Spring Boot

Czym jest Spring Boot i dlaczego zdobył dużą popularność?

Spring Boot jest mikro-frameworkiem typu open-source wprowadzonym przez firmę Pivotal. Dostarcza on programistom platformę do rozpoczęcia pracy z automatycznie konfigurowalną aplikacją Spring klasy produkcyjnej. Dzięki niemu programiści mogą szybko rozpocząć pracę, nie tracąc czasu na przygotowanie i konfigurację aplikacji Spring.

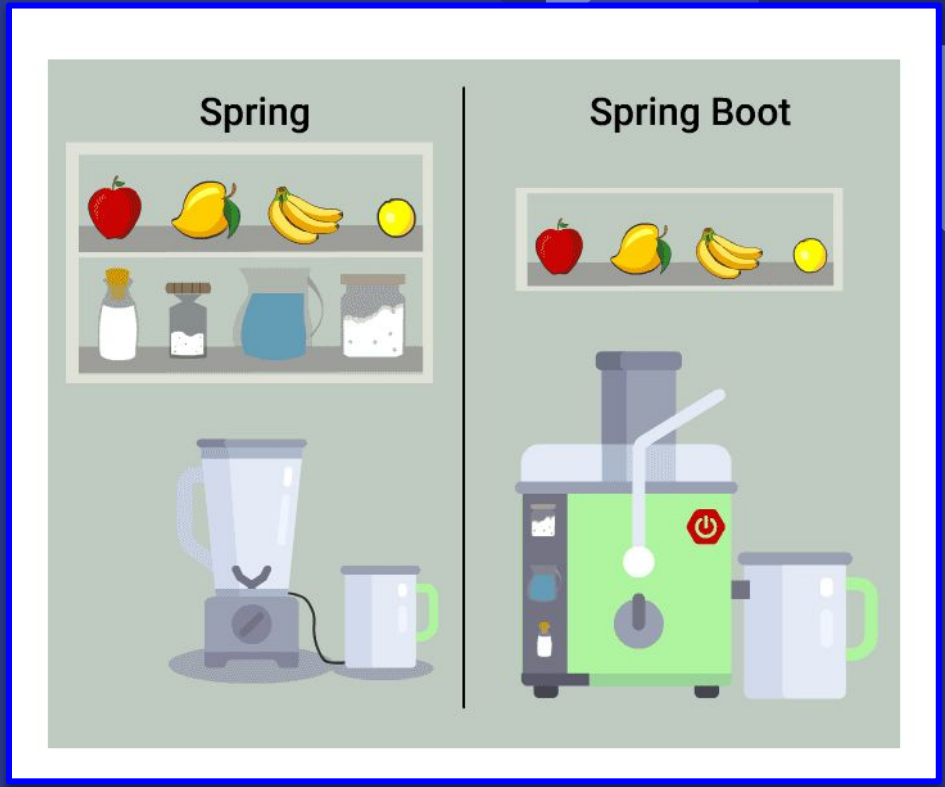
Za popularnością Spring Boot stoi przede wszystkim wykorzystywanie Javy, która jest jednym z najpopularniejszych języków programowania na świecie. Poza tym, Spring Boot jest narzędziem, które pomaga szybko uruchomić aplikację bez konieczności martwienia się o poprawną i bezpieczną konfigurację.

Ponadto, społeczność użytkowników jest ogromna. Istnieje mnóstwo darmowych materiałów do nauki i kursów.

Spring a Spring Boot

Framework Spring koncentruje się na szybkim wstrzykiwaniu wymaganych zależności (dependency injection) oraz na tworzeniu aplikacji luźno ze sobą powiązanych (loose coupling).

Z kolei Spring Boot koncentruje się na skróceniu długości kodu, a także zapewnieniu łatwego sposobu uruchamiania aplikacji Spring.



Główne cechy

- Łatwiejsze tworzenie samodzielnych aplikacji Spring
- Wbudowany bezpośrednio Tomcat, Jetty lub Undertow (bez potrzeby rozmieszczania plików WAR)
- Dostarczanie zależności startowych, aby uprościć konfigurację
- Automatyczne konfigurowanie Springa i bibliotek innych firm
- Dostarczanie funkcji gotowych do wykorzystania w produkcji, takich jak metryki, kontrole stanu i zewnętrzna konfiguracja
- Całkowity brak generowania kodu i brak wymagań co do konfiguracji XML

Zalety i wady

- mniej kodu źródłowego
- brak konieczności konfiguracji XML
- łatwy start i zarządzanie aplikacjami
- duża społeczność
- wbudowany Tomcat
- duża ilość funkcjonalności dostępna od początku

- mniejsza kontrola aplikacji
- problematyczna konwersja na projekt typu Spring Boot
- nieodpowiedni do projektów na dużą skalę (duża ilość nieużywanych zależności skutkuje dużym rozmiarem pakietu)

Wstępna konfiguracja projektu

Gradle – jest narzędziem służącym do budowania projektów. Pozwala ono na zautomatyzowanie tego procesu. Jest oparty na koncepcjach Apache Ant i Apache Maven. Ponadto wykorzystuje on język dziedzinowy – DSL (ang. Domain Specific Language), który ułatwia wykonywanie standardowych zadań związanych z budowaniem projektu. Zamiast formy XML używanej przez Apache Maven do deklarowania konfiguracji projektu używa dedykowanego pliku .gradle.

Build.gradle

```
plugins {
    id 'org.springframework.boot' version '2.1.3.RELEASE'
}

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

dependencies {
    compile 'org.springframework.boot:spring-boot-starter-web'
    testCompile 'org.springframework.boot:spring-boot-starter-test'
    testCompile 'org.testng:testng:6.14.3'
}

bootJar {
    baseName = 'bsg5-chapter7'
    version =  '1.0.0'
}

sourceCompatibility = 11
targetCompatibility = 11

test {
    useTestNG()
}
```

Pierwsze kroki ze Spring Boot’em cz.1

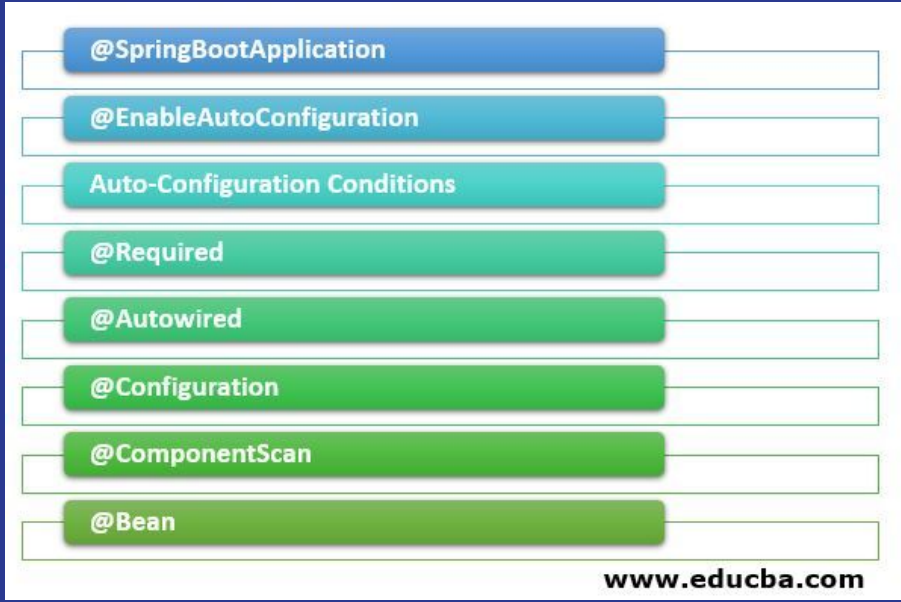
7.3.1 Building the Application

Listing 7-3. chapter7/src/main/java/com/bsg5/chapter7/Chapter7Application.java

```
package com.bsg5.chapter7;

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

@SpringBootApplication
public class Chapter7Application {

    public static void main(String[] args) {
        SpringApplication.run(Chapter7Application.class, args);
    }
}
```

```
public class Greeting {
    String message;

    public Greeting(String message) {
        this.message = message;
    }

    public Greeting() {
    }

    public String getMessage() {
        return message;
    }

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

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Greeting)) return false;
        Greeting greeting = (Greeting) o;
        return Objects.equals(getMessage(), greeting.getMessage());
    }

    @Override
    public int hashCode() {
        return Objects.hash(getMessage());
    }
}
```

Pierwsze kroki ze Spring Boot’em cz.2



Listing 7-5. chapter7/src/main/java/com/bsg5/chapter7/GreetingController.java

```
package com.bsg5.chapter7;

import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {
    @RequestMapping(value = {"/greeting/{name}", "/greeting"})
    Greeting greeting(@PathVariable(required = false) String name) {
        String object = name != null ? name : "world";

        /* Jack Griffin is the name of the "Invisible Man." */
        if (object.equalsIgnoreCase("jack griffin")) {
            return new Greeting("I don't know who you are.");
        } else {
            return new Greeting("Hello, " + object + "!");
        }
    }
}
```

Testowanie Spring Boot

Pierwszym podejściem w testowaniu aplikacji Spring Boot jest bezpośrednie wywołanie metod kontrolera. W takim podejściu faktycznie sprawdzamy czy logika została poprawnie zaimplementowana, jednak pomijamy dużą część funkcjonalności samego kontrolera.

```
@Test(dataProvider = "greetingData")
public void testDirectGreeting(String name, String greeting) {
    assertEquals(
            greetingController.greeting(name).getMessage(),
}
```

Testowanie Spring Boot

Drugim podejściem jest użycie *TestRestTemplate*. Dzięki niemu, testujemy nasz kontroler symulując prawdziwe zapytania HTTP, co pozwala nam lepiej odwzorować realne przykłady użycia. Pozwala nam też sprawdzić czy kontroler zwraca poprawne odpowiedzi HTTP

```
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class TestGreetingController extends
AbstractTestNGSpringContextTests {
    @Autowired
    private GreetingController greetingController;
    @LocalServerPort
    private int port;
    @Autowired
    private TestRestTemplate restTemplate;

    @DataProvider
    Object[][] greetingData() {
        return new Object[][]{
                new Object[]{null, "Hello, world!"},
                new Object[]{"World", "Hello, World!"},
                new Object[]{"Andrew", "Hello, Andrew!"},
                new Object[]{"Jack Griffin", "I don't know who you are."}
        };
    }

    @Test(dataProvider = "greetingData")
    public void testRestGreeting(String name, String greeting) {
        String url = "http://localhost:" + port + "/greeting/" +
            (name != null ? name : "");
        ResponseEntity<Greeting> result =
            restTemplate.getForEntity(url, Greeting.class);
        assertEquals(result.getStatusCode(), HttpStatus.OK);
        assertEquals(result.getBody().getMessage(), greeting);
    }

    @Test(dataProvider = "greetingData")
    public void testDirectGreeting(String name, String greeting) {
        assertEquals(
                greetingController.greeting(name).getMessage(),
                greeting);
    }
}
```

Konfiguracja w Spring Boot

Konfigurowanie aplikacji Spring Boot nie jest wymagane. Anotacja @SpringBootApplication robi to za nas. Powinna ona wykryć i połączyć wszystkie komponenty naszej aplikacji w danym pakiecie, pod warunkiem, że nie robimy niczego nadzwyczajnego. Spring Boot automatycznie wykryje każdy Bean w aktualnym pakiecie.

Obsługa wyjątków

W celu utworzenia własnego wyjątku wystarczy stworzyć nową klasę, która będzie rozszerzała np. RuntimeException.

Dla takiej klasy możemy za pomocą adnotacji
@ResponseStatus(HTTP_CODE)
wybrać jaki kod http ma być zwracany wraz z jego wystąpieniem.
Ta adnotacja mówi klasie @Controller na jaki kod http ma zostać zmapowany rzucony wyjątek.

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class ArtistNotFoundException extends RuntimeException {
```

Przechwytywanie wyjątków

Do przechwytywania wyjątków możemy utworzyć osobną klasę, do której wystarczy dodać adnotację @ControllerAdvice.

Następnie tworzymy w niej metodę do której dodajemy kolejną adnotację @ExceptionHandler, w której podajemy jakiego rodzaju wyjątki mają być przetwarzane przez tę metodę.

```
@ControllerAdvice
public class ApiExceptionHandler {
    @ExceptionHandler(value = { IllegalArgumentException.class, IllegalStateException.class })
    public ResponseEntity<Object> handleException()...
```

Integracja z bazą danych

Integrację z bazą danych rozpoczynamy od dodania zależności:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Przykład wskazuje na załączenie bazy H2, w przypadku chęci dołączenia innych serwerów bazodanowych wystarczy zmienić zależność, np. dla PostgreSQL:

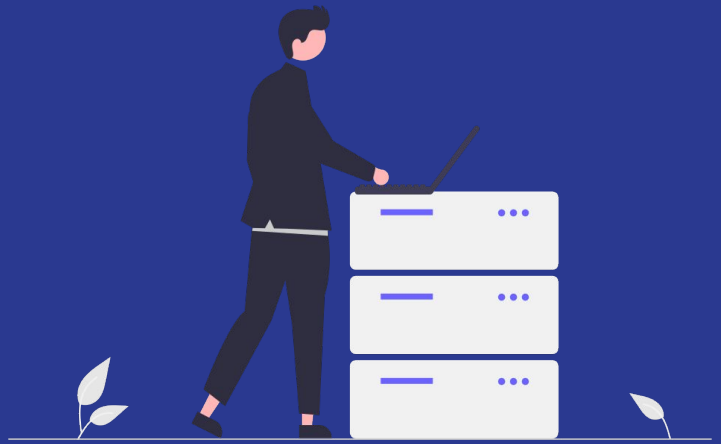
```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

Konfiguracja połączenia

Domyślna konfiguracja Spring Boot powoduje połączenie do tzw. in-memory store z wartościami:

- **sa** jako nazwa użytkownika
- pustym hasłem

Konfigurację tę można zmienić w pliku *application.properties* bądź poprzez plik YAML *application.yaml*



Konfiguracja połączenia

application.properties

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

YML

```
spring:
  datasource:
    url: jdbc:h2:mem:mydb
    username: sa
    password: password
    driverClassName: org.h2.Driver
  jpa:
    spring.jpa.database-platform: org.hibernate.dialect.H2Dialect
```

Przy wykorzystaniu H2 pamięć bazy domyślnie będzie ulotna, tj. resetowana przy restarcie aplikacji. W celu zmiany takiej sytuacji należy wykorzystać przechowywanie danych w pliku, zmieniając wartość spring.datasource.url:

```
spring.datasource.url=jdbc:h2:file:/data/demo
```

```
spring:
  datasource:
    url: jdbc:h2:file:/data/demo
```

Inicjalizacja danych

W celu inicjalizacji bazy danymi można stworzyć plik data.sql, który będzie zwykłym skryptem SQL, a następnie umieścić go w katalogu src/main/resources, który będzie automatycznie uruchamiany przez Spring Boot. Rozwiązanie takie może być wykorzystywane np. przy testowaniu aplikacji.

CRUD Repository

Interfejs CrudRepository zapewnia dostęp do operacji typu CRUD dla określonego typu. Poniżej przedstawiono przykład wyszukiwania elementu poprzez jego ID.

CityRepository.java

```
@Repository
public interface CityRepository extends CrudRepository<City, Long> {

}
```

Dzięki rozszerzeniu CrudRepository otrzymujemy dostęp do podstawowych metod dla danego modelu. W naszym przypadku jest to City.

ICityService.java

```
public interface ICityService {

    Optional<City> findById(Long id);
}
```

ICityService zapewnia dostęp do metody wyszukującej miasta po jego ID.

CityService.java

```
@Service
public class CityService implements ICityService {

    private final CityRepository cityRepository;

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

    @Override
    public Optional<City> findById(Long id) {

        return cityRepository.findById(id);
    }
}
```

CityService zawiera implementację metody findById, do pobrania danych z bazy wykorzystujemy CityRepository wstrzyknięte do CityService

Dziękujemy za uwagę

Wykonali:

Grzegorz Podwika
Dominik Pepaś
Michał Mamla
Sebastian Smulski
Mariusz Morawski
Patryk Paluch
Dawid Aleksandrowicz