



Développement d'applications modulaires en Java

Chouki Tibermacine

Chouki.Tibermacine@umontpellier.fr



Plan de l'ECUE

1. (Rappels sur le) Développement d'applications Web avec Java
2. Modulariser les applications Java avec Spring
3. Bien structurer une application Web avec Spring MVC
4. Auto-configurer une application Web avec Spring Boot
5. Sécuriser une application Web avec Spring Security
6. Gérer des données massives avec Apache Kafka et Spring
7. Tester une application Web Spring
8. Écrire des applications Web (API) réactives avec Spring WebFlux

Plan du cours

- 1. Automatisation des tests en Java avec JUnit**
2. Introduction aux tests avec Spring
3. Tests unitaires avec un backend SQL
4. Tests d'intégration

Premiers tests automatisés en Java

- Je vous invite à consulter d'abord le cours sur les assertions et JUnit, disponible dans le dépôt Git
- C'est un cours de prog par objets
- Si vous n'avez pas beaucoup pratiqué les tests avant ce cours, lisez attentivement le contenu du cours, et réalisez les exemples
- Sinon, parcourez rapidement les diapos du cours et passez à la suite de ce cours

Plan du cours

1. Automatisation des tests en Java avec JUnit
2. Introduction aux tests avec Spring
3. Tests unitaires avec un backend SQL
4. Tests d'intégration

Tests dans une application Web

- Vous avez probablement remarqué le dossier test créé par Spring Initializr ou par l'IDE pour votre projet Gradle
- Ce dossier comporte une classe qui porte le nom de votre projet + ApplicationTests
- Cette classe est annotée @SpringBootTest
- Cette annotation indique à Spring Boot qu'il faut rechercher une classe principale de configuration (annotée @SpringBootApplication, par ex) et l'utiliser pour démarrer/créer le contexte de l'application (tous les objets qui composent l'application : contrôleurs, modèles, ...)
- La classe de test comporte des méthodes de test (cas de tests)
- On peut définir plusieurs classes de test annotées de cette façon

Un premier test Spring

Modifier la classe en :

- ajoutant un attribut auto-injecté par une référence à l'un de vos contrôleurs. Par exemple :

```
@Autowired  
private HomeController controller;
```

- Dans la méthode de test ajouter une assertion :

```
@Test  
public void contextLoads() throws Exception {  
    assertThat(controller).isNotNull();  
}
```

Ajouter un import statique des méthodes assertXxx() :

```
import static org.assertj.core.api.Assertions.*;
```

Exécuter le test

- Démarrer le test en exécutant la tâche verification/test (tool-window Gradle à droite), ou bien clique bouton droit sur la méthode puis Run
- Vous allez voir le test passer au vert
- C'est normal, parce que le contrôleur est bien instancié par Spring Boot lors de la création du contexte de l'application
- L'étape suivante est de tester le comportement de l'app Web
- Tester l'envoi d'une requête HTTP et vérifier le contenu de la réponse (voir diapo suivante)

Classe de test d'une requête HTTP

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

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void greetingShouldReturnDefaultMessage() throws
        Exception {
        assertThat(this.restTemplate.getForObject("http://
            localhost:" + port + "/",
            String.class)).contains("Hello , World");
    }
}
```

Classe de test d'une requête HTTP -suite-

Plusieurs facilités offertes par Spring, utilisées dans l'exemple précédent :

- L'utilisation de `webEnvironment=RANDOM_PORT` permet le démarrage du serveur avec un port aléatoire (utile pour éviter les conflits de ports dans un environnement de tests)
- L'injection du port avec l'annotation `@LocalServerPort`
- L'attribut de type `TestRestTemplate` permet de tester l'envoi de requêtes HTTP ; il a été auto-injecté avec `Autowired` (bean disponible grâce à l'annotation `@SpringBootTest`)

Surcoût de démarrer un serveur Web

- Pour éviter le surcoût induit par un serveur Web, on peut utiliser Spring MockMvc
- Ceci permet de tester tout ce qu'il y a derrière le serveur Web comme application, mais sans devoir démarrer le serveur
- Il suffit d'ajouter l'annotation suivante à la classe de test :
`@AutoConfigureMockMvc`

Exemple avec Spring MockMvc

```
@SpringBootTest
@AutoConfigureMockMvc
public class TestingWebApplicationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void shouldReturnDefaultMessage() throws Exception {
        this.mockMvc.perform(get("/")).andDo(print())
            .andExpect(status().isOk())
            .andExpect(content()
                .string(containsString("Hello , World")));
    }
}
```

Ceci permet de démarrer tout le contexte d'application Spring (tous ses objets), mais pas le serveur

Exemple avec Spring WebMvcTest

- On peut limiter les tests à la couche Web avec @WebMvcTest

```
@WebMvcTest
public class WebLayerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void shouldReturnDefaultMessage() throws Exception {
        this.mockMvc.perform(get("/")).andDo(print())
            .andExpect(status().isOk())
            .andExpect(content().string(
                containsString("Hello , World")));
    }
}
```

Noter l'utilisation de MockMvc pour envoyer une requête HTTP, obtenir la réponse et enchaîner avec des assertions sur cette réponse

Exemple avec Spring WebMvcTest -suite-

- Dans l'exemple précédent, Spring Boot instancie les objets de la couche Web seulement (pas tout le contexte)
- Dans une application avec plusieurs contrôleurs, on peut restreindre les tests à un seul contrôleur avec `@WebMvcTest(HomeController.class)`
- Si le contrôleur a des dépendances vers d'autres objets de l'application (des objets `@Service`, par exemple, sur lesquels il invoque des méthodes), on doit utiliser un framework pour créer un Mock (simulacre) pour les simuler, comme Mockito :
<https://site.mockito.org/>
Tuto : <https://www.vogella.com/tutorials/Mockito/article.html>
- Dans Spring, l'intégration de Mockito est transparente, avec des objets `MockBean` (toute la config de ce framework est faite automatiquement)

Exemple avec intégration de Mockito

```
@Controller
public class GreetingController {
    private final GreetingService service;
    public GreetingController(GreetingService service) {
        this.service = service;
    }
    @RequestMapping("/greeting")
    public @ResponseBody String greeting() {
        return service.greet();
    }
}
```

```
@Service
public class GreetingService {
    public String greet() {
        return "Hello, World";
    }
}
```

Exemple avec intégration de Mockito -suite-

```
@WebMvcTest( GreetingController.class )
public class WebMockTest {
    @Autowired
    private MockMvc mockMvc;
    @MockBean
    private GreetingService service;
    @Test
    public void greetingShouldReturnMessageFromService() throws
        Exception {
        when(service.greet()).thenReturn("Hello , Mock");
        this.mockMvc.perform(get("/greeting")).andDo(print())
            .andExpect(status().isOk()).andExpect(content()
                .string(containsString("Hello , Mock")));
    }
}
```

GreetingService est remplacé par un mock (créé et injecté par Spring) dont le comportement est décrit par la 1ère ligne de code de la méthode de test

Plan du cours

1. Automatisation des tests en Java avec JUnit
2. Introduction aux tests avec Spring
3. Tests unitaires avec un backend SQL
4. Tests d'intégration

Stratégie de test pour une app Web Spring

Réaliser des tests unitaires pour (dans l'ordre) :

1. les contrôleurs
2. les services, le cas échéant
3. les repositories

Ensuite, réaliser des tests d'intégration de ces trois couches

1. Tester les contrôleurs

Utiliser MockMvc tel qu'on l'a vu dans la diapo [12](#) pour :

- exécuter des requêtes HTTP vers le contrôleur
- valider les réponses HTTP
- valider les entêtes HTTP
- valider les corps des réponses

Écrire un test

- La première partie de la méthode de test (la section *Given* d'un *test case*) comporte la mise en place du Mock :

```
@WebMvcTest( GreetingController.class )
public class WebMockTest {
    @Autowired
    private MockMvc mockMvc;
    @MockBean
    private GreetingService service;
    @Test
    public void greetingShouldReturnMessageFromService()
        throws Exception {
        when(service.greet()).thenReturn("Hello, Mock");
        // ...
    }
}
```

Ici, il faut lire l'instruction de la façon suivante : quand (*when*) la méthode *greet()* est invoquée sur l'objet simulé, celle-ci doit retourner la valeur indiquée en paramètre de *thenReturn*

Écrire un test -suite-

- La 2ème partie comporte l'exécution de la requête HTTP (When d'un test case) et la validation de la réponse :

```
this .mockMvc.perform(get("/greeting")).andDo(print())
// Verification du header
.andExpect(status().isOk())
// Verification du contenu
.andExpect(content()
    .string(containsString("Hello , Mock")));

```

- `andExpect()` permet de tester une hypothèse (en prenant en paramètre un *result matcher*) et de chaîner les tests

- `status()` et `content()` sont des méthodes importées statiquement du framework Spring Test :

`org.springframework.test.web.servlet.result.MockMvcResultMatchers`

- `containsString()` provient de Hamcrest (framework de matchers) : <http://hamcrest.org/JavaHamcrest/tutorial>

Un exemple plus élaboré - test service REST

```
@Test
@DisplayName("GET /product/1 - Found")
void testGetProductByIdFound() throws Exception {
    // Setup our mocked service
    Product mockProduct = new Product( id: 1, name: "Product Name", quantity: 10, version: 1 );
    doReturn(Optional.of(mockProduct)).when(service).findById(1);

    // Execute the GET request
    mockMvc.perform(get( urlTemplate: "/product/{id}", ...uriVars: 1))

        // Validate the response code and content type
        .andExpect(status().isOk())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8_VALUE))

        // Validate the headers
        .andExpect(header().string(HttpHeaders.ETAG, value: "\"1\""))
        .andExpect(header().string(HttpHeaders.LOCATION, value: "/product/1"))

        // Validate the returned fields
        .andExpect(jsonPath( expression: "$.id", is( value: 1)))
        .andExpect(jsonPath( expression: "$.name", is( value: "Product Name")))
        .andExpect(jsonPath( expression: "$.quantity", is( value: 10)))
        .andExpect(jsonPath( expression: "$.version", is( value: 1)));
}
```

Ici, service est un service simulé avec le mock. Quand sa méthode findById(1) est invoqué, il doit retourner l'objet mockProduct

Dans l'exemple précédent

- Est-ce que vous avez remarqué le header ETag ?
- A quoi sert-il ?
- Un header très utile dans la définition d'API REST, notamment pour gérer les versions des ressources (users, locations, ...)
- Utile dans la gestion des requêtes PUT (mise à jour d'une ressource) pour détecter les conflits (si jamais lors de la mise à jour, la version d'une ressource a changé depuis le dernier GET (elle a été mise à jour par un autre utilisateur). Il y a donc un conflit : erreur HTTP 409 renvoyée normalement)
- Plus de détails ici :

<https://www.baeldung.com/etags-for-rest-with-spring>

Toujours dans l'exemple précédent

- Est-ce que vous connaissez JsonPath ?
- Un langage de requêtes de données JSON (requêtes déclaratives à la manière de XPath)
- Un interprète de ce langage est fourni via la méthode `jsonPath` de `MockMvcResultMatchers` de Spring
- \$ dans l'exemple veut dire l'élément racine de la donnée JSON dans la réponse HTTP
- Plus de détails sur JsonPath :

<https://www.baeldung.com/guide-to-jayway-jsonpath>

Test service REST - Get Not Found

```
@Test
@DisplayName("GET /product/1 – Not Found")
void testGetProductByIdNotFound() throws Exception {
    // Setup our mocked service
    doReturn(Optional.empty()).when(service).findById(1);

    // Execute the GET request
    mockMvc.perform(get(urlTemplate: "/product/{id}",
        ...uriVars: 1))
        // Validate that we get a 404 Not Found response
        .andExpect(status().isNotFound());
}
```

- On doit tester le scénario lorsque la requête HTTP réussit, mais aussi lorsque ça ne réussit pas
- Ici, lorsqu'on fait un GET d'un produit qui n'existe pas, on teste qu'on a bien en retour un 404

Test service REST - POST qui réussit

```
@Test
@DisplayName("POST /product - Success")
void testCreateProduct() throws Exception {
    // Setup mocked service
    Product postProduct = new Product( name: "Product Name", quantity: 10);
    Product mockProduct = new Product( id: 1, name: "Product Name", quantity: 10, version: 1);
    doReturn(mockProduct).when(service).save(any());

    mockMvc.perform(post( urlTemplate: "/product")
        .contentType(MediaType.APPLICATION_JSON)
        .content(asJsonString(postProduct)))

        // Validate the response code and content type
        .andExpect(status().isCreated())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8_VALUE))

        // Validate the headers
        .andExpect(header().string(HttpHeaders.ETAG, value: "\"1\""))
        .andExpect(header().string(HttpHeaders.LOCATION, value: "/product/1"))

        // Validate the returned fields
        .andExpect(jsonPath( expression: "$.id", is( value: 1)))
        .andExpect(jsonPath( expression: "$.name", is( value: "Product Name")))
        .andExpect(jsonPath( expression: "$.quantity", is( value: 10)))
        .andExpect(jsonPath( expression: "$.version", is( value: 1)));
}
```

- Deux objets ici, l'un passé en paramètre de la requête POST et l'autre utilisé comme Mock

Dans l'exemple précédent

- Noter la préparation des données de la requête dans l'invocation à `perform()`
- `asJsonString()` est une méthode qui n'est pas montrée, mais qui utilise simplement la bibliothèque Jackson pour sérialiser l'objet en JSON (voir diapo 59)
- On s'assure ensuite que la réponse a le *status Ok* (HTTP 200)
- On vérifie ensuite les headers et le contenu (charge utile) de la réponse

Des exemples, que vous pouvez copier, sont donnés à la fin du cours

Test service REST - PUT qui réussit

```
@Test
@DisplayName("PUT /product/1 - Success")
void testProductPutSuccess() throws Exception {
    // Setup mocked service
    Product putProduct = new Product(name: "Product Name", quantity: 10);
    Product mockProduct = new Product(id: 1, name: "Product Name", quantity: 10, version: 1);
    doReturn(Optional.of(mockProduct)).when(service).findById(1);
    doReturn(toBeReturned: true).when(service).update(any());

    mockMvc.perform(put(urlTemplate: "/product/{id}", ...uriVars: 1)
        .contentType(MediaType.APPLICATION_JSON)
        .header(HttpHeaders.IF_MATCH, ...values: 1)
        .content(asJsonString(putProduct)))

    // Validate the response code and content type
    .andExpect(status().isOk())
    .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8_VALUE))

    // Validate the headers
    .andExpect(header().string(HttpHeaders.ETAG, value: "\"2\""))
    .andExpect(header().string(HttpHeaders.LOCATION, value: "/product/1"))

    // Validate the returned fields
    .andExpect(jsonPath(expression: "$.id", is(value: 1)))
    .andExpect(jsonPath(expression: "$.name", is(value: "Product Name")))
    .andExpect(jsonPath(expression: "$.quantity", is(value: 10)))
    .andExpect(jsonPath(expression: "$.version", is(value: 2)));
}
```

- Là aussi, deux objets sont utilisés (param requête et Mock)

Dans l'exemple précédent

- Noter la préparation des données de la requête dans l'invocation à `perform()` avec cette fois un header `IF_MATCH` pour vérifier le ETag
- On s'assure que la réponse a le *status Created* (HTTP 201)
- On vérifie ensuite les headers et le contenu (charge utile) de la réponse

Test service REST - PUT qui échoue - Conflit

```
@Test
@DisplayName("PUT /product/1 – Version Mismatch")
void testProductPutVersionMismatch() throws Exception {
    // Setup mocked service
    Product putProduct = new Product( name: "Product Name", quantity: 10);
    Product mockProduct = new Product( id: 1, name: "Product Name", quantity: 10, version: 2);
    doReturn(Optional.of(mockProduct)).when(service).findById(1);
    doReturn(toBeReturned: true).when(service).update(any());

    mockMvc.perform(put( urlTemplate: "/product/{id}", ...uriVars: 1)
        .contentType(MediaType.APPLICATION_JSON)
        .header(HttpHeaders.IF_MATCH, ...values: 1)
        .content(asJsonString(putProduct)))

        // Validate the response code and content type
        .andExpect(status().isConflict());
}
```

- Noter la valeur de `IF_MATCH` (égale à 1) différente de 2 dans l'objet Mock
- Le numéro de version est utilisé dans cet exemple comme ETag
- La réponse attendue doit avoir comme *status Conflict* (HTTP 409)

Test service REST - PUT qui échoue - Not Found

```
@Test
@DisplayName("PUT /product/1 – Not Found")
void testProductPutNotFound() throws Exception {
    // Setup mocked service
    Product putProduct = new Product( name: "Product Name", quantity: 10);
    doReturn(Optional.empty()).when(service).findById(1);

    mockMvc.perform(put( urlTemplate: "/product/{id}", ...uriVars: 1)
        .contentType(MediaType.APPLICATION_JSON)
        .header(HttpHeaders.IF_MATCH, ...values: 1)
        .content(asJsonString(putProduct)))

        // Validate the response code and content type
        .andExpect(status().isNotFound());
}
```

- Noter la réponse du service simulé par un `Optional.empty()` (absence de valeur pour simuler une ressource introuvable)
- La réponse attendue doit avoir comme `status Not Found` (HTTP 404)

Test service REST - DELETE

Tester :

- Delete qui réussit (une ressource qui existe)
- Delete qui échoue (une ressource qui n'existe pas)
- Delete qui échoue (erreur interne, dans la base de données par ex)

Test service REST - DELETE qui réussit

```
@Test
@DisplayName("DELETE /product/1 – Success")
void testProductDeleteSuccess() throws Exception {
    // Setup mocked product
    Product mockProduct = new Product( id: 1, name: "Product Name", quantity: 10, version: 1);

    // Setup the mocked service
    doReturn(Optional.of(mockProduct)).when(service).findById(1);
    doReturn( toBeReturned: true).when(service).delete( id: 1);

    // Execute our DELETE request
    mockMvc.perform(delete( urlTemplate: "/product/{id}", ...uriVars: 1))
        .andExpect(status().isOk());
}
```

- On teste simplement le fait que le *status* de la réponse est Ok

Test service REST - DELETE qui échoue

```
@Test
@DisplayName("DELETE /product/1 - Not Found")
void testProductDeleteNotFound() throws Exception {
    // Setup the mocked service
    doReturn(Optional.empty()).when(service).findById(1);

    // Execute our DELETE request
    mockMvc.perform(delete( urlTemplate: "/product/{id}", ...uriVars: 1))
        .andExpect(status().isNotFound());
}

@Test
@DisplayName("DELETE /product/1 - Failure")
void testProductDeleteFailure() throws Exception {
    // Setup mocked product
    Product mockProduct = new Product( id: 1, name: "Product Name", quantity: 10, version: 1);

    // Setup the mocked service
    doReturn(Optional.of(mockProduct)).when(service).findById(1);
    doReturn( toBeReturned: false).when(service).delete( id: 1);

    // Execute our DELETE request
    mockMvc.perform(delete( urlTemplate: "/product/{id}", ...uriVars: 1))
        .andExpect(status().isInternalServerError());
}
```

Contrôleur DELETE, qui était testé

```
/**  
 * Deletes the product with the specified ID.  
 * @param id      The ID of the product to delete.  
 * @return        A ResponseEntity with one of the following status codes:  
 *               200 OK if the delete was successful  
 *               404 Not Found if a product with the specified ID is not found  
 *               500 Internal Service Error if an error occurs during deletion  
 */  
@DeleteMapping("/product/{id}")  
public ResponseEntity<?> deleteProduct(@PathVariable Integer id) {  
  
    logger.info(s: "Deleting product with ID {}", id);  
  
    // Get the existing product  
    Optional<Product> existingProduct = productService.findById(id);  
  
    return existingProduct.map(p -> {  
        if (productService.delete(p.getId())) {  
            return ResponseEntity.ok().build();  
        } else {  
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();  
        }  
    }).orElse(ResponseEntity.notFound().build());  
}
```

2. Tester les services

- Comme vu dans les cours précédents, les services implémentent la logique métier (pour décharger le contrôleur de cette tâche)
- Ils servent comme intermédiaire entre les contrôleurs et les repositories
- Dans les exemples donnés précédemment, on a 4 méthodes : find, save, update et delete

Interface du service à tester

```
public interface ProductService {  
    /**  
     * Returns the product with the specified id.  
     *  
     * @param id           ID of the product to retrieve.  
     * @return            The requested Product if found.  
     */  
    Optional<Product> findById(Integer id);  
  
    /**  
     * Returns all products in the database.  
     *  
     * @return            All products in the database.  
     */  
    List<Product> findAll();  
  
    /**  
     * Updates the specified product, identified by its id.  
     *  
     * @param product    The product to update.  
     * @return           True if the update succeeded, otherwise false.  
     */  
    boolean update(Product product);  
  
    /**  
     * Saves the specified product to the database.  
     *  
     * @param product    The product to save to the database.  
     * @return           The saved product.  
     */  
    Product save(Product product);
```

Implémentation du service à tester

```
@Service
public class ProductServiceImpl implements ProductService {

    private final ProductRepository productRepository;

    public ProductServiceImpl(ProductRepository productRepository) { this.productRepository = productRepository; }

    @Override
    public Optional<Product> findById(Integer id) { return productRepository.findById(id); }

    @Override
    public List<Product> findAll() { return productRepository.findAll(); }

    @Override
    public boolean update(Product product) { return productRepository.update(product); }

    @Override
    public Product save(Product product) {
        product.setVersion(1);
        return productRepository.save(product);
    }

    @Override
    public boolean delete(Integer id) { return productRepository.delete(id); }
}
```

Tester le service – find

```
/**  
 * The service that we want to test.  
 */  
@Autowired  
private ProductService service;  
  
/**  
 * A mock version of the ProductRepository for use in our tests.  
 */  
@MockBean  
private ProductRepository repository;  
  
@Test  
@DisplayName("Test findById Success")  
void testfindByIdSuccess() {  
    // Setup our mock  
    Product mockProduct = new Product( id: 1, name: "Product Name", quantity: 10, version: 1);  
    doReturn(Optional.of(mockProduct)).when(repository).findById(1);  
  
    // Execute the service call  
    Optional<Product> returnedProduct = service.findById(1);  
  
    // Assert the response  
    Assertions.assertThat(returnedProduct.isPresent(), message: "Product was not found");  
    Assertions.assertThat(returnedProduct.get(), mockProduct, message: "Products should be the same");  
}
```

- Noter l'utilisation du même mécanisme de Mock, mais cette fois pour le repository
- On utilise ici des assertions JUnit pour vérifier le résultat

Tester le service – find Not Found

```
@Test
@DisplayName("Test findById Not Found")
void testfindByIdNotFound() {
    // Setup our mock
    Product mockProduct = new Product( id: 1, name: "Product Name", quantity: 10, version: 1);
    doReturn(Optional.empty()).when(repository).findById(1);

    // Execute the service call
    Optional<Product> returnedProduct = service.findById(1);

    // Assert the response
    Assertions.assertFalse(returnedProduct.isPresent(), message: "Product was found, when it shouldn't be");
}
```

- On crée un Mock pour le repository, comme précédemment
- On invoque findById() sur le service, comme précédemment
- On fait un assertFalse() ici

Tester le service – findAll

```
@Test
@DisplayName("Test findAll")
void testfindAll() {
    // Setup our mock
    Product mockProduct = new Product( id: 1, name: "Product Name", quantity: 10, version: 1);
    Product mockProduct2 = new Product( id: 2, name: "Product Name 2", quantity: 15, version: 3);
    doReturn(Arrays.asList(mockProduct, mockProduct2)).when(repository).findAll();

    // Execute the service call
    List<Product> products = service.findAll();

    Assertions.assertEquals( expected: 2, products.size(), message: "findAll should return 2 products");
}
```

- On crée un Mock pour le repository, comme précédemment, avec une liste de deux objets simulés
- On invoque findAll() sur le service
- Enfin, on fait un assertEquals()

Tester le service – save

```
@Test
@DisplayName("Test save product")
void testSave() {
    Product mockProduct = new Product( id: 1, name: "Product Name", quantity: 10);
    doReturn(mockProduct).when(repository).save(any());
    Product returnedProduct = service.save(mockProduct);

    Assertions.assertNotNull(returnedProduct, message: "The saved product should not be null");
    Assertions.assertEquals( expected: 1, returnedProduct.getVersion().intValue(),
        message: "The version for a new product should be 1");
}
```

- Deux assertions ici. Vérifier que : 1) save retourne un objet et 2) le numéro de version est égal à 1

Il faudrait prévoir aussi des tests pour les autres méthodes (update et delete)

3. Tester le repository

- Dans le repository, méthodes équivalentes à celles du service
- Elles interagissent directement avec la BdD
- Pour tester un repository (les requêtes SQL), on a le choix entre :
 - Ne pas interagir avec une BdD et utiliser à la place un Mock pour la DataSource : solution simple et efficace, mais ne permet pas de vérifier concrètement si les requêtes SQL s'exécutent correctement
 - Tester sur une vraie BdD : permet de s'assurer que les requêtes SQL sont correctes et si jamais on les change/optimise plus tard, on peut re-tester leur bon fonctionnement

Pour la même raison, on recommande plutôt des tests unitaires pour les repositories et non pas des tests d'intégration

Config dédiée pour tester sur une vraie BdD

- Il faut mettre en place un environnement de test BdD dédié
- On a vu la solution basée sur les profils dans le cours sur Spring Boot (avec un fichier application-test.properties)
Ce qui est indiqué dans le fichier application.properties sera chargé pour tous les profils
- N'importe quelle classe de bean (annotée @Component, @Configuration, @Controller, ...) peut être annotée @Profile("test") pour limiter sa portée au profil
- On a vu dans le cours sur Spring Boot comment on peut activer un profil avec une option de la JVM (-D...). On peut le faire aussi comme paramètre de config Spring Boot :
`spring.profiles.active=test`

Config dédiée pour tester sur une vraie BdD -suite-

- Souvent, on utilise, pour des raisons de performance, des bases de données *in-memory* pour les tests, comme H2 :

```
spring.datasource.driver-class-name=org.h2.Driver  
spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=-1  
spring.datasource.username=sa  
spring.datasource.password=sa
```

A déclarer dans un fichier application-test.properties

- Ensuite, nous allons utiliser DBUnit (framework de tests de bases de données)
- Nous allons l'utiliser pour créer les tables et insérer des données minimalistes (un dataset de base) et aussi pour rafraîchir/enrichir la BdD à chaque exécution de test

Concrètement pour utiliser DBUnit

- Ajouter les dépendances suivantes dans votre script de build Gradle :

```
testImplementation('org.dbunit:dbunit:2.7.0')
testImplementation('com.github.database-rider:rider-
    core:1.18.0')
testImplementation('com.github.database-rider:rider-
    junit5:1.18.0')
testImplementation('com.github.springtestdbunit:spring-
    test-dbunit:1.3.0')
testImplementation('com.h2database:h2:1.4.200')
```

- Ensuite écrire une classe de tests :

```
@ExtendWith({DBUnitExtension.class,
             SpringExtension.class})
@SpringBootTest
@ActiveProfiles("test")
public class UserRepositoryTest { /* ... */ }
```

Concrètement pour utiliser DBUnit -suite-

- Préparer un fichier Yaml pour le dataset minimal et le stocker comme : src/test/resources/datasets/users.yml :

```
users:  
  - id: 1  
    prenom: "Yan"  
    nom: "Brooks"  
    age: 47  
  - id: 2  
    prenom: "Dan"  
    nom: "Waynes"  
    age: 54
```

- Première ligne : nom de la table
- Les autres champs doivent correspondre aux champs de la table

Concrètement pour utiliser DBUnit -suite-

- Définir un fichier schema.sql pour créer la table dans src/main/resources :

```
CREATE TABLE IF NOT EXISTS users (
    id INTEGER NOT NULL AUTO_INCREMENT,
    prenom VARCHAR(30) NOT NULL,
    nom VARCHAR(30) NOT NULL,
    age INTEGER NOT NULL,
    PRIMARY KEY (id)
);
```

Concrètement pour utiliser DBUnit -suite-

- Dans la classe de test ajouter :

```
@Autowired
private DataSource dataSource;
public ConnectionHolder getConnectionHolder() {
    return () -> dataSource.getConnection();
}
@Autowired
private UserRepository repository;
@Test
@DataSet("users.yml")
void testfindAll() {
    List<User> users =
        Lists.newArrayList(repository.findAll());
    Assertions.assertEquals(2, users.size(),
        "Expected 2 users in the database");
}
```

Ici, on teste si findAll retourne bien deux objets (ceux du .yml)

Dans le précédent test

- Ouvrir sur votre IDE l'annotation DataSet (appui sur Ctrl+l'annotation) pour voir les options possibles :
 - value : dataset à charger
 - strategy : on a laissé ici la valeur par défaut, qui est CLEAN_INSERT, qui veut dire effacer et insérer dans la base les données du dataset à chaque exécution du test case
Autres valeurs possibles : INSERT (ajout de façon incrémentale), UPDATE (mettre à jour les données de la base avec le nouveau dataset), ...
 - On peut également paramétriser cette annotation pour mettre en place des scripts SQL qui s'exécutent avant, après le test, ...

Enfin, on doit ajouter des tests cases pour les autres méthodes du repository utilisées dans l'application (cas de succès et échec)

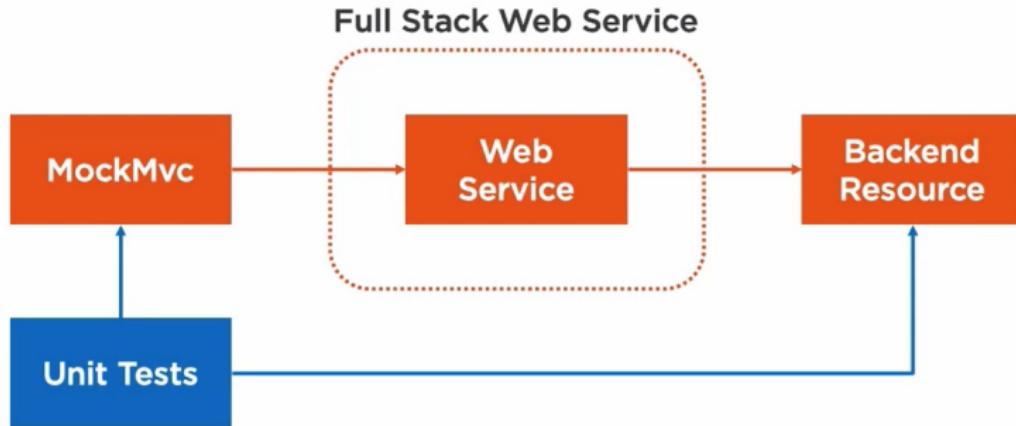
Plan du cours

1. Automatisation des tests en Java avec JUnit
2. Introduction aux tests avec Spring
3. Tests unitaires avec un backend SQL
4. Tests d'intégration

Idée générale

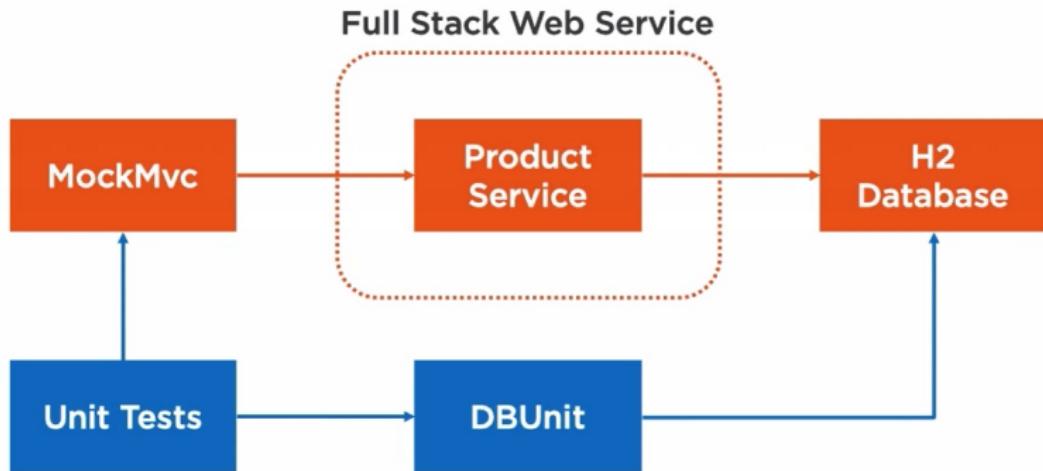
- Tester si tout ce qui a été développé et testé séparément fonctionne bien une fois connecté ensemble
- Tester si une fois les contrôleurs, reliés aux services, et les services aux repositories, le tout fonctionne correctement
- Pourquoi ?
 - Vérifier que la configuration (connexion des composants) est correcte
 - S'assurer que la fonctionnalité end-to-end (sans le front) marche correctement

Test d'intégration



- Nous utiliserons MockMVC pour éviter le démarrage d'un serveur Web (et simuler des requêtes/réponses HTTP)
- Nous utiliserons aussi une BdD in-memory (H2) pour simuler la source de données

Test d'intégration avec MockMvc et H2



- Nous utiliserons aussi DBUnit pour tester les repositories

Classe de test d'intégration

```
@ExtendWith({ DBUnitExtension.class , SpringExtension.class })
@SpringBootTest
@Profile("test")
@AutoConfigureMockMvc
public class UserServiceIntegrationTest {
    @Autowired
    MockMvc mockMvc;
    @Autowired
    private DataSource dataSource;
    public ConnectionHolder getConnectionHolder() {
        // Return a function that retrieves a connection
        // from our data source
        return () -> dataSource.getConnection();
    }
    // ...
}
```

- Noter l'utilisation de l'attribut auto-injecté MockMvc

Méthode de test d'intégration - Get - Found

```
@Test
@DisplayName("GET /user/1 - Found")
@DataSet("users.yml")
void test GetUserByIdFound() throws Exception {
    // Execute the GET request
    mockMvc.perform(get("/user/{id}", 1))
        // Validate the status code and content type
        .andExpect(status().isOk())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON))
    // Validate the returned fields
    .andExpect(jsonPath("$.id", is(1)))
    .andExpect(jsonPath("$.prenom", is("Yan")))
    .andExpect(jsonPath("$.nom", is("Brooks")))
    .andExpect(jsonPath("$.age", is(47)));
}
```

Méthode de test d'intégration - imports

- Prévoir les imports statiques suivants :

```
import static org.springframework.test.web.servlet.  
    request.MockMvcRequestBuilders.*;  
import static org.springframework.test.web.servlet.  
    result.MockMvcResultMatchers.*;  
import static org.springframework.test.web.servlet.  
    result.MockMvcResultHandlers.*;  
import static org.mockito.Mockito.*;  
import static org.hamcrest.Matchers.*;
```

Méthode de test d'intégration - Get - Not Found

```
@Test
@DisplayName( "GET /user/99 - Not Found")
@DataSet( "users.yml")
void test GetUserByIdNotFound() throws Exception {
    // Execute the GET request
    mockMvc.perform(get("/user/{id}",99))
    // Validate the status code and content type
    .andExpect(status().isNotFound());
}
```

Méthode de test d'intégration - Post - Success

```
@Test
@DisplayName( "POST /user - Success")
@DataSet( "users.yml")
void testCreateUser() throws Exception {
    // Setup User to create
    String prenom="Lisa", nom="Holmes"; int age=26;
    User user = new User(prenom,nom,age);
    // Execute the POST request
    mockMvc.perform(post("/user")
        .contentType(MediaType.APPLICATION_JSON)
        .content(asJsonString(user)))
    // Validate the status code and content type
    .andExpect(status().isCreated()).andExpect(
        content().contentType(MediaType.APPLICATION_JSON))
    // Validate the returned fields
    .andExpect(jsonPath("$.id",is(3)))
    .andExpect(jsonPath("$.prenom",is(prenom)))
    .andExpect(jsonPath("$.nom",is(nom)))
    .andExpect(jsonPath("$.age",is(age)));
}
```

Méthode de test d'intégration - sérialisation

```
private String asJsonString(Object obj) {  
    try {  
        return new ObjectMapper().writeValueAsString(obj);  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }  
}
```

Prévoir enfin des méthodes de test pour les autres opérations du service :

- mise à jour d'un user (put), avec succès, conflit, not found
- suppression d'un user (delete), avec succès, not found

Conclusion

- Tests automatiques très importants dans le développement d'une application Web
- Aident à vérifier le comportement correct de l'application
- Font qu'on peut faire des changements sans avoir peur de casser l'application
- TDD (*Test-Driven Development*) : écrire d'abord les tests, qui échouent tous au départ, et ensuite développer l'application pour faire passer les tests au vert l'un après l'autre (paradigme un peu lent au départ, mais développement sûr)
- Spring offre plein de facilités pour écrire des tests, en intégrant de façon transparente/facile plusieurs autres frameworks JUnit, Mockito, Hamcrest, ...

Conclusion -suite- et références biblio

- Pour aller plus loin ("out of scope" de ce cours), faire les tests end-to-end en intégrant le front, avec des frameworks comme Cypress (<https://www.cypress.io/>) : tester les interfaces utilisateurs, les navigations, les vérifier et ensuite rejouer cela à chaque changement dans l'application

Références biblio :

- Site Web de Spring Testing :
<https://spring.io/guides/gs/testing-web/>
- Tutoriels sur Pluralsight et Baeldung

