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
        return new ResponseEntity<Greeting>(updatedGreeting, HttpStatus.OK);
    }

    @RequestMapping(value = "/greetings/{id}", method = RequestMethod.DELETE)
    public ResponseEntity<Greeting> deleteGreeting(@PathVariable("id") Long id) {
        greetingService.delete(id);
        return new ResponseEntity<Greeting>(HttpStatus.NO_CONTENT);
    }

}
```

Anwendungseigenschaftendatei für MySQL-Datenbank

```
#mysql config
spring.datasource.url=jdbc:mysql://localhost:3306/springboot
spring.datasource.username=root
spring.datasource.password=Welcome@123
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto = update

spring.jpa.hibernate.naming-strategy=org.hibernate.cfg.DefaultNamingStrategy

#initialization
spring.datasource.schema=classpath:/data/schema.sql
```

SQL-Datei

```
drop table if exists greeting;
create table greeting (
    id bigint not null auto_increment,
    text varchar(100) not null,
    primary key(id)
);
```

pom.xml-Datei

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org</groupId>
```

```
<artifactId>springboot</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>war</packaging>
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.1.RELEASE</version>
</parent>

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

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

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>

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

Erstellen eines ausführbaren JAR

Sie können die Anwendung über die Befehlszeile mit Maven ausführen. Sie können auch eine einzelne ausführbare JAR-Datei erstellen, die alle erforderlichen Abhängigkeiten, Klassen und Ressourcen enthält, und diese ausführen. Dies macht es einfach, den Service als Anwendung während des gesamten Entwicklungszyklus, in verschiedenen Umgebungen usw. zu liefern, zu implementieren und bereitzustellen.

Führen Sie die Anwendung mit `./mvnw spring-boot:run` . Oder Sie können die JAR-Datei mit `./mvnw clean package` . Dann können Sie die JAR-Datei ausführen:

```
java -jar target/springboot-0.0.1-SNAPSHOT.jar
```

Federstiefel + Spring Data JPA online lesen: https://riptutorial.com/de/spring-boot/topic/6203/federstiefel-plus-spring-data-jpa

Kapitel 7: Installieren der Spring Boot-CLI

Einführung

Mit der Spring Boot-CLI können Sie Spring Boot-Anwendungen einfach über die Befehlszeile erstellen und mit ihnen arbeiten.

Bemerkungen

Nach der Installation kann die Spring Boot-CLI mit dem `spring` ausgeführt werden:

So erhalten Sie eine Befehlszeilenhilfe:

```
$ spring help
```

So erstellen und führen Sie Ihr erstes Spring Boot-Projekt aus:

```
$ spring init my-app
$ cd my-app
$ spring run my-app
```

Öffnen Sie Ihren Browser für `localhost:8080` :

```
$ open http://localhost:8080
```

Sie erhalten die Seite mit der Whitelabel-Fehlermeldung, da Sie Ihrer Anwendung noch keine Ressourcen hinzugefügt haben. Sie können jedoch nur die folgenden Dateien verwenden:

```
my-app/
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src/
    ├── main/
    │   ├── java/
    │   │   └── com/
    │   │       └── example/
    │   │           └── DemoApplication.java
    │   └── resources/
    │       └── application.properties
    └── test/
        └── java/
            └── com/
                └── example/
                    └── DemoApplicationTests.java
```

- `mvnw` und `mvnw.cmd` - Maven-Wrapper-Skripts, die Maven bei der ersten Verwendung (falls erforderlich) herunterladen und installieren.
- `pom.xml` - Die Maven-Projektdefinition

- `DemoApplication.java` - die Hauptklasse, mit der Ihre Spring Boot-Anwendung `DemoApplication.java` .
- `application.properties` - Eine Datei für externalisierte Konfigurationseigenschaften. (Kann auch die Erweiterung `.yml` .)
- `DemoApplicationTests.java` - Ein `DemoApplicationTests.java` , der die Initialisierung des Spring Boot-Anwendungskontexts überprüft.

Examples

Manuelle Installation

Auf der Download-Seite können Sie die neueste Version manuell herunterladen und entpacken.

- spring-boot-cli-1.5.1.RELEASE-bin.zip
- spring-boot-cli-1.5.1.RELEASE-bin.tar.gz

Installieren Sie unter Mac OSX mit HomeBrew

```
$ brew tap pivotal/tap
$ brew install springboot
```

Installieren Sie unter Mac OSX mit MacPorts

```
$ sudo port install spring-boot-cli
```

Auf jedem Betriebssystem mit SDKMAN installieren!

SDKMAN! ist der Software Development Kit Manager für Java. Es kann zum Installieren und Verwalten von Versionen der Spring Boot-CLI sowie von Java, Maven, Gradle usw. verwendet werden.

```
$ sdk install springboot
```

Installieren der Spring Boot-CLI online lesen: https://riptutorial.com/de/spring-boot/topic/9031/installieren-der-spring-boot-cli

Kapitel 8: REST-Dienste

Parameter

Anmerkung	Säule
@Regler	Gibt an, dass eine kommentierte Klasse ein "Controller" (Webcontroller) ist.
@RequestMapping	Anmerkung zum Mapping von Webanforderungen auf bestimmte Handler-Klassen (wenn wir mit class verwendet wurden) und / oder Handler-Methoden (wenn wir mit Methoden verwendet wurden)
method = RequestMethod.GET	Art der HTTP-Anforderungsmethoden
ResponseBody	Anmerkungen, die angeben, dass ein Methodenrückgabewert angezeigt wird, sollten an den Webantworthauptteil gebunden werden
@RestController	@Controller + ResponseBody
@ResponseEntity	Die Erweiterung von HttpEntity, die einen HttpStatus-Statuscode hinzufügt, kann den zurückgegebenen http-Code steuern

Examples

Erstellen eines REST-Service

1. Erstellen Sie ein Projekt mit STS (Spring Starter Project) oder Spring Initializr (unter https://start.spring.io).
2. Fügen Sie eine Webabhängigkeit in Ihrer pom.xml hinzu:

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

oder geben Sie **web** in das Suchfeld Nach `Search for dependencies` suchen ein, fügen Sie eine Webabhängigkeit hinzu und laden Sie ein gezipptes Projekt herunter.

3. Erstellen Sie eine Domänenklasse (z. B. Benutzer)

```
public class User {

       private Long id;
```

```
    private String userName;

    private String password;

    private String email;

    private String firstName;

    private String lastName;

    public Long getId() {
        return id;
    }

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

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getEmail() {
        return email;
    }

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

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @Override
    public String toString() {
```

```
            return "User [id=" + id + ", userName=" + userName + ", password=" + password + ",
email=" + email
                    + ", firstName=" + firstName + ", lastName=" + lastName + "]";
        }

        public User(Long id, String userName, String password, String email, String firstName,
String lastName) {
            super();
            this.id = id;
            this.userName = userName;
            this.password = password;
            this.email = email;
            this.firstName = firstName;
            this.lastName = lastName;
        }

        public User() {}
    }
```

4. Erstellen Sie die UserController-Klasse und fügen Sie `@Controller` , `@RequestMapping` Anmerkungen hinzu

```
    @Controller
    @RequestMapping(value = "api")
    public class UserController {
    }
```

5. Definieren Sie die statische Liste Benutzervariable, um die Datenbank zu simulieren, und fügen Sie der Liste zwei Benutzer hinzu

```
    private static List<User> users = new ArrayList<User>();

    public UserController() {
        User u1 = new User(1L, "shijazi", "password", "shijazi88@gmail.com", "Safwan",
"Hijazi");
        User u2 = new User(2L, "test", "password", "test@gmail.com", "test", "test");
        users.add(u1);
        users.add(u2);
    }
```

6. Neue Methode erstellen, um alle Benutzer in der statischen Liste zurückzugeben (getAllUsers)

```
    @RequestMapping(value = "users", method = RequestMethod.GET)
    public @ResponseBody List<User> getAllUsers() {
         return users;
    }
```

7. Führen Sie die Anwendung [von `mvn clean install spring-boot:run`] aus und rufen Sie diese URL `http://localhost:8080/api/users`

8. Wir können die Klasse mit Anmerkungen versehen `@RestController` , und in diesem Fall können wir die Response von allen Methoden in dieser Klasse entfernen `(@RestController = @Controller + ResponseBody)` , ein weiterer Punkt wir die Rückkehr http Code steuern können ,

wenn wir verwenden `ResponseEntity` , Wir werden dieselben vorherigen Funktionen implementieren, jedoch `@RestController` und `ResponseEntity`

```
@RestController
@RequestMapping(value = "api2")
public class UserController2 {

    private static List<User> users = new ArrayList<User>();

    public UserController2() {
        User u1 = new User(1L, "shijazi", "password", "shijazi88@gmail.com", "Safwan",
"Hijazi");
        User u2 = new User(2L, "test", "password", "test@gmail.com", "test", "test");
        users.add(u1);
        users.add(u2);
    }

    @RequestMapping(value = "users", method = RequestMethod.GET)
    public ResponseEntity<?> getAllUsers() {
       try {
           return new ResponseEntity<>(users, HttpStatus.OK);
       } catch (Exception e) {
           return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
       }
    }
}
```

Versuchen Sie jetzt, die Anwendung auszuführen, und rufen Sie diese URL http: // localhost: 8080 / api2 / users auf

Erstellen eines Ruhedienstes mit JERSEY und Spring Boot

Jersey ist eines der vielen verfügbaren Frameworks zum Erstellen von Rest Services. In diesem Beispiel wird beschrieben, wie Rest Services mit Jersey und Spring Boot erstellt werden

1.Projekt-Setup

Sie können ein neues Projekt mit STS oder über die Seite Spring Initializr erstellen. Berücksichtigen Sie beim Erstellen eines Projekts die folgenden Abhängigkeiten:

1. Trikot (JAX-RS)
2. Netz

2. Controller erstellen

Lassen Sie uns einen Controller für unseren Jersey Web Service erstellen

```
@Path("/Welcome")
@Component
public class MyController {
    @GET
```

```
    public String welcomeUser(@QueryParam("user") String user){
        return "Welcome "+user;
    }
}
```

`@Path("/Welcome")` Annotation `@Path("/Welcome")` gibt dem Framework an, dass dieser Controller auf den URI-Pfad / Welcome antworten soll

`@QueryParam("user")` Anmerkung zeigt den Rahmen , dass wir einen Abfrageparameter mit dem Namen erwarten `user`

3.Wiring Jersey-Konfigurationen

Lassen Sie uns nun Jersey Framework mit Spring Boot konfigurieren: Erstellen Sie eine Klasse, eher eine `org.glassfish.jersey.server.ResourceConfig` die `org.glassfish.jersey.server.ResourceConfig` :

```
@Component
@ApplicationPath("/MyRestService")
public class JerseyConfig extends ResourceConfig {
    /**
     * Register all the Controller classes in this method
     * to be available for jersey framework
     */
    public JerseyConfig() {
        register(MyController.class);
    }

}
```

`@ApplicationPath("/MyRestService")` gibt dem Framework an, dass nur Anforderungen, die an den Pfad `/MyRestService` sind, vom Jersey-Framework verarbeitet werden sollen, andere Anforderungen sollten weiterhin vom Spring-Framework verarbeitet werden.

Es ist eine gute Idee, die Konfigurationsklasse mit `@ApplicationPath` zu kommentieren, andernfalls werden alle Anforderungen von Jersey bearbeitet, und wir können sie nicht umgehen und können bei Bedarf einen Feder-Controller verwenden.

4.Done

Starten Sie die Anwendung und lösen Sie eine Beispiel-URL aus (Angenommen, Sie haben Spring Boot für die Ausführung auf Port 8080 konfiguriert):

`http://localhost:8080/MyRestService/Welcome?user=User`

Sie sollten eine Nachricht in Ihrem Browser sehen:

Willkommener Benutzer

Und Sie sind fertig mit Ihrem Jersey-Webdienst mit Spring Boot

REST-API mit RestTemplate (GET) verwenden

Um einen REST - API mit verbrauchen `RestTemplate` , ein Frühlings - Boot - Projekt mit dem Spring - Boot initialzr zu schaffen und sicherzustellen , dass die **Web** - Abhängigkeit angefügt:

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

Wenn Sie Ihr Projekt eingerichtet haben , erstellen Sie eine `RestTemplate` Bean. Dies kann in der bereits generierten Hauptklasse oder in einer separaten Konfigurationsklasse (einer mit `@Configuration` annotierten `@Configuration`) erfolgen:

```
@Bean
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

Erstellen Sie anschließend eine Domänenklasse, ähnlich wie Sie beim Erstellen eines REST-Diensts vorgehen müssen .

```
public class User {
    private Long id;
    private String username;
    private String firstname;
    private String lastname;

    public Long getId() {
        return id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
```

```
        this.lastname = lastname;
    }
}
```

Aktivieren `RestTemplate` in Ihrem Client die RestTemplate `RestTemplate` :

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

Um eine REST-API zu verwenden, die einen einzelnen Benutzer zurückgibt, können Sie jetzt Folgendes verwenden:

```
String url = "http://example.org/path/to/api";
User response = restTemplate.getForObject(url, User.class);
```

Wenn Sie eine REST-API verwenden, die eine Liste oder ein Array von Benutzern zurückgibt, haben Sie zwei Möglichkeiten. Entweder verbrauchen Sie es als Array:

```
String url = "http://example.org/path/to/api";
User[] response = restTemplate.getForObject(url, User[].class);
```

Oder verbrauchen Sie es mit der `ParameterizedTypeReference` :

```
String url = "http://example.org/path/to/api";
ResponseEntity<List<User>> response = restTemplate.exchange(url, HttpMethod.GET, null, new
ParameterizedTypeReference<List<User>>() {});
List<User> data = response.getBody();
```

`RestTemplate.exchange()` Sie, dass Sie bei der Verwendung von `ParameterizedTypeReference` die erweiterte `RestTemplate.exchange()` Methode verwenden und eine Unterklasse davon erstellen müssen. Im obigen Beispiel wird eine anonyme Klasse verwendet.

REST-Dienste online lesen: https://riptutorial.com/de/spring-boot/topic/1920/rest-dienste

Kapitel 9: Scannen von Paketen

Einführung

In diesem Thema werde ich einen Überblick über das Scannen von Frühlingsbootpaketen geben.

Einige grundlegende Informationen finden Sie in Spring Boot-Dokumenten unter folgendem Link (using-boot-structuring-your-code). Ich werde jedoch versuchen, genauere Informationen bereitzustellen.

Spring Boot und Spring im Allgemeinen bieten eine Funktion zum automatischen Durchsuchen von Paketen nach bestimmten Anmerkungen, um `beans` und `configuration` zu erstellen.

Parameter

Anmerkung	Einzelheiten
@SpringBootApplication	Anmerkung zur Hauptfederboot-Anwendung. einmal in der Anwendung verwendet, enthält eine Hauptmethode und dient als Hauptpaket für das Scannen von Paketen
@SpringBootConfiguration	Gibt an, dass eine Klasse eine Spring Boot-Anwendung bereitstellt. Sollte nur einmal in der Anwendung deklariert werden, normalerweise automatisch durch Setzen von `@SpringBootApplication`
@EnableAutoConfiguration	Aktivieren Sie die automatische Konfiguration des Spring-Anwendungskontexts. Sollte nur einmal in der Anwendung deklariert werden, normalerweise automatisch durch Setzen von `@SpringBootApplication`
@ComponentScan	Wird verwendet, um die automatische Paketsuche für ein bestimmtes Paket und dessen untergeordnete Pakete auszulösen oder um eine benutzerdefinierte Paketsuche festzulegen
@Aufbau	Wird verwendet, um eine oder mehrere `@Bean` Methoden zu deklarieren. Kann durch automatisches Scannen von Paketen ausgewählt werden, um eine oder mehrere `@Bean` Methoden anstelle der herkömmlichen XML-Konfiguration zu deklarieren
@Bohne	Gibt an, dass eine Methode eine Bean erzeugt, die vom Spring-Container verwaltet wird. Normalerweise werden mit `@Bean` versehene `@Bean` Methoden in mit `@Configuration` versehenen `@Configuration` Klassen abgelegt, die von `@Configuration` zur Erstellung von Java-Konfigurations-Beans ausgewählt werden.

Anmerkung	Einzelheiten
@Komponente	Durch das Deklarieren einer Klasse als `@Component` sie zu Kandidaten für die automatische Erkennung, wenn auf Annotation basierende Konfigurations- und Klassenpfad-Scans verwendet werden. Normalerweise wird eine mit `@Component` annotierte Klasse in der Anwendung zu einer `bean`
@Repository	Ursprünglich definiert von Domain-Driven Design (Evans, 2003) als „ein Mechanismus zur Speicherung eingekapselt wird . Es wird verwendet , um ein usualy , um anzuzeigen , `Repository` für `spring data`
@Bedienung	In der Praxis sehr ähnlich zu `@Component` . Ursprünglich von Domain-Driven Design (Evans, 2003) definiert als "eine Operation, die als Schnittstelle angeboten wird, die im Modell allein steht und keinen gekapselten Zustand hat".
@Regler	Gibt an, dass eine kommentierte Klasse ein "Controller" ist (z. B. ein Webcontroller).
@RestController	Eine praktische Annotation, die selbst mit `@Controller` und `@ResponseBody` kommentiert `@ResponseBody` . Wird standardmäßig automatisch ausgewählt, da sie die `@Controller` Anmerkung enthält, die standardmäßig ausgewählt ist.

Examples

@SpringBootApplication

Die einfachste Methode, Ihren Code mithilfe von Spring Boot für eine gute automatische `@SpringBootApplication` zu strukturieren, ist die Verwendung der Annotation `@SpringBootApplication` . Diese Anmerkung enthält drei weitere Anmerkungen, die beim automatischen Scannen `@SpringBootConfiguration` sind: `@SpringBootConfiguration` , `@EnableAutoConfiguration` , `@ComponentScan` (weitere Informationen zu jeder Anmerkung im Abschnitt `Parameters`).

`@SpringBootApplication` wird `@SpringBootApplication` im Hauptpaket abgelegt und alle anderen Komponenten werden in Paketen unter dieser Datei abgelegt:

```
com
 +- example
     +- myproject
         +- Application.java (annotated with @SpringBootApplication)
         |
         +- domain
         |   +- Customer.java
         |   +- CustomerRepository.java
         |
         +- service
         |   +- CustomerService.java
```

```
        |
        +- web
            +- CustomerController.java
```

Wenn nicht anders angegeben, erkennt Spring Boot `@Configuration` , `@Component` , `@Repository` , `@Service` , `@Controller` , `@RestController` Anmerkungen automatisch unter den gescannten Paketen (`@Configuration` und `@RestController` werden ausgewählt, da sie von `@Component` und `@Controller` entsprechend `@Controller` werden).

Basiscode-Beispiel:

```
@SpringBootApplication
public class Application {

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

}
```

Pakete / Klassen explizit setzen

Seit Version **1.3** können Sie Spring Boot auch zum Scannen bestimmter Pakete `scanBasePackages` indem Sie `scanBasePackages` oder `scanBasePackageClasses` in `@SpringBootApplication` anstatt `@ComponentScan` anzugeben.

1. `@SpringBootApplication(scanBasePackages = "com.example.myproject")` - `com.example.myproject` als zu scannendes Basispaket fest.
2. `@SpringBootApplication(scanBasePackageClasses = CustomerController.class)` - Die typsichere Alternative zu `scanBasePackages` legt das Paket `CustomerController.java` (`com.example.myproject.web`) als zu scannendes Basispaket fest.

Auto-Konfiguration ausschließen

Eine weitere wichtige Funktion ist die Möglichkeit, bestimmte Klassen der automatischen Konfiguration mithilfe von `excludeName` oder `excludeName exclude` (`excludeName` existiert seit Version **1.3**).

1. `@SpringBootApplication(exclude = DemoConfiguration.class)` - schließt `DemoConfiguration` von der automatischen `DemoConfiguration` aus.
2. `@SpringBootApplication(excludeName = "DemoConfiguration")` - tut dies unter Verwendung des vollständig klassifizierten Namens der Klasse.

@ComponentScan

Sie können `@ComponentScan` um komplexere `@ComponentScan` zu konfigurieren. Es gibt auch `@ComponentScans` , die als Container-Annotation fungieren, die mehrere `@ComponentScan` Annotationen zusammenfasst.

Grundlegende Codebeispiele

```
@ComponentScan
public class DemoAutoConfiguration {
}
```

```
@ComponentScans({@ComponentScan("com.example1"), @ComponentScan("com.example2")})
public class DemoAutoConfiguration {
}
```

Die Angabe von `@ComponentScan` ohne Konfiguration `@SpringBootApplication` wie `@SpringBootApplication` und durchsucht alle Pakete unter der mit dieser Annotation annotierten Klasse.

In diesem Beispiel werde ich einige nützliche Attribute von `@ComponentScan` :

1. **basePackages** - kann verwendet werden, um bestimmte zu **durchsuchende** Pakete **anzugeben** .
2. **useDefaultFilters** - Durch Setzen dieses Attributs auf false (Standardwerte true) können Sie sicherstellen, dass spring `@Component` , `@Repository` , `@Service` oder `@Controller` automatisch `@Controller` .
3. **includeFilters** - kann verwendet werden, *um* bestimmte **Federanmerkungen** / Regex-Muster für die **Paketsuche einzubeziehen** .
4. **excludeFilters** - kann verwendet werden, *um* bestimmte **Federanmerkungen** / Regex-Muster für die **Paketsuche** *auszuschließen* .

Es gibt viele weitere Attribute, aber diese werden am häufigsten verwendet, um das Scannen von Paketen anzupassen.

Erstellen Sie Ihre eigene Autokonfiguration

Spring Boot basiert auf vielen vorgefertigten übergeordneten Projekten zur automatischen Konfiguration. Sie sollten bereits mit Spring Boot Starter-Projekten vertraut sein.

Sie können leicht ein eigenes Starterprojekt erstellen, indem Sie die folgenden einfachen Schritte ausführen:

1. Erstellen Sie einige `@Configuration` Klassen, um Standard-Beans zu definieren. Sie sollten externe Eigenschaften so weit wie möglich verwenden, um Anpassungen zuzulassen, und versuchen, Auto-Configuration Helper-Annotationen wie `@AutoConfigureBefore` , `@AutoConfigureAfter` , `@ConditionalOnBean` , `@ConditionalOnMissingBean` usw. zu verwenden. Weitere Informationen zu jeder Anmerkung finden Sie in den offiziellen Anmerkungen zu den Bedingungen
2. Platzieren Sie eine bzw. `@Configuration` , die alle `@Configuration` Klassen zusammenfassen.
3. Erstellen Sie eine Datei mit dem Namen `spring.factories` und platzieren Sie sie in `src/main/resources/META-INF` .
4. `spring.factories` in `org.springframework.boot.autoconfigure.EnableAutoConfiguration` Eigenschaft `org.springframework.boot.autoconfigure.EnableAutoConfiguration` mit durch Kommas getrennten Werten Ihrer `@Configuration` Klassen fest:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.mycorp.libx.autoconfigure.LibXAutoConfiguration,\
com.mycorp.libx.autoconfigure.LibXWebAutoConfiguration
```

Mit dieser Methode können Sie Ihre eigenen Autokonfigurationsklassen erstellen, die per Spring-Boot ausgewählt werden. Spring-Boot durchsucht automatisch alle Abhängigkeiten von Maven / Gradle nach einer `spring.factories` Datei. Wenn sie eine findet, fügt sie alle `@Configuration` angegebenen `@Configuration` Klassen zum automatischen Konfigurationsprozess hinzu.

Stellen Sie sicher , dass Ihre `auto-configuration` Starter Projekt nicht enthält `spring boot maven plugin` , weil sie das Projekt als eine ausführbare JAR - Paket und wird nicht durch den Classpath geladen werden , wie beabsichtigt - Feder Boot nicht in der Lage Ihr finden `spring.factories` und lädt Ihre Konfiguration nicht

Scannen von Paketen online lesen: https://riptutorial.com/de/spring-boot/topic/9354/scannen-von-paketen

Kapitel 10: Spring Boot + JPA + REST

Bemerkungen

In diesem Beispiel werden Spring Boot, Spring Data JPA und Spring Data REST verwendet, um ein einfaches, von JPA verwaltetes Domänenobjekt über REST bereitzustellen. Das Beispiel antwortet mit dem HAL JSON-Format und macht eine auf `/person` zugreifbare URL verfügbar. Die Maven-Konfiguration enthält eine H2-In-Memory-Datenbank, um ein schnelles Standup zu unterstützen.

Examples

Spring Boot Startup

```
package com.example;

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

@SpringBootApplication
public class Application {

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

}
```

Domain-Objekt

```
package com.example.domain;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;

//simple person object with JPA annotations

@Entity
public class Person {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column
    private String firstName;

    @Column
    private String lastName;
```

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

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

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

}
```

Repository-Schnittstelle

```
package com.example.domain;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;
import org.springframework.data.rest.core.annotation.RestResource;

//annotation exposes the
@RepositoryRestResource(path="/person")
public interface PersonRepository extends JpaRepository<Person,Long> {

    //the method below allows us to expose a search query, customize the endpoint name, and
specify a parameter name
    //the exposed URL is GET /person/search/byLastName?lastname=
    @RestResource(path="/byLastName")
    Iterable<Person> findByLastName(@Param("lastName") String lastName);

    //the methods below are examples on to prevent an operation from being exposed.
    //For example DELETE; the methods are overridden and then annotated with
RestResouce(exported=false) to make sure that no one can DELETE entities via REST
    @Override
    @RestResource(exported=false)
    default void delete(Long id) { }

    @Override
    @RestResource(exported=false)
    default void delete(Person entity) { }

    @Override
    @RestResource(exported=false)
    default void delete(Iterable<? extends Person> entities) { }
```

```
    @Override
    @RestResource(exported=false)
    default void deleteAll() { }

    @Override
    @RestResource(exported=false)
    default void deleteAllInBatch() { }

    @Override
    @RestResource(exported=false)
    default void deleteInBatch(Iterable<Person> arg0) { }


}
```

Maven-Konfiguration

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.0.RELEASE</version>
    </parent>
    <groupId>com.example</groupId>
    <artifactId>spring-boot-data-jpa-rest</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>spring-boot-data-jpa-rest</name>
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-rest</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
```

```
        </dependency
    </dependencies>
</project>
```

Spring Boot + JPA + REST online lesen: https://riptutorial.com/de/spring-boot/topic/6507/spring-boot-plus-jpa-plus-rest

Kapitel 11: Spring Boot + Ruhezustand + Web-Benutzeroberfläche (Thymeleaf)

Einführung

Dieser Thread befasst sich mit der Erstellung einer Springboot-Anwendung mit Hibernate- und Thymyleaf-Vorlagenmodul.

Bemerkungen

Überprüfen Sie auch die Thymeleaf-Dokumentation

Examples

Abhängigkeiten von Maven

Dieses Beispiel basiert auf Spring Boot 1.5.1.RELEASE. mit folgenden Abhängigkeiten:

```
<!-- Spring -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!-- Lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
<!-- H2 -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
<!-- Test -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

In diesem Beispiel werden Spring Boot JPA, Thymeleaf und Webstarter verwendet. Ich benutze

Lombok, um Getter und Setter einfacher zu generieren, aber das ist nicht zwingend. H2 wird als einfach zu konfigurierende In-Memory-Datenbank verwendet.

Ruhezustand der Konfiguration

Lassen Sie uns zunächst einen Überblick darüber geben, was wir benötigen, um den Ruhezustand richtig einzurichten.

1. `@EnableTransactionManagement` und `@EnableJpaRepositories` - wir wollen Transaktionsmanagement und verwenden Spring Data Repositories.
2. `DataSource` - Hauptdatenquelle für die Anwendung. In-Memory h2 für dieses Beispiel verwenden.
3. `LocalContainerEntityManagerFactoryBean` - Entitätsmanager-Factory für Frühling, die `HibernateJpaVendorAdapter` .
4. `PlatformTransactionManager` - Haupttransaktionsmanager für mit `@Transactional` versehene Komponenten von `@Transactional` .

Konfigurationsdatei:

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = "com.example.repositories")
public class PersistanceJpaConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:testdb;mode=MySQL;DB_CLOSE_DELAY=-
1;DB_CLOSE_ON_EXIT=FALSE");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource)
{
        LocalContainerEntityManagerFactoryBean em = new
LocalContainerEntityManagerFactoryBean();
        em.setDataSource(dataSource);
        em.setPackagesToScan(new String[] { "com.example.models" });
        JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        em.setJpaVendorAdapter(vendorAdapter);
        em.setJpaProperties(additionalProperties());
        return em;
    }

    @Bean
    public PlatformTransactionManager
transactionManager(LocalContainerEntityManagerFactoryBean entityManagerFactory, DataSource
dataSource) {
        JpaTransactionManager tm = new JpaTransactionManager();
        tm.setEntityManagerFactory(entityManagerFactory.getObject());
        tm.setDataSource(dataSource);
        return tm;
```

```
    }

    Properties additionalProperties() {
        Properties properties = new Properties();
        properties.setProperty("hibernate.hbm2ddl.auto", "update");
        properties.setProperty("hibernate.dialect", "org.hibernate.dialect.H2Dialect");
        return properties;
    }

}
```

Entitäten und Repositories

Eine einfache Entität: Verwenden der `@Getter` und `@Setter` Setter-Anmerkungen von Lombok, um Getter und Setter für uns zu generieren

```
@Entity
@Getter @Setter
public class Message {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    private String message;

}
```

Ich verwende UUID-basierte IDs und Lombok, um Getter und Setter zu generieren.

Ein einfaches Repository für die Entität oben:

```
@Transactional
public interface MessageRepository extends CrudRepository<Message, String> {
}
```

Weitere Informationen zu Ersatzlagern: Spring Data Docs

Stellen Sie sicher, dass sich die Entitäten in einem Paket befinden, das in `em.setPackagesToScan` (definiert in der Bean `LocalContainerEntityManagerFactoryBean`) und in einem in `basePackages` (in der Annotation `@EnableJpaRepositories`) definierten Paket zugeordnet ist.

Thymeleaf-Ressourcen und Federsteuerung

Um Thymeleaf-Vorlagen verfügbar zu machen, müssen Controller definiert werden.

Beispiel:

```
@Controller
@RequestMapping("/")
public class MessageController {

    @Autowired
```

```
    private MessageRepository messageRepository;

    @GetMapping
    public ModelAndView index() {
        Iterable<Message> messages = messageRepository.findAll();
        return new ModelAndView("index", "index", messages);
    }

}
```

Dieser einfache Controller fügt `MessageRepository` und übergibt alle Nachrichten an eine Vorlagendatei mit dem Namen `index.html` , die sich in `src/main/resources/templates` , und schließlich in `/index` .

Auf die gleiche Weise können wir andere Vorlagen im Vorlagenordner ablegen (standardmäßig bis Frühling zu `src/main/resources/templates`), ihnen ein Modell übergeben und sie dem Kunden zur Verfügung stellen.

Andere statische Ressourcen sollten in einem der folgenden Ordner abgelegt werden, die standardmäßig im Spring Boot verfügbar sind:

```
/META-INF/resources/
/resources/
/static/
/public/
```

Thymeleaf `index.html` Beispiel:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
    <head th:fragment="head (title)">
        <title th:text="${title}">Index</title>
        <link rel="stylesheet" th:href="@{/css/bootstrap.min.css}"
href="../../css/bootstrap.min.css" />
    </head>
    <body>
        <nav class="navbar navbar-default navbar-fixed-top">
          <div class="container-fluid">
            <div class="navbar-header">
              <a class="navbar-brand" href="#">Thymeleaf</a>
            </div>
          </div>
        </nav>
        <div class="container">
            <ul class="nav">
                <li><a th:href="@{/}" href="messages.html"> Messages </a></li>
            </ul>
        </div>
    </body>
</html>
```

- `bootstrap.min.css` befindet sich im Ordner `src/main/resources/static/css` . Sie können die Syntax `@{}` , um andere statische Ressourcen über den relativen Pfad abzurufen.

Spring Boot + Ruhezustand + Web-Benutzeroberfläche (Thymeleaf) online lesen:

https://riptutorial.com/de/spring-boot/topic/9200/spring-boot-plus-ruhezustand-plus-web-benutzeroberflache--thymeleaf-

Kapitel 12: Spring Boot + Spring Data Elasticsearch

Einführung

Spring Data Elasticsearch ist eine Spring Data- Implementierung für Elasticsearch, die die Integration mit der Elasticsearch- Suchmaschine ermöglicht.

Examples

Integration von Spring Boot und Spring Data Elasticsearch

In diesem Beispiel werden wir ein Feder-Daten-Elasticsearch-Projekt implementieren, um POJO in der Elasticsearch zu speichern. Wir werden ein Musterprojekt sehen, das folgende Aufgaben erfüllt:

- Fügen Sie ein `Greeting(id, username, message)` in elasticsearch ein.
- Alle Begrüßungselemente abrufen, die eingefügt wurden.
- Aktualisieren Sie ein Grußelement.
- Löschen Sie ein Begrüßungselement.
- Bekommen Sie einen Begrüßungsartikel nach ID.
- Holen Sie sich alle Begrüßungsartikel nach Benutzername.

Spring Boot und Spring Data Elasticsearch Integration

In diesem Beispiel sehen wir eine Maven-basierte Federstiefelanwendung, die die Federdaten-Elasticsuche integriert. Hier werden wir Folgendes tun und die entsprechenden Codesegmente sehen.

- Fügen Sie ein `Greeting(id, username, message)` in elasticsearch ein.
- Holen Sie sich alle Artikel von elasticsearch
- Aktualisieren Sie ein bestimmtes Element.
- Einen bestimmten Artikel löschen
- Holen Sie sich einen bestimmten Artikel nach ID.
- Holen Sie sich einen bestimmten Artikel nach Benutzername.

Projektkonfigurationsdatei (pom.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
    <groupId>org.springdataes</groupId>
    <artifactId>springdataes</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.6.RELEASE</version>
    </parent>

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

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <goals>
                            <goal>repackage</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Wir werden Spring Boot der Version `1.5.6.RELEASE` und Spring Data Elasticsearch der jeweiligen Version verwenden. Für dieses Projekt müssen wir elasticsearch-2.4.5 ausführen, um unsere Apis zu testen.

Eigenschaftendatei

Wir legen die Projekteigenschaftsdatei (namens `applications.properties`) im `resources` Folgendes enthält:

```
elasticsearch.clustername = elasticsearch
elasticsearch.host = localhost
elasticsearch.port = 9300
```

Wir verwenden den Standard-Clusternamen, Host und Port. Standardmäßig wird der `9300` Port als Transportport verwendet und der `9200` Port wird als http-Port bezeichnet. Um den Standard-Clusternamen anzuzeigen, klicken Sie auf http: // localhost: 9200 / .

Hauptklasse (Application.java)

```
package org.springdataes;

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

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

`@SpringBootApplication` ist eine Kombination aus `@Configuration` `@EnableAutoConfiguration` , `@EnableWebMvc` `@Configuration` , `@EnableAutoConfiguration` `@EnableWebMvc` und `@ComponentScan` Anmerkungen. Die `main()` -Methode verwendet die `SpringApplication.run()` -Methode von Spring Boot, um eine Anwendung zu starten. Da wir keine XML-Konfiguration benötigen, handelt es sich bei dieser Anwendung um eine reine Java-Frühling-Anwendung.

Elasticsearch-Konfigurationsklasse (ElasticsearchConfig.java)

```
package org.springdataes.config;

import org.elasticsearch.client.Client;
import org.elasticsearch.client.transport.TransportClient;
import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.common.transport.InetSocketTransportAddress;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.data.elasticsearch.core.ElasticsearchOperations;
import org.springframework.data.elasticsearch.core.ElasticsearchTemplate;
import
org.springframework.data.elasticsearch.repository.config.EnableElasticsearchRepositories;

import java.net.InetAddress;

@Configuration
@PropertySource(value = "classpath:applications.properties")
@EnableElasticsearchRepositories(basePackages = "org.springdataes.dao")
public class ElasticsearchConfig {
    @Value("${elasticsearch.host}")
    private String EsHost;

    @Value("${elasticsearch.port}")
    private int EsPort;

    @Value("${elasticsearch.clustername}")
    private String EsClusterName;

    @Bean
    public Client client() throws Exception {
        Settings esSettings = Settings.settingsBuilder()
                .put("cluster.name", EsClusterName)
                .build();
```

```
        return TransportClient.builder()
                .settings(esSettings)
                .build()
                .addTransportAddress(new
InetSocketTransportAddress(InetAddress.getByName(EsHost), EsPort));
    }

    @Bean
    public ElasticsearchOperations elasticsearchTemplate() throws Exception {
        return new ElasticsearchTemplate(client());
    }
}
```

`ElasticsearchConfig` Klasse `ElasticsearchConfig` konfiguriert die `ElasticsearchConfig` für dieses Projekt und stellt eine Verbindung mit der `ElasticsearchConfig` . Hier wird `@PropertySource` zum Lesen der Datei `application.properties` verwendet, in der der Clustername, der Elasticsearch-Host und der Port gespeichert werden. `@EnableElasticsearchRepositories` wird verwendet, um Elasticsearch-Repositorys zu aktivieren, die die Pakete der annotierten Konfigurationsklasse standardmäßig nach Spring Data-Repositorys scannen. `@Value` wird hier zum Lesen der Eigenschaften aus der Datei `application.properties` verwendet.

Die `Client()` Methode erstellt eine Transportverbindung mit Elasticsearch. Die obige Konfiguration richtet einen Embedded Elasticsearch Server ein, der vom `ElasticsearchTemplate` . Die Bean " `ElasticsearchTemplate` verwendet den `Elasticsearch Client` und stellt eine benutzerdefinierte Ebene zum Bearbeiten von Daten in Elasticsearch bereit.

Modellklasse (Gruß.java)

```
package org.springdataes.model;

import org.springframework.data.annotation.Id;
import org.springframework.data.elasticsearch.annotations.Document;
import java.io.Serializable;

@Document(indexName = "index", type = "greetings")
public class Greeting implements Serializable{

    @Id
    private String id;

    private String username;

    private String message;

    public Greeting() {
    }

    public String getId() {
        return id;
    }

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

    public String getUsername() {
```

```
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

Hier haben wir unsere `Greeting` mit einer @Document-Annotation versehen, mit der wir auch Indexeinstellungen wie Name, Anzahl der Shards oder Anzahl der Repliken ermitteln können. Eines der Attribute der Klasse muss eine `id` , entweder durch Annotation mit `@Id` oder durch Verwendung eines der automatisch gefundenen Namen `id` oder `documentId` . Hier wird der `id` -Feldwert automatisch generiert, wenn wir keinen Wert für das `id` Feld festlegen.

Elasticsearch-Repository-Klasse (GreetingRepository.class)

```
package org.springdataes.dao;

import org.springdataes.model.Greeting;
import org.springframework.data.elasticsearch.repository.ElasticsearchRepository;
import java.util.List;

public interface GreetingRepository extends ElasticsearchRepository<Greeting, String> {
    List<Greeting> findByUsername(String username);
}
```

Hier haben wir das `ElasticsearchRepository` erweitert, das uns viele Apis bietet, die wir nicht extern definieren müssen. Dies ist die Basis-Repository-Klasse für `elasticsearch` basierte Domänenklassen. Da `Spring` basierte Repository-Klassen erweitert werden, haben wir den Vorteil, dass der für die Implementierung von Datenzugriffsschichten für verschiedene Persistenzspeicher erforderliche Boilerplate-Code vermieden wird.

Hier haben wir eine Methode `findByUsername(String username)` deklariert, die in eine Übereinstimmungsabfrage konvertiert wird, die mit dem Benutzernamen mit dem `username` der `Greeting` übereinstimmt und die Ergebnisliste zurückgibt.

Dienste (GreetingService.java)

```
package org.springdataes.service;

import org.springdataes.model.Greeting;
import java.util.List;

public interface GreetingService {
    List<Greeting> getAll();
    Greeting findOne(String id);
```

```
    Greeting create(Greeting greeting);
    Greeting update(Greeting greeting);
    List<Greeting> getGreetingByUsername(String username);
    void delete(String id);
}
```

Service Bean (GreetingServiceBean.java)

```
package org.springdataes.service;

import com.google.common.collect.Lists;
import org.springdataes.dao.GreetingRepository;
import org.springdataes.model.Greeting;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class GreetingServiceBean implements GreetingService {

    @Autowired
    private GreetingRepository repository;

    @Override
    public List<Greeting> getAll() {
        return Lists.newArrayList(repository.findAll());
    }

    @Override
    public Greeting findOne(String id) {
        return repository.findOne(id);
    }

    @Override
    public Greeting create(Greeting greeting) {
        return repository.save(greeting);
    }

    @Override
    public Greeting update(Greeting greeting) {
        Greeting persitedGreeting = repository.findOne(greeting.getId());
        if(persitedGreeting == null) {
            return null;
        }
        return repository.save(greeting);
    }

    @Override
    public List<Greeting> getGreetingByUsername(String username) {
        return repository.findByUsername(username);
    }

    @Override
    public void delete(String id) {
        repository.delete(id);
    }
}
```

In oben Klasse haben wir `@Autowired` die `GreetingRepository` . Wir können einfach die `CRUDRepository` Methoden und die von uns in der Repository-Klasse mit dem `GreetingRepository` Objekt deklarierte Methode aufrufen.

In der `getAll()` -Methode finden Sie möglicherweise eine Zeile `Lists.newArrayList(repository.findAll())` . Wir haben dies getan, um `repository.findAll()` in `List<>` `Iterable` zu konvertieren, da es eine `Iterable` List `Iterable` .

Controller-Klasse (GreetingController.java)

```
package org.springdataes.controller;

import org.springdataes.model.Greeting;
import org.springdataes.service.GreetingService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api")
public class GreetingController {

    @Autowired
    private GreetingService greetingService;

    @ResponseBody
    @RequestMapping(value = "/greetings", method = RequestMethod.GET)
    public ResponseEntity<List<Greeting>> getAll() {
        return new ResponseEntity<List<Greeting>>(greetingService.getAll(), HttpStatus.OK);
    }

    @ResponseBody
    @RequestMapping(value = "/greetings", method = RequestMethod.POST)
    public ResponseEntity<Greeting> insertGreeting(@RequestBody Greeting greeting) {
        return new ResponseEntity<Greeting>(greetingService.create(greeting),
HttpStatus.CREATED);
    }

    @ResponseBody
    @RequestMapping(value = "/greetings", method = RequestMethod.PUT)
    public ResponseEntity<Greeting> updateGreeting(@RequestBody Greeting greeting) {
        return new ResponseEntity<Greeting>(greetingService.update(greeting),
HttpStatus.MOVED_PERMANENTLY);
    }

    @ResponseBody
    @RequestMapping(value = "/greetings/{id}", method = RequestMethod.DELETE)
    public ResponseEntity<Greeting> deleteGreeting(@PathVariable("id") String idd) {
        greetingService.delete(idd);
        return new ResponseEntity<Greeting>(HttpStatus.NO_CONTENT);
    }

    @ResponseBody
    @RequestMapping(value = "/greetings{id}", method = RequestMethod.POST)
    public ResponseEntity<Greeting> getOne(@PathVariable("id") String idd) {
        return new ResponseEntity<Greeting>(greetingService.findOne(idd), HttpStatus.OK);
```

```
    }

    @ResponseBody
    @RequestMapping(value = "/greetings/{name}", method = RequestMethod.GET)
    public ResponseEntity<List<Greeting>> getByUserName(@PathVariable("name") String name) {
        return new ResponseEntity<List<Greeting>>(greetingService.getGreetingByUsername(name),
HttpStatus.OK);
    }
}
```

Bauen

So erstellen Sie diese Maven-Anwendung

```
mvn clean install
```

Obiger Befehl entfernen Sie zuerst alle Dateien im `target` und das Projekt. Nach dem `springdataes-1.0-SNAPSHOT.jar` des Projekts erhalten Sie die ausführbare .jar-Datei mit dem Namen `springdataes-1.0-SNAPSHOT.jar` . Wir können die Hauptklasse (`Application.java`) ausführen, um den Prozess zu starten, oder indem Sie einfach den obigen Jar ausführen, indem Sie Folgendes eingeben:

```
java -jar springdataes-1.0-SNAPSHOT.jar
```

Überprüfen der APIs

Führen Sie den folgenden Befehl aus, um ein Begrüßungselement in die Elasticsearch einzufügen

```
curl -H "Content-Type: application/json" -X POST -d '{"username":"sunkuet02","message": "this
is a message"}' http://localhost:8080/api/greetings
```

Sie sollten das untenstehende Ergebnis gerne erhalten

```
{"id":"AV2ddRxBcuirs1TrVgHH","username":"sunkuet02","message":"this is a message"}
```

Sie können die Get-API auch überprüfen, indem Sie Folgendes ausführen:

```
curl -H "Content-Type: application/json" -X GET http://localhost:8080/api/greetings
```

Du solltest bekommen

```
[{"id":"AV2ddRxBcuirs1TrVgHH","username":"sunkuet02","message":"this is a message"}]
```

Sie können andere Apis überprüfen, indem Sie die obigen Schritte befolgen.

Offizielle Dokumentationen:

- https://projects.spring.io/spring-boot/
- https://projects.spring.io/spring-data-elasticsearch/

Spring Boot + Spring Data Elasticsearch online lesen: https://riptutorial.com/de/spring-boot/topic/10928/spring-boot-plus-spring-data-elasticsearch

Kapitel 13: Spring Boot-Hibernate-REST-Integration

Examples

Hibernate-Unterstützung hinzufügen

1. Fügen Sie pom.xml die **Spring-Boot-Starter-data-jpa-** Abhängigkeit hinzu. Sie können das **Versionskennzeichen** überspringen, wenn Sie **Spring-Boot-Starter-Parent** als übergeordnetes **Element** Ihrer **pom.xml verwenden** . Die nachstehende Abhängigkeit bringt den Ruhezustand und alles, was mit JPA zusammenhängt, in Ihr Projekt (Referenz).

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

2. Fügen Sie der **pom.xml den** Datenbanktreiber hinzu. Dieser hier ist für die H2-Datenbank (Referenz).

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

3. Aktivieren Sie die Debugprotokollierung für Hibernate in **application.properties**

   logging.level.org.hibernate.SQL = debug

oder in **application.yml**

```
logging:
  level:
    org.hibernate.SQL: debug
```

4. Fügen Sie unter **$ {project.home} / src / main / java /** dem gewünschten Paket eine Entitätsklasse hinzu, zum Beispiel unter **com.example.myproject.domain** (reference):

```
package com.example.myproject.domain;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import java.io.Serializable;

@Entity
public class City implements Serializable {
```

```
    @Id
    @GeneratedValue
    public Long id;

    @Column(nullable = false)
    public String name;
}
```

5. Fügen Sie **import.sql** zu **$ {project.home} / src / main / resources / hinzu** . **Fügen** Sie **INSERT-** Anweisungen in die Datei ein. Diese Datei wird für die Auffüllung des Datenbankschemas bei jedem Start der App (Referenz) verwendet:

```
insert into city(name) values ('Brisbane');

insert into city(name) values ('Melbourne');
```

6. Fügen Sie die Repository-Klasse unter **$ {project.home} / src / main / java /** dem gewünschten Paket hinzu, beispielsweise unter **com.example.myproject.service** (Referenz):

```
package com.example.myproject.service;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import java.io.Serializable;

import com.example.myproject.domain.City;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.Repository;

interface CityRepository extends Repository<City, Long> {

    Page<City> findAll(Pageable pageable);

    Page<City> findByName(String name);
}
```

Im Grunde ist es das! An dieser Stelle können Sie bereits mit den Methoden von **com.example.myproject.service.CityRepository** auf die Datenbank **zugreifen** .

REST-Unterstützung hinzufügen

1. Fügen Sie pom.xml die **Spring-Boot-Starter-Web-** Abhängigkeit hinzu. Sie können das **Versions-** Tag überspringen, wenn Sie **Spring-Boot-Starter-Parent** als übergeordnetes **Element** Ihrer **pom.xml** (Referenz) verwenden.

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

```
    </dependency>
```

2. Fügen Sie dem gewünschten Paket den REST-Controller hinzu, beispielsweise zu **com.example.myproject.web.rest** (Referenz):

```
package com.example.myproject.web.rest;

import java.util.Map;
import java.util.HashMap;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.http.HttpServletRequest;

@RestController
public class VersionController {
    @RequestMapping("/api/version")
    public ResponseEntity get() {
        final Map<String, String> responseParams = new HashMap();
        responseParams.put("requestStatus", "OK");
        responseParams.put("version", "0.1-SNAPSHOT");

        return ResponseEntity.ok().body(responseParams.build());
    }
}
```

3. Starten Sie die Spring Boot-Anwendung (Referenz).

4. Ihr Controller ist unter der Adresse **http: // localhost: 8080 / api / version erreichbar** .

Spring Boot-Hibernate-REST-Integration online lesen: https://riptutorial.com/de/spring-boot/topic/6818/spring-boot-hibernate-rest-integration

Kapitel 14: Spring-Boot + JDBC

Einführung

Mit Spring Boot kann eine relationale SQL-Datenbank erstellt und beibehalten werden. Sie können wählen, ob Sie mit Spring Boot eine Verbindung zu einer H2 in der Speicherdatenbank herstellen möchten, oder Sie wählen möglicherweise eine Verbindung zu MySql DataBase. Wenn Sie CRUD-Vorgänge für Ihre Datenbank ausführen möchten, können Sie dies mit dem JdbcTemplate-Bean tun. Dieses Bean wird automatisch von Spring Boot bereitgestellt. Spring Boot hilft Ihnen dabei, indem Sie die automatische Konfiguration einiger häufig verwendeter Beans für JDBC bereitstellen.

Bemerkungen

Um zu beginnen, gehen Sie in Ihrer sts-Finsternis auf neu -> Spring Starter Project -> geben Sie Ihre Maven-Koordinaten ein -> und fügen Sie die nächsten Abhängigkeiten hinzu:

Auf der Registerkarte SQL -> add JDBC + fügen Sie MySql hinzu (wenn MySql Ihre Wahl ist).

Für MySQL müssen Sie auch den MySQL-Java-Connector hinzufügen.

In Ihrer Spring Boot-Datei application.properties (Ihre Spring Boot-Konfigurationsdatei) müssen Sie Ihre Datenquellenberechtigungsnachweise für MySql DB konfigurieren:

1. spring.datasource.url
2. spring.datasource.username
3. spring.datasource.password
4. spring.datasource.driver-klassenname

zum Beispiel:

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Fügen Sie unter dem Ressourcenordner die nächsten zwei Dateien hinzu:

1. schema.sql -> Jedes Mal, wenn Sie Ihre Anwendung ausführen, führt Spring Boot diese Datei aus, in der Sie Ihr DB-Schema schreiben, Tabellen und ihre Beziehungen definieren.

2. data.sql -> Jedes Mal, wenn Sie Ihre Anwendung ausführen, führt Spring Boot diese Datei aus. In dieser Datei werden Daten geschrieben, die als Initialisierung in Ihre Tabelle eingefügt werden.

Spring Boot stellt Ihnen die JdbcTemplate-Bean automatisch zur Verfügung, sodass Sie sie sofort wie folgt verwenden können:

```
@Autowired
private JdbcTemplate template;
```

ohne andere Konfigurationen.

Examples

schema.sql-Datei

```
CREATE SCHEMA IF NOT EXISTS `backgammon`;
USE `backgammon`;

DROP TABLE IF EXISTS `user_in_game_room`;
DROP TABLE IF EXISTS `game_users`;
DROP TABLE IF EXISTS `user_in_game_room`;

CREATE TABLE `game_users`
(
    `user_id` BIGINT NOT NULL AUTO_INCREMENT,
    `first_name` VARCHAR(255) NOT NULL,
    `last_name` VARCHAR(255) NOT NULL,
    `email` VARCHAR(255) NOT NULL UNIQUE,
    `user_name` VARCHAR(255) NOT NULL UNIQUE,
    `password` VARCHAR(255) NOT NULL,
    `role` VARCHAR(255) NOT NULL,
    `last_updated_date` DATETIME NOT NULL,
    `last_updated_by` BIGINT NOT NULL,
    `created_date` DATETIME NOT NULL,
    `created_by` BIGINT NOT NULL,
    PRIMARY KEY(`user_id`)
);

DROP TABLE IF EXISTS `game_rooms`;

CREATE TABLE `game_rooms`
(
    `game_room_id` BIGINT NOT NULL AUTO_INCREMENT,
    `name` VARCHAR(255) NOT NULL,
    `private` BIT(1) NOT NULL,
    `white` BIGINT DEFAULT NULL,
    `black` BIGINT DEFAULT NULL,
    `opened_by` BIGINT NOT NULL,
    `speed` BIT(3) NOT NULL,
    `last_updated_date` DATETIME NOT NULL,
    `last_updated_by` BIGINT NOT NULL,
    `created_date` DATETIME NOT NULL,
    `created_by` BIGINT NOT NULL,
    `token` VARCHAR(255) AS (SHA1(CONCAT(`name`, "This is a qwe secret 123", `created_by`,
`created_date`))),
    PRIMARY KEY(`game_room_id`)
);

CREATE TABLE `user_in_game_room`
(
    `user_id` BIGINT NOT NULL,
    `game_room_id` BIGINT NOT NULL,
    `last_updated_date` DATETIME NOT NULL,
    `last_updated_by` BIGINT NOT NULL,
```

```
    `created_date` DATETIME NOT NULL,
    `created_by` BIGINT NOT NULL,
    PRIMARY KEY(`user_id`, `game_room_id`),
    FOREIGN KEY (`user_id`) REFERENCES `game_users`(`user_id`),
    FOREIGN KEY (`game_room_id`) REFERENCES `game_rooms`(`game_room_id`)
);
```

Erste JdbcTemplate-Startanwendung

```
@SpringBootApplication
@RestController
public class SpringBootJdbcApplication {

    @Autowired
    private JdbcTemplate template;

    @RequestMapping("/cars")
    public List<Map<String,Object>> stocks(){
        return template.queryForList("select * from car");
    }

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

data.sql

```
insert into game_users values(..., ..., ..., ...);
insert into game_users values(..., ..., ..., ...);
insert into game_users values(..., ..., ..., ...);
```

Spring-Boot + JDBC online lesen: https://riptutorial.com/de/spring-boot/topic/9834/spring-boot-plus-jdbc

Kapitel 15: Springboot + JPA + MongoDB

Examples

CRUD-Vorgang in MongoDB mit JPA

Kundenmodell

```
package org.bookmytickets.model;

import org.springframework.data.annotation.Id;

public class Customer {

    @Id
    private String id;
    private String firstName;
    private String lastName;

    public Customer() {}

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public Customer(String id,String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return String.format(
                "Customer[id=%s, firstName='%s', lastName='%s']",
                id, firstName, lastName);
    }

    public String getId() {
        return id;
    }

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

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
```

```
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

}
```

Kundencontroller

```
package org.bookmytickets.controller;

import java.util.List;

import org.bookmytickets.model.Customer;
import org.bookmytickets.repository.CustomerRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(value = "/customer")
public class CustomerController {

    @Autowired
    private CustomerRepository repository;

    @GetMapping("")
    public List<Customer> selectAll(){
        List<Customer> customerList = repository.findAll();
        return customerList;
    }

    @GetMapping("/{id}")
    public List<Customer> getSpecificCustomer(@PathVariable String id){
        return repository.findById(id);
    }

    @GetMapping("/search/lastName/{lastName}")
    public List<Customer> searchByLastName(@PathVariable String lastName){
        return repository.findByLasttName(lastName);
    }

    @GetMapping("/search/firstname/{firstname}")
    public List<Customer> searchByFirstName(@PathVariable String firstName){
        return repository.findByFirstName(firstName);
    }

    @PostMapping("")
    public void insert(@RequestBody Customer customer) {
        repository.save(customer);
    }

    @PatchMapping("/{id}")
    public void update(@RequestParam String id, @RequestBody Customer customer) {
```

```
        Customer oldCustomer = repository.finedById(id);
        if(customer.getFirstName() != null) {
            oldCustomer.setFristName(customer.getFirstName());
        }
        if(customer.getLastName() != null) {
            oldCustomer.setLastName(customer.getLastName());
        }
        repository.save(oldCustomer);
    }

    @DeleteMapping("/{id}")
    public void delete(@RequestParam String id) {
        Customer deleteCustomer = repository.findById(id);
        repository.delete(deleteCustomer);
    }
}
```

Kunden-Repository

```
package org.bookmytickets.repository;

import java.util.List;

import org.bookmytickets.model.Customer;
import org.springframework.data.mongodb.repository.MongoRepository;

public interface CustomerRepository extends MongoRepository<Customer, String> {
    public Customer findByFirstName(String firstName);
    public List<Customer> findByLastName(String lastName);
}
```

pom.xml

Bitte fügen Sie folgende Abhängigkeiten in der Datei pom.xml hinzu:

```
<dependencies>

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

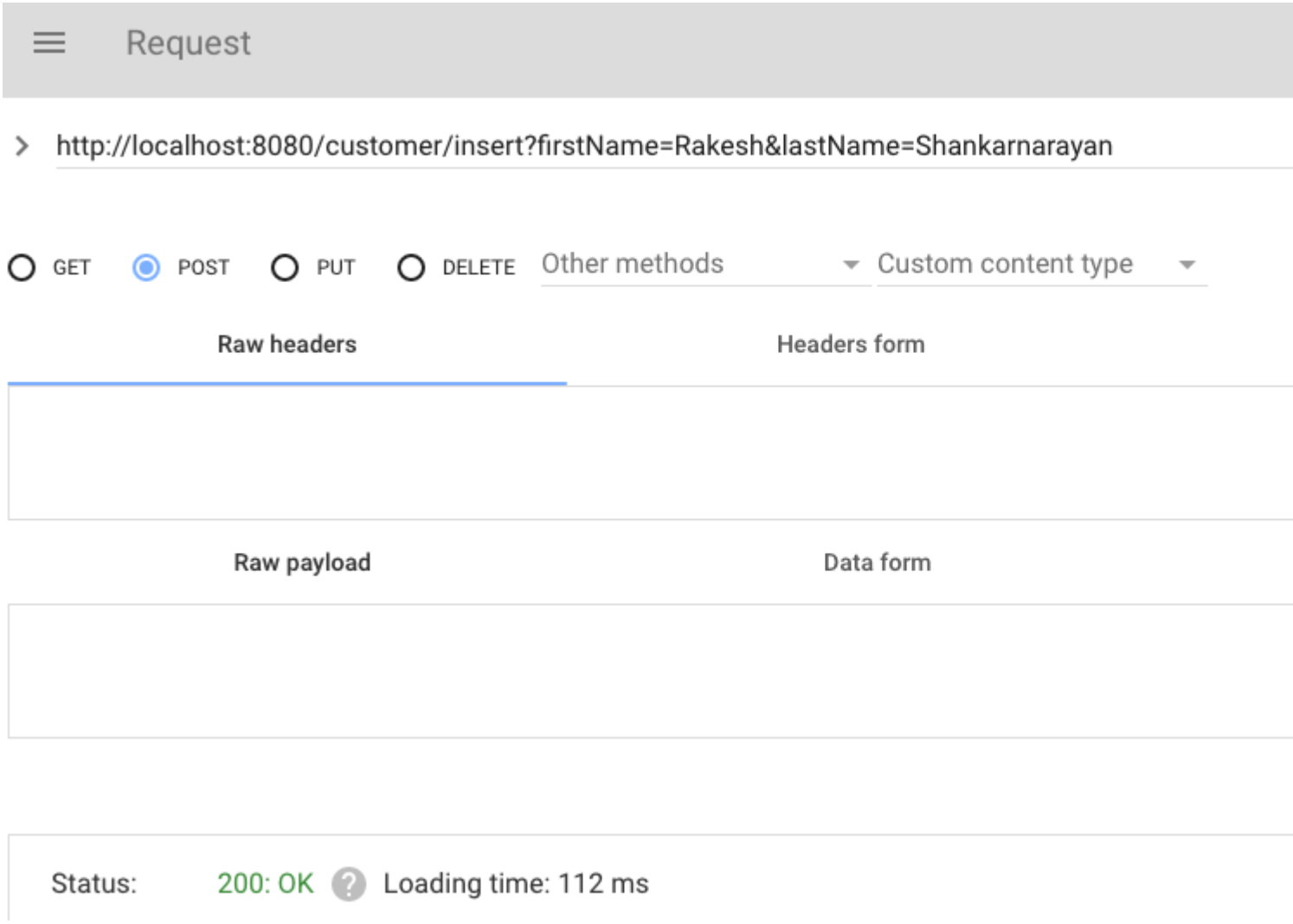
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-mongodb</artifactId>
        </dependency>

</dependencies>
```

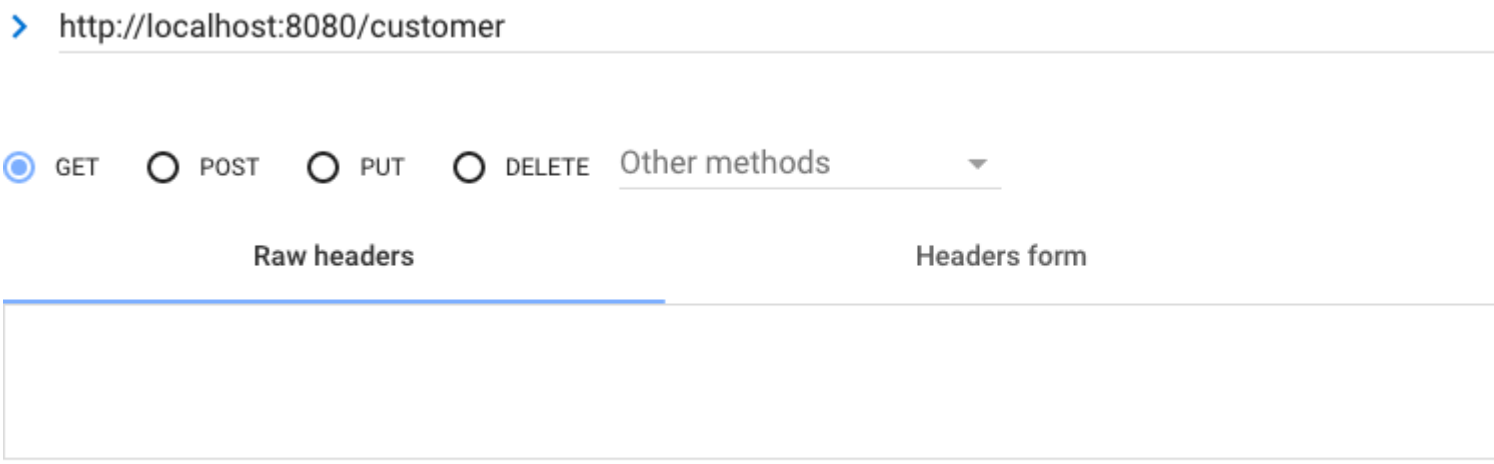
Fügen Sie Daten mit dem Restclient ein: POST-Methode

Um unsere Anwendung zu testen, verwende ich den Rest Rest Client, eine Chrome-Erweiterung:

Hier ist also die Momentaufnahme zum Einfügen der Daten:



Anforderungs-URL abrufen



Anforderungsergebnis abfragen:



Springboot + JPA + MongoDB online lesen: https://riptutorial.com/de/spring-boot/topic/3398/springboot-plus-jpa-plus-mongodb

Kapitel 16: Spring-Boot Microservice mit JPA

Examples

Anwendungsklasse

```
package com.mcf7.spring;

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

@SpringBootApplication
public class SpringDataMicroServiceApplication {

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

Buchmodell

```
package com.mcf7.spring.domain;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import java.io.Serializable;

@lombok.Getter
@lombok.Setter
@lombok.EqualsAndHashCode(of = "isbn")
@lombok.ToString(exclude="id")
@Entity
public class Book implements Serializable {

    public Book() {}

    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    private long id;

    @NotNull
    @Size(min = 1)
    private String isbn;

    @NotNull
    @Size(min = 1)
    private String title;

    @NotNull
    @Size(min = 1)
    private String author;
```

```
    @NotNull
    @Size(min = 1)
    private String description;
}
```

Nur ein Hinweis, da hier ein paar Dinge los sind, wollte ich sie schnell brechen.

Alle Annotationen mit @lombok generieren einige Kesselplatten unserer Klasse

```
@lombok.Getter  //Creates getter methods for our variables

@lombok.Setter  //Creates setter methods four our variables

@lombok.EqualsAndHashCode(of = "isbn") //Creates Equals and Hashcode methods based off of the
isbn variable

@lombok.ToString(exclude="id") //Creates a toString method based off of every variable except
id
```

Wir haben auch die Validierung in diesem Objekt genutzt

```
@NotNull  //This specifies that when validation is called this element shouldn't be null

@Size(min = 1)  //This specifies that when validation is called this String shouldn't be
smaller than 1
```

Buchspeicher

```
package com.mcf7.spring.domain;

import org.springframework.data.repository.PagingAndSortingRepository;

public interface BookRepository extends PagingAndSortingRepository<Book, Long> {
}
```

Grundlegendes Spring-Repository-Muster, außer dass wir ein Paging- und Sortier-Repository für zusätzliche Funktionen wie ... Paging und Sortieren aktiviert haben :)

Validierung aktivieren

```
package com.mcf7.spring.domain;

import org.springframework.validation.Errors;
import org.springframework.validation.Validator;


public class BeforeCreateBookValidator implements Validator{
    public boolean supports(Class<?> clazz) {
        return Book.class.equals(clazz);
    }

    public void validate(Object target, Errors errors) {
        errors.reject("rejected");
```

```
    }
}
```

Laden von Testdaten

```
package com.mcf7.spring.domain;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class DatabaseLoader implements CommandLineRunner {
    private final BookRepository repository;

    @Autowired
    public DatabaseLoader(BookRepository repository) {
        this.repository = repository;
    }

    public void run(String... Strings) throws Exception {
        Book book1 = new Book();
        book1.setIsbn("6515616168418510");
        book1.setTitle("SuperAwesomeTitle");
        book1.setAuthor("MCF7");
        book1.setDescription("This Book is super epic!");
        repository.save(book1);
    }
}
```

Nur das Laden einiger Testdaten sollte idealerweise nur unter einem Entwicklungsprofil hinzugefügt werden.

Validator hinzufügen

```
package com.mcf7.spring.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.data.rest.core.event.ValidatingRepositoryEventListener;
import org.springframework.data.rest.webmvc.config.RepositoryRestConfigurerAdapter;
import org.springframework.validation.Validator;
import org.springframework.validation.beanvalidation.LocalValidatorFactoryBean;

@Configuration
public class RestValidationConfiguration extends RepositoryRestConfigurerAdapter {

    @Bean
    @Primary
    /**
     * Create a validator to use in bean validation - primary to be able to autowire without
qualifier
     */
    Validator validator() {
        return new LocalValidatorFactoryBean();
    }
```

```
    @Override
    public void configureValidatingRepositoryEventListener(ValidatingRepositoryEventListener
validatingListener) {
        Validator validator = validator();
        //bean validation always before save and create
        validatingListener.addValidator("beforeCreate", validator);
        validatingListener.addValidator("beforeSave", validator);
    }
}
```

Gradle Build-Datei

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'io.spring.gradle:dependency-management-plugin:0.5.4.RELEASE'
    }
}

apply plugin: 'io.spring.dependency-management'
apply plugin: 'idea'
apply plugin: 'java'

dependencyManagement {
    imports {
        mavenBom 'io.spring.platform:platform-bom:2.0.5.RELEASE'
    }
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.springframework.boot:spring-boot-starter-web'
    compile 'org.springframework.boot:spring-boot-starter-data-jpa'
    compile 'org.springframework.boot:spring-boot-starter-data-rest'
    compile 'org.springframework.data:spring-data-rest-hal-browser'
    compile 'org.projectlombok:lombok:1.16.6'
    compile 'org.springframework.boot:spring-boot-starter-validation'
    compile 'org.springframework.boot:spring-boot-actuator'

    runtime 'com.h2database:h2'

    testCompile 'org.springframework.boot:spring-boot-starter-test'
    testCompile 'org.springframework.restdocs:spring-restdocs-mockmvc'
}
```

Spring-Boot Microservice mit JPA online lesen: https://riptutorial.com/de/spring-boot/topic/6557/spring-boot-microservice-mit-jpa

Kapitel 17: Testen im Spring Boot

Examples

So testen Sie eine einfache Spring-Boot-Anwendung

Wir haben ein Beispiel für eine Spring-Boot-Anwendung, die Benutzerdaten in MongoDB speichert, und wir verwenden Rest-Services, um Daten abzurufen

Zunächst gibt es eine Domänenklasse, z. B. POJO

```
@Document
public class User{
    @Id
    private String id;

    private String name;

}
```

Ein entsprechendes Repository basierend auf Spring Data MongoDB

```
public interface UserRepository extends MongoRepository<User, String> {
}
```

Dann unser User Controller

```
@RestController
class UserController {

    @Autowired
    private UserRepository repository;

    @RequestMapping("/users")
    List<User> users() {
        return repository.findAll();
    }

    @RequestMapping(value = "/Users/{id}", method = RequestMethod.DELETE)
    @ResponseStatus(HttpStatus.NO_CONTENT)
    void delete(@PathVariable("id") String id) {
        repository.delete(id);
    }

    // more controller methods
}
```

Und zum Schluss noch unsere Spring Boot Applikation

```
@SpringBootApplication
public class Application {
    public static void main(String args[]){
```

```
     SpringApplication.run(Application.class, args);
    }
}
```

Wenn beispielsweise John Cena, The Rock und TripleHHH die einzigen drei Benutzer in der Datenbank wären, würde eine Anfrage an / Benutzer die folgende Antwort geben:

```
$ curl localhost:8080/users
[{"name":"John Cena","id":"1"},{"name":"The Rock","id":"2"},{"name":"TripleHHH","id":"3"}]
```

Um den Code zu testen, überprüfen wir nun, ob die Anwendung funktioniert

```
@RunWith(SpringJUnit4ClassRunner.class)   // 1
@SpringApplicationConfiguration(classes = Application.class)   // 2
@WebAppConfiguration   // 3
@IntegrationTest("server.port:0")   // 4
public class UserControllerTest {

    @Autowired   // 5
    UserRepository repository;

    User cena;
    User rock;
    User tripleHHH;

    @Value("${local.server.port}")   // 6
    int port;

    @Before
    public void setUp() {
        // 7
        cena = new User("John Cena");
        rock = new User("The Rock");
        tripleHHH = new User("TripleHH");

        // 8
        repository.deleteAll();
        repository.save(Arrays.asList(cena, rock, tripleHHH));

        // 9
        RestAssured.port = port;
    }

    // 10
    @Test
    public void testFetchCena() {
        String cenaId = cena.getId();

        when().
                get("/Users/{id}", cenaId).
        then().
                statusCode(HttpStatus.SC_OK).
                body("name", Matchers.is("John Cena")).
                body("id", Matchers.is(cenaId));
    }

    @Test
    public void testFetchAll() {
```

```
        when().
                get("/users").
        then().
                statusCode(HttpStatus.SC_OK).
                body("name", Matchers.hasItems("John Cena", "The Rock", "TripleHHH"));
    }

    @Test
    public void testDeletetripleHHH() {
        String tripleHHHId = tripleHHH.getId();

        when()
                .delete("/Users/{id}", tripleHHHId).
        then().
                statusCode(HttpStatus.SC_NO_CONTENT);
    }
}
```

Erläuterung

1. Wie bei jedem anderen Spring-basierten Test benötigen wir den `SpringJUnit4ClassRunner` damit ein Anwendungskontext erstellt wird.
2. Die Annotation `@SpringApplicationConfiguration` ähnelt der Annotation `@ContextConfiguration` Sie wird verwendet, um anzugeben, welche Anwendungskontexte im Test verwendet werden sollen. Darüber hinaus wird eine Logik zum Lesen von Spring Boot-spezifischen Konfigurationen, Eigenschaften usw. ausgelöst.
3. `@WebAppConfiguration` muss vorhanden sein, um Spring mitzuteilen, dass ein `WebApplicationContext` für den Test geladen werden soll. Außerdem enthält es ein Attribut zum Angeben des Pfads zum Stammverzeichnis der Webanwendung.
4. `@IntegrationTest` Spring Boot mit, dass der eingebettete Webserver gestartet werden soll. Durch Angabe von Doppelpunkt- oder gleichwertigen Namen / Wert-Paaren kann jede Umgebungsvariable überschrieben werden. In diesem Beispiel überschreibt `"server.port:0"` die Standardporteinstellung des Servers. Normalerweise verwendet der Server die angegebene Portnummer, der Wert 0 hat jedoch eine besondere Bedeutung. Wenn der Wert 0 ist, wird Spring Boot angewiesen, die Ports in der Hostumgebung zu scannen und den Server an einem beliebigen verfügbaren Port zu starten. Dies ist nützlich, wenn verschiedene Dienste auf den Entwicklungsmaschinen und dem Build-Server unterschiedliche Ports belegen, die möglicherweise mit dem Anwendungsport kollidieren. In diesem Fall wird die Anwendung nicht gestartet. Zweitens: Wenn wir mehrere Integrationstests mit unterschiedlichen Anwendungskontexten erstellen, können diese auch kollidieren, wenn die Tests gleichzeitig ausgeführt werden.
5. Wir haben Zugriff auf den Anwendungskontext und können mithilfe von Autowiring Springbeans injizieren.
6. Der `@Value("${local.server.port}”)` wird in die tatsächlich verwendete Portnummer aufgelöst.
7. Wir erstellen einige Entitäten, die wir zur Validierung verwenden können.
8. Die MongoDB-Datenbank wird für jeden Test gelöscht und neu initialisiert, sodass immer ein bekannter Zustand geprüft wird. Da die Reihenfolge der Tests nicht definiert ist, besteht die Möglichkeit, dass der Test testFetchAll () fehlschlägt, wenn er nach dem Test testDeletetripleHHH () ausgeführt wird.
9. Wir weisen Rest Assured an, den richtigen Port zu verwenden. Es handelt sich um ein Open

Source-Projekt, das eine Java-DSL zum Testen unruhiger Dienste bereitstellt
10. Tests werden mithilfe von Rest Assured implementiert. Wir können die Tests mithilfe des TestRestTemplate oder eines anderen http-Clients implementieren. Ich verwende jedoch Rest Assured, da wir mit RestDocs eine präzise Dokumentation schreiben können

Laden verschiedener yaml-Dateien [oder Eigenschaften] oder Überschreiben einiger Eigenschaften

Wenn wir `@SpringApplicationConfiguration` , wird die Konfiguration von `application.yml` [Eigenschaften] verwendet, was in bestimmten Situationen nicht angemessen ist. `@TestPropertySource` die Eigenschaften zu überschreiben, können wir die Annotation `@TestPropertySource` .

```
@TestPropertySource(
        properties = {
                "spring.jpa.hibernate.ddl-auto=create-drop",
                "liquibase.enabled=false"
        }
)
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(Application.class)
public class ApplicationTest{

    // ...

}
```

Wir können das **properties-** Attribut von `@TestPropertySource` , um die `@TestPropertySource` spezifischen **Eigenschaften** zu überschreiben. In obigem Beispiel **überschreiben** wir die Eigenschaft `spring.jpa.hibernate.ddl-auto` , `create-drop` zu `create-drop` . Und `liquibase.enabled` auf `false` .

Laden einer anderen Yml-Datei

Wenn Sie verschiedene **Yml-** Dateien **vollständig** zum Testen laden möchten, können Sie das Attribut **location** auf `@TestPropertySource` .

```
@TestPropertySource(locations="classpath:test.yml")
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(Application.class)
public class ApplicationTest{

    // ...

}
```

Alternativ Optionen

Option 1:

Sie können auch eine andere **Yml-** Datei laden, indem Sie eine **Yml-** Datei in das Verzeichnis **`test > resource`**

Option 2:

Verwenden der Annotation `@ActiveProfiles`

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = Application.class)
@ActiveProfiles("somename")
public class MyIntTest{
}
```

Sie können sehen, dass wir `@ActiveProfiles` Annotation verwenden, und wir übergeben den **Namen** als Wert.

Erstellen Sie eine Datei namens **`application-somename.yml`** und der Test lädt diese Datei.

Testen im Spring Boot online lesen: https://riptutorial.com/de/spring-boot/topic/1985/testen-im-spring-boot

Kapitel 18: ThreadPoolTaskExecutor: Konfiguration und Verwendung

Examples

Anwendungskonfiguration

```
@Configuration
@EnableAsync
public class ApplicationConfiguration{

    @Bean
    public TaskExecutor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(2);
        executor.setThreadNamePrefix("executor-task-");
        executor.initialize();
        return executor;
    }

}
```

ThreadPoolTaskExecutor: Konfiguration und Verwendung online lesen: https://riptutorial.com/de/spring-boot/topic/7497/threadpooltaskexecutor--konfiguration-und-verwendung

Kapitel 19: Vollständig responsive Spring Boot-Webanwendung mit JHipster

Examples

Erstellen Sie Spring Boot App mit jHipster unter Mac OS

Mit jHipster können Sie eine Spring Boot-Webanwendung mit einem REST-API-Backend und einem AngularJS- und Twitter-Bootstrap-Frontend bootstrappen.

Mehr zu jHipster hier: jHipster-Dokumentation

Brühen installieren:

```
 /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Weitere Informationen zur Installation von brew finden Sie hier: Installieren Sie Brew

Installieren Sie Gradle

Gradle ist ein Abhängigkeitsverwaltungs- und Buildsystem.

```
brew install gradle
```

Installiere Git

Git ist ein Versionskontrolle-Tool

```
brew install git
```

Installieren Sie NodeJS

Mit NodeJS haben Sie Zugriff auf npm, den Knotenpaket-Manager, der zur Installation anderer Tools erforderlich ist.

```
brew install node
```

Installieren Sie Yeoman

Yeoman ist ein Generator

```
npm install -g yo
```

Installieren Sie Bower

Bower ist ein Werkzeug zur Verwaltung von Abhängigkeiten

```
npm install -g bower
```

Installieren Sie Gulp

Gulp ist ein Aufgabenläufer

```
npm install -g gulp
```

Installieren Sie den jHipster Yeoman Generator

Dies ist der jHipster-Generator

```
npm install -g generator-jhipster
```

Erstellen Sie eine Anwendung

Öffnen Sie ein Terminalfenster.

Navigieren Sie zu dem Stammverzeichnis, in dem Sie Ihre Projekte aufbewahren. Erstellen Sie ein leeres Verzeichnis, in dem Sie Ihre Anwendung erstellen

```
mkdir myapplication
```

Wechseln Sie in dieses Verzeichnis

```
cd myapplication/
```

Geben Sie zum Generieren Ihrer Anwendung ein

```
yo jhipster
```

Sie werden mit den folgenden Fragen aufgefordert

Welche Art von Anwendung möchten Sie erstellen?

Ihre Art der Anwendung hängt davon ab, ob Sie eine Microservices-Architektur verwenden möchten oder nicht. Eine vollständige Erklärung zu Microservices finden Sie hier, wenn Sie sich nicht sicher sind, ob Sie die Standardeinstellung "Monolithic Application" verwenden.

Wählen Sie standardmäßig *Monolithische Anwendung,* wenn Sie sich nicht sicher sind

Wie lautet Ihr Standard-Java-Paketname?

Ihre Java-Anwendung verwendet dies als Root-Paket.

Welche Art der Authentifizierung möchten Sie verwenden?

Verwenden Sie standardmäßig die sitzungsbasierte *Spring Security,* wenn Sie sich nicht sicher sind

Welche Art von Datenbank möchten Sie verwenden?

Welche Entwicklungsdatenbank möchten Sie verwenden?

Dies ist die Datenbank, die Sie für Ihr "Entwicklungsprofil" verwenden werden. Sie können entweder verwenden:

Verwenden Sie standardmäßig H2, wenn Sie sich nicht sicher sind

H2, läuft im Arbeitsspeicher. Dies ist die einfachste Möglichkeit, JHipster zu verwenden, aber Ihre Daten gehen verloren, wenn Sie Ihren Server neu starten.

Möchten Sie den Hibernate 2nd Level Cache verwenden?

Hibernate ist der von JHipster verwendete JPA-Anbieter. Aus Gründen der Performance empfehlen wir Ihnen dringend, einen Cache zu verwenden und ihn an die Anforderungen Ihrer Anwendung anzupassen. Wenn Sie sich dafür entscheiden, können Sie entweder Ehcache (lokaler Cache) oder Hazelcast (verteilter Cache für die Verwendung in einer Clusterumgebung) verwenden.

Möchten Sie eine Suchmaschine in Ihrer Anwendung verwenden? Elasticsearch wird mit Spring Data Elasticsearch konfiguriert. Weitere Informationen finden Sie in unserem Elasticsearch Guide.

Wählen Sie Nein, wenn Sie sich nicht sicher sind

Möchten Sie gruppierte HTTP-Sitzungen verwenden?

Standardmäßig verwendet JHipster eine HTTP-Sitzung nur zum Speichern der Authentifizierungs- und Autorisierungsinformationen von Spring Security. Sie können natürlich auch mehr Daten in Ihre HTTP-Sitzungen einfügen. Die Verwendung von HTTP-Sitzungen kann zu Problemen führen, wenn Sie in einem Cluster ausgeführt werden, insbesondere wenn Sie keinen Lastenausgleich mit „ständigen Sitzungen“ verwenden. Wenn Sie Ihre Sitzungen innerhalb Ihres Clusters replizieren möchten, wählen Sie diese Option aus, um Hazelcast konfigurieren zu lassen.

Wählen Sie Nein, wenn Sie sich nicht sicher sind

Möchten Sie WebSockets verwenden? Websockets können mit Spring Websocket aktiviert werden. Wir bieten auch ein vollständiges Beispiel, um Ihnen zu zeigen, wie Sie das Framework effizient nutzen können.

Wählen Sie Nein, wenn Sie sich nicht sicher sind

Möchten Sie Maven oder Gradle verwenden? Sie können Ihre generierte Java-Anwendung entweder mit Maven oder Gradle erstellen. Maven ist stabiler und reifer. Gradle ist flexibler, einfacher erweiterbar und mehr Hype.

Wählen Sie *Gradle,* wenn Sie sich nicht sicher sind

Möchten Sie den LibSass Stylesheet-Präprozessor für Ihr CSS verwenden? Node-sass ist eine großartige Lösung, um das Design von CSS zu vereinfachen. Um effizient genutzt zu werden, müssen Sie einen Gulp-Server betreiben, der automatisch konfiguriert wird.

Wählen Sie Nein, wenn Sie sich nicht sicher sind

Möchten Sie die Übersetzungsunterstützung mit Angular Translate aktivieren? Standardmäßig bietet JHipster eine hervorragende Internationalisierungsunterstützung, sowohl auf der Clientseite mit Angular Translate als auch auf der Serverseite. Die Internationalisierung erhöht jedoch den Aufwand und ist etwas komplexer in der Verwaltung. Sie können sich also entscheiden, diese Funktion nicht zu installieren.

Wählen Sie Nein, wenn Sie sich nicht sicher sind

Welche Testframeworks möchten Sie verwenden? Standardmäßig bietet JHipster Java-Einheiten- / Integrationstests (mithilfe der Spring-JUnit-Unterstützung) und JavaScript-Einheitentests (mithilfe von Karma.js) an. Optional können Sie Unterstützung für Folgendes hinzufügen:

Wählen Sie keine, wenn Sie sich nicht sicher sind. Sie haben standardmäßig Zugriff auf junit und Karma.

Vollständig responsive Spring Boot-Webanwendung mit JHipster online lesen: https://riptutorial.com/de/spring-boot/topic/6297/vollstandig-responsive-spring-boot-webanwendung-mit-jhipster

Kapitel 20: Zwischenspeicherung mit Redis mit Spring Boot für MongoDB

Examples

Warum zwischenspeichern?

Leistung ist heute eine der wichtigsten Kennzahlen, die wir beim Entwickeln eines Webservices / einer Anwendung auswerten müssen. Die Kundenbindung ist für jedes Produkt von entscheidender Bedeutung. Aus diesem Grund ist es äußerst wichtig, die Leistung zu verbessern und die Ladezeiten der Seiten zu reduzieren.

Wenn Sie einen Webserver ausführen, der mit einer Datenbank interagiert, können seine Vorgänge zu einem Engpass werden. MongoDB ist hier keine Ausnahme, und wenn unsere MongoDB-Datenbank größer wird, können sich die Dinge wirklich verlangsamen. Dieses Problem kann sich sogar verschlimmern, wenn der Datenbankserver vom Webserver getrennt wird. In solchen Systemen kann die Kommunikation mit der Datenbank einen großen Overhead verursachen.

Glücklicherweise können wir eine Methode namens Caching verwenden, um die Dinge zu beschleunigen. In diesem Beispiel werden wir diese Methode vorstellen und sehen, wie wir sie verwenden können, um die Leistung unserer Anwendung mithilfe von Spring Cache, Spring Data und Redis zu verbessern.

Das Basissystem

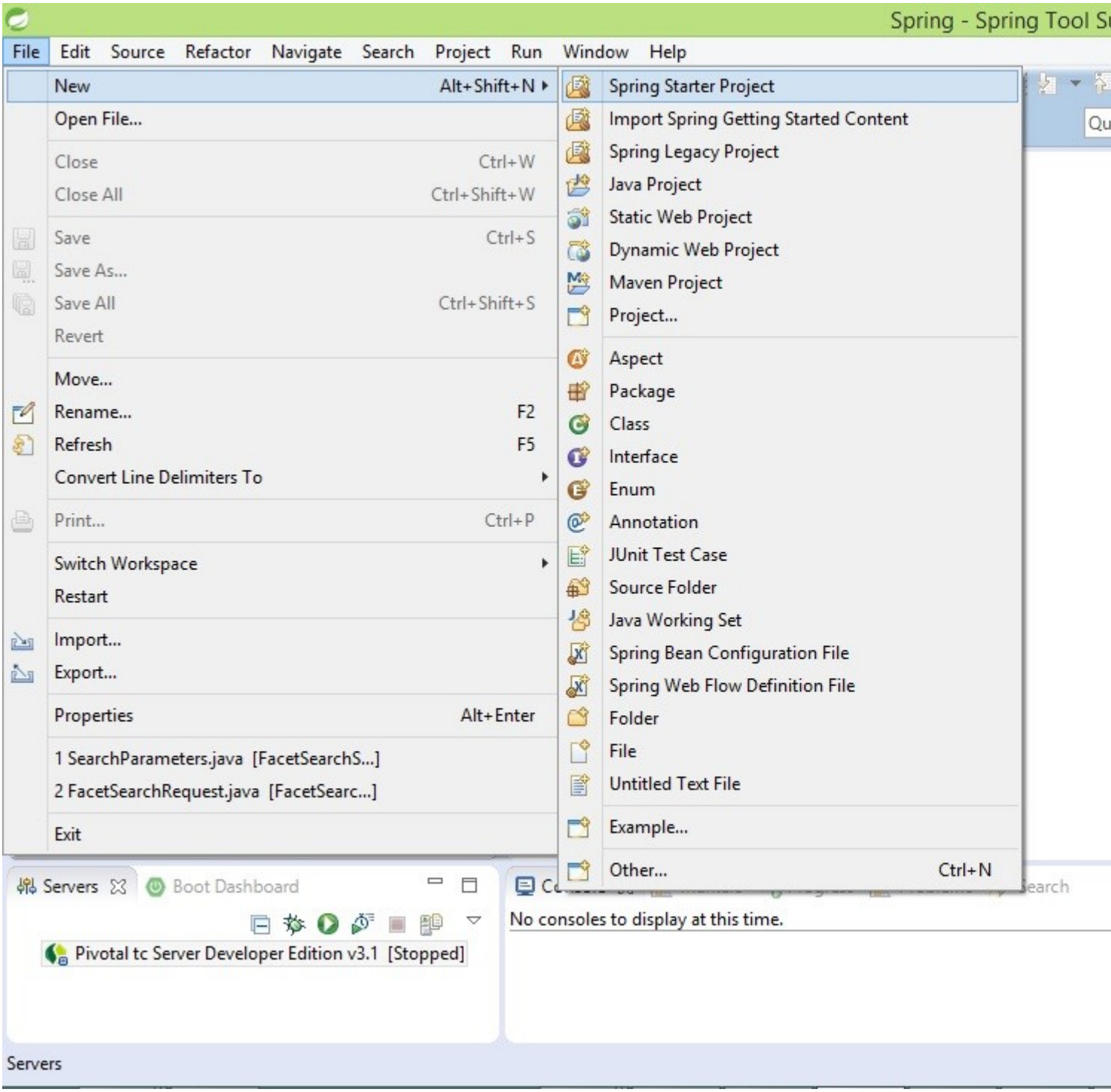
Als ersten Schritt erstellen wir einen grundlegenden Webserver, der Daten in MongoDB speichert. Für diese Demonstration nennen wir sie "schnelle Bibliothek". Der Server verfügt über zwei grundlegende Vorgänge:

`POST /book` : Dieser Endpunkt erhält den Titel, den Autor und den Inhalt des Buches und erstellt einen Bucheintrag in der Datenbank.

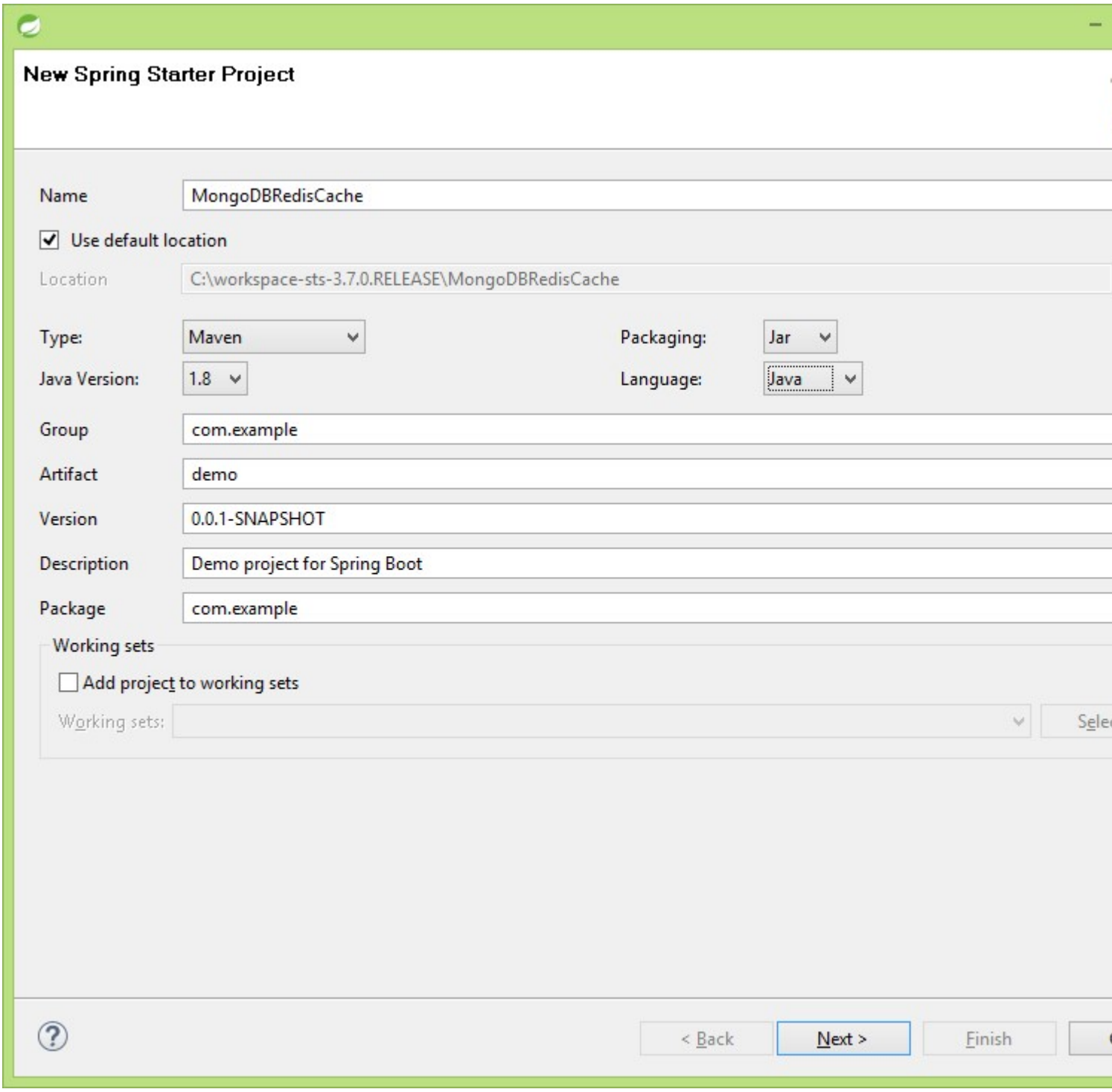
`GET /book/ {title}` : Dieser Endpunkt erhält einen Titel und gibt seinen Inhalt zurück. Wir gehen davon aus, dass Titel Bücher eindeutig identifizieren (daher gibt es nicht zwei Bücher mit demselben Titel). Eine bessere Alternative wäre natürlich die Verwendung einer ID. Der Einfachheit halber verwenden wir jedoch einfach den Titel.

Dies ist ein einfaches Bibliothekssystem, wir werden jedoch später erweiterte Funktionen hinzufügen.

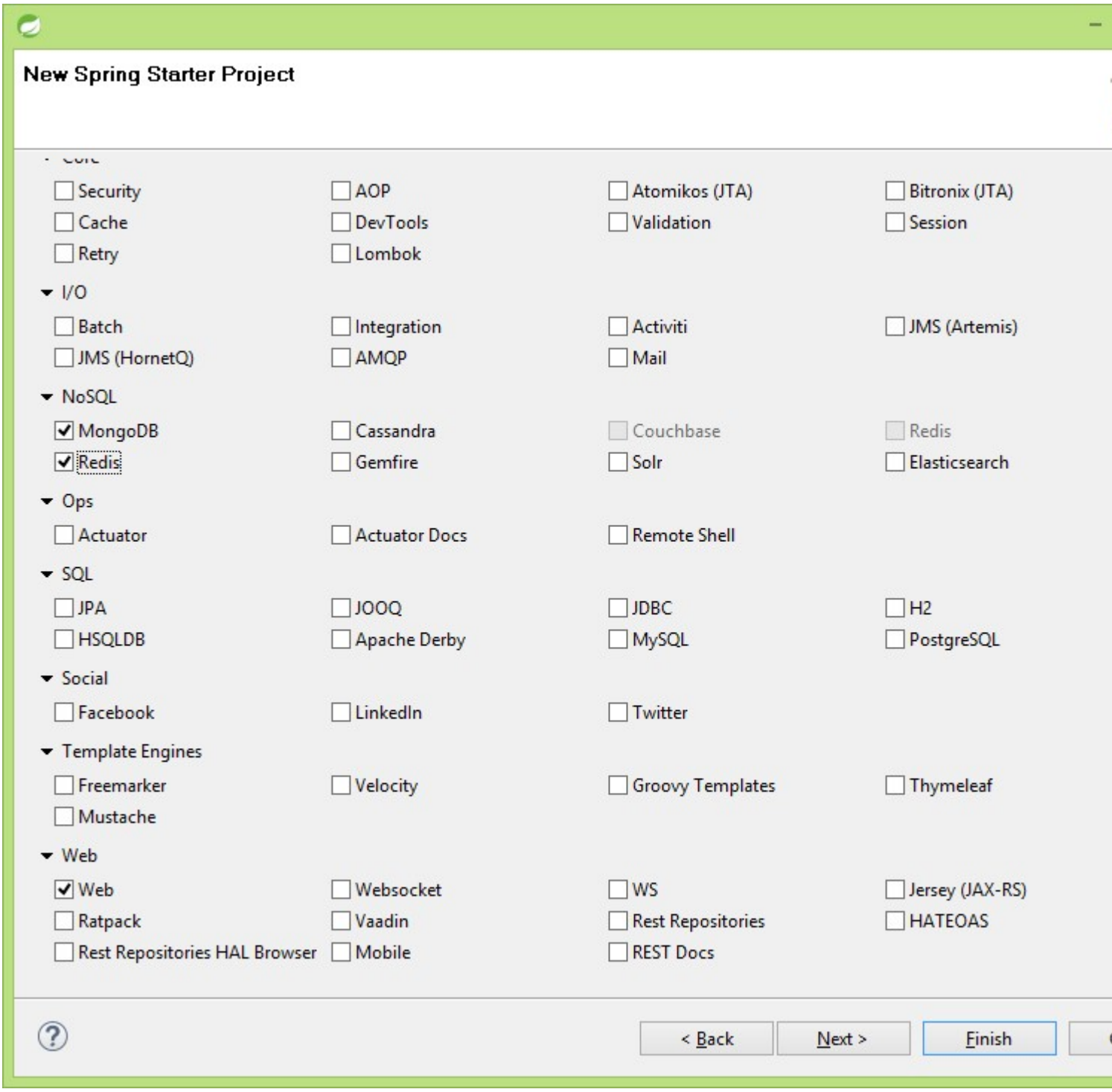
Jetzt erstellen wir das Projekt mit Spring Tool Suite (mit Eclipse erstellen) und Spring Starter Project



Wir bauen unser Projekt mit Java und für den Aufbau verwenden wir maven, wählen Sie Werte aus und klicken Sie auf Weiter



Wählen Sie im Web-Modul MongoDB, Redis aus NOSQL und Web aus und klicken Sie auf Fertig stellen. Wir verwenden Lombok für die automatische Generierung von Setters und Get-Werten von Modellwerten. Daher müssen wir die Lombok-Abhängigkeit zum POM hinzufügen


New Spring Starter Project
Security
AOP
Atomikos (JTA)
Bitronix (JTA)
Cache
DevTools
Validation
Session
Retry
Lombok
I/O
Batch
Integration
Activiti
JMS (Artemis)
JMS (HornetQ)
AMQP
Mail
NoSQL
MongoDB
Cassandra
Couchbase
Redis
Redis
Gemfire
Solr
Elasticsearch
Ops
Actuator
Actuator Docs
Remote Shell
SQL
JPA
JOOQ
JDBC
H2
HSQLDB
Apache Derby
MySQL
PostgreSQL
Social
Facebook
LinkedIn
Twitter
Template Engines
Freemarker
Velocity
Groovy Templates
Thymeleaf
Mustache
Web
Web
Websocket
WS
Jersey (JAX-RS)
Ratpack
Vaadin
Rest Repositories
HATEOAS
Rest Repositories HAL Browser
Mobile
REST Docs
< Back
Next >
Finish

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-mongodb</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

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

MongoDbRedisCacheApplication.java enthält die Hauptmethode, mit der Spring Boot Application add ausgeführt wird

Erstellen Sie eine Modellklasse Book, die ID, Buchtitel, Autor, Beschreibung und Annotation mit @Data enthält, um automatische Setter und Getter aus dem Jar-Projekt lombok zu erstellen

```
package com.example;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.index.Indexed;
import lombok.Data;
@Data
public class Book {
    @Id
    private String id;
    @Indexed
    private String title;
    private String author;
    private String description;
}
```

Spring Data erstellt alle grundlegenden CRUD-Operationen automatisch für uns. Erstellen wir also BookRepository.Java, das das Buch nach Titel findet und das Buch löscht

```
package com.example;
import org.springframework.data.mongodb.repository.MongoRepository;
public interface BookRepository  extends MongoRepository<Book, String>
{
    Book findByTitle(String title);
    void delete(String title);
}
```

Erstellen Sie einen WebservicesController, der Daten in MongoDB speichert und Daten über idTitle (@PathVariable String title) abruft.

```
package com.example;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class WebServicesController {
    @Autowired
    BookRepository repository;
    @Autowired
    MongoTemplate mongoTemplate;
    @RequestMapping(value = "/book", method = RequestMethod.POST)
    public Book saveBook(Book book)
    {
        return repository.save(book);
    }
    @RequestMapping(value = "/book/{title}", method = RequestMethod.GET)
    public Book findBookByTitle(@PathVariable String title)
    {
        Book insertedBook = repository.findByTitle(title);
        return insertedBook;
    }
}
```

Zwischenspeicher hinzufügen Bisher haben wir einen grundlegenden Bibliotheks-Webdienst erstellt, der aber überhaupt nicht erstaunlich schnell ist. In diesem Abschnitt versuchen wir, die findBookByTitle () -Methode zu optimieren, indem die Ergebnisse zwischengespeichert werden.

Um eine bessere Vorstellung davon zu bekommen, wie wir dieses Ziel erreichen, gehen wir zurück zum Beispiel der Menschen, die in einer traditionellen Bibliothek sitzen. Nehmen wir an, sie möchten das Buch mit einem bestimmten Titel finden. Zuerst schauen sie sich am Tisch um, ob sie ihn schon dorthin gebracht haben. Wenn ja, ist das großartig! Sie hatten gerade einen Cache-Treffer, der ein Element im Cache findet. Wenn sie es nicht gefunden haben, hatten sie einen Cache-Fehler, was bedeutet, dass sie das Element nicht im Cache gefunden haben. Im Falle eines fehlenden Artikels müssen sie in der Bibliothek nach dem Buch suchen. Wenn sie es finden, behalten sie es auf ihrem Tisch oder fügen es in den Cache ein.

In unserem Beispiel folgen wir genau demselben Algorithmus für die findBookByTitle () -Methode. Wenn Sie nach einem Buch mit einem bestimmten Titel gefragt werden, suchen wir es im Cache. Wenn nicht gefunden, suchen wir im Hauptspeicher nach unserer MongoDB-Datenbank.

Redis verwenden

Durch das Hinzufügen von Spring-Boot-Daten-Redis zu unserem Klassenpfad kann der Spring-Boot seine Magie vollbringen. Es erstellt alle erforderlichen Vorgänge durch automatische Konfiguration

Lassen Sie uns nun die Methode mit einer Zeile unter der Überschrift kommentieren, damit der

Sprung booten kann

```
@Cacheable (value = "book", key = "#title")
```

Um aus dem Cache zu löschen, wenn ein Datensatz gelöscht wird, kommentieren Sie ihn einfach mit der unteren Zeile in BookRepository und überlassen Sie Spring Boot die Cache-Löschung für uns.

```
@CacheEvict (value = "book", key = "#title")
```

Um die Daten zu aktualisieren, müssen Sie der Methode eine Zeile unterhalb der Zeile hinzufügen und den Spring Boot-Handle zulassen

```
@CachePut(value = "book", key = "#title")
```

Den vollständigen Projektcode finden Sie bei GitHub

Zwischenspeicherung mit Redis mit Spring Boot für MongoDB online lesen: https://riptutorial.com/de/spring-boot/topic/6496/zwischenspeicherung-mit-redis-mit-spring-boot-fur-mongodb

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit Spring-Boot	Andy Wilkinson, Brice Roncace, Community, GVArt, imdzeeshan, ipsi, kodiak, loki2302, M. Deinum, Marvin Frommhold, Matthew Fontana, MeysaM, Misa Lazovic, Moshiour, Nikem, pinkpanther, rajadilipkolli, RamenChef, Ronnie Wang, Slava Semushin, Szobi, Tom
2	Bereitstellen einer Beispielanwendung mithilfe von Spring-Boot auf Amazon Elastic Beanstalk	Praveen Kumar K S
3	Controller	Amit Gujarathi
4	Eine Spring-Boot-Anwendung an MySQL anschließen	koder23, sunkuet02
5	Erstellen und Verwenden mehrerer application.properties-Dateien	Patrick
6	Federstiefel + Spring Data JPA	dunni, Johir, naXa, Sanket, Sohlowmawn, sunkuet02
7	Installieren der Spring Boot-CLI	Robert Thornton
8	REST-Dienste	Andy Wilkinson, CAPS LOCK, g00glen00b, Johir, Kevin Wittek, Manish Kothari, odedia, Ronnie Wang, Safwan Hijazi, shyam, Soner
9	Scannen von Paketen	Tom
10	Spring Boot + JPA + REST	incomplete-co.de
11	Spring Boot + Ruhezustand + Web-Benutzeroberfläche (Thymeleaf)	ppeterka, rajadilipkolli, Tom

12	Spring Boot + Spring Data Elasticsearch	sunkuet02
13	Spring Boot-Hibernate-REST-Integration	Igor Bljahhin
14	Spring-Boot + JDBC	Moshe Arad
15	Springboot + JPA + MongoDB	Andy Wilkinson, Matthew Fontana, Rakesh, RubberDuck, Stephen Leppik
16	Spring-Boot Microservice mit JPA	Matthew Fontana
17	Testen im Spring Boot	Ali Dehghani, Aman Tuladhar, rajadilipkolli, Slava Semushin
18	ThreadPoolTaskExecutor: Konfiguration und Verwendung	Rosteach
19	Vollständig responsive Spring Boot-Webanwendung mit JHipster	anataliocs, codependent
20	Zwischenspeicherung mit Redis mit Spring Boot für MongoDB	rajadilipkolli