Spring Boot : Tests

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Spring Boot

Deux catégories de tests

- tests manuels :
  - coûteux
  - répétitif (corriger et retester)
- tests automatiques (via des programmes)

Spring Boot

Plusieurs niveaux de tests

- tests unitaires
- tests d'intégration
- tests fonctionnels
- tests de charge
- tests d'acceptation

Spring Boot

tests unitaires

- Permettent de tester une **unité (partie) isolée** de l'application (souvent une fonction ou une méthode).
- Indiquent au développeur que le code fait **bien les choses**.

Spring Boot

tests unitaires

- Permettent de tester une **unité (partie) isolée** de l'application (souvent une fonction ou une méthode).
- Indiquent au développeur que le code fait **bien les choses**.

Remarque

Les tests qui communiquent avec une base de données
ne sont pas des tests unitaires, mais plutôt des tests d'intégration.

Spring Boot

tests d'intégration

- permettent de vérifier que les **unités isolées** fonctionnent correctement ensemble.
- peuvent nécessiter de composants extérieurs (Service web, base de données...).

Spring Boot

Autres niveaux de tests

- **tests fonctionnels** (de bout-en-bout ou **end-to-end** en anglais) : sont écrits du point de vue de l'utilisateur final depuis l'interface utilisateur, pour vérifier que l'application répond aux exigences.
  Ils indiquent au développeur que le code fait **les bonnes choses**.

- **tests d'acceptation** : permettent de vérifier que l'**application** respecte bien le besoin fonctionnel.

- **tests de charge** (système) : permettent de vérifier si un système peut gérer une charge spécifiée : le nombre d'utilisateurs simultanés...

Spring Boot

Et les tests de non-régression ?

Ils permettent de vérifier que la livraison de nouvelles fonctionnalités n'aura pas d'effet de bord sur les fonctionnalités existantes (programme préalablement testé).

Spring Boot

Autres tests

- **tests de performance**
- **tests de fiabilité**
- ...

Spring Boot

Création de projet **Spring Boot**

- Aller dans `File > New > Other`
- Chercher `Spring`, dans `Spring Boot` sélectionner `Spring Starter Project` et cliquer sur `Next >`
- Saisir
  - `first-spring-test` dans `Name`,
  - `com.example` dans `Group`,
  - `firstspringtest` dans `Artifact`,
  - `com.example.demo` dans `Package`
- Cliquer sur `Next >`
- Chercher et cocher les cases correspondantes aux `Spring Data JPA`, `MySQL Driver`, `Spring Boot DevTools` et `Lombok` et `Spring Web` puis cliquer sur `Next >`
- Valider en cliquant sur `Finish`

Spring Boot

Explication

- Le package contenant le point d'entrée de notre application (la classe contenant le `puclic static void main`) est `com.example.demo`
- Tous les autres packages `dao`, `model`... doivent être dans le package `demo`.

Spring Boot

Pour la compatibilité d'*Apache Tomcat* avec les JSP, on ajoute la dépendance suivante

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

Spring Boot

Pour la compatibilité d'*Apache Tomcat* avec les JSP, on ajoute la dépendance suivante

```
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

Pour utiliser la `JSTL`, on ajoute la dépendance suivante

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
</dependency>
```

Spring Boot

Contenu de `application.properties`

```
spring.mvc.view.prefix=/views/
spring.mvc.view.suffix=.jsp
```

Spring Boot

Considérons le contrôleur `HomeController`

```
package com.example.demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HomeController {

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

Spring Boot

Contenu de `home.jsp` (à créer dans `webapp/views`)

```
<%@ page language="java" contentType="text/html;
   charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Home page</title>
</head>
<body>
        <h1>Hello world!</h1>
</body>
</html>
```

Spring Boot

Tests unitaires

Programme permettant de vérifier le bon fonctionnement d'une partie (ou une unité) de l'application.

Spring Boot

Tests unitaires

Programme permettant de vérifier le bon fonctionnement d'une partie (ou une unité) de l'application.

Objectif

Trouver un maximum d'erreurs pour les corriger.

Spring Boot

Tests unitaires

Programme permettant de vérifier le bon fonctionnement d'une partie (ou une unité) de l'application.

Objectif

Trouver un maximum d'erreurs pour les corriger.

Remarque

Si le test ne détecte pas d'erreurs $\not\Rightarrow$ il n'y en a pas.

Spring Boot

Pour créer une classe de test

- Faire un clic droit sur `src/test/java`
- Aller dans `New > JUnit Test Case`
- Saisir `com.example.demo.controller` dans `Package`
- Saisir `HomeControllerTests` dans `Name`
- Cliquer sur `Browse` en face de `Class under test`
- Saisir `HomeController` puis sélectionner **`HomeController`** – `com.example.demo.controller` et valider
- Cliquer sur `Next` puis cocher la case correspondante de `home` dans `HomeController`
- Cliquer sur `Finish`

Spring Boot

Contenu généré pour `HomeControllerTests`

```
package com.example.demo.controller;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class HomeControllerTests {

	@Test
	void testHome() {
		fail("Not yet implemented");
	}

}
```

Spring Boot

Écrivons notre première assertion

```
package com.example.demo.controller;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class HomeControllerTests {

        HomeController homeController = new HomeController();

        @Test
        void testHome() {
                assertEquals("home", homeController.home());
        }

}
```

Spring Boot

Pour tester

- Faire un clic droit sur le la classe de test
- Aller dans `Run As > JUnit Test`

Spring Boot

Ajoutons un paramètre de requête aux routes `home` et `/`

```
package com.example.demo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class HomeController {

        @GetMapping({ "/home", "/" })
        public String home(@RequestParam String nom, Model
           model) {
                model.addAttribute("nom", nom);
                return "home";
        }
}
```

Spring Boot

Affichons le paramètre de requête envoyé par le contrôleur dans `home.jsp`

```
<%@ page language="java" contentType="text/html;
   charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Home page</title>
</head>
<body>
        <h1>Hello ${nom}</h1>
</body>
</html>
```

Spring Boot

Pour tester si la vue a correctement reçue la valeur envoyée par le contrôleur

```
package com.example.demo.controller;

import static org.assertj.core.api.Assertions.assertThat;

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 org.springframework.boot.web.server.LocalServerPort;

@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class HomeControllerTests {

        @Autowired
        private TestRestTemplate restTemplate;

        @LocalServerPort
        private int port;

        @Test
        public void testHome() throws Exception {
                assertThat(this.restTemplate.getForObject("http://localhost:" + port + "/home?
                  nom=wick", String.class))
                                .contains("Hello wick");
        }

}
```

Spring Boot

Explication

- `webEnvironment=RANDOM_PORT` permet de démarrer le serveur avec un port aléatoire (utile pour éviter les conflits dans les environnements de test).
- `@LocalServerPort` permet d'injecter le port généré.

Spring Boot

Explication

- `webEnvironment=RANDOM_PORT` permet de démarrer le serveur avec un port aléatoire (utile pour éviter les conflits dans les environnements de test).
- `@LocalServerPort` permet d'injecter le port généré.

Remarque 1

`@RunWith(SpringRunner.class)` est une annotation **JUnit 4** qui n'est plus nécessaire dans la version **Jupiter**, idem pour `@ExtendWith(SpringExtension.class)`.

Spring Boot

Remarque 2

Pour simuler le démarrage du serveur (sans le faire réellement), on peut utiliser les **Mock**.

Solution avec les mocks

```
package com.example.demo.controller;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.model;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.servlet.MockMvc;

@SpringBootTest
@AutoConfigureMockMvc
class HomeControllerTests {

        @Autowired
        private MockMvc mockMvc;

        @Test
        public void testHome() throws Exception {
                this.mockMvc.perform(get("/home").param("nom", "wick"))
                .andDo(print())
                .andExpect(status().isOk())
                .andExpect(view().name("home"))
                .andExpect(model().attribute("nom", "wick1"));
        }

}
```

Spring Boot

Explication

- `get(...)` permet de spécifier le verbe **HTTP** et le chemin.
- `param` permet d'envoyer un paramètre.
- `andDo(print())` permet d'afficher le résultat dans la console.
- `andExpect()` permet d'asserter sur le status serveur, le nom de la vue, les attributs du model...

Spring Boot

Remarques

Les **Mocks** n'ont pas accès au contenu d'une page **JSP** car les pages **JSP** sont rendues par un conteneur de servlet et les **Mocks** n'ont pas accès au conteneur de servlet.

Spring Boot

Remarques

Les **Mocks** n'ont pas accès au contenu d'une page **JSP** car les pages **JSP** sont rendues par un conteneur de servlet et les **Mocks** n'ont pas accès au conteneur de servlet.

`MockMvc` **VS** `TestRestTemplate`

- `MockMvc` permet de tester coté serveur.
- `TestRestTemplate` permet de tester coté client.

Spring Boot

Créons une entité `Personne` dans `com.example.demo.model`

```
@NoArgsConstructor
@AllArgsConstructor
@Data
@Entity
@RequiredArgsConstructor
public class Personne {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long num;
  @NonNull
  private String nom;
  @NonNull
  private String prenom;
}
```

Spring Boot

Préparons notre interface DAO `PersonneRepository`

```
package com.example.demo.dao;

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

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository<Personne,
   Long> {

}
```

Ajoutons les propriétés suivantes dans `application.properties`

```
spring.datasource.url=jdbc:mysql://localhost:3306/springtest?serverTimezone=UTC&
   characterEncoding=UTF-8&useUnicode=yes&createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

Ajoutons les propriétés suivantes dans `application.properties`

```
spring.datasource.url=jdbc:mysql://localhost:3306/springtest?serverTimezone=UTC&
   characterEncoding=UTF-8&useUnicode=yes&createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```


Explication

- La chaîne `serverTimezone=UTC` permet d'éviter un bug du connecteur `MySQL` concernant l'heure système.
- La chaîne `characterEncoding=UTF-8&useUnicode=yes` permet d'ajouter les caractères accentués dans la base de données.
- La chaîne `createDatabaseIfNotExist=true` permet de créer la base de données si elle n'existe pas.
- La propriété `spring.jpa.hibernate.naming.physical-strategy = org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl` permet de forcer **Hibernate** à utiliser les mêmes noms pour les tables et les colonnes que les entités et les attributs.

Spring Boot

Pour alimenter la base de données avec quelques données au démarrage de l'application

```
@SpringBootApplication
public class FirstSpringTestApplication implements ApplicationRunner {

    @Autowired
    private PersonneRepository personneRepository;

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

    @Override
    public void run(ApplicationArguments args) throws Exception {
        Personne personne1 = new Personne("wick", "john");
        Personne personne2 = new Personne("dalton", "jack");
        Personne personne3 = new Personne("maggio", "carol");
        Personne personne4 = new Personne("cohen", "sophie");
        personneRepository.saveAll(Arrays.asList(personne1, personne2,
          personne3, personne4));
  }
}
```

Spring Boot

Considérons le contrôleur REST suivant

```
@RestController
public class PersonneRestController {

        @Autowired
        private PersonneRepository personneRepository;

        @GetMapping("/personnes")
        public List<Personne> getPersonnes() {
                return personneRepository.findAll();
        }

        @GetMapping("/personnes/{id}")
        public Personne getPersonne(@PathVariable("id") long id) {
                var personne = personneRepository.findById(id).orElse(null);
                if (personne == null)
                        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Personne avec
                          id = " + id + " est introuvable");
                return personne;
        }

        @PostMapping("/personnes")
        public Personne addPersonne(@RequestBody Personne personne) {
                return personneRepository.save(personne);
        }
}
```

Spring Boot

Commençons par créer la classe de test

```
package com.example.demo.controller;

import org.junit.jupiter.api.Test;

class PersonneRestControllerTests {

        @Test
        void testGetPersonnes() {
        }

        @Test
        void testGetPersonne() {
        }

        @Test
        void testAddPersonne() {
        }

}
```

Spring Boot

Ajoutons le mock

```
@SpringBootTest
@AutoConfigureMockMvc
class PersonneRestControllerTests {

        @Autowired
        private MockMvc mockMvc;

        @Test
        void testGetPersonnes() {
        }

        @Test
        void testGetPersonne() {
        }

        @Test
        void testAddPersonne() {
        }

}
```

Ajoutons le mock

```
@SpringBootTest
@AutoConfigureMockMvc
class PersonneRestControllerTests {

        @Autowired
        private MockMvc mockMvc;

        @Test
        void testGetPersonnes() throws Exception {
                this.mockMvc
                .perform(get("/personnes"))
                .andDo(print())
                .andExpect(status().isOk())
                .andExpect(content().contentType("application/json"))
                .andExpect(jsonPath("$[0].nom").exists())
                .andExpect(jsonPath("$[0].nom").value("wick"));
        }
        @Test
        void testGetPersonne() throws Exception {
                this.mockMvc
                .perform(get("/personnes/1"))
                .andDo(print())
                .andExpect(status().isOk())
                .andExpect(content().contentType("application/json"))
                .andExpect(jsonPath("$.nom").exists())
                .andExpect(jsonPath("$.nom").value("wick"));
        }
        @Test
        void testAddPersonne() {
        }
}
```

Spring Boot

Explication

- `$[0]` permet d'accéder au premier objet du tableau **JSON** retourné.
- `$.nom` permet d'accéder à l'attribut `nom` de l'objet **JSON** retourné.

Spring Boot

Explication

- `$[0]` permet d'accéder au premier objet du tableau **JSON** retourné.
- `$.nom` permet d'accéder à l'attribut `nom` de l'objet **JSON** retourné.

Remarque

En testant le code précédent, on obtient un échec car les méthodes `getPersonnes` et `getPersonne` utilisent l'interface `PersonneRepository`.

Spring Boot

Explication

- `$[0]` permet d'accéder au premier objet du tableau **JSON** retourné.
- `$.nom` permet d'accéder à l'attribut `nom` de l'objet **JSON** retourné.

Remarque

En testant le code précédent, on obtient un échec car les méthodes `getPersonnes` et `getPersonne` utilisent l'interface `PersonneRepository`.

Solution

Mocker le bean `PersonneRepository`.

Spring Boot

Commençons par mocker le bean `PersonneRepository`

```
@SpringBootTest
@AutoConfigureMockMvc
class PersonneRestControllerTests {

        @Autowired
        private MockMvc mockMvc;

        @MockBean
        private PersonneRepository personneRepository;

        // + le reste du code précédent

}
```

Définissons le comportement de `findAll` et `findById` puis tester (le test doit passer)

```
@Test
void testGetPersonnes() throws Exception {
        Personne personne1 = new Personne("wick", "john");
        Personne personne2 = new Personne("dalton", "jack");
        Personne personne3 = new Personne("maggio", "carol");
        Personne personne4 = new Personne("cohen", "sophie");
        when(personneRepository.findAll()).thenReturn(Arrays.asList(personne1, personne2,
           personne3, personne4));
        this.mockMvc
        .perform(get("/personnes"))
        .andDo(print())
        .andExpect(status().isOk())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$[0].nom").exists())
        .andExpect(jsonPath("$[0].nom").value("wick"));
}

@Test
void testGetPersonne() throws Exception {
        when(personneRepository.findById(1L)).thenReturn(Optional.of(new Personne("wick", "john
           ")));
        this.mockMvc
        .perform(get("/personnes/1"))
        .andDo(print())
        .andExpect(status().isOk())
        .andExpect(content().contentType("application/json"))
        .andExpect(jsonPath("$.nom").exists())
        .andExpect(jsonPath("$.nom").value("wick"));
}
```

Spring Boot

Pour vérifier que le mock a été appelé avec la bonne valeur (le test doit passer)

```
@Test
void testGetPersonne() throws Exception {
        when(personneRepository.findById(1L)).thenReturn(Optional.of(new Personne("wick", "john
           ")));
        this.mockMvc
        .perform(get("/personnes/1"))
        .andDo(print())
        .andExpect(status().isOk())
        .andExpect(content().contentType("application/json"))
        .andExpect(jsonPath("$.nom").exists())
        .andExpect(jsonPath("$.nom").value("wick"));
    verify(personneRepository).findById(1L);
}
```

Spring Boot

Pour utiliser les assertions Jupiter (assertThat...) sans passer par `andExpect`, on peut utiliser `doReturn` (le test doit passer)

```
@Test
void testGetPersonne() throws Exception {
        when(personneRepository.findById(1L)).thenReturn(Optional.of(new Personne("wick", "john
           ")));
        final MvcResult result = this.mockMvc
        .perform(get("/personnes/1"))
        .andDo(print())
        .andReturn();
        assertThat(result.getResponse().getContentAsString()).contains("wick");
}
```

Spring Boot

Les deux annotations `@SpringBootTest` **et** `@AutoConfigureMockMvc` **peuvent être remplacées par** `@WebMvcTest`

```
@WebMvcTest(PersonneRestController.class)
class PersonneRestControllerTests {

        // le code précédent

}
```

Spring Boot

Spring Boot nous permet de tester

- les contrôleurs
- mais aussi les repositories

Spring Boot

Commençons par définir une méthode `findByNomContaining` dans `PersonneRepository` que l'on souhaite tester plus tard

```
package com.example.demo.dao;

import java.util.List;

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

import com.example.demo.model.Personne;

public interface PersonneRepository extends JpaRepository<Personne,
   Long> {
        List<Personne> findByNomContaining(String s);
}
```

Spring Boot

Ensuite créons notre classe de test

```
package com.example.demo.dao;

import org.junit.jupiter.api.Test;

class PersonneRepositoryTests {

        @Test
        void testFindByNomContaining() {
        }

}
```

Spring Boot

Pour indiquer que l'on souhaite tester un repository, on ajoute l'annotation suivante

```
package com.example.demo.dao;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

@DataJpaTest
class PersonneRepositoryTests {

        @Test
        void testFindByNomContaining() {
        }

}
```

Spring Boot

Ensuite, injectons le repository à tester

```
package com.example.demo.dao;

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

@DataJpaTest
class PersonneRepositoryTests {

        @Autowired
        private PersonneRepository personneRepository;

        @Test
        void testFindByNomContaining() {
        }

}
```

Ajoutons les assertions (Le test ne passera pas)

```
package com.example.demo.dao;

import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

@DataJpaTest
class PersonneRepositoryTests {

    @Autowired
    private PersonneRepository personneRepository;

    @Test
    void testFindByNomContaining() {
        var wicks = personneRepository.findByNomContaining("wick");
        assertThat(wicks).isNotEmpty();
        assertThat(wicks).hasSize(1);
    }

}
```

Spring Boot

Question

Pourquoi le test a-t-il échoué ?

Spring Boot

Question

Pourquoi le test a-t-il échoué ?

Réponse

Par défaut, **DataJpaTest** remplace le **DataSource** par une base de données embarquée.

Spring Boot

Pour indiquer à Spring que l'on ne souhaite pas remplacer notre base de données par une autre de test, on ajoute l'annotation suivante

```
package com.example.demo.dao;

import static org.assertj.core.api.Assertions.assertThat;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;


@DataJpaTest
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
class PersonneRepositoryTests {

        @Autowired
        private PersonneRepository personneRepository;

        @Test
        void testFindByNomContaining() {
                var wicks = personneRepository.findByNomContaining("wick");
                assertThat(wicks).isNotEmpty();
                assertThat(wicks).hasSize(1);
        }

}
```