Spring Boot Jumpstart

Setup Projekt

Analog dem Spring Hello World Projekt erstellen wir ein Spring Boot Projekt mit Rest Support. Die Screenshots zeigen die Variante mit Eclipse, Spring Initializer und Maven: Falls Sie den JumpStart mit Maven und Eclipse anwenden möchten so folgen Sie dieser Anweisung analog dem zuvor erstellten Hello Maven Projekt. Sie können die Namen und Packages beliebig anpassen, folgen Sie aber immer einer einheitlichen Struktur.

AutoCompleteController

Erstellen Sie die Klasse AutoCompleteController z.B. im Package ch.std.jumpstart.rest und definieren Sie den REST Endpoint via @RestController Annotation:package ch.std.jumpstart.rest;

import java.util.List;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class CityAutoCompleteController {
 @GetMapping("/rest/auto/cities")
 public List<String> autoComplete(@RequestParam(value = "value", defaultValue="") String value) {
  List<String> cities = new CitiesBean().find(value);
  return cities;
 }
}Programmieren Sie das CitiesBean und definieren Sie einige Städte oder Gemeinden der Schweiz als static List initialisiert in einem static Block:package ch.std.jumpstart.beans;

import java.util.ArrayList;
import java.util.List;

public class CitiesBean {
 private static List<String> cities;
 static {
  cities = new ArrayList<>();
 cities.add("Aarau");
  ...
 }

 public List<String> find(String find) {
  List<String> foundCities = new ArrayList<>();
  ...
  return foundCities;
  }
}Programmieren Sie die Methode List find(String find) aus, so dass der gesuchte Value Case Insensitive mit einem oder mehrerer Elemente via Contains matched.Starten Sie die Applikation und verifizieren Sie das Resultat via Browser z.B. wie folgt:

MockMvc

Wir programmieren nun den ersten Rest Controller Unit Test.Die Testklasse wurde ja schon via spring.io erstellt. Passen Sie den Test nun gemäss dem Script an:package ch.std.jumpstart;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.skyscreamer.jsonassert.JSONAssert;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;

import ch.std.jumpstart.rest.CityAutoCompleteController;

@ExtendWith(SpringExtension.class)
@WebMvcTest(CityAutoCompleteController.class)
public class JumpstartApplicationTests {
 @Autowired
 private MockMvc mvc;

 @Test
 public void contextLoads() {
 }

 @Test
 public void testCityAutoCompleteController() throws Exception {
  MvcResult mvcResult = mvc.perform(get("/rest/auto/cities?value=Bern").contentType(MediaType.APPLICATION_JSON))
  .andExpect(status().isOk()).andReturn();
  String expected = "[Bern]";
  String actual = mvcResult.getResponse().getContentAsString();
 JSONAssert.assertEquals(expected, actual, false);
 }
}Führen Sie den Unit Test aus, er sollte korrekt (grün) ausgeführt werden.Weitere Infos zum Testen und den Annotations finden wir unter: https://spring.io/guides/gs/testing-web/

JPA Entity

Unsere Städte sollen nun aus einer relationalen Datenbank gelesen werden. Den Datenbank Zugriff bauen wir mit dem Java Persistence API (JPA).In einem ersten Schritt programmieren wir die Entitäts Klasse City gemässe dem Skript:package ch.std.jumpstart.jpa;

import java.io.Serializable;

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

@Entity
@Table(name = "city")
public class City implements Serializable {

 private static final long serialVersionUID = 2347460802149202192L;

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

 @Column(name = "name", unique=true)
 private String name;

 public City() {
 }

 public City(String name) {
  this.name = name;
 }

 public Long getId() {
  return id;
 }

 public String getName() {
  return name;
 }

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

 @Override
 public String toString() {
  return "City [id=" + id + ", name=" + name + "]";
 }

}Die JPA Entity Klasse wird leider nicht kompilieren, weil die JPA Unterstützung im Projekt fehlt.JPA wird als Spring Starter Projekt als Maven Dependeny via Datei pom.xml definiert:<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>Damit sollte die JPA Klasse kompilieren, aber die Applikation startet nicht und gibt den folgenden Fehler:**********
APPLICATION FAILED TO START
****

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class

Action:

Consider the following:
  If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.
  If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).Um das Problem zu beheben ergänzen wir die Datei pom.xml mit der folgenden Dependency:<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>
</dependency>Damit verwendet Spring Boot per default die In-Memory Datenbank H2.Mit den Starter Dependencies werden die Abhängigkeiten zu anderen Libraries gekapselt und damit vereinfacht.Zur Zeit existieren ca. 30 Spring Boot Starter Dependencies. Eine Liste hierzu findet man hier:https://github.com/spring-projects/spring-boot/tree/master/spring-boot-project/spring-boot-starters Wir wenden die folgenden JPA Annotations an:@Entity, damit werden Java Klassen als Persistenz-Objekte definiert, default ist jeder Entität eine Datenbank Tabelle zugeordnet.@Id, definiert den Primary Key mit Strategie Auto Inkrement.@Column, definiert eine Spalte in der Tabelle, das Type Mapping ist hier Default. Das Attribute unique definiert die Index Beziehung.Detaillierte Infos zu JPA finden Sie im Internet, z.B. unter dem Link https://www.tutorialspoint.com/de/jpa/jpa_introduction.htm

JPA Repository

Spring Data vereinfacht den Zugriff auf die Datenbank mit SQL über Repositories.Repositories sind eigentlich nur Interfaces, die Funktionen für den Zugriff via gekapseltem SQL anbieten, generell basierend auf Prepared Statements.Die Grundidee ist die, dass man bereits aus der Signatur einer Interface(!)-Methode den bzw. die Namen des/der Query-Parameter ableiten kann. Spring liefert zur Laufzeit eine entsprechende Implementierung, die die entsprechende Query über das JPA Criteria API aufbaut und auch ausführt. Damit ist ein Grossteil von Queries in der Praxis sehr schnell formuliert. In unserem Beispiel definieren wir die Methode findByName() für die exakte Suche nach einer City an.Zur Methode findByNameLike() definieren wir die Query via @Query Annotation selber.Integrieren Sie das folgende Repository in das Jumpstart Projekt:package ch.std.jumpstart.repository;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import ch.std.jumpstart.jpa.City;

public interface CityRepository extends JpaRepository<City, Long> {

    City findByName(String name);

    @Query("SELECT c FROM City c WHERE c.name LIKE %?1%")
 List<City> findByNameLike(String name);

}Das folgende UML

Klassendiagramm zeigt die Repository Hierarchie auf:

JPA Controller

Wir bauen nun einen eigenen REST Controller, welcher auf Cities die Autocomplete Funktion über die Datenbank und dem CityRepository anwendet. Achten Sie darauf, dass wir einen neuen REST Endpoint verwenden, da solcher mit dem bestehenden In-Memory AutoComplete Controller kollidiert. Zur Zeit arbeiten wir mit der In-Memory Datenbank H2. Die CityRepository Instanz wird durch Spring via Constructor Injection automatisch zugewiesen.Integrieren Sie die Klasse CityAutoJpaCompleteController gemäss dem folgenden Listing in das Jumpstart Projekt:package ch.std.jumpstart.rest;

import java.util.List;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import ch.std.jumpstart.jpa.City;
import ch.std.jumpstart.repository.CityRepository;

@RestController
public class CityAutoJpaCompleteController {

 private CityRepository cityRepository;

 public CityAutoJpaCompleteController(CityRepository cityRepository) {
  this.cityRepository = cityRepository;
 }

 @GetMapping("/rest/auto/citiesjpa")
 public City[] autoComplete(@RequestParam(value = "value", defaultValue = "") String value) {
  List<City> cities = cityRepository.findByNameLike(value);
  return cities.toArray(new City[0]);
 }
}

CommandLineRunner

Mit dem CommandLineRunner laden wir ein Bean, welches die Programmlogik nach dem Laden des Application Context ausführen kann. Damit laden wir die Städte des CitiesBean in die In-Memory H2 Datenbank. Erweitern Sie die Klasse JumpstartApplication gemäss dem folgenden Listing:package ch.std.jumpstart;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Profile;

import ch.std.jumpstart.beans.CitiesBean;
import ch.std.jumpstart.jpa.City;
import ch.std.jumpstart.repository.CityRepository;

@SpringBootApplication
public class JumpstartApplication {

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

 @Bean
 @Profile("!test")
 public CommandLineRunner setup(CityRepository repository) {
  return (args) -> {
   for (String city : CitiesBean.getCities()) {
  repository.save(new City(city));
   }
  };
 }

}Starten Sie die Jumpstart Applikation testen Sie solche mit der folgenden URL http://localhost:8080/rest/auto/citiesjpa. Das folgene oder ein ähnliches Resultat sollte der Browser anzeigen:Leider funktioniert unser Unit Test nicht mehr. Der Maven Build via Konsolemvn clean installergibt leider einen Fehler. Dies ist in der Praxis jederzeit möglich wenn nicht sogar die Regel, dass Code Anpassungen zu Testfehlern führen. Der Fehler liegt am Datenbank Zugriff via CommandLineRunner, da der Test nicht korrekt aufgesetzt ist. Wir umgehen das Problem aktuell, indem wir den CommandLineRunner deaktvieren via Test Profile. Ergänzen Sie die Klasse JumpstartApplicationTests mit der folgenden Annotation:@ActiveProfiles("test")Mit der @Profile Annotation !test schliessen wir den CommandLineRunner gezielt für gesuchte Tests aus, indem solche @ActiveProfiles anwenden.Verifizieren Sie via Konsole ob der Build und damit der Test korrekt funktionieren. Dies sollte wieder der Fall sein.

H2 Console

Die In-Memory Datenbank H2 bietet eine Console und damit den Zugriff auf die H2 Datenbank. Die H2 Konsole aktivieren wir über die folgende Maven Dependency (pom.xml):<dependency>
<groupId>com.h2database</groupId>
<artifactId>h2</artifactId>
<scope>runtime</scope>
  </dependency>Die

Dependency haben wir schon definiert, aber nicht den scope runtime.Weiter passen wir die Datei application.properties wie folgt
an:spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
spring.datasource.url=jdbc:h2:mem:testdbWir starten die Applikation neu und öffnen im Browser die folgende URL http://localhost:8080/h2-console. Die H2 Console Login Site wird angezeigt:Passen Sie die JDBC URL an zu "jdbc:h2:mem:testdb" und verbinden Sie mit der H2 Datenbank (kein Passwort erforderlich):Wir haben nun den Zugriff auf die H2 Datenbank und können via SQL die Daten lesen und manipulieren.

Integration Test

Spring Integration Tests ermöglichen das Testen der Applikation auf der Stufe der REST Schnittstelle mit einem echt gestarteten Backend und Spring Context.Das folgenden Listing zeigt die Klasse JumpstartApplicationIntegrationTests, welche einen Integrationstest definiert:package ch.std.jumpstart;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;

import ch.std.jumpstart.jpa.City;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class JumpstartApplicationIntegrationTests {

 @Autowired
 private TestRestTemplate restTemplate;

 @Test
 public void contextLoads() {
 }

 @Test
 public void testCityAutoCompleteController() throws Exception {
  City[] cities = this.restTemplate.getForObject("/rest/auto/citiesjpa?value=Bern", City[].class);
 assertNotNull(cities);
  assertEquals(1, cities.length);
  City city = cities[0];
 assertNotNull(city);
  assertEquals("Bern", city.getName());
 }
}Bei diesem Test handelt es sich um einen Integrationstest. Solcher wird über die Annotation @SpringBootTest definiert. Mit dem RANDOM_PORT Web Environment wird für diesen Test eine komplette Server Instanz gestartet. Der verwendete Port kann über die @LocalServerPort Annotation einem Attribute zugewiesen werden. Mit der Klasse TestRestTemplate wenden wir den Rest Controller über die dynamisch erstellte URL an. TestRestTemplate basiert auf dem Spring Boot RestTemplate, welche als Rest Client in Anwendungen eingesetzt wird.Integrieren Sie den Test in das Projekt Jumpstart und testen Sie solchen via Maven Clean Install und Terminal.

SQL Log und Trace

JPA erstellt im Hintergrund teils kopmlexe SQL Anweisungen. Solche wollen wir im Log sichten können. Wir schalten über die application.properties das SQL Log mit Level DEBUG ein. Damit werden die SQL Statements im Log angezeigt:
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACEMit dem Start der Applikation oder mit dem Ausführen der Tests werden die generierten SQL Statements angezeigt:Eine Übersicht über die Spring Common Application Properties findet man unter dem
Link:https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html.

Data Transfer Object (DTO)

REST Controllers dürfen keine Domain Instanzen als Request empfangen oder als Response zurückgeben. Wir realisieren die Trennung für den JumpStart mit dem Data Transfer Object (DTO):package ch.std.jumpstart.dto;

public class CityDTO {

 private Long id;

 private String name;

 public CityDTO() {
 }

 public Long getId() {
  return id;
 }

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

 public String getName() {
  return name;
 }

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

 @Override
 public String toString() {
  return "CityDTO [id=" + id + ", name=" + name + "]";
 }

}Data Transfer Objects (DTO's) wurden von Martin Fowler beschrieben und entsprechen einem Softwarepattern. Integrieren Sie die CityDTO Klasse in das

Jumpstart Projekt. Aktuell ändert sich an der Funktionalität noch nichts.

ModelMapper

Die Konvertierung zwischen Domain und DTO Instanzen kann selber im Java Programmcode oder z.B. über ein Tool wie dem ModelMapper programmiert werden. Wir verwenden für unser Mapping zwischen City und CityDTO Instanzen den ModelMapper (siehe auch http://modelmapper.org/.Zuerst definieren wir die Maven Dependency wie folgt in der Datei pom.xml:<dependency>
 <groupId>org.modelmapper</groupId>
 <artifactId>modelmapper</artifactId>
 <version>2.3.5</version>
</dependency>ModelMapper ist keine Spring Boot Ressource und muss dem Spring Framework zugeführt werden als Spring Bean. Hierzu programmieren wir die folgenden Configuration Klasse:package ch.std.jumpstart.config;

import org.modelmapper.ModelMapper;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ApplicationConfig {

  @Bean
  public ModelMapper modelMapper() {
 ModelMapper modelMapper = new ModelMapper();
  return modelMapper;
 }

}Wir integrieren das Entity to DTO Mapping in unseren CityAutoJpaCompleteController REST Service:package ch.std.jumpstart.rest;

import java.util.List;

import org.modelmapper.ModelMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import ch.std.jumpstart.dto.CityDTO;
import ch.std.jumpstart.jpa.City;
import ch.std.jumpstart.repository.CityRepository;

@RestController
public class CityAutoJpaCompleteController {

 private CityRepository cityRepository;
 private ModelMapper modelMapper;

 public CityAutoJpaCompleteController(CityRepository cityRepository, ModelMapper modelMapper) {
  this.cityRepository = cityRepository;
 this.modelMapper = modelMapper;
 }

 @GetMapping("/rest/auto/citiesjpa")
 public CityDTO[] autoComplete(@RequestParam(value = "value") String value) {
 List<City> cities = (List<City>) cityRepository.findByNameLike(value);
  return cities.stream().map(city -> convertToDTO(city)).toArray(CityDTO[]::new);
 }

 private CityDTO convertToDTO(City city) {
  return this.modelMapper.map(city, CityDTO.class);
 }
}Führen Sie die Applikation aus und verifizieren Sie die URL http://localhost:8080/rest/auto/citiesjpa.Korrigieren Sie die Unit Tests, so dass alles korrekt via "mvn clean install" funktioniert.

CityService

REST Controllers sollten die Business Logik an Service Components delegieren. Wir realisieren dies mit dem CityService:package ch.std.jumpstart.service;

import java.util.List;

import org.springframework.stereotype.Service;

import ch.std.jumpstart.jpa.City;
import ch.std.jumpstart.repository.CityRepository;

@Service
public class CityService {

 private CityRepository cityRepository;

 public CityService(CityRepository cityRepository) {
  this.cityRepository = cityRepository;
 }

 public List<City> find(String value) {
  return cityRepository.findByNameLike(value);
 }

}CityService ist ein Spring @Service und wird via Dependency Injection automatisch geladen. Der City Service greift via Repository auf die Datenbank zu.Den REST Service CityAutoJpaCompleteController bauen wir so um, dass er nicht mehr direkt auf die Datenbank (Repository) zugreift, sondern indirekt via CityService:package ch.std.jumpstart.rest;

import java.util.List;

import org.modelmapper.ModelMapper;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import

ch.std.jumpstart.dto.CityDTO;
import ch.std.jumpstart.jpa.City;
import ch.std.jumpstart.service.CityService;

@RestController
public class CityAutoJpaCompleteController {

 private CityService cityService;

 private ModelMapper modelMapper;

 public CityAutoJpaCompleteController(CityService cityService, ModelMapper modelMapper) { // autowired
  this.cityService = cityService;
 this.modelMapper = modelMapper;
 }

 @GetMapping("/rest/auto/citiesjpa")
 public CityDTO[] autoComplete(@RequestParam(value = "value", defaultValue="") String value) {
  List<City> cities = (List<City>) this.cityService.find(value);
  return cities.stream().map(city -> convertToDTO(city)).toArray(CityDTO[]::new);
 }

 private CityDTO convertToDTO(City city) {
  return this.modelMapper.map(city, CityDTO.class);
 }
}Integrieren Sie den CityService in die Jumpstart Applikation und testen Sie solche aus. Verifizieren Sie, dass alle Unit Tests korrekt funktionieren.

Run Code after Startup

Spring Boot ermöglicht diverse Optionen um Programmcode nach dem Application Startup auszuführen. In unserem Fall nutzen wie die @EventListener Annotation und das ApplicationReadyEvent: package ch.std.jumpstart;

import java.net.InetAddress;
import java.net.URL;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Profile;
import org.springframework.context.event.EventListener;
import org.springframework.core.env.Environment;

import ch.std.jumpstart.beans.CitiesBean;
import ch.std.jumpstart.jpa.City;
import ch.std.jumpstart.repository.CityRepository;

@SpringBootApplication
public class JumpstartApplication {

 Logger logger = LoggerFactory.getLogger(JumpstartApplication.class);

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

 public JumpstartApplication(Environment environment) {
  this.environment = environment;
 }

 @Bean
 @Profile("!test")
 public CommandLineRunner setup(CityRepository repository) {
  return (args) -> {
   for (String city : CitiesBean.getCities()) {
    repository.save(new City(city));
   }
  };
 }

@Value("#{servletContext.contextPath}")
private String contextPath;

 private Environment environment;

 @EventListener(ApplicationReadyEvent.class)
 public void onApplicationReadyEvent(ApplicationReadyEvent event) {
  try {
   String ip = InetAddress.getLocalHost().getHostAddress();
   int port = environment.getProperty("server.port", Integer.class, 8080);
   String host = InetAddress.getLocalHost().getHostName();
   //String contextPath = environment.getProperty("server.servlet.context-path", String.class, "/");
 String protocol = "http";
   URL accessURL = new URL(protocol, host, port, contextPath);
   logger.info("Access URL = " + accessURL);
  } catch (Exception e) {
   logger.error(e.getMessage(), e);
  }
 }

}Das Listing zeigt den Zugriff auf Application Properties via @Value Annotation oder via Environment Map.Weiter definieren wir in den Application Properties den Server Port und Context:server.port=8080
server.servlet.context-path=/jumpstartHier schreiben wir die Access URL ins Log. Hierzu haben wir die application.properties angepasst und den Port sowie den Context Path definiert. Altenativ gibt es z.B. @PostConstruct Annotation oder die @Component Constructor Injection via @Autowired Annotation angewendet auf den Konstruktor. Bisher haben wir Programmcode via CommandLineRunner beim Startup ausgeführt.Mit dem Start der Jumpstart Applikation wird die Access URL korrekt beim Startup angezeigt:Damit schliessen wir den Jumpstart ab.

Lösung

Eine mögliche Lösung finden Sie als Maven Projekt jumpstart.zip.Ein Beispiel mit Pageable REST Support findet man unter dem Link jumpstart-pageable.zip.

Kontakt

Simtech AG
Finkenweg 23
3110 Münsingen
Schweiz

Impressum

Das Copyright für sämtliche Inhalte dieser Website liegt bei Simtech AG, Schweiz.
Beachten Sie auch unsere Hinweise zum Urheberrecht, Datenschutz und Haftungsauschluss.
Jeder Hinweis auf Fehler nehmen wir gerne entgegen.

Copyright

2024 Simtech AG, All rights reserved, Powered by stack.ch written in Golang by Daniel Schmutz

https://www.simtech-ag.ch/try