

## Spring API - A3 - Erreurs et Tests

Dans cette séance, nous découvrirons comment gérer correctement les erreurs dans Spring Boot, et comment tester efficacement toutes les couches de l'application.

### 01 . Introduction

Reprenez l'API développée au cours des sujets précédents : elle servira pour ce sujet et permettra d'avoir une API simple, déjà écrite.


N'oubliez pas d'avoir exécuté MongoDB dans un terminal

### 02 . Gestion des erreurs

Dans chacune des parties de l'application, une erreur (exception) peut arriver. Spring Boot possède une gestion par défaut des exceptions, mais il serait pertinent de contrôler totalement ce qui est renvoyé au client en cas d'erreur.

Nous allons donc définir, pour notre application, un ensemble de classes d'exception et un gestionnaire global d'exceptions.

#### 2.1 Création d'une classe d'erreur

Créez un sous-dossier  `errors` dans l'application (au même niveau que les dossiers `controllers`, `services`, etc.).

Placez dans ce dossier une classe abstraite `APIException` qui hérite de `RuntimeException`. Cette classe contiendra une propriété en lecture seule `Status`, de type `HttpStatus` (classe présente dans `org.springframework.http`). Le constructeur prendra en paramètre un `HttpStatus` et une chaîne de caractères, initialisera les attributs ainsi que les attributs hérités.

Toutes les exceptions que notre application lèvera devront hériter de cette classe.

Nous avons, par exemple, dans `PersonCustomRepositoryImpl`, une opération `updateAgeAllPersons` qui lève une exception `RuntimeException` : nous allons remplacer ceci par une exception personnalisée.

Créez, dans `errors`, une classe abstraite `ObjectNotFoundException` qui hérite de `APIException`. Son constructeur prendra une chaîne message en paramètre et doit initialiser son ancêtre avec le statut `HttpStatus.NOT_FOUND` et le message reçu.

Créez une classe `PersonsNotFoundException` qui hérite de `ObjectNotFoundException`.

Son constructeur ne prendra aucun paramètre, et doit initialiser son ancêtre avec le message « No persons in database ».

Une fois la classe créée, modifiez `updateAgeAllPersons` pour lever cette dernière. En passant, remplacez `if(result==null)` par `if(result.getMatchedCount()==0)`.

Créez ensuite une classe `PersonNotFoundException` qui hérite de `ObjectNotFoundException`. Son constructeur prendra en paramètre une chaîne de caractères `id`, et doit initialiser son ancêtre avec le message « the id <id> is not found ».

Créez enfin une classe `ApiFileNotFoundException` qui hérite de `ObjectNotFoundException`. Son constructeur prendra en paramètre une chaîne de caractères `fileName` et doit initialiser son ancêtre avec le message « the file <FileName> is not found or not readable. ».

Modifiez l'opération `getMethodName` du contrôleur et levez l'exception `PersonNotFoundException` au lieu d'une autre.


Retirez enfin la signature `throws` de `getMethodName`. L'opération ne compile plus car le `readFromPath` peut lever des exceptions...

Modifiez alors cette dernière opération (et sa documentation) :

```
/**
 * @param fileName the name of the file, into the base folder
 * @return bytes of the file
 * @throws ApiFileNotFoundException if file not found or not readable
 */

public byte[] readFromPath(String fileName) {
    byte[] bytes=null;
    try{
        try(FileInputStream stream = new
FileInputStream(Paths.get(this.basePath,fileName).toString()))
        {
            bytes = stream.readAllBytes();
        }
    }
    catch(IOException ex){
        throw new ApiFileNotFoundException(fileName);
    }
    return bytes;
}
```

## 2.2 Création du gestionnaire d'exception

Ce gestionnaire est une classe que l'on situe, en général, dans le dossier  `controllers`.

Créez donc dans ce dossier une classe `ApiExceptionHandler` qui hérite de `ResponseEntityExceptionHandler`. Annotez cette classe avec `@ControllerAdvice` pour que Spring Boot l'initialise automatiquement.

Placez l'opération suivante dans cette classe :

```
@ExceptionHandler(value = { ApiException.class })
protected ResponseEntity<Object> handleApiException(ApiException ex,
WebRequest request) {
    return handleExceptionInternal(ex, ex.getMessage(), new
HttpHeaders(), ex.getStatus(), request);
}
```

Cette opération va déclarer un gestionnaire d'exception (ExceptionHandler) pour la classe APIException (donc pour toutes nos autres classes qui en héritent...) et utiliser le comportement par défaut (envoyer une réponse au client) en précisant le message et le statut http.

Il peut arriver que d'autres exceptions soient levées : nous pouvons rajouter un autre gestionnaire pour RuntimeException (qui est une classe ancêtre) et utiliser le même comportement mais avec le statut http 500 (HttpStatus.INTERNAL\_SERVER\_ERROR).

Vous pouvez, si vous avez le temps, modifier le service pour ne pas renvoyer des Optional mais lever une exception en cas d'erreur.

## 2.3 Tests

Lancez votre serveur et essayez, avec un client comme Thunder Client, d'appeler les API. Vérifiez que les messages d'erreur soient bien ceux attendus.

Vous avez à présent la technique pour personnaliser les réponses d'erreur de l'API en utilisant proprement le système d'exceptions.


## 03 . Introduction pour tester les différentes couches

Les **tests unitaires** sont bien entendus très importants, et ils le sont encore plus quand l'application est un serveur !

Il est important de faire un test complet (dit « end-to-end ») : lancer le serveur, faire une requête depuis un client, etc., mais si le test échoue il est très complexe de savoir quelle couche ne fonctionne pas. Il convient donc de bien tester couche par couche, indépendamment les unes des autres !

Nous allons utiliser le framework JUnit (Jupiter), classique en Java, et très complet.

### 3.1 Tester les modèles

Ce sont les tests les plus faciles... mais les moins utiles car les modèles étant très simples à coder, rares sont les bugs qui s'y cachent. Mais ce n'est pas une raison pour ne pas les faire ! Normalement ils devraient être déjà faits, mais si ce n'est pas le cas, commencez par créer dans le dossier  test une classe `TestPerson`. Une classe de test doit toujours être **publique**.

Rajoutez un cas de test, c'est-à-dire une simple opération annotée par `@Test`. Par exemple, pour simplement tester la création d'une personne :

```
@Test
public void testPerson(){
    Person p = new Person("toto",15, null);
    assertEquals("toto", p.getName());
    assertEquals(15, p.getAge());
    assertNull(p.getPhoto());
}
```

Exécutez le test pour vérifier. (Vous devriez avoir un bouton play qui apparait à coté de votre fonction et de votre classe)

### 3.2 Tester les services

Le test de la couche « service » se complique : en effet cette couche nécessite la couche « repository » pour fonctionner. Or nous n'avons pas envie de tester l'empilement des deux couches, ni de devoir modifier notre base de données pour faire des tests ! Il nous faut donc un dépôt **factice** (mock repository) !

Nous n'avons pas besoin de créer nous-même cette classe : le framework contient une classe Mockito qui permet de créer automatiquement des objets avec la même interface qu'un autre !

Nous allons utiliser ces fonctionnalités pour créer les tests de la couche service.

Créez dans le dossier `test` une classe publique `TestPersonService`.

Créez une opération de test `testAddPerson` qui servira à tester l'ajout de personne dans le dépôt.

Il faut commencer par créer le dépôt factice, grâce à la classe Mockito

```
PersonRepository repository = Mockito.mock(PersonRepository.class);
```

Ensuite, il faut créer le service, en lui transmettant ce dépôt :

```
PersonService service = new PersonService(repository);
```

Nous allons ensuite effectuer l'ajout d'une personne, via le service :

```
PersonData result = service.add("toto",15,null);
```

Nous pouvons bien entendu tester que le retour correspond à ce que l'on souhaitait :

```
assertEquals("toto", result.getName());
assertEquals(15, result.getAge());
assertNull(result.getPhoto());
ObjectId id = new ObjectId(result.getId());
assertEquals(id.toHexString(), result.getId());
```

Mais cela ne signifie pas que l'ajout dans le dépôt est correct depuis le service, uniquement que la personne a bien été décorée avec un id correct (ce qui est déjà bien).

Le dépôt étant factice, nous ne pouvons pas aller voir dedans... mais heureusement le framework permet de savoir si les bons appels ont été faits !

Nous allons pour cela utiliser la classe `ArgumentCaptor` qui va « capturer » ce qui s'est passé dans le dépôt factice.

Pour vérifier que l'opération insert du dépôt a bien été appelée avec la bonne personne, il suffit de faire le code suivant :

```
ArgumentCaptor<PersonData> argumentCaptor =
ArgumentCaptor.forClass(PersonData.class);
Mockito.verify(repository,
Mockito.times(1)).insert(argumentCaptor.capture());
PersonData resultPersonData = argumentCaptor.getValue();
assertSame(resultPersonData, result);
```

Lancez les tests : ils devraient normalement passer.

Notre classe de test n'est pas encore terminée : il reste des cas de tests à faire... et certains vont être un peu plus complexes.

Si nous souhaitons tester, par exemple, le `findById` du service, il va falloir un dépôt factice qui « contient » des personnes, pour que l'on puisse vérifier...

En fait notre dépôt factice ne contient rien du tout, mais il peut garder la trace des appels qui ont été faits, et l'on peut définir ce qu'il doit renvoyer dans certains cas. C'est un peu complexe, mais très puissant pour faire des tests complets.

Créez un cas de test `testFindById`, et commencez par créer le dépôt factice et le service, comme tout à l'heure.

Nous allons ensuite créer deux **personnes factices** qui sont sensées être présentes dans le dépôt :

```
PersonData personData1 = Mockito.mock(PersonData.class);
PersonData personData2 = Mockito.mock(PersonData.class);
```

Nous allons ensuite indiquer quoi renvoyer quand l'opération `getId` sera appelée :

```
ObjectId id1 = ObjectId.get();
ObjectId id2 = ObjectId.get();
Mockito.when(personData1.getId()).thenReturn(id1.toHexString());
Mockito.when(personData2.getId()).thenReturn(id2.toHexString());
```

Tout se passe comme si les objets `personData1` et `personData2` contenaient les identifiants `id1` et `id2`, mais ce n'est pas le cas : tout est factice !

Il faut ensuite indiquer ce que doit **renvoyer** l'opération findById du dépôt factice...

C'est-à-dire, personData1 si l'opération est appelée avec id1, personData2 si l'opération est appelée avec id2... et renvoyer vide dans les autres cas. Ce qui se fait avec le code suivant :

```
Mockito.when(repository.findById(any())).thenAnswer(invocation->{
    Optional<PersonData> result = Optional.empty();
    ObjectId id = (ObjectId)invocation.getArguments()[0];
    if(id.equals(id1))
        result = Optional.of(personData1);
    else if(id.equals(id2))
        result = Optional.of(personData2);
    return result;
});
```

Maintenant que nous avons bien expliqué au dépôt factice comment réagir, nous allons effectuer un premier test, et vérifier qu'avec un id connu (comme id1) nous avons la bonne personne renvoyée (personData1, donc) :

```
PersonData test1 = service.findById(id1.toHexString());
assertEquals(personData1, test1);
```

Le même test pourrait se faire avec id2 et personData2, mais il est pertinent de faire un test avec un id **inconnu**, pour voir ce qu'il se passe (dans ce cas, le service doit **lever** une **exception**) :

```
ObjectId id3 = new ObjectId();
assertThrows(PersonNotFoundException.class, ()->{
    service.findById(id3.toHexString());
});
```

Lancez les tests : tout devrait bien se passer !

Créons un dernier test, pour vérifier si tout est compris : tester le findByName du service.

Créez ce cas de test (assez proche du précédent) !



Le temps manquera pour finir, mais bien entendu, toutes les opérations du service sont testables et doivent être testées.

Le contrôleur peut être également testé de la même manière.

Pour tester le dépôt, il faut le relier à une base de données test, c'est un peu plus complexe... mais faisable...