Spring Boot 2 Blog

Eigentlich wird Spring Boot immer wieder als Microservices Framework bezeichnet, doch dies ist nicht ganz korrekt. Spring Boot ist eher eine pragmatische Umsetzung des Spring Frameworks und bietet etliche Vereinfachungen. Seit der Lancierung im Jahr 2013 mit Version 1.x wurde 2018 die Version 2.x freigegeben. In diesem Blog lernen Sie die Key Features von Spring Boot 2 kennen und gewinnen damit eine kompakte Übersicht zu diesem effizienten Ökosystem.

Was ist Spring Boot

Spring Boot basiert auf dem Spring Framework 5 und bedingt Java 8. Mit Spring Boot lassen sich Java Web Anwendungen bauen, die als Single Jar ausführbar sind: Single Fat Jar Web oder Fat War DeploymentInsbesondere der Single Jar Ansatz ist für den Betrieb via Docker Container der interessante Ansatz. Spring Boot verzichtet auf XML Konfigurationen und hat zum Ziel, den Konfigurationsaufwand minim zu halten. Spring Boot arbeitet mit dem "Opinionated Defaults Configuration" Ansatz. Alle eingesetzten Beans werden standardmässig konfiguriert. Wenn man z.B. den Spring-Boot-Starter für JPA (Java Persistence API) referenziert, so wird automatisch eine In-Memory-Datenbank, ein Hibernate Entity Manager und eine einfache Datenquelle konfiguriert. Dies ist ein Beispiel für eine meinungsfähige Standardkonfiguration, die Sie überschreiben können, damit erhalten Sie sofort eine gute Ausgangslage.Die folgende Grafik zeigt die vereinfachte Darstellung der Spring Boot Architektur:

Spring Boot Features

- Standalone Spring Anwendungen erstellen.
- Einbetten von Tomcat, Jetty oder Undertow direkt (keine Bereitstellung von WAR-Dateien erforderlich).
- Starter Dependencies für die Vereinfachung der Build Konfiguration.
- Falls immer möglich automatische Konfiguration von Spring und 3rd party libraries.
- Metrik Tools, Health Checks und ausgelagerte Konfiguration via Properties oder YAML Dateien. Damit ist die gleiche Applikation auf unterschiedlichen Umgebungen ausführbar.
- Keine Code Generierung und keine XML Konfiguration.

Spring Boot Hello World

Das folgende Listing zeigt einen ersten einfachen Spring Boot Hello World REST Service:package ch.std.hello.rest;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {
  private static final String template = "Hello, %s!";
  @GetMapping("/hello")
  public String greeting(@RequestParam(value="name", defaultValue="World") String name) {
    return String.format(template, name);
  }
}
Der Maven oder Gradle Build erstellt ein ausführbares FAT Jar:java -jar springboothellomaven-0.0.1-SNAPSHOT.jarDas FAT Jar entspricht der folgenden Struktur:Nach dem Startup finden wir unseren Hello REST Service unter der folgenden URL http://localhost:8080/hello:

Spring Data

Spring Data bietet die vereinfachte Untersützung für den Zugriff auf SQL und NoSQL Datenbanken via Repository- und benutzerdefinierte Objekt-Mapping-Abstraktionen mit JDBC, JPA, LDAP, MongoDB etc.. SQL Queries können via Repository und Methodennamen beschrieben und ausgeführt werden. Aus dem Namen der Methode erstellt Spring Data die gesuchte Query.Das Vorgehen bei der Integration von Datenbank zeigen wir anhand einem klassischen Beispiel mit JPA und einem Spring Data Repository.Hierzu verwenden wir die folgende JPA Klasse:package ch.std.jumpstart.jpa;
import java.io.Serializable;
@Entity
@Table(name = "city")
public class City implements Serializable {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  @Column(name = "name", unique=true, length=128)
  private String name;

...
}
Der Zugriff erfolgt nun über die Repository Klasse und über Methodennamen oder eigene JPQL Abfragen definiert durch die @Query Annotation: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 {
  Optional findByName(String name);
  @Query("SELECT c FROM City c WHERE c.name LIKE %?1%")
  public List<City> findByNameLike(String name);
}Der Zugriff auf das Repository erfolgt nun je nach Design direkt im REST Controller:package ch.std.jumpstart.rest;
...
import ch.std.jumpstart.jpa.City;
import ch.std.jumpstart.repository.CityRepository;
@RestController
public class CityController {
  @Autowired
  CityRepository cityRepository;
  @GetMapping("/rest/cities")
  public City[] getCities(@RequestParam(value = "value", required = false) String value) {
    List<City> cities = null;
    if (value == null) {
      cities = (List<City>) cityRepository.findAll();
    } else {
      cities = (List<City>) cityRepository.findByNameLike(value);
    }
    return cities.toArray(new City[0]);
  }
  @GetMapping("/rest/city/{id}")
  public City getCityById(@PathVariable Long id) {
    City city = cityRepository.findById(id).orElseThrow(() -> new CityNotFoundException(id));
    return city;
  }
  ...
}Die Rückgabe von JPA Entity Instanzen direkt im REST Controller via JSON ist keine gute Praxis. Die JSON Response muss von der inneren JPA Struktur getrennt werden, z.B. via Map:@GetMapping("/city/{id}")
public Map getCityById(@PathVariable Long id) {
  City city = cityRepository.findById(id).orElseThrow(() -> new CityNotFoundException(id));
  if (city == null) {
    return null
  }
  return city.toMap();
}...

public Map toMap() {
  Map map = new LinkedHashMap<>();
  map.put("id", this.id);
  map.put("name", this.name);
  return map;
}
Das direkte Rendering über die toMap-Methode via JPA Entity ist auch nicht optimal, besser wäre eine explizite Trennung via DTO (Data Transfer Object) oder Renderer Ebene.

Generic REST Controller

Ein noch besseres REST Controller Design sehen sie im Blog Generic REST Endpoint mit Spring Boot

Spring @DataJpaTest

Mit der Spring @DataJpaTest Annotation können Repositories einfach getestet werden. In Kombination mit der SpringRunner.class sind damit JPA Komponenten effizient mit der InMemory Datenbank H2 testbar.Alternativ kann mit einer realen Datenbank wie mysql gearbeitet werden. Das folgende Listing zeigt den JPA Test für das CityRepository:package ch.std.jumpstart.repository;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;
...
import ch.std.jumpstart.jpa.City;
import ch.std.jumpstart.repository.BookRepository.IsbnTitleOnly;
@RunWith(SpringRunner.class)
@DataJpaTest
@ActiveProfiles("test")
public class CityRepositoryIntegrationTest {
  @Autowired
  private CityRepository cityRepository;
  private City city;
  private List<City> cityList;
  @Before
  public void setup() {
    city = new City("Bern");
    this.cityRepository.save(city);
    this.cityList = new Array<>();
    for (int i = 0; i < 10; i++) {
      City localCity = new City("city-" + i);
      this.cityRepository.save(localCity);
      this.cityList.add(localCity);
    }
  }
  @After
  public void tearDown() {
    this.cityRepository.deleteAllInBatch();
  }
  @Test
  public void testFindById() {
    City foundCity = this.cityRepository.findById(city.getId()).orElse(null);
    assertNotNull(foundCity);
    assertEquals(this.city, foundCity);
  }
  @Test
  public void testFindByName() {
    City foundCity = this.cityRepository.findByName("Bern").orElse(null);
    assertNotNull(foundCity);
    assertEquals(this.city, foundCity);
  }
}

Spring @SpringBootTest

Spring Boot Integrationstests lassen sich mit der @SpringBootTest Annotation schnell und einfach erstellen. Via RandomPort wird der Endpoint und damit die RESTController gestartet und solche sind nun via TestRestTemplate, der REST Client Klasse, testbar. Das folgende Listing zeigt den Integrationstest zum City Controller:package ch.std.jumpstart;
...
import ch.std.jumpstart.jpa.City;
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@ActiveProfiles("test")
public class CityControllerTests {
  @LocalServerPort
  private int port;
  @Autowired
  private ServletContext servletContext;
  @Autowired
  private TestRestTemplate restTemplate;
  private List<City> citiesList = new ArrayList<>();
  @Before
  public void setup() {
 this.citiesList.clear();
    String postUrl = this.getUrl("/rest/city");
 HttpEntity<City> request = new HttpEntity<>(new City("Bern"));
  City bernCity = this.restTemplate.postForObject(postUrl, request, City.class);
 this.citiesList.add(bernCity);
  }
  @Test
  public void testGetCities() throws Exception {
    String url = this.getUrl("/rest/cities?value=Bern");
    City[] cities = this.restTemplate.getForObject(url, City[].class);
    assertNotNull(cities);
 assertEquals(1, cities.length);
    City city = cities[0];
    assertNotNull(city);
 assertEquals("Bern", city.getName());
  }
  @Test
  public void testGetAllCities() throws Exception {
    String url = this.getUrl("/rest/cities");
 City[] cities = this.restTemplate.getForObject(url, City[].class);
    assertNotNull(cities);
 assertEquals(this.citiesList.size(), cities.length);
    for (City city : this.citiesList) {
 assertTrue(citiesList.contains(city));
    }
  }
  @Test
  public void testGetCityById() throws Exception {
    String url = this.getUrl("/rest/cities?value=Bern");
    City[] cities = this.restTemplate.getForObject(url, City[].class);
    assertNotNull(cities);
 assertEquals(1, cities.length);
    City city = cities[0];
    String urlById = this.getUrl("/rest/city/" + city.getId());
    City cityById = this.restTemplate.getForObject(urlById, City.class);
    assertEquals(city, cityById);
 }
  public String getUrl(String path) {
    return "http://localhost:" + port + servletContext.getContextPath() + path;
  }
}

Feedback

War dieser Blog für Sie wertvoll. Wir danken für jede Anregung und Feedback

Kontakt

Simtech AG
Finkenweg 23
3110 Münsingen
Schweiz

Impressum

Copyright

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