# **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


1. Introduction
2. Avant de commencer
3. Premier exemple : application MVC
4. Solution avec `TestRestTemplate`
   - `@SpringBootTest`
   - `@LocalServerPort`
   - `TestRestTemplate`
5. Solution avec `MockMvc`
   - `@AutoConfigureMockMvc`
   - `MockMvc`
6. Deuxième exemple : API REST
   - `@MockBean`
   - Tests unitaires (cas d'un service)
   - Tests unitaires (cas d'un contrôleur)
   - Tests d'intégration (cas d'un contrôleur)
7. `@WebMvcTest`
8. `@DataJpaTest`

# 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
  - `cours-spring-test` dans `Name`,
  - `com.example` dans `Group`,
  - `cours-spring-test` 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`

**Dans `application.properties`, on ajoute les données concernant la connexion à la base de données et la configuration de `Hibernate`**

```
# server properties
server.servlet.context-path=/web
server.port=8080

# datasource properties
spring.datasource.url = jdbc:mysql://localhost:3306/cours_test?createDatabaseIfNotExist=true
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=

# spring data jpa properties
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

**Dans `application.properties`, on ajoute les données concernant la connexion à la base de données et la configuration de `Hibernate`**

```
# server properties
server.servlet.context-path=/web
server.port=8080

# datasource properties
spring.datasource.url = jdbc:mysql://localhost:3306/cours_test?createDatabaseIfNotExist=true
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=

# spring data jpa properties
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```



**Explication**

- Les propriétés ajoutées remplacent les beans utilisés par **Spring MVC**.
- La chaîne `createDatabaseIfNotExist=true` permet de créer la base de données si elle n'existe pas.

# 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`.

**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 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 les dépendances suivantes**

```
<!-- https://mvnrepository.com/artifact/jakarta.servlet.jsp.jstl/
   jakarta.servlet.jsp.jstl-api -->
<dependency>
    <groupId>jakarta.servlet.jsp.jstl</groupId>
    <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
    <version>3.0.0</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.glassfish.web/jakarta.
   servlet.jsp.jstl -->
<dependency>
    <groupId>org.glassfish.web</groupId>
    <artifactId>jakarta.servlet.jsp.jstl</artifactId>
</dependency>
```

# Spring Boot

**Contenu de** `application.properties`

```
# spring mvc 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>
```

**Commençons par modifier le point d'entrée pour ajouter des tuples dans la base de données**

```
package com.example.demo;

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

import com.example.demo.model.Personne;
import com.example.demo.service.PersonneService;

import lombok.AllArgsConstructor;

@SpringBootApplication
@AllArgsConstructor
public class CoursSpringTestApplication {

    private PersonneService personneService;

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

    @Bean
    public CommandLineRunner start() {
        return args -> {
            personneService.save(new Personne("Wick", "John", 45));
            personneService.save(new Personne("Linus", "Benjamin", 30));
            personneService.save(new Personne("Pradel", "Jacques", 65));
        };
    }
}
```

# 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 `HomeControllerTest` 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** `HomeControllerTest`

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

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

import org.junit.jupiter.api.Test;

class HomeControllerTest {

	@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 HomeControllerTest {

        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

## Comment tester si

- le contrôleur injectait quelques composants (services, repositories...) ?
- une action de contrôleur devait récupérer des `@RequestParam` ou des `@PathVariable` ?

# Spring Boot

## Comment tester si

- le contrôleur injectait quelques composants (services, repositories...) ?
- une action de contrôleur devait récupérer des `@RequestParam` ou des `@PathVariable` ?

## Plusieurs solutions : utiliser

- **TestRestTemplate**
- **MockMvc**
- ...

# Spring Boot

## TestRestTemplate

- Configurable avec `@SpringBootTest`.
- Démarre le conteneur de Servlets (un numéro de port est nécessaire).
- Permet de tester coté client (Impossible de vérifier le modèle ou le nom d'une vue) .

## MockMvc

- Mock d'une **Servlet**
- Ne démarre pas le conteneur de Servlets (le numéro de port n'est nécessaire).
- Permet de tester coté serveur (Possibilité de vérifier le modèle ou le nom d'une vue).

# Spring Boot

**Commençons par ajouter `@SpringBootTest` pour pouvoir configurer l'environnement d'exécution**

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

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest()
class HomeControllerTest {

        @Test
        public void testHome() throws Exception {
        }

}
```

# Spring Boot

## Remarque

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

# 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.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;

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

        @Test
        public void testHome() throws Exception {
        }

}
```

# 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.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;

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

        @Test
        public void testHome() throws Exception {
        }

}
```

`webEnvironment=RANDOM_PORT` permet de démarrer le serveur avec un port aléatoire (utile pour éviter les conflits dans les environnements de test).

# Spring Boot

**Utilisons `@LocalServerPort` pour récupérer le numéro de port généré**

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

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

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.web.server.LocalServerPort;

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

        @LocalServerPort
        private int port;

        @Test
        public void testHome() throws Exception {
        }

}
```

# Spring Boot

**Injectons `TestRestTemplate` pour tester notre contrôleur**

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

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.test.web.server.LocalServerPort;

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

    @Autowired
    private TestRestTemplate restTemplate;

    @LocalServerPort
    private int port;

    @Test
    public void testHome() throws Exception {
    }

}
```

# Spring Boot

**Vérifions que notre contrôleur récupère et passe correctement le paramètre à la vue**

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

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.test.web.server.LocalServerPort;

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

    @Autowired
    private TestRestTemplate restTemplate;

    @LocalServerPort
    private int port;

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

}
```

# Spring Boot

## Ne pas confondre TestRestTemplate avec RestTemplate

- RestTemplate : classe permettant de créer un client REST (consommer une API **REST**)
- TestRestTemplate : classe permettant de tester une API **REST**

# Spring Boot

## Remarque

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

# Spring Boot

**Commençons par activer l'auto-configuration des** `MockMvc`

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

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;

@AutoConfigureMockMvc
class HomeControllerTest {

        @Test
        public void testHome() throws Exception {
        }

}
```

# Spring Boot

**Injectons ensuite** `MockMvc`

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

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.test.web.servlet.MockMvc;

@AutoConfigureMockMvc
class HomeControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testHome() throws Exception {
    }

}
```

# Spring Boot

**Sans oublier d'ajouter l'annotation** `@SpringBootTest`

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

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;

@AutoConfigureMockMvc
@SpringBootTest
class HomeControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testHome() throws Exception {
    }

}
```

**Et enfin le test**

```
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;

@AutoConfigureMockMvc
@SpringBootTest
class HomeControllerTest {

    @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", "wick"));
    }

}
```

# 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

## Remarque

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

## Pour la suite, allons dans `cours-spring-rest` récupérer les packages suivants

- `com.example.demo.controller`
- `com.example.demo.service`
- `com.example.demo.model`
- `com.example.demo.dao`

# Spring Boot

## Préparons les classes de test suivantes

- Dans `com.example.demo.controller`
  - `PersonneRestControllerIntegrationTest`
  - `PersonneRestControllerUnitTest`
- Dans `com.example.demo.service`
  - `PersonneServiceUnitTest`

# Spring Boot

## Rappel

- **Tests unitaires** : on teste chaque unité séparément, donc les beans injectés dans la classes doivent être remplacés par des mocks
- **Tests d'intégration** : on teste plusieurs unités ensemble, les beans ne seront pas remplacés par des mocks

# Spring Boot

**Commençons par créer la classe** `PersonneRestControllerUnitTest`

```
class PersonneRestControllerUnitTest {

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

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

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

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

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

}
```

# Spring Boot

**Et** `PersonneServiceUnitTest`

```
class PersonneServiceUnitTest {

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

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

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

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

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

}
```

# Spring Boot

**Et** `PersonneRestControllerIntegrationTest`

```
class PersonneRestControllerIntegrationTest {

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

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

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

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

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

}
```

# Spring Boot

## Ensuite

- Ajoutons l'annotation `@SpringBootTest` de test dans les trois classes de test
- Ajoutons l'annotation `@@AutoConfigureMockMvc` uniquement dans `PersonneRestControllerUnitTest` et `PersonneRestControllerIntegrationTest`
- Injectons `MockMvc` uniquement dans `PersonneRestControllerUnitTest` et `PersonneRestControllerIntegrationTest`

# Spring Boot

**Contenu de** `PersonneRestControllerUnitTest`

```
@SpringBootTest
@AutoConfigureMockMvc
class PersonneRestControllerUnitTest {

    @Autowired
    private MockMvc mockMvc;

    // + le contenu précédent
}
```

# Spring Boot

**Contenu de** `PersonneRestControllerIntegrationTest`

```
@SpringBootTest
@AutoConfigureMockMvc
class PersonneRestControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    // + le contenu précédent

}
```

# Spring Boot

**Contenu de** `PersonneServiceUnitTest`

```
@SpringBootTest
class PersonneServiceUnitTest {

    // + le contenu précédent

}
```

# Spring Boot

## Remarque

- Les classes de test unitaire doivent être isolées et indépendantes.
- Elles ne doivent pas utiliser des beans comme les services, les repositories...
- Elles doivent les remplacer par des **Mock**

# Spring Boot

**Utilisons `@MockBean` dans `PersonneServiceUnitTest`**

```
@SpringBootTest
class PersonneServiceUnitTest {

    @MockBean
    private PersonneRepository personneRepository;

    // + le contenu précédent

}
```

# Spring Boot

**Et aussi dans** `PersonneRestControllerUnitTest`

```
@SpringBootTest
@AutoConfigureMockMvc
class PersonneRestControllerUnitTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private PersonneServiceImpl personneService;

    // + le contenu précédent

}
```

# Spring Boot

**Injectons aussi `@PersonneService` pour comparer ses résultats avec les résultats attendus**

```
@SpringBootTest
class PersonneServiceUnitTest {

    @MockBean
    private PersonneRepository personneRepository;


    @Autowired
    private PersonneService personneService;
```

# Spring Boot

**Commençons par** `testFindAll` **dans** `@PersonneServiceUnitTest`

```
@Test
void testFindAll() {
    List<Personne> fakedPersonnes = List.of(
        new Personne("Wick", "John", 45),
        new Personne("Linus", "Benjamin", 30),
        new Personne("Pradel", "Jacques", 65)
    );
    when(personneRepository.findAll()).thenReturn(fakedPersonnes);
    var result = personneService.findAll();
    assertIterableEquals(fakedPersonnes, result);
}
```

# Spring Boot

**Commençons par** `testFindAll` **dans** `@PersonneServiceUnitTest`

```
@Test
void testFindAll() {
    List<Personne> fakedPersonnes = List.of(
        new Personne("Wick", "John", 45),
        new Personne("Linus", "Benjamin", 30),
        new Personne("Pradel", "Jacques", 65)
    );
    when(personneRepository.findAll()).thenReturn(fakedPersonnes);
    var result = personneService.findAll();
    assertIterableEquals(fakedPersonnes, result);
}
```

**Ensuite** `testFindById`

```
@Test
void testFindById() {
    int id = 1;
    Optional<Personne> pers = Optional.of(new Personne(1, "Wick", "John", 45));
    when(personneRepository.findById(id)).thenReturn(pers);
    var res = personneService.findById(id);
    assertEquals(pers.get(), res);
    verify(personneRepository, times(1)).findById(id);
}
```

**Et** `testSave`

```
@Test
void testSave() {
    var pers = new Personne(4, "Wick", "John", 45);
    when(personneRepository.save(pers)).thenReturn(pers);
    Personne resultat = personneService.save(pers);
    assertEquals(pers, resultat);
    verify(personneRepository, times(1)).save(pers);
}
```

**Et** `testSave`

```
@Test
void testSave() {
    var pers = new Personne(4, "Wick", "John", 45);
    when(personneRepository.save(pers)).thenReturn(pers);
    Personne resultat = personneService.save(pers);
    assertEquals(pers, resultat);
    verify(personneRepository, times(1)).save(pers);
}
```

**Et** `testDeleteById`

```
@Test
void testDeleteById() {
    int id = 1;
    personneService.deleteById(id);
    verify(personneRepository, times(1)).deleteById(id);
}
```

**Et** `testSave`

```
@Test
void testSave() {
    var pers = new Personne(4, "Wick", "John", 45);
    when(personneRepository.save(pers)).thenReturn(pers);
    Personne resultat = personneService.save(pers);
    assertEquals(pers, resultat);
    verify(personneRepository, times(1)).save(pers);
}
```

**Et** `testDeleteById`

```
@Test
void testDeleteById() {
    int id = 1;
    personneService.deleteById(id);
    verify(personneRepository, times(1)).deleteById(id);
}
```

**Et enfin** `testUpdate`

```
@Test
void testUpdate() {
    var pers = new Personne(3, "Doe", "Johnny", 25);
    when(personneRepository.save(pers)).thenReturn(pers);
    Personne resultat = personneService.save(pers);
    assertEquals(pers, resultat);
    verify(personneRepository, times(1)).save(pers);
}
```

# Spring Boot

**Commençons par** `get` **dans** `@PersonneRestControllerUnitTest`

```
@Test
void testGetPersonnes() throws Exception {
    List<Personne> fakedPersonnes = List.of(
        new Personne("Wick", "John", 45),
        new Personne("Linus", "Benjamin", 30),
        new Personne("Pradel", "Jacques", 65)
    );
    when(personneRepository.findAll()).thenReturn(fakedPersonnes);
    mockMvc
    .perform(get("/personnes"))
//  .andDo(print())
    .andExpect(status().isOk())
    .andExpect(content().contentType(MediaType.APPLICATION_JSON))
    .andExpect(jsonPath("$").isNotEmpty())
    .andExpect(jsonPath("$[0].nom").value("Wick"));
    verify(personneService, times(1)).findAll();
}
```

# 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.

**Et `get` selon l'identifiant (cas de réussite)**

```
@Test
void testGetPersonne_Success() throws Exception {
    int id = 1;
    var pers = new Personne(1, "Wick", "John", 45);
    when(personneService.findById(id)).thenReturn(pers);
    mockMvc
    .perform(get("/personnes/1"))
//  .andDo(print())
    .andExpect(status().isOk())
    .andExpect(content().contentType(MediaType.APPLICATION_JSON))
    .andExpect(jsonPath("$").isNotEmpty())
    .andExpect(jsonPath("$.nom").value("Wick"));
    verify(personneService, times(1)).findById(id);
}
```

**Et `get` selon l'identifiant (cas de réussite)**

```
@Test
void testGetPersonne_Success() throws Exception {
    int id = 1;
    var pers = new Personne(1, "Wick", "John", 45);
    when(personneService.findById(id)).thenReturn(pers);
    mockMvc
    .perform(get("/personnes/1"))
//  .andDo(print())
    .andExpect(status().isOk())
    .andExpect(content().contentType(MediaType.APPLICATION_JSON))
    .andExpect(jsonPath("$").isNotEmpty())
    .andExpect(jsonPath("$.nom").value("Wick"));
    verify(personneService, times(1)).findById(id);
}
```

**Et le cas échec**

```
@Test
void testGetPersonne_NotFoundException() throws Exception {
    int id = 1000;
    when(personneService.findById(id)).thenThrow(new ResponseStatusException(HttpStatus.
      NOT_FOUND));
    mockMvc
    .perform(get("/personnes/1000"))
//  .andDo(print())
    .andExpect(status().isNotFound());
    verify(personneService, times(1)).findById(id);
}
```

**Et `delete` (cas de réussite)**

```
@Test
void testDeletePersonne_Success() throws Exception {
    int id = 1;
    var pers = new Personne(1, "Wick", "John", 45);
    when(personneService.findById(id)).thenReturn(pers);
    doNothing().when(personneService).deleteById(id);
    mockMvc
    .perform(delete("/personnes/1"))
    .andExpect(status().isNoContent());
    verify(personneService, times(1)).findById(id);
    verify(personneService, times(1)).deleteById(id);
}
```

**Et `delete` (cas de réussite)**

```
@Test
void testDeletePersonne_Success() throws Exception {
    int id = 1;
    var pers = new Personne(1, "Wick", "John", 45);
    when(personneService.findById(id)).thenReturn(pers);
    doNothing().when(personneService).deleteById(id);
    mockMvc
    .perform(delete("/personnes/1"))
    .andExpect(status().isNoContent());
    verify(personneService, times(1)).findById(id);
    verify(personneService, times(1)).deleteById(id);
}
```

**Et le cas échec**

```
@Test
void testDeletePersonne_NotFoundException() throws Exception {
    int id = 1000;
    when(personneService.findById(id)).thenReturn(null);
    mockMvc
    .perform(delete("/personnes/1000"))
//  .andDo(print())
    .andExpect(status().isNotFound());
    verify(personneService, times(1)).findById(id);
    verify(personneService, never()).deleteById(id);
}
```

# Spring Boot

**Pour envoyer des données sérialisées avec `MockMvc`, il faut injecter `ObjectMapper`**

```
@Autowired
private ObjectMapper objectMapper;
```

# Spring Boot

**Pour envoyer des données sérialisées avec `MockMvc`, il faut injecter `ObjectMapper`**

```
@Autowired
private ObjectMapper objectMapper;
```

**Et `post`**

```
@Test
void testAddPersonne() throws Exception {
    var pers = new Personne("Wick", "John", 45);

    when(personneService.save(pers)).then(invocation -> {
        pers.setId(111);
        return pers;
    });
    String json = objectMapper.writeValueAsString(pers);
    System.out.println(json);
    var result = mockMvc.perform(post("/personnes")
            .content(json)
            .contentType(MediaType.APPLICATION_JSON));
    result
            .andExpect(status().isCreated())
            .andExpect(content().contentType(MediaType.APPLICATION_JSON))
            .andExpect(jsonPath("$.id").value(111))
            .andExpect(jsonPath("$.nom").value("Wick"));
    verify(personneService, times(1)).save(any(Personne.class));
}
```

# Spring Boot

## Exercice

Implémentez et testez tous les cas possibles pour `put`.

# Spring Boot

**Commençons par** `get` **dans** `@PersonneRestControllerIntegrationTest`

```
@Test
void testGetPersonnes() throws Exception {
  mockMvc.perform(get("/personnes"))
   .andDo(print())
   .andExpect(status().isOk())
   .andExpect(content().contentType(MediaType.APPLICATION_JSON))
   .andExpect(jsonPath("$").isNotEmpty())
   .andExpect(jsonPath("$[0].nom").value("Wick"));
}
```

# Spring Boot

`get` **selon l'identifiant et** `delete`

```
@Test
void testGetPersonne_Success() throws Exception {
  mockMvc.perform(get("/personnes/1"))
  .andExpect(status().isOk())
  .andExpect(content().contentType(MediaType.APPLICATION_JSON))
  .andExpect(jsonPath("$").isNotEmpty())
  .andExpect(jsonPath("$.nom").value("Wick"));
}

@Test
void testGetPersonne_NotFounfException() throws Exception {
  mockMvc.perform(get("/personnes/1000"))
  .andExpect(status().isNotFound());
}

@Test
void testDeletePersonne_Success() throws Exception {
  mockMvc.perform(delete("/personnes/1"))
  .andExpect(status().isNoContent());
}

@Test
void testDeletePersonne_NotFounfException() throws Exception {
  mockMvc.perform(delete("/personnes/1000"))
  .andExpect(status().isNotFound());
}
```

# Spring Boot

## Exercice

Implémentez et testez tous les cas possibles pour `post` et `put`.

# Spring Boot

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

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

        // le code précédent

}
```

# Spring Boot

## **Spring Boot** nous permet de tester

- les contrôleurs
- les services
- et 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);
        }

}
```