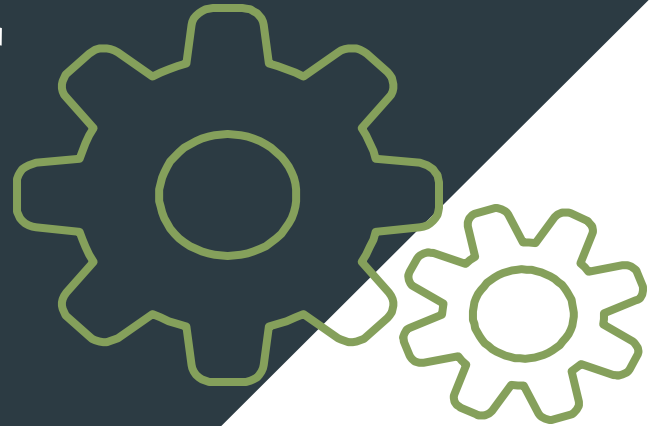


# Structurer et tester une application Spring



# Plan

- Architecture n-tiers
- Test d'intégration et test unitaire
- Tester une application

# Structure d'une application

**Architecture n-tiers**



## Stéréotypes Spring

`@Repository`, `@Component`, `@Controller`, `@RestController` sont des annotations qui permettent deux choses :

- Elles précisent à Spring qu'il s'agit de Beans à intégrer à son contexte applicatif
- Elles donnent du **SENS** à une classe en précisant sa fonction globale au sein de l'application.

Cette double fonctionnalité d'une annotation est appelé un **Stéréotype**



# Objet

## Diviser pour mieux régner

Le sens apporté à une classe permet de lui associer une fonction spécifique plutôt que de créer des Objets “fourre tout” responsables de toute l'application.

C'est une extension du “**Principe de responsabilité unique**” introduit par la POO. “Une classe ne doit changer que pour une seule raison”.



## Modèle n-tiers

### Plusieurs strates pour autant d'usages

L'une des conceptualisations de cette séparation des tâches au sein des classes d'une application a été théorisée par la création du modèle n-tiers.

Dans cette architecture, l'application est divisée en au moins trois couches:

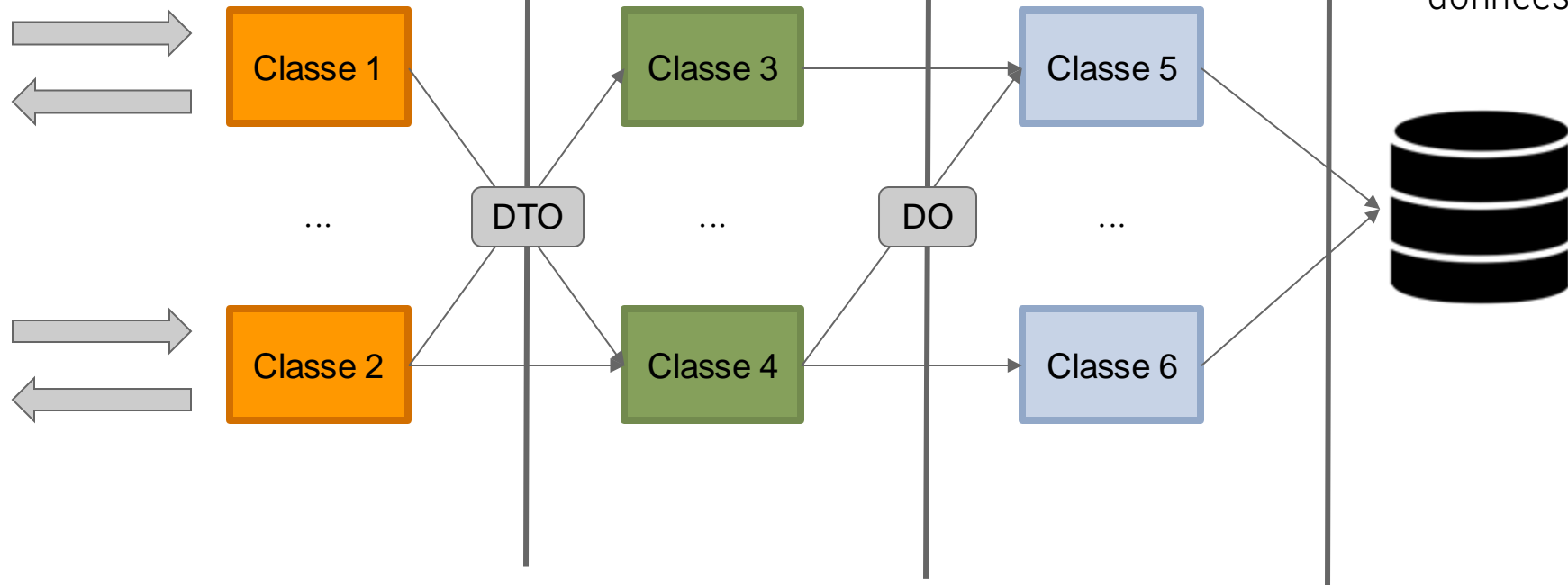
- La présentation
- La logique
- La persistance des données

Couche présentation

Couche de logique

Couche de persistance

Système de stockage des données





# Modèle n-tiers

## Couche présentation

Cette couche est responsable de l'interaction de l'application avec l'utilisateur ou le client. Elle peut prendre de nombreuses formes:

- Une page Web
- Une API Rest
- Une application android
- ...





# Modèle n-tiers

## Couche logique

Cette couche est porteuse de l'intelligence de l'application, de son domaine métier. On y trouvera toutes les règles métiers

C'est elle qui fournira les données calculées à la couche de présentation sous la forme de “**Services**” à appeler depuis les classes de la couche de présentation :

- Un service de récupération des élèves d'une classe
- Un service de calcul d'une fiche de paie
- ...



# Modèle n-tiers

## Couche persistance des données

C'est la **seule** couche interagissant avec le système de stockage des données. On parle de **DAO** pour Data Acces Object.

Elle n'interprète en rien les données extraites et se contente de les propager à la couche Logique qui saura en faire l'usage demandé par la couche de présentation.

Là encore, plusieurs formes sont possibles:

- Lecture de fichiers
- Interaction avec une base de données
- Appel vers un Web Service externe



## Modèle n-tiers

### DO vs DTO : des POJOs

- **DO** : Data Object. Ne fait que porter les données issues du système de stockage
- **DTO** : Data Transfer Object. Sert à encapsuler plusieurs données à transférer ensemble. Il peut être composés de plusieurs BO, voire de nouveaux objets, ou d'autres DTO



## Modèle n-tiers

### Programmation par interface

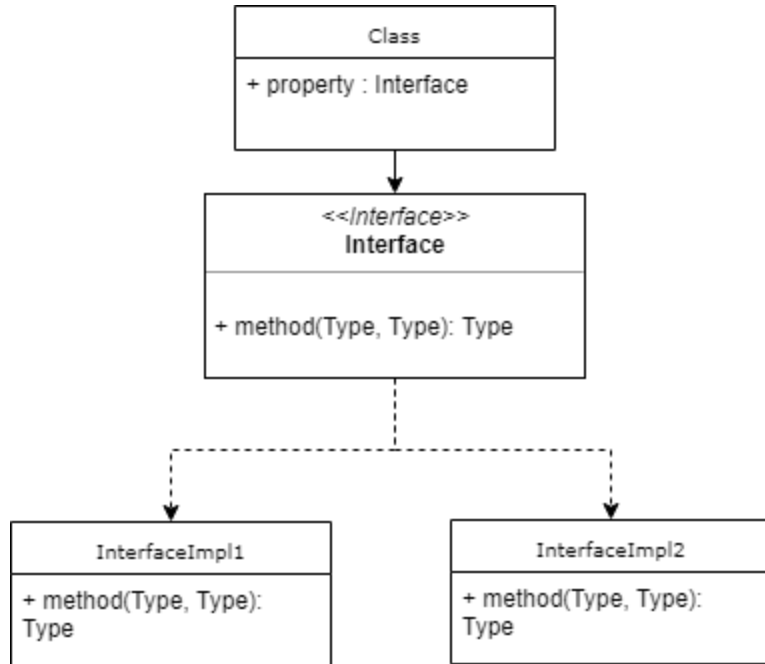
Cette philosophie se rapproche du “Contrat de Service”: seul le résultat compte, pas l’implémentation.

C’est un phénomène grandement facilité par Spring et la programmation par interface Java.


Le passage d’une implémentation à l’autre est transparent tant que la couche supérieure utilise toujours l’interface et non pas l’implémentation.



## Modèle n-tiers



Le passage de InterfaceImpl2 à InterfaceImpl1 doit être totalement transparent pour Class



**Une couche ne peut  
appeler que la  
couche inférieure!**



## Lien avec Spring

### Appeler une couche du même niveau

Il est dans certains cas possible d'appeler un élément de même niveau que celui que l'on souhaite modifier.

Il serait par exemple possible d'appeler un Service depuis un autre Service.

**MAIS**

Cela pourrait créer des dépendances cycliques!



## Lien avec Spring

### Stéréotypes

Les Stéréotypes Spring servent à identifier à quelle couche appartient une classe . Par défaut, ils sont tous des singletons.

Ces annotations doivent être posées sur les implémentations.

Présentation : `@Controller`, `@RestController`

Persistence : `@Repository`

Logique : `@Service`





## Lien avec Spring

### Component

`@Component` est un cas particulier.

Il permet à Spring d'ajouter cette classe en tant que Bean, sans préciser de sens, et donc de place dans le modèle n-tiers.

Il est utilisés pour certains composants très spécifiques qui touchent plus souvent à la mécanique de Spring qu'aux applications en elle même (Interceptor, etc)



## Modèle n-tiers

Il n'existe que 3 types de couche.

Une seule couche de présentation est présente à la fois.

Une seule couche de persistance peut être présente à la fois.

Si plusieurs sources de données différentes interviennent, c'est un service qui doit les agréger



**Un DAO ne gère qu'une seule source de données**



## Modèle n-tiers

### Pourquoi parle-t-on de modèle n-tiers?

Il peut exister une infinité de couches de logique.

Elles peuvent être structurées entre elles pour créer des “Super Services”.

On parle aussi parfois d’**Aggregator** ou d’**Orchestrator**.

Exemple: Un service de paie peut avoir besoin des règles de gestion d’un service de calcul, mais aussi de la liste des employés fournis par un service de gestion du personnel.

Couche présentation

Classe 1

...

Classe 2

DTO

Couche de  
logique 1

Classe 3

...

Classe 4

DTO

...

Couche de  
logique n

Classe n

DTO

Classe n'

DO

Couche de  
persistance

Classe 1

...

Classe 1

# Tester une application

**Test d'intégration et test unitaire**



# Un test unitaire

## Définition

Procédé permettant de s'assurer du bon fonctionnement d'une petite portion de code appelé "unité".

Une unité est la plus petite portion de code qui puisse être isolé en conservant du sens, une entrée et une sortie.

En Java, c'est une méthode permettant de s'assurer du bon fonctionnement d'une autre méthode.



# Un test unitaire

## Définition

Un test unitaire doit être :

- **Indépendant** : il ne doit dépendre d'aucun autre test pour fonctionner.  
Seuls ses propres pré requis doivent être suffisants
- **Déterministe** : Son rejeu dans les mêmes conditions doit toujours parvenir au même résultat



## Un test d'intégration

### Définition

Procédé permettant de s'assurer du bon fonctionnement de **plusieurs unités** de code entre elles.

Ils prennent place dans un plus haut niveau d'abstraction et s'assurent du respect du fonctionnement global de plusieurs couches ensemble.

Ils peuvent prendre de très nombreuses formes en Java et dans le contexte d'un projet Spring.

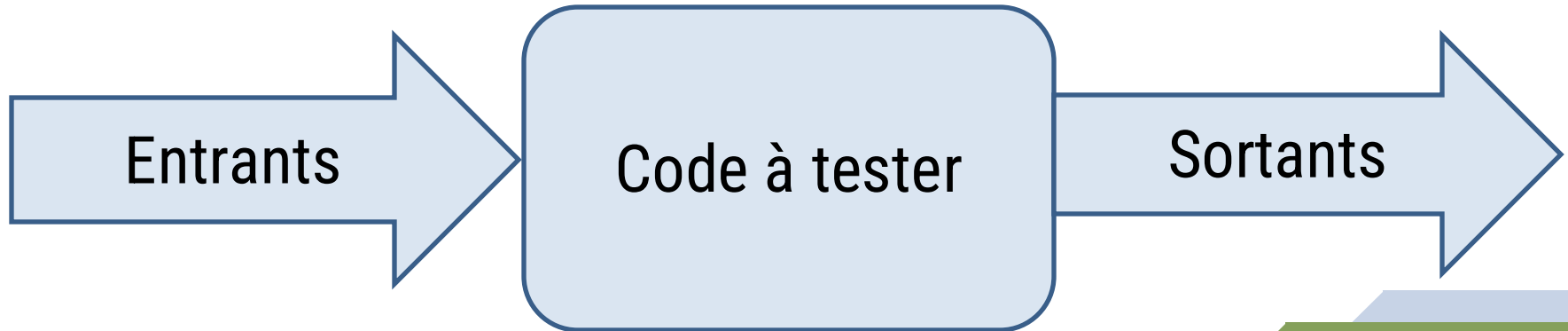




## Cérémoniel

Déroulé d'un test :

1. Fournir des entrants
2. Dérouler le code à tester
3. Vérifier les éléments en sortie du code à tester



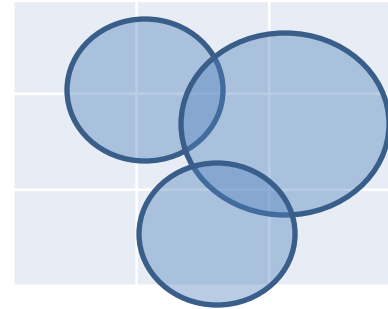


# TI vs TU

## Tests unitaires

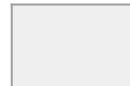
x	x	x
x	x	x
x	x	x

## Tests d'intégration



Test d'intégration

x Test unitaire



Règle de gestion, micro feature, ...



## Bouchonnage

Les tests unitaire et d'intégration n'éprouvent pas la totalité de l'application

Par exemple, si l'on teste le bon fonctionnement d'un service de la couche logique seul, il faut émuler le comportement de la couche persistance qui fournit le données.

On parle alors de **Bouchonnage** ou de **Mock**



## Utilité

### Outil de non régression ultime

Ils offrent un filet de sécurité : Après une modification dans du code, il est toujours possible de savoir s'il est fonctionnel, ou pas.

Il sert à documenter le fonctionnement de l'application : en absence de spécifications, on peut presque parler de Code As Document

# Tester une application

**Tester avec Spring Boot**



## Structure

### Aboresence

Le code de Production se trouve dans src/main/java

Les tests se trouvent dans src/test/java

Une classe de test doit exister par classe de Production, hors POJOs.

La classe de test est à placer dans le même package que la classe qu'elle teste

Convention de nommage : <NomDeMaClasse>Test.java



### Spring-boot-starter-test

Spring Boot fournit une dépendance contenant tous les éléments nécessaires à la rédaction des tests unitaires et d'intégration

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-test</artifactId>  
  <scope>test</scope>  
</dependency>
```



## Librairie de test

### Fonctionnement

Une méthode de test est à annoter avec l'annotation method-level `@Test`

Elle permet au moteur de Junit de repérer les méthodes de tests dans une classe de Test.

Les méthodes de test doivent être publiques, et ne renvoient rien

`@Test`

```
public void myTestMethod() {  
    //test things  
}
```





### Assertions

La vérification des données sortantes est composée d'une ou plusieurs **"assertions"**. C'est un ensemble de méthodes fournies par la librairie Junit.

Ces méthodes lèvent une exception lorsque la condition qu'elles doivent vérifier n'est pas respectée. Cette exception stoppe l'exécution de la méthode de test en cours d'exécution, considérant le test comme en échec.

Elles se trouvent dans le package **org.junit.assert.\***



## Librairie de test

<code>assertEquals(Object o1, Object o2)</code>	<b>Vérifie que o1 et o2 sont égaux. Exécute la méthode <code>.equals()</code> de o1.</b>
<code>assertFalse(boolean b)</code>	<b>Vérifie que b est faux</b>
<code>assertNull(Object o1)</code>	<b>Vérifie que o1 est null</b>
<code>assertArrayEquals(Object[] o1, Object[] o2)</code>	<b>Vérifie que les tableaux o1 et o2 sont égaux.</b>



## Librairie de test

### Initialisation des données

L'annotation method-level `@BeforeEach` permet à une méthode d'être jouée avant chaque test.

En général une seule méthode est annotée avec par classe de test.

Elle doit être publique et ne renvoie rien

```
@BeforeEach
```

```
public void init() {
```

```
    //init things
```

```
}
```



## Librairie de test

### Nettoyage des données

L'annotation method-level `@AfterEach` permet à une méthode d'être jouée après chaque test.

En général une seule méthode est annotée avec par classe de test.

Elle doit être publique et ne renvoie rien

`@AfterEach`

```
public void tearDown() {  
    //init things  
}
```



## Bouchonnage



### Entre couches

Le bouchonnage des données, le plus souvent provenant d'une couche inférieure, est assuré par **Mockito**.

Cas d'utilisation classique :

“Je veux tester mon service

Je bouchonne le retour de mon DAO pour être certain des données et garantir le déterminisme et l'indépendance de mon test.”



## Bouchonnage



### Fonctionnement

Mockito s'appuie le plus souvent sur un système d'annotations et de méthodes.

Pour qu'elles fonctionnent, la classe de test doit être annotée avec cette annotation class-level:

```
@ExtendWith (MockitoExtension.class)
```

Elle précise à Junit que la classe de test doit être exécuté avec un démarrage spécifique.



# Bouchonnage

## Annotations

`@Mock` permet d'identifier un objet comme bouchonné. Mockito va en créer sa propre version sur laquelle il sera possible de donner des comportements spécifiques.

Le plus souvent, il s'agit des objets injectés par Spring dans la classe que l'on souhaite tester

`@Mock`

```
private MonService objectToMock;
```



# Bouchonnage

## Annotations

`@InjectMocks` permet d'identifier un objet comme celui dans lequel les éléments bouchonnés vont prendre place. Il s'agit le plus généralement de la classe à tester.

`@InjectMocks`

```
private MaClasse classToTest;
```





# Bouchonnage

## Syntaxe

L'émulation d'un comportement spécifique sur une méthode d'un objet bouchonné par l'utilisation de méthode de la librairie.

Pour retourner une valeur spécifique depuis un objet bouchonné :

```
doReturn (theMockValue) .when (objectToMock) .methodToMock (expectedParameter) ;
```

Avec cette syntaxe, `theMockValue` sera retourné lors de l'appel `objectToMock.methodToMock (exepctedParameter)`



## Bouchonnage

### Syntaxe

Pour ne rien faire lors de l'appel à une méthode ne retournant rien :

```
doNothing().when(objectToMock).methodToMock(expectedParameter);
```

Avec cette syntaxe, rien ne sera exécuté retourné lors de l'appel

```
objectToMock.methodToMock(exepctedParameter)
```



# Bouchonnage

## Syntaxe

Pour lever une exception spécifique depuis un objet bouchonné :

```
doThrow (new  
MyException() ).when (objectToMock) .methodToMock (expectedParameter) ;
```

Avec cette syntaxe, Une instance `MyException` sera levée lors de l'appel `objectToMock.methodToMock (expectedParameter)`



# Bouchonnage

## Usage

En règle général, l'initialisation des bouchons au travers de la méthode `.when()` se fait dans une méthode annotée `@BeforeEach`

Il est possible de surcharger ce comportement par défaut, commun à toute la classe de test, dans chaque méthode de test spécifique.



# Bouchonnage

## Paramètres et matchers

Il n'est parfois pas possible de connaître précisément la valeur d'un paramètre lors de l'appel à une méthode bouchonnée.

La méthode de comparaison s'appuyant sur la méthode `.equals()` du paramètre, il est parfois plus simple d'appliquer un comportement quelque soit la valeur du paramètre.

Mockito propose une solution à cette inconnue via l'API des **Matchers** et notamment avec les méthodes `any*`



# Bouchonnage

## Matchers: syntaxe

```
doNothing().when(objectToMock).methodToMock(anyString());  
doNothing().when(objectToMock).methodToMock(anyInt());  
doNothing().when(objectToMock).methodToMock(anyDouble());  
doNothing().when(objectToMock).methodToMock(any(MyClass.class));  
;
```



## Bouchonnage

### Verification

Mockito propose de vérifier qu'une méthode bouchonnée a bien été appelée, le nombre de fois souhaité, avec les paramètres attendus.

Cela est permis avec la méthode **verify()**

```
verify(objectToMock, times(1)).methodToMock(anyInt()) ;
```

Le mécanisme des matchers fonctionne également pour cette méthode

```
@ExtendWith(MockitoExtension.class)
public class MyClassTest {

    @Mock
    private MonService objectToMock;

    @InjectMocks
    private MyClass myclass;

    @BeforeEach
    public void init() {
        doNothing().when(objectToMock.methodToMock(anyInt()));
    }

    @Test
    public void test() {
        assertEquals("test", myclass.methodToTest());
        verify(objectToMock, times(1)).methodToMock(anyInt());
    }
}
```





## Tests d'intégration

### Philosophie

Une classe de test telle que la précédente ne fait pas intervenir le contexte Spring : Mockito se charge lui même du cycle de vie des objets.

On considère qu'un test faisant intervenir le contexte Spring est un test d'intégration. Il peut en effet faire interagir plusieurs couches de l'application entre elles, voire toute l'application elle même.



## Tests d'intégration

### Annotations

Pour utiliser le contexte Spring sur une classe de test, il faut lui ajouter l'annotation class-level `@ExtendWith(SpringExtension.class)`

Il faut ensuite préciser comment alimenter le contexte la configuration :

- `@ContextConfiguration(classes = {MyConfigurationClass.class})` permet d'ajouter la classe `MyConfigurationClass` à la configuration du contexte
- `@SpringBootTest` permet de reprendre tout le contexte de l'application. Remplace aussi le `@ExtendWith`



## Tests d'intégration

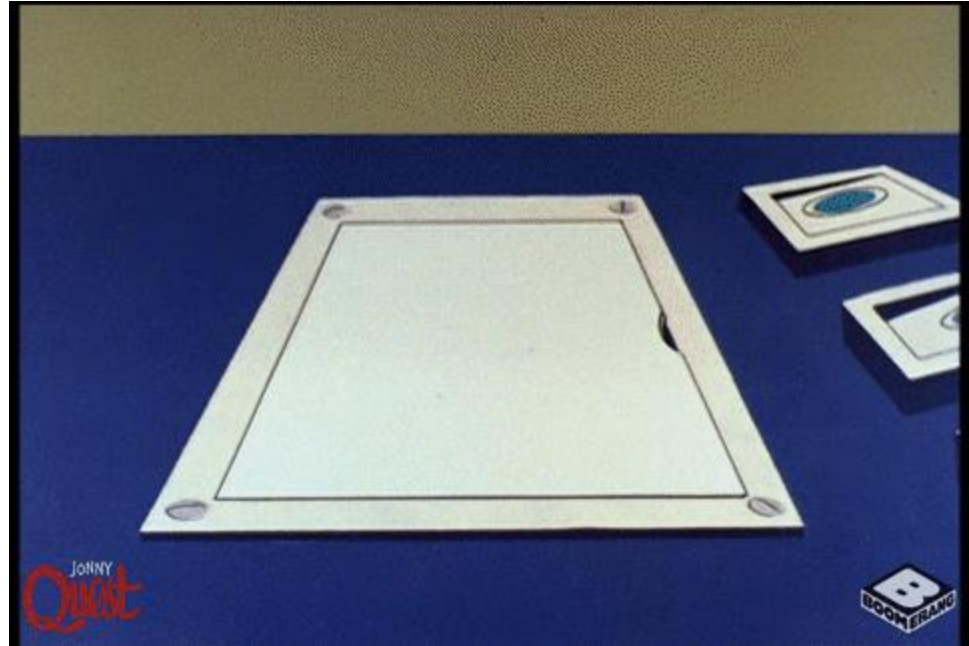
### Injections

Une classe de en contexte Spring peut donc bénéficier de l'injection des dépendances, comme le ferait l'application.

Il est donc possible d'y injecter

- Un service pour vérifier qu'il peut bien récupérer des données de la base et les exploiter (En l'état, seule l'injection via `@Autowired` fonctionne dans les tests)
- Un controller REST pour vérifier que la stack globale fonctionne

**Les tests ne doivent pas  
utiliser les environnements  
de production pour leurs  
données !!**





## Tests d'intégration

### Properties

Il est vivement recommandé d'utiliser un fichier de configuration `application.properties` ou `application.yml` différent pour les tests.

Il doit se trouver dans `src/test/resources`.



## Tests d'intégration

### Surcharge des propriétés

Il est possible de surcharger spécifiquement certaines propriétés depuis un fichier supplémentaire, ou certaines valeurs spécifiques via l'annotation class-level `@TestPropertySource`

```
@TestPropertySource(locations = "/file.properties",  
    properties = "com.example.key=value")
```

L'exemple ci-dessus charge le contenu du fichier `file.properties`, et écrase la valeur de la clé `"com.example.key"` par la valeur `"value"`