

## Auteurs

Franck BANSEPT



Github / CI CD



Spring boot



# SPRING 3



[https://fr.wikipedia.org/wiki/Licence\\_Creative\\_Commons](https://fr.wikipedia.org/wiki/Licence_Creative_Commons)



BY (Attribution) : signature de l'auteur initial et des auteurs ayant modifié le document



SA (Share alike) : partage de l'œuvre, avec obligation de rediffuser selon la même licence ou une licence similaire (version ultérieure ou localisée)



NC (Non Commercial) : interdiction de tirer un profit commercial de l'œuvre sans autorisation de l'auteur initial

Licence Creative Commons : Attribution-NonCommercial-ShareAlike (BY-NC-SA)

N'importe quelle personne morale ou physique est libre d'utiliser, modifier, partager pour un usage non commercial ce document si elle respecte les conditions suivantes :

- Intégrer les noms des différents auteurs ayant participés à sa réalisation
- Ne pas intégrer de contenu offensant ou contraire à la législation en vigueur
- Intégrer cette même licence ou une licence équivalente

Note : Ce document peut être utilisé en tant que support d'une formation rémunérée, mais il ne peut être vendu indépendamment.

# >> Code couleur des diapositives du cours

>> Fil rouge

Contient des manipulations / ajout ou modification de code à effectuer

>> Pour aller plus loin

Contient des cours ou des notions supplémentaires qui ne sont pas essentiel à la manipulation, mais permettant de comprendre les mécanismes poussés

>> Alternative

Ou une méthode alternative à la méthode préconisé / standard

>> Tests

Contient les tests unitaires / d'intégration concernant la fonctionnalité mise en place

>> TP

Contient des exercices de manipulation pour le TP



## >> Sujets abordés dans ce cours

Ce cours **n'a pas pour unique but** la découverte du framework **Spring**

Il a été structuré et complété de façons à étudier **des concepts plus vastes** que l'on peut retrouver dans **d'autres langages / framework**.

Ainsi il permet une approche sur des sujet variés tel que la modélisation d'architecture, l'inversion de contrôle, les conception d'API, les concepts avancés de programmation orienté objet, les tests unitaires, les ORM, le design pattern MVC, les gestionnaires de dépendances, la conception de base de données, la sécurité des applications web, les systèmes d'authentification et d'autorisation, l'architecture des systèmes, le déploiement d'application ...

Voyez ce cours comme l'occasion de mettre en place tous ces concepts en place, plutôt que simplement le voir comme "Comment mettre en place une application Spring"

Merci de votre attention :)



# >> Sujets abordés dans ce cours

## Programmation Orienté Objet

- Héritage, Interface
- Structurer l'application avec un couplage faible
- Design pattern Injection de dépendance, MVC

## Conception

- Représenter l'architecture avec Merise et UML  
*(Diagramme de classe, diagramme de séquence, diagramme de cas d'utilisation)*

## Sécurité

- Créer une gestion de droits
- Connexion par token (JWT)
- Gérer les failles (Injection SQL, CSRF, HTTPS)
- Créer des tests unitaires / fonctionnalité
- Limiter les appels (anti brut force / bot)
- Connexion Auth2
- Log et supervision sur une Stack ELK

## DevOps

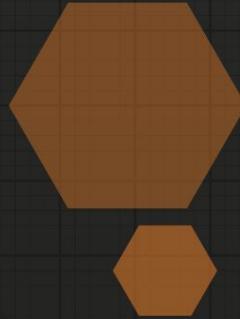
- Déployer sur docker
- Créer une pipeline CI / CD

## Conception

- Représenter l'architecture avec Merise et UML  
*(Diagramme de classe, diagramme de séquence, diagramme de cas d'utilisation)*

## Sécurité

- Créer une gestion de droits
- Connexion par token (JWT)
- Gérer les failles (Injection SQL, CSRF, HTTPS)
- Créer des tests unitaires / fonctionnalité



Qu'est ce qu'un framework ?

## >> Framework (*cadriel*)

Un framework, dont la traduction littérale serait “Cadre de travail” porte plutôt bien son nom.

Comme son nom l'indique, il apporte une **structuration à respecter** pour les développeurs.

Pour un **développeur isolé** il peut sembler **inutile** voir même **plus complexe** à utiliser que du code qui n'en utiliserait aucun.

**L'intérêt apparaît surtout lorsque l'on travail en équipe** (2, 10 voir des centaines de personnes dans le cadre d'un projet open source par exemple)

Si 2 développeurs respectent les mêmes règles imposées par le framework, alors ils pourront plus facilement travailler ensemble.

Une personne qui rejoint une équipe de développeur sera productive plus rapidement car elle aura la connaissance du cadre de travail dès son arrivé.



## >> Framework VS bibliothèque

L'erreur courante que l'on fait sur l'utilité des framework est qu'on les confonds avec les bibliothèques.

Une **bibliothèque** n'est qu'un ensemble de fonctionnalités (regroupé sous forme de classes, modules...)

Les **frameworks** entendent imposer une **structure** (*par exemple MVC que l'on verra plus tard*)

Imposer un certain nombre de **bibliothèques**

Voir même une philosophie ou des pratiques (*Inversion de contrôle, usage d'annotations ...*)

Un framework pourrait très bien ne posséder aucune fonctionnalité, et tout simplement consister à imposer des règles, des conventions, des structures ...

Mais les frameworks sont fournis en général avec un ensemble bibliothèques afin d'harmoniser ces dernières, toujours dans le but d'imposer un cadre de travail commun (et donc des fonctionnalités communes)

Un exemple parfait et la différence entre la bibliothèque React (qui n'impose aucune structure) et des frameworks comme Angular ou Vue (*a noter que la bibliothèque React dispose du framework Next*)



# Démarrer un projet spring

Dans ce chapitre :

- Comprendre l'architecture 3 tiers à mettre en place
- Comprendre la notion d'API
- Installer et configurer l'environnement
- L'arborescence d'un projet Spring Boot

# Présentation

*Chapitre : Démarrer un projet spring*

## >> Pré-requis

### Obligatoire

Afin de pouvoir suivre ce module, il est nécessaire de maîtriser le langage **JAVA**, ainsi que les concepts de la **POO** qui lui sont associés.

*Type primitifs, Classes, héritage, interface, classes anonymes, encapsulation, gestion des exceptions ...*

Avoir l'IDE **IntelliJ** installé (*minimum version community*), une **connexion Internet**, une base de donnée **MySQL** avec un utilisateur ayant les droits maximum (dont vous connaissez le mot de passe).

### De préférence

Maîtriser les règles de passage d'un **MCD Merise** ou d'un **Diagramme de classe UML** vers un schéma de base de données (*Création de clé étrangères, Table de jointure ...*)



# >> Architecture à mettre en place

**Spring** est un **framework JAVA**

Nous allons apprendre à créer des **serveurs** permettant de mettre à disposition une **API REST** (*Application Programming Interface*) (*REpresentational State Transfer*) afin que différents clients puissent s'y connecter (*navigateurs, application mobile, appareil connectés...*).

*Le code respectera une **architecture MVC** (Model, View, Controller).*

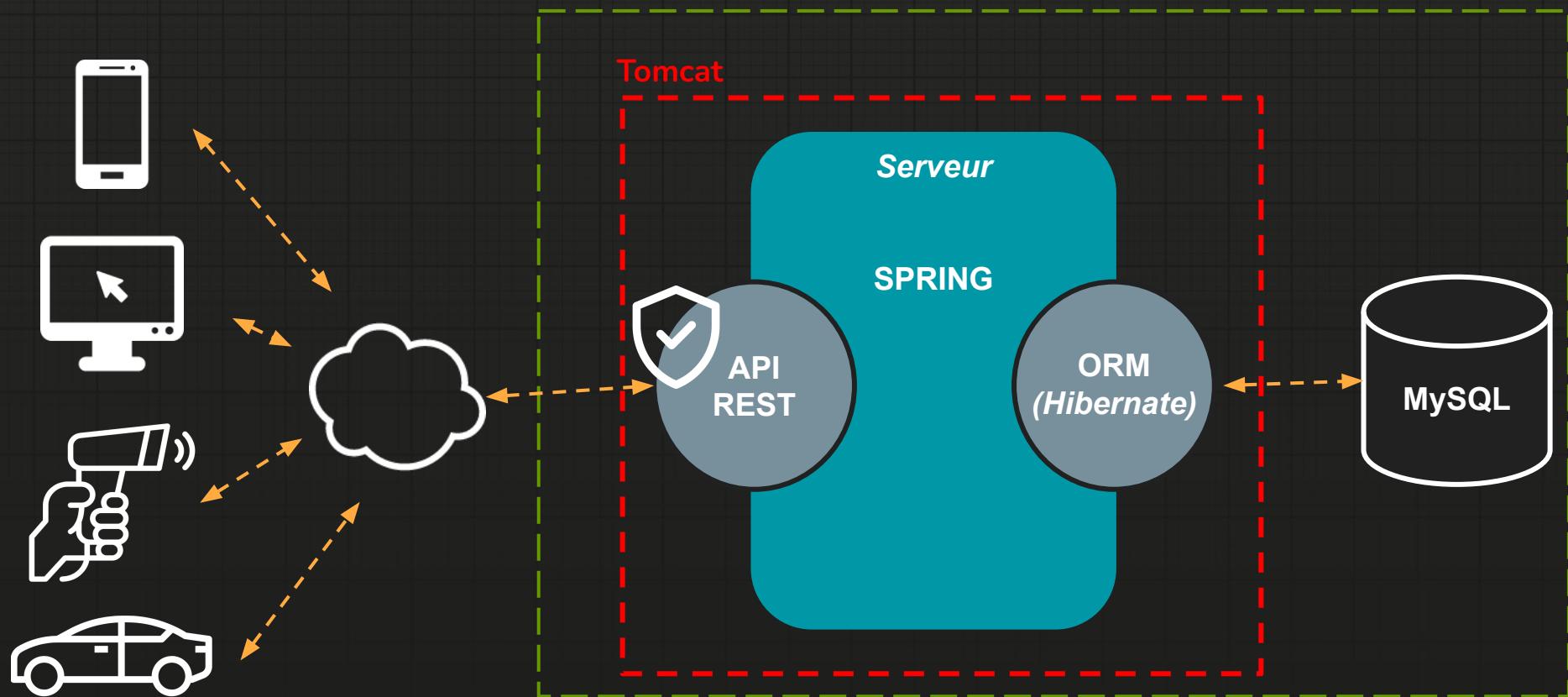
Le serveur pourra via l'**ORM** (*Object Relational Mapping*) **Hibernate**, contacter une base de donnée et effectuer des opérations **CRUD** (*Create Read Update Delete*) sans que nous ayons à effectuer la moindre **requête SQL** (*Structured Query Language*) (*le langage permettant de modifier les données et la structure d'une base de donnée*)

L'**API** (*Application Programming Interface*) sera protégée par un système d'authentification par **JWT** (*JSON Web Token*) compatible multi-serveur, et permettant de rendre disponible certaines ressources que pour certains utilisateurs (*administrateur, simple utilisateur ...*)



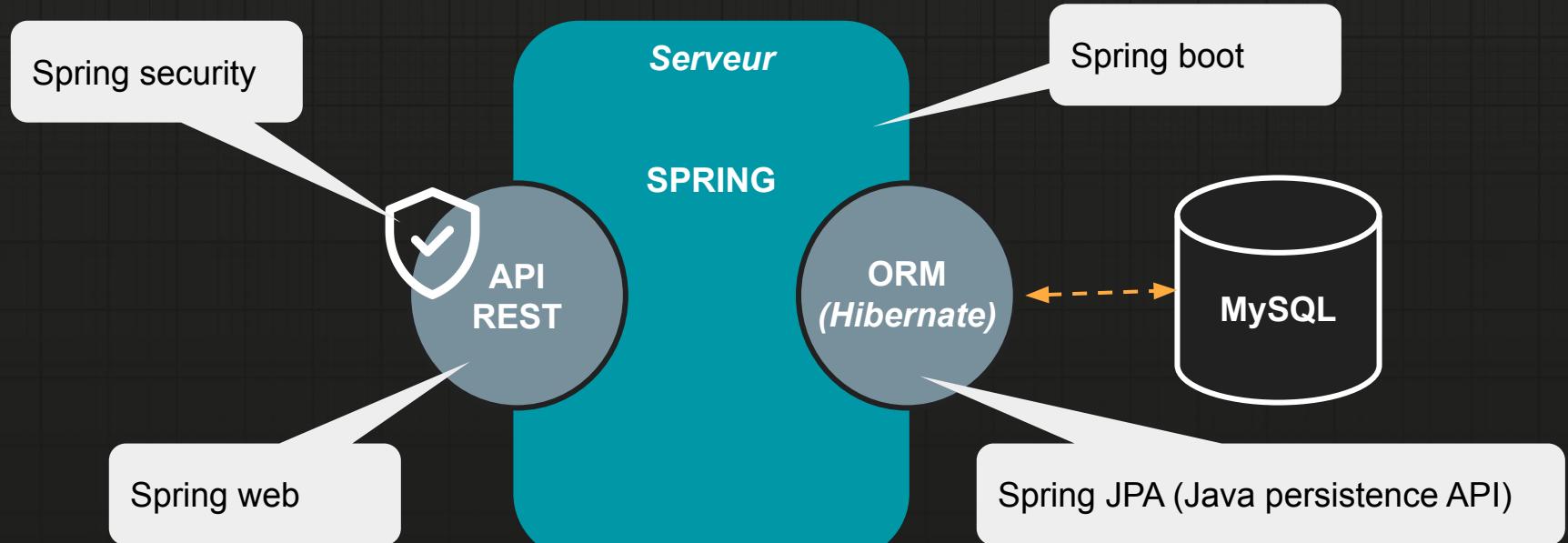
# >> Architecture à mettre en place

Serveur physique / VM



# >> Les différentes dépendances de Spring

Spring se décompose en un certain nombre de **dépendances** que l'on peut ajouter selon nos besoins. Nous utiliserons le **gestionnaire de dépendances Maven** afin de les télécharger.



# Installation

*Chapitre : Démarrer un projet spring*

# Maven™

Maven est un **gestionnaire de dépendances** au même titre que **Gradle**.

Il est comparable à **NPM** pour **javascript** ou à **Composer** pour **PHP**.

Outre la gestion des dépendances, **Maven** permet de gérer le **cycle de vie** d'une application (*build, test unitaire, déploiement ...*) ainsi que les profils des développeurs (*mot de passe dans l'application, URL locales ...*)

Il utilise pour cela un fichier de configuration **pom.xml** (*Project Object Model*) qui va définir les propriétés du projet (*version de java employé, liste des dépendances, profils ...*)





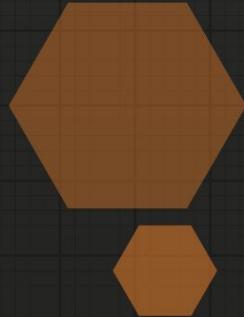
# SPRING BOOT

**Spring boot** permet de réaliser des applications basées sur **SPRING** avec un minimum de configuration.

Il permet de mettre en place une application **Spring** rapidement en respectant une arborescence type.

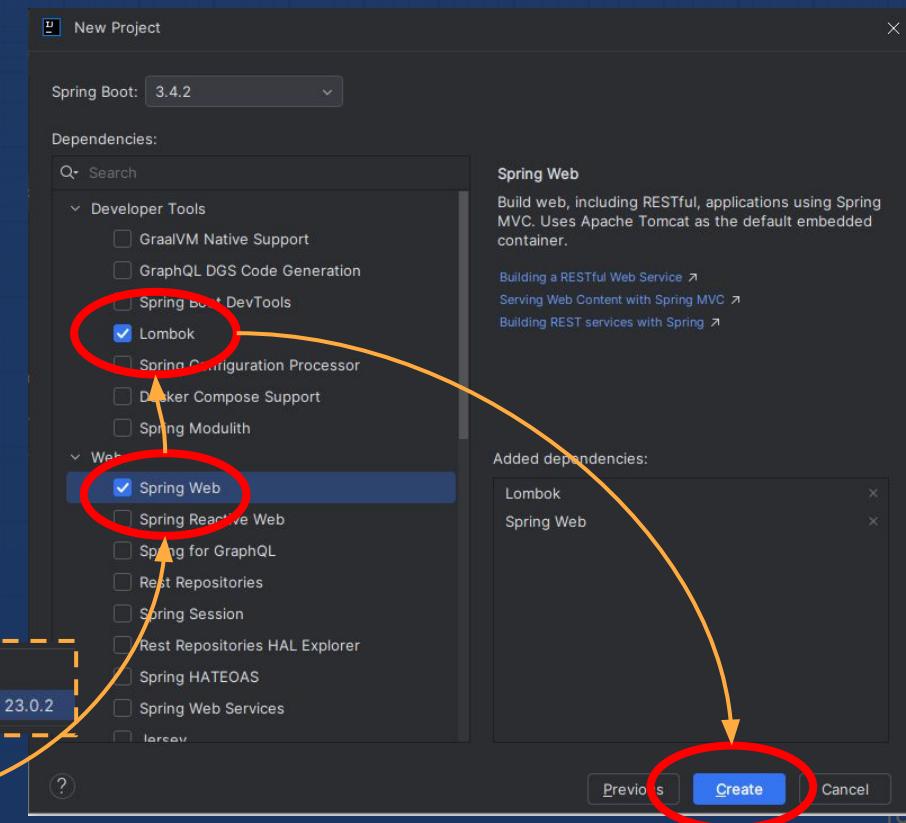
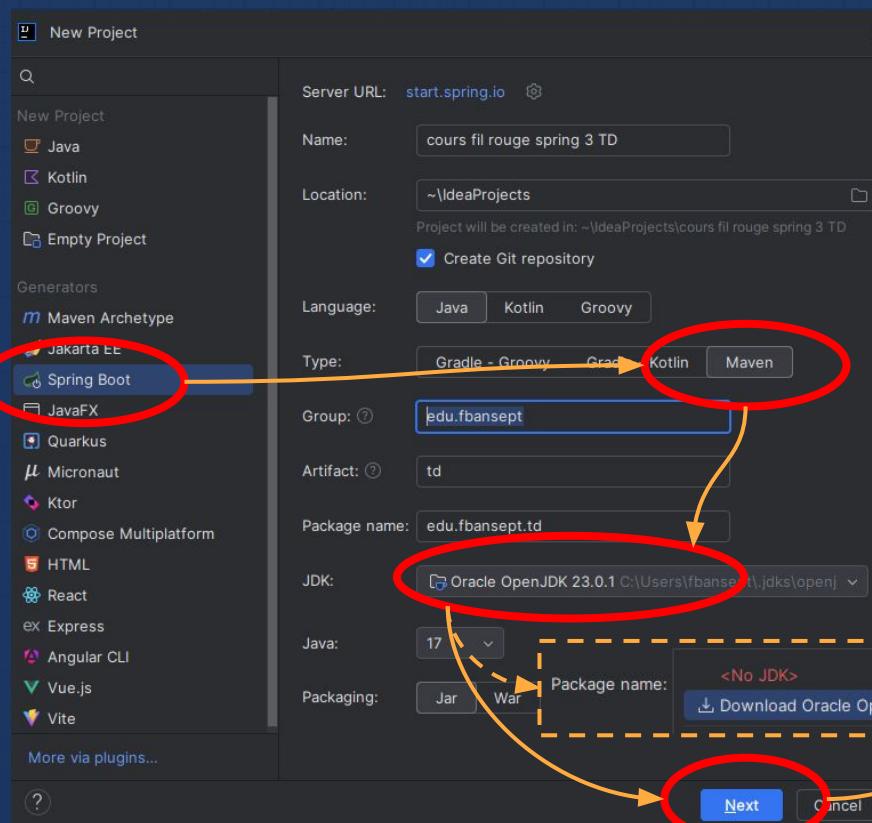
Il embarque également un serveur **Tomcat** où il déploie les applications pendant la phase de développement.





# Créer une application Spring boot avec IntelliJ Ultimate

*Chapitre : Démarrer un projet spring*

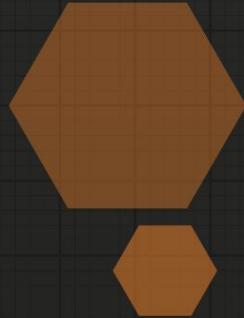


Si vous obtenez cette erreur, utiliser la suggestion de IntelliJ pour définir le JDK à utiliser (17 ou +)

The screenshot shows the IntelliJ IDEA interface with the following details:

- File:** CoursFilRougeSpring3TdApplication.java
- Warning:** Project .JDK is not defined
- Code:**

```
1 package edu.fbansupt.td;
2
3 > import ...
4
5
6 @SpringBootApplication new *
7 public class CoursFilRougeSpring3TdApplication {
8
9     >     public static void main(String[] args) { SpringApplication.run(CoursFilRougeSpring3TdA
10
11
12
13 }
```
- Toolbars:** A red circle highlights the "Setup SDK" button in the top right corner of the toolbar.



# Créer une application Spring boot sans IntelliJ Ultimate (Avec *la version community*)

*Chapitre : Démarrer un projet spring*

>>  **spring initializr**

**Spring initializr** va nous permettre de créer un projet **Spring boot** via **Maven**, en générant une arborescence type et le fichier **pom.xml** correspondant à nos besoins.

Il est disponible à l'adresse : <https://start.spring.io/>

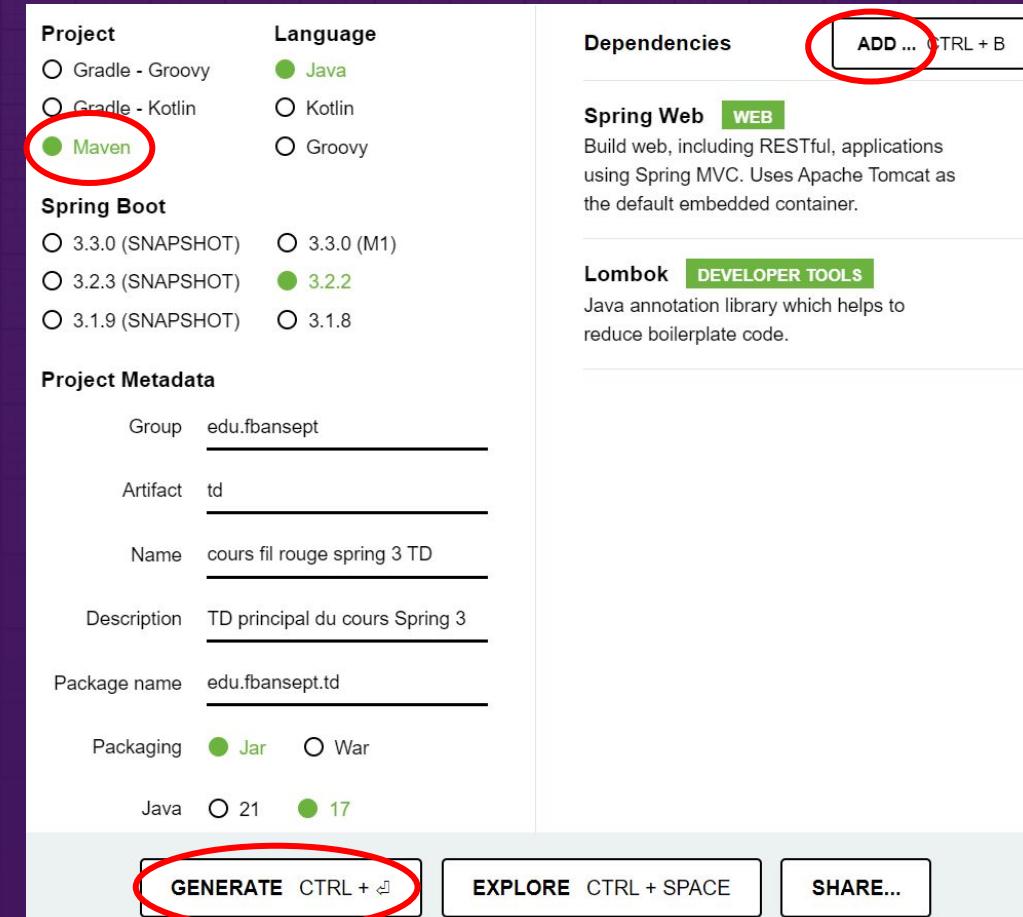
- Project : Maven project
- Langage : JAVA
- Spring boot : laissez la version recommandée
- Metadata : personnalisé à votre convenance
- Packaging : jar
- JAVA : 17

Au niveau des dépendances, sélectionnez uniquement :

### Spring Web et Lombok

*(Nous ajouterons les autres dépendances au moment venu, car elles requièrent certaines configurations)*

Cliquez sur **generate**, un dossier compressé est alors téléchargé.



The screenshot shows the Spring Initializr configuration page. The 'Project' section has 'Maven' selected (circled in red). The 'Language' section has 'Java' selected. Under 'Spring Boot', the recommended version '3.2.2 (SNAPSHOT)' is selected. The 'Project Metadata' section includes fields for Group (edu.fbansept), Artifact (td), Name (cours fil rouge spring 3 TD), Description (TD principal du cours Spring 3), Package name (edu.fbansept.td), and Packaging (Jar, selected). Java version 17 is also selected. At the bottom, the 'GENERATE' button is circled in red, along with the 'ADD ...' button in the 'Dependencies' section.

## >> Ouvrir le projet

Décompresser l'archive à l'endroit de votre choix (*évitez le bureau ou le dossier de téléchargement*)

Via l'interface d'IntelliJ, cliquez sur File/Open... et sélectionnez le fichier **pom.xml**.

Une fenêtre de dialogue vous proposera alors d'ouvrir ce fichier en tant que projet (*as project*).

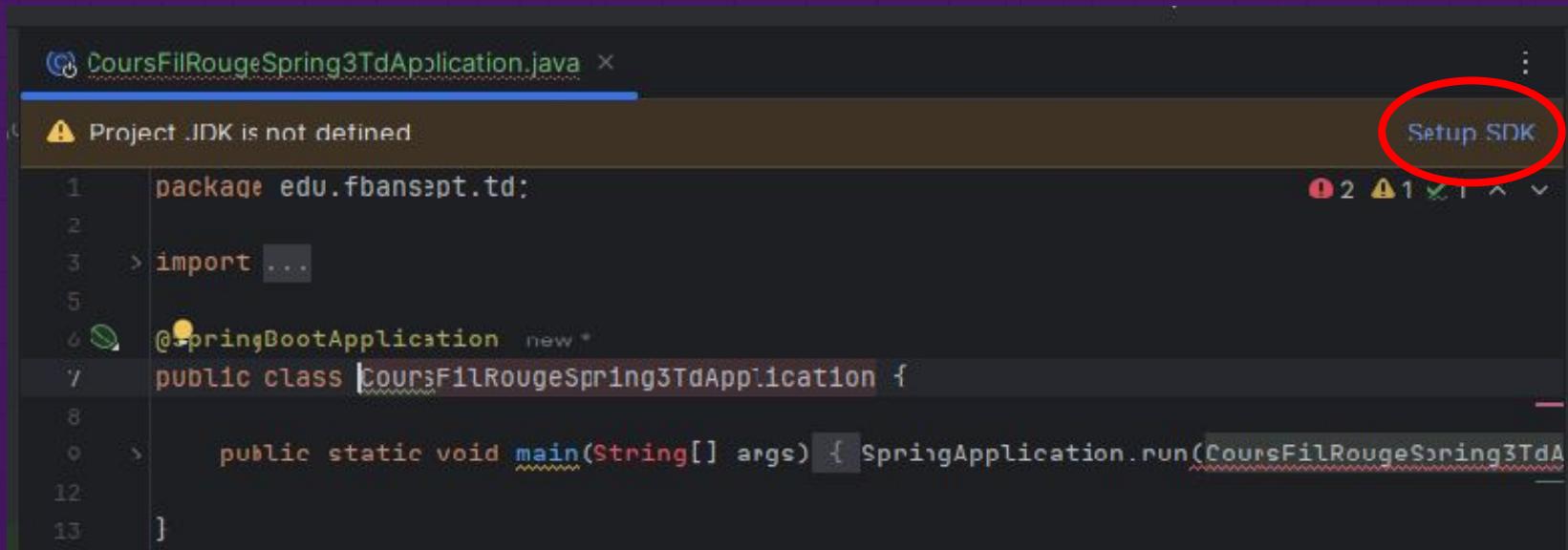
Validez, Maven va alors lire le fichier pom.xml et télécharger les dépendances (cela peut prendre entre 30 et 5 min selon votre connexions).

Vous pouvez voir en bas à droite de l'interface une barre de chargement indiquant si une action est en train de se dérouler.

Une fois terminé, une arborescence est alors visible dans le side menu situé en haut à gauche de l'interface

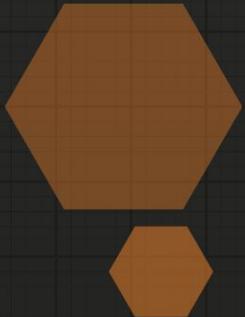


Si vous obtenez cette erreur, utiliser la suggestion de IntelliJ pour définir le JDK à utiliser (17 ou +)



The screenshot shows the IntelliJ IDEA interface with a Java file named `CoursFilRougeSpring3TdApplication.java` open. A yellow warning icon is displayed next to the text "Project .JDK is not defined". In the top right corner of the code editor, there is a button labeled "Setup SDK" which is circled in red. The code itself contains standard Java imports and a Spring Boot application class definition.

```
1 package edu.fbansupt.td;
2
3 > import ...
4
5
6 @SpringBootApplication new *
7 public class CoursFilRougeSpring3TdApplication {
8
9 >     public static void main(String[] args) { SpringApplication.run(CoursFilRougeSpring3TdA
10
11
12 }
13 }
```



# Lancer le projet

*Chapitre : Démarrer un projet spring*



## >> Démarrer le projet

Une fois l'application démarrez, (vois slide précédente) la console s'ouvre (*onglet Run*) et doit se terminer par les lignes :

```
XXX      : XXXXXXXXX  
XXX      : Tomcat started on port(s): 8080 (http) with context path ''  
XXX      : Started CoursFilRougeSpring3TdApplication in X seconds (JVM running for X)
```

Si ce n'est pas le cas, c'est certainement parce que le port par défaut de Tomcat (*le port **8080***) est déjà utilisé par une autre application (*un pare feu, une autre application Tomcat ...*)

Voir la slide suivante pour corriger le problème ....



## >> Windows : Trouver un processus par son port et l'arrêter

Dans un invite de commande (CMD) en mode administrateur tapez la commande suivante :

```
netstat -ano | findstr :8080
```

Cela affichera une ligne comme :

TCP	0.0.0.0:8080	0.0.0.0:0	LISTENING	12345
-----	--------------	-----------	-----------	-------

Le dernier nombre (**12345**) est l'ID du processus (PID) qui utilise le port 8080.

Copier le **PID**, et executer la commande suivante en remplaçant le PID :

```
taskkill /F /PID 12345
```



## >> Linux/MacOS : Trouver un processus par son port et l'arrêter

Dans un terminal taper la commande suivante :

```
lsof -i :8080
```

Cela affichera une ligne comme :

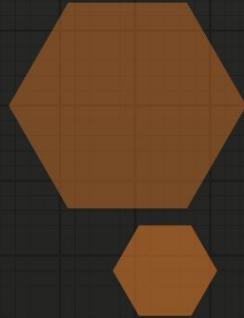
COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE/OFF	NODE	NAME
java	12345	user	67u	IPv6	0t0			TCP *:http-alt (LISTEN)

Le dernier nombre (**12345**) est l'ID du processus (PID) qui utilise le port 8080.

Copier le **PID**, et executer la commande suivante en remplaçant le PID :

```
kill -9 12345
```





# Fixer la configuration pour Lombok

*Chapitre : Démarrer un projet spring*

## >> Fixer un problème avec Lombok apparu avec la version 3 de Spring

Une modification est à effectuer dans le fichier pom.xml afin de faire fonctionner la bibliothèque lombok.

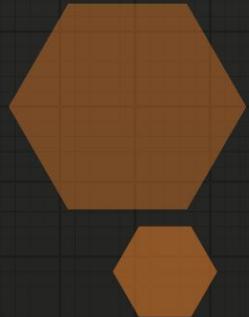
Remplacer les noeuds <plugin> présent dans le noeud <plugins> par la configuration ci contre :

Cela forcera **maven** à charger au minimum la version 1.18.22 de **lombok** qui est la première version compatible avec JAVA 17 (*utilisée par Spring 3*)

Puis mettre à jour l'application via l'icône qui apparaît alors en haut à droite.



```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>17</source>
        <target>17</target>
        <annotationProcessorPaths>
          <path>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>${lombok.version}</version>
          </path>
        </annotationProcessorPaths>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <excludes>
          <exclude>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
          </exclude>
        </excludes>
      </configuration>
    </plugin>
  </plugins>
</build>
```



# Configurer IntelliJ

*Chapitre : Démarrer un projet spring*

## >> Configurer IntelliJ (optionnel)

Modifier le theme : File (menu principal) / Settings / Appearance (menu vertical) / Use custom font

Modifier la taille de la police des menu :File (menu principal) / Settings / Editor(menu vertical) / Font(menu vertical) / Size

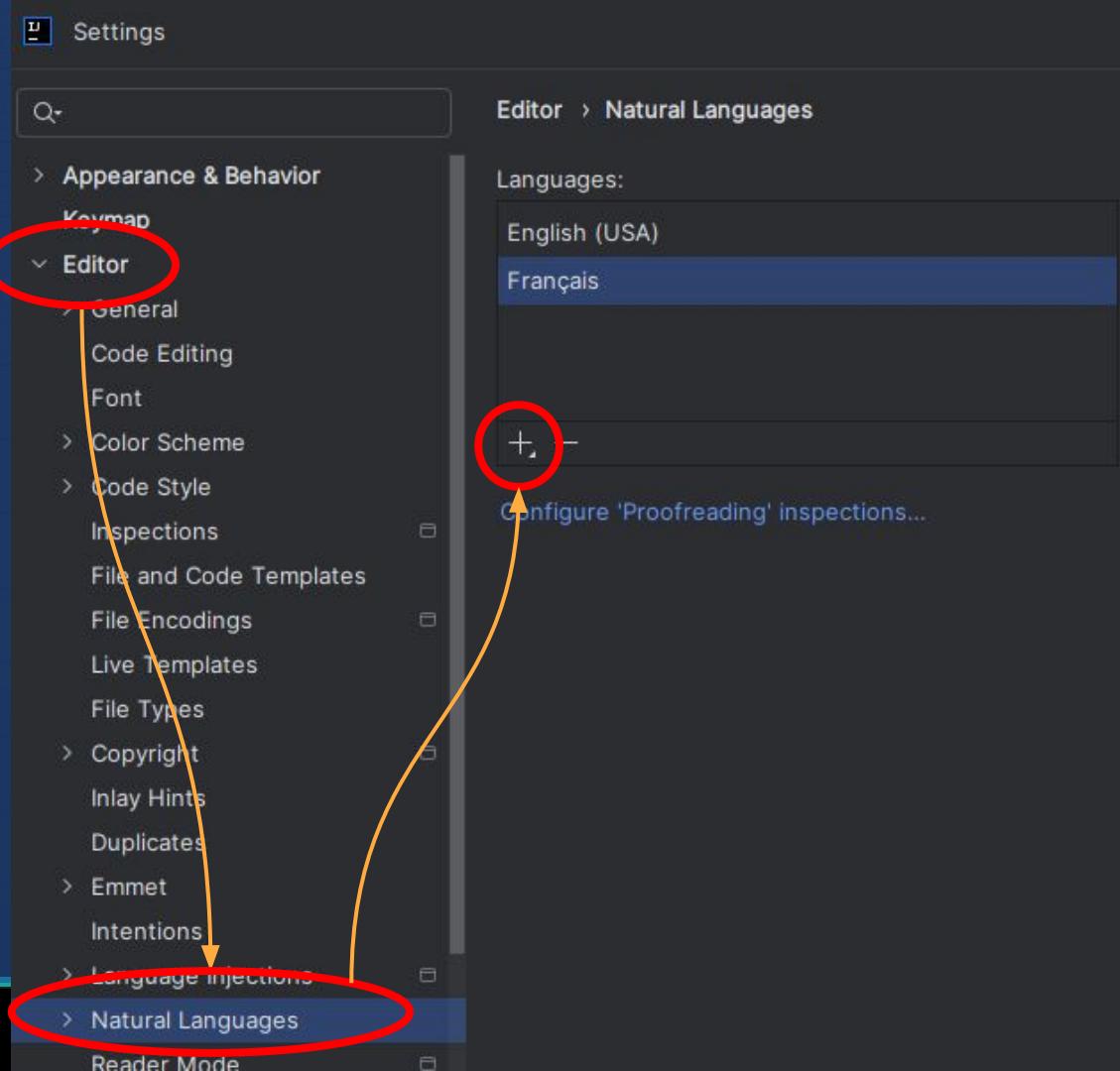
Modifier la taille de la police de l'éditeur : File (menu principal) / Settings / Editor(menu vertical) / Font(menu vertical) / Size

Raccourcis : File (menu principal) / Settings / Editor(menu vertical) / Keymap

*Note : si votre clavier ne dispose pas de pavé numérique, il est recommandé d'ajouter le raccourci "ctrl + :" pour commenter votre code (cherchez "comment" dans le menu Keymap)*

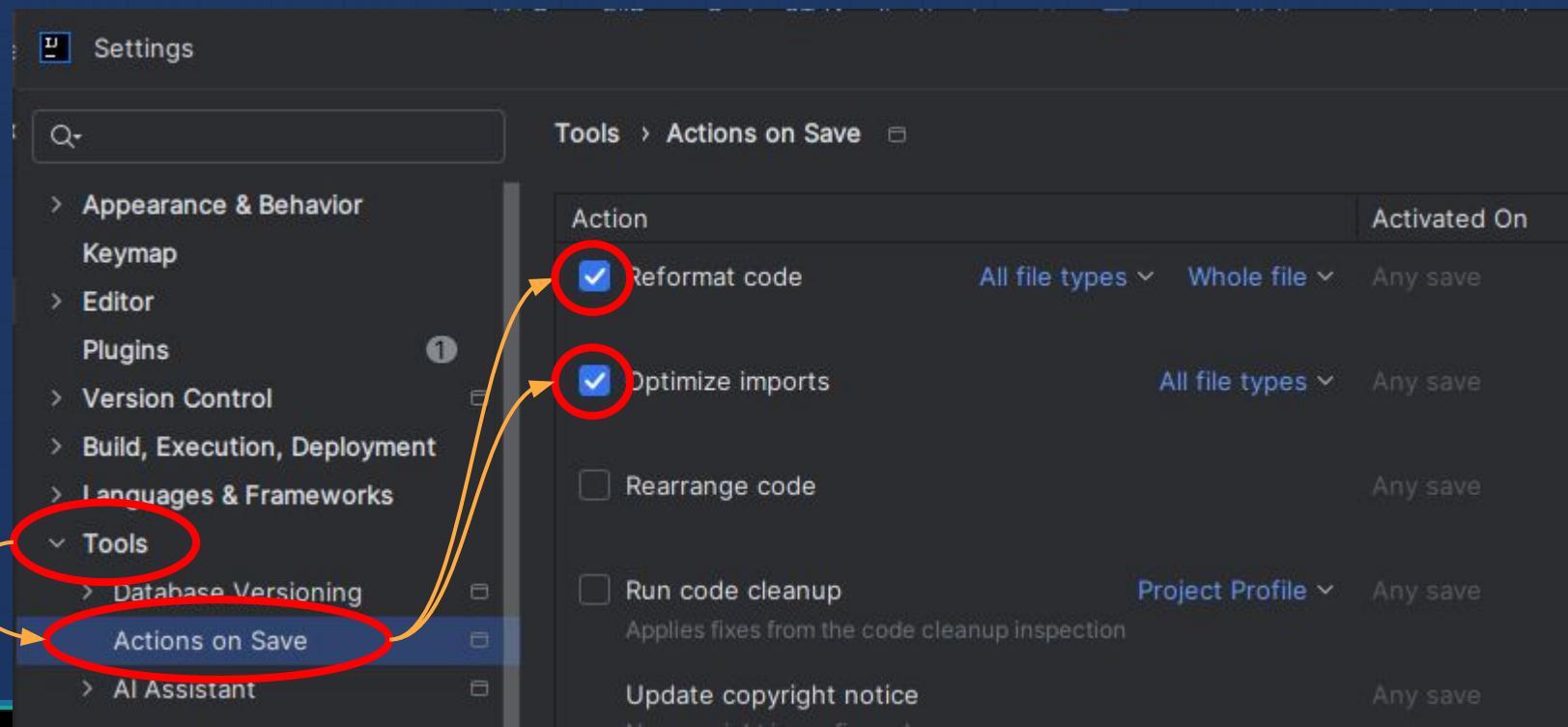


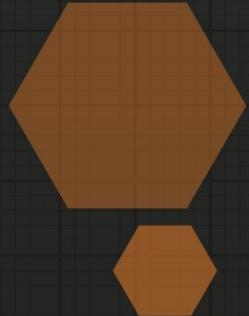
## &gt;&gt; Ajouter un dictionnaire français



## >> Reformater automatiquement

En sauvegardant le fichier (win/linux : `ctrl + S`, macOs : `cmd + S`) le fichier sera reformatted (*indentation, retour à la ligne*).  
Et les import inutiles seront supprimés





# Configurer les Linter

*Chapitre : Démarrer un projet spring*

# Qu'est ce qu'un Linter (*Analyseur de conformité du code*)

Un linter est un outil qui analyse statiquement le code source pour détecter des problèmes potentiels, tels que :

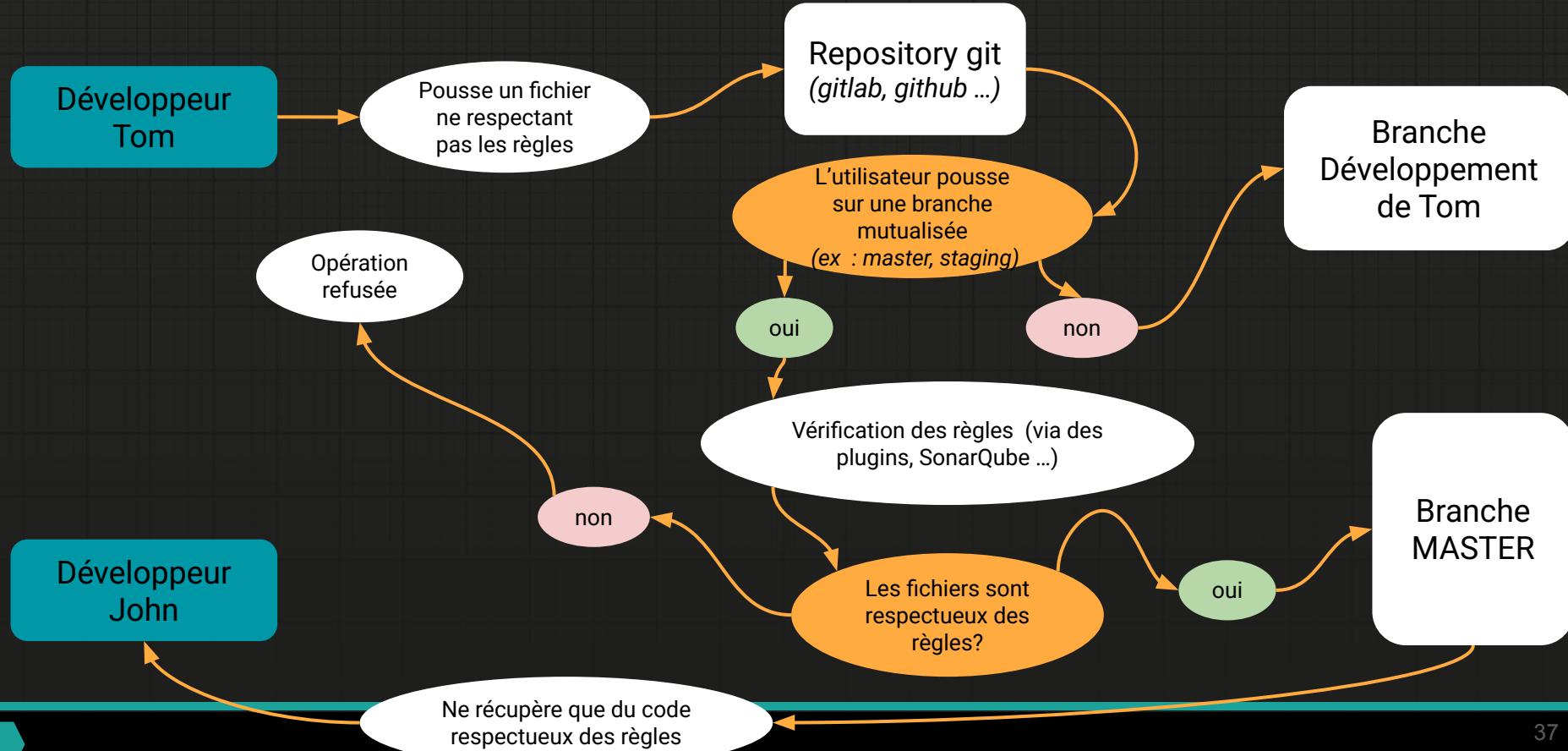
- Les erreurs de style et de formatage (*indentation, espaces, conventions de nommage...*)
- Les mauvaises pratiques de codage
- Les violations de règles spécifiques définies dans un fichier de configuration

Ils permettent de forcer le développeur à mettre en place de bonne pratique de code, ainsi que de forcer une équipe à utiliser le même formatage et ainsi limiter les problème de fusion de code et faciliter sa lecture.

*Note : il n'y a pas de traduction exacte en français, on utilise donc le terme "linter" indistinctement en Français, mais on peut le traduire par : "Analyseur de conformité du code"*



# DevOps : empêcher des commit de code qui enfreint les règles



# >> Comparatif Linter gratuits

## SonarQube community

Avantages :

- C'est un produit très populaire
- Un seul logiciel pour toutes les fonctionnalités (*vérification des règles, qualité du code, découverte de bug*)
- Il permet de générer des dashboards sur la qualité du code

Désavantages :

- Il nécessite un serveur pour générer des rapports (local ou hébergé)
- Il nécessite un serveur pour bloquer des merges sur github (hébergé)

## CheckStyle + PMD + SpotBug

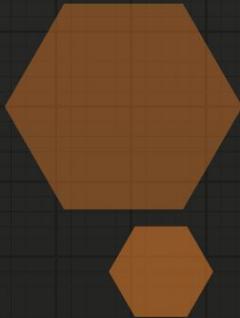
Avantages :

- Il permet de bloquer les merges sans serveur

Désavantages :

- Combinaison de 3 logiciels à configurer
- Pas de dashboards natif sur la qualité du code





# SonarQube

Note : afficher les rapports nécessite docker  
Note 2 : bloquer un commit nécessite un serveur dédié

*Chapitre : Démarrer un projet spring*

## &gt;&gt; Configurer SonarQube for IDE (SonarLint)

The screenshot shows the IntelliJ IDEA settings interface, specifically the 'Plugins' section under 'Marketplace'. A search bar at the top has 'sonar' typed into it. Below the search bar, a red circle highlights the search results, which show 'Search Results (6)' and 'Sort By: Relevance'. The first result, 'SonarQube for IDE', is highlighted with a red circle and has an 'Install' button. A large callout bubble with a blue border and white text points to this 'Install' button with the text 'Install / Restart IDE'. To the right of the main search area, there are tabs for 'Inspection', 'Security', 'Code Quality', and 'Static Analysis'. The 'SonarQube for IDE' plugin card includes a 'SonarSource' link, an 'Install' button, and a 'Plugin homepage' link.

Settings

Plugins Marketplace Installed

Appearance & Behavior Keymap Editor Plugins Version Control Build, Execution, Deploy Languages & Framework Tools Backup and Sync Advanced Settings

sonar

Search Results (6) Sort By: Relevance

SonarQube for IDE ↓ 9,4M ⭐ 3.64 SonarSource

SonarAnalyzer ↓ 75,8K ⭐ 4.08 Yu Junyang

Sonargraph Integration ↓ 11,6K hello2morrow, Inc.

Swiss-AS Dev-Tools ↓ 2,4K ⭐ 4.56 Alain Tavan

Inspection Security Code Quality Static Analysis

SonarQube for IDE

SonarSource Plugin homepage ↗

Install

Additional Info

IDE forms a powerful end-to-end code quality platform to enrich the CI/CD pipeline, ensuring any code edits or additions are clean. In Connected Mode, your team can share common language rulesets, project analysis settings and more.

SonarQube for IDE is a powerful open-source tool for developers

## &gt;&gt; Analyser un fichier avec SonarQube for IDE (SonarLint)

The screenshot shows a Java file named `CoursFilRougeSpring3TdApplication.java` in a code editor. An orange arrow points to line 14, which contains the code `System.out.println("Hello world");`. A red circle highlights the SonarQube icon in the bottom-left corner of the editor.

The SonarQube interface displays the following information:

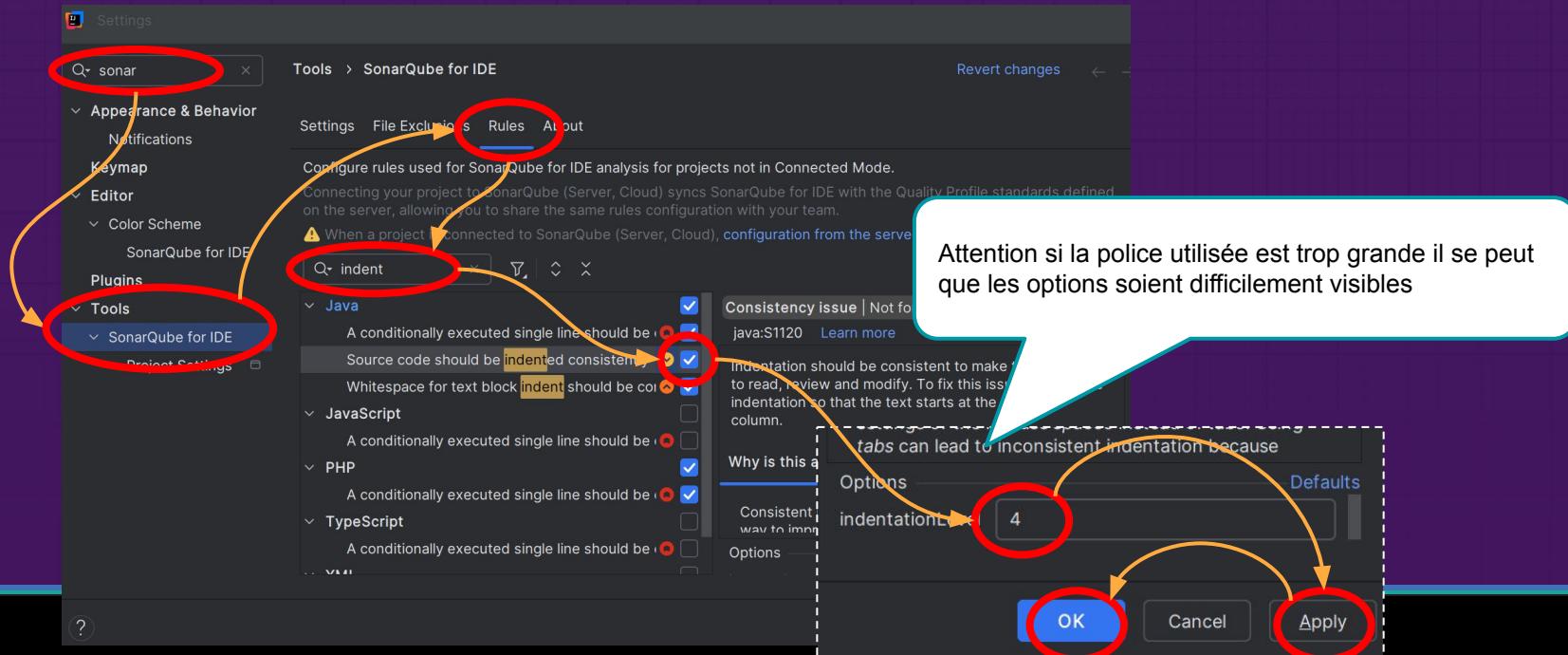
- Rule:** Standard outputs should not be used ..
- Locations:** CoursFilRougeSpring3TdApplication.java (1 issue)
- Issue:** (14, 8) Replace this use of System.out by a logger. 20 days ago
- Type de règle:** Adaptability issue | No modular
- Identifier de la règle:** java:S106
- Explanations:** In software development, logs serve as a record of events within an application, providing crucial insights for debugging. When logging, it is essential to ensure that the logs are:

## &gt;&gt; Ajouter / configurer une règle avec SonarQube

Dans settings (mac : preference) / Tools (menu vertical) / SonarQube for IDE / Rules (onglet)

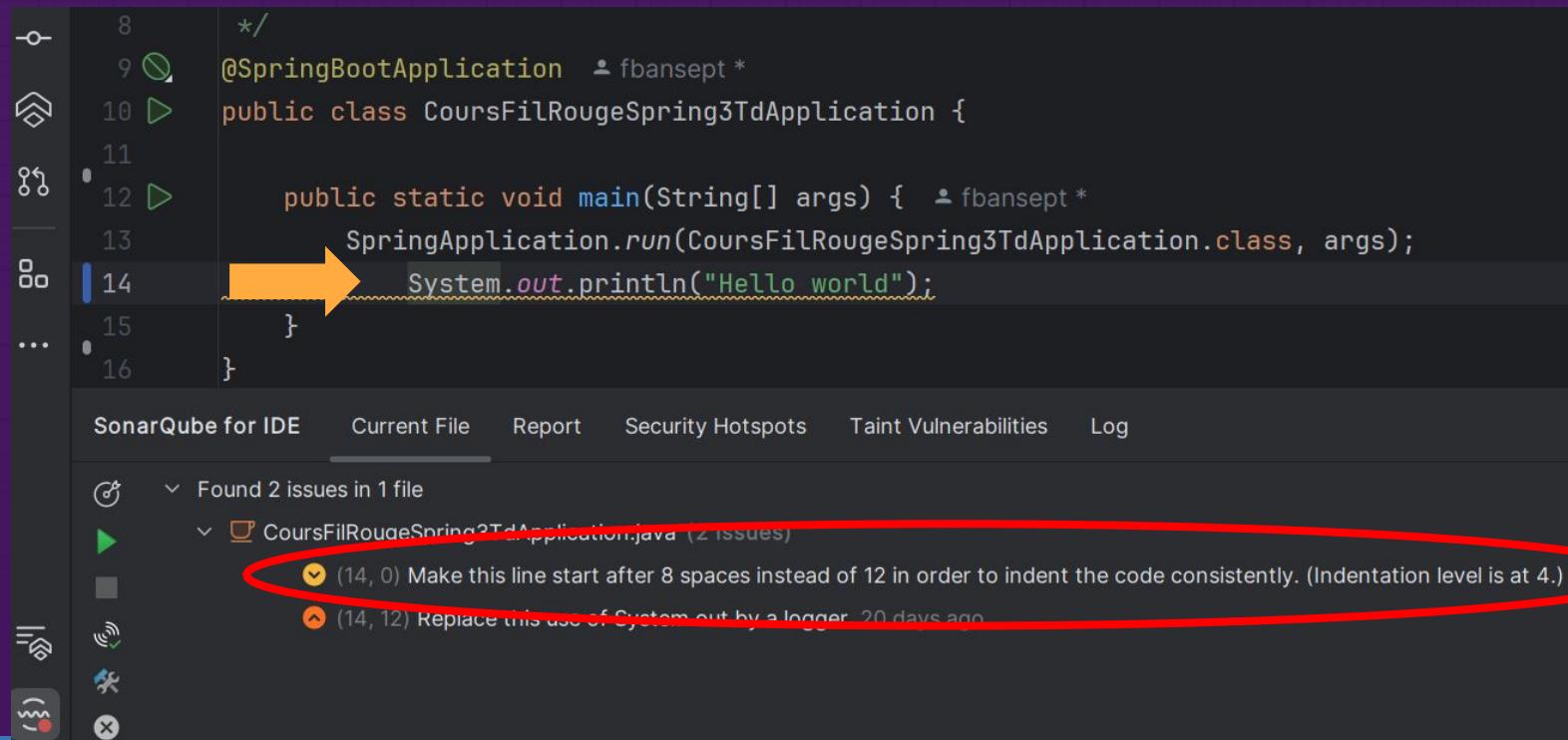
Recherchez "indent" afin d'activer la règle imposant une indentation constante entre 2 instructions en cochant la case

Modifiez le nombre d'espace représentant une indentation à 4 puis cliquez sur "Apply" puis "OK"



## &gt;&gt; Ajouter / configurer une règle avec SonarQube

Une nouvelle erreur doit apparaître en cas de mauvaise indentation



```
8 */  
9 @SpringBootApplication *  
10 public class CoursFilRougeSpring3TdApplication {  
11  
12     public static void main(String[] args) { *  
13         SpringApplication.run(CoursFilRougeSpring3TdApplication.class, args);  
14         System.out.println("Hello world");  
15     }  
16 }
```

SonarQube for IDE    Current File    Report    Security Hotspots    Taint Vulnerabilities    Log

Found 2 issues in 1 file

CoursFilRougeSpring3TdApplication.java (2 issues)

- (14, 0) Make this line start after 8 spaces instead of 12 in order to indent the code consistently. (Indentation level is at 4.) fix
- (14, 12) Replace this use of System.out by a logger. 20 days ago

## >> Installer le serveur SonarQube via Docker

<https://medium.com/@johanesimarmata/sonarqube-guide-improve-code-quality-and-code-security-4fa0c7b02ef2>



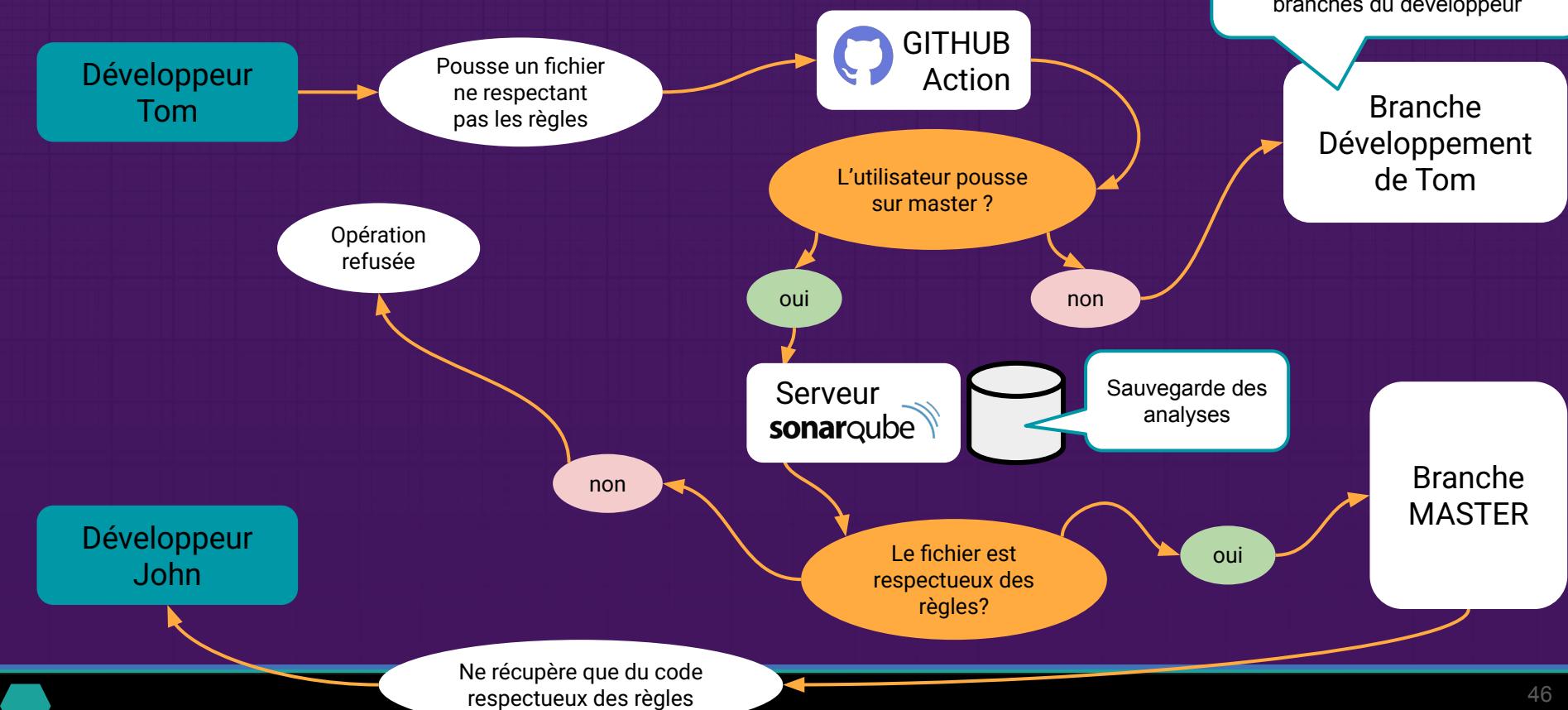
## >> Bloquer un commit via SonarQube

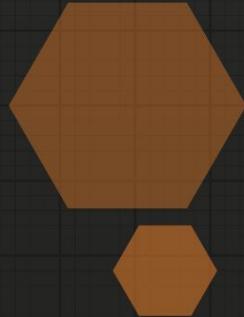
<https://docs.sonarsource.com/sonarqube-server/latest/setup-and-upgrade/install-the-server/installing-sonarqube-from-docker/>

github app : <https://github.com/settings/apps>



>> Ajouter la vérification des règles lors d'un push





Alternative à SonarQube community:

CheckStyle + PMD + SpotBugs

*Chapitre : Démarrer un projet spring*

## >> Descriptif des logiciels de linting

### **Checkstyle**

Outil d'analyse de code statique qui se concentre principalement sur le style de codage et les conventions. Il aide les programmeurs à écrire du code Java qui respecte une norme de codage définie, en automatisant le processus de vérification des règles de style.

### **PMD**

Analyseur de code source qui recherche des problèmes potentiels dans le code, tels que des bugs, du code inutilisé, et des expressions inefficaces. Il est plus orienté vers la détection de mauvaises pratiques de codage et de problèmes de conception.

### **SpotBugs**

Analyse le bytecode Java pour détecter des bugs potentiels. Il est le successeur de FindBugs et est utilisé pour identifier des problèmes qui peuvent ne pas être évidents dans le code source, comme les fuites de ressources ou les erreurs de logique.



## &gt;&gt; Configurer le Linter CheckStyle

The screenshot shows the IntelliJ IDEA settings interface. On the left, the sidebar lists various configuration sections like Appearance & Behavior, Editor, and Plugins. The Plugins section is currently selected, indicated by a blue background and a notification badge '1'. A search bar at the top right contains the query 'CheckStyle'. Below it, the 'Marketplace' tab is active, showing search results for 'CheckStyle'.

**Search Results (37)**

- CheckStyle-IDEA** by Jamie Shiell (4.33 stars, 4.8M downloads) - **Install**
- Regexp Tester** by Sergey Evdokimov (4.26 stars, 473K downloads) - **Install**
- Flutter Pub Version Checker** by Paulina Szklarska (4.85 stars, 171.7K downloads) - **Install**

**CheckStyle-IDEA** by Jamie Shiell ([Plugin homepage](#))

This plugin provides both real-time and on-demand scanning of Java files with Checkstyle from within IDEA.

Please note this is not an official part of Checkstyle - they neither endorse nor bear responsibility for this plugin. Please see the README for full details.

## >> Ajouter un fichier de règles personnalisées

Il existe un certain nombre de conventions de nommage / indentation / bonne pratiques ...

Chaque organisation est libre de suivre ses propres règles.

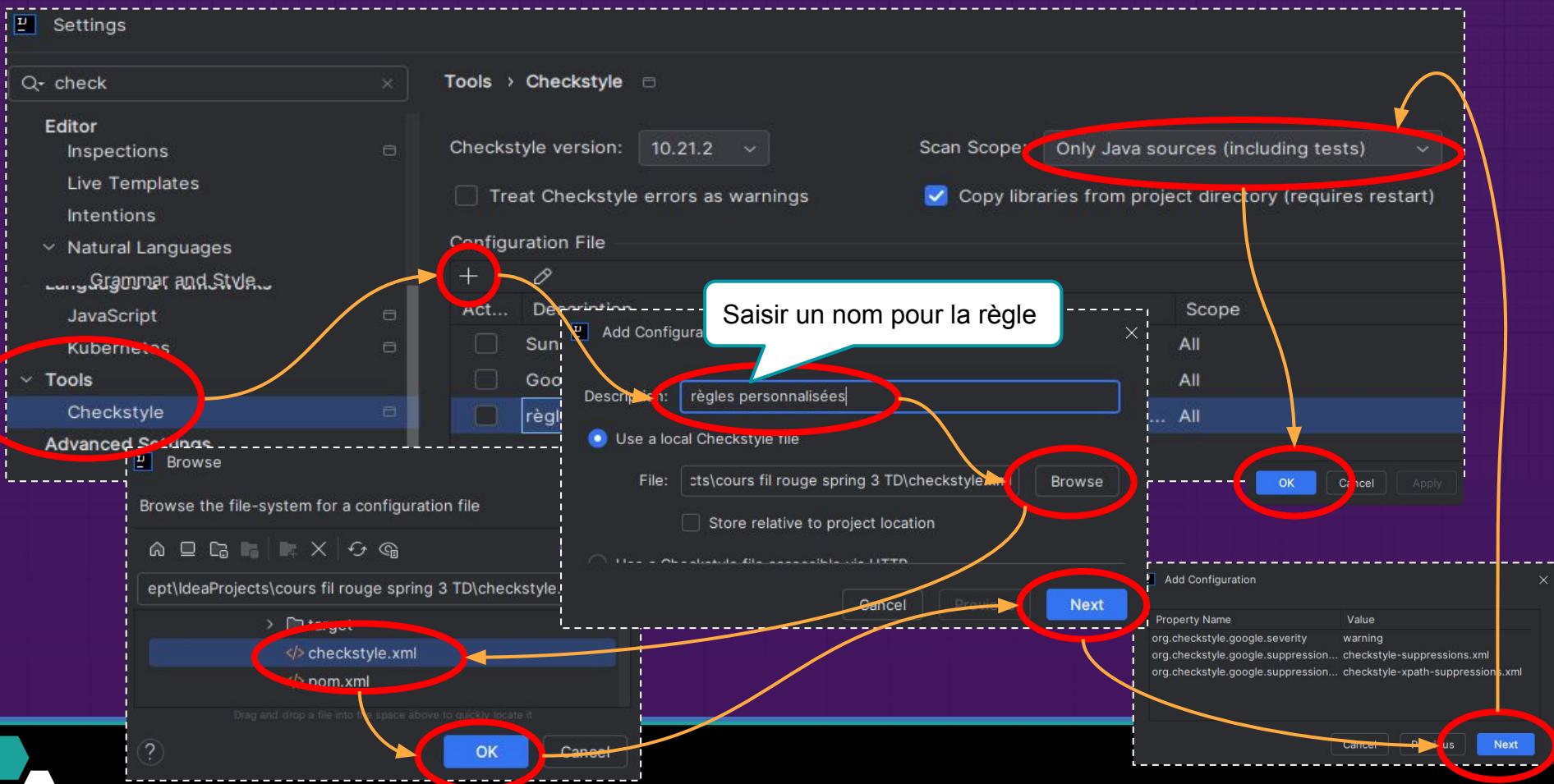
Nous allons nous baser sur les règles de Google qui sont assez répandues :

[https://github.com/checkstyle/checkstyle/blob/master/src/main/resources/google\\_checks.xml](https://github.com/checkstyle/checkstyle/blob/master/src/main/resources/google_checks.xml)

Copiez les règles et placez les dans un nouveau fichier “checkstyle.xml” à la racine du projet (*au même niveau que le fichier pom.xml*)



## Dans Settings (Preferences sur MacOs)

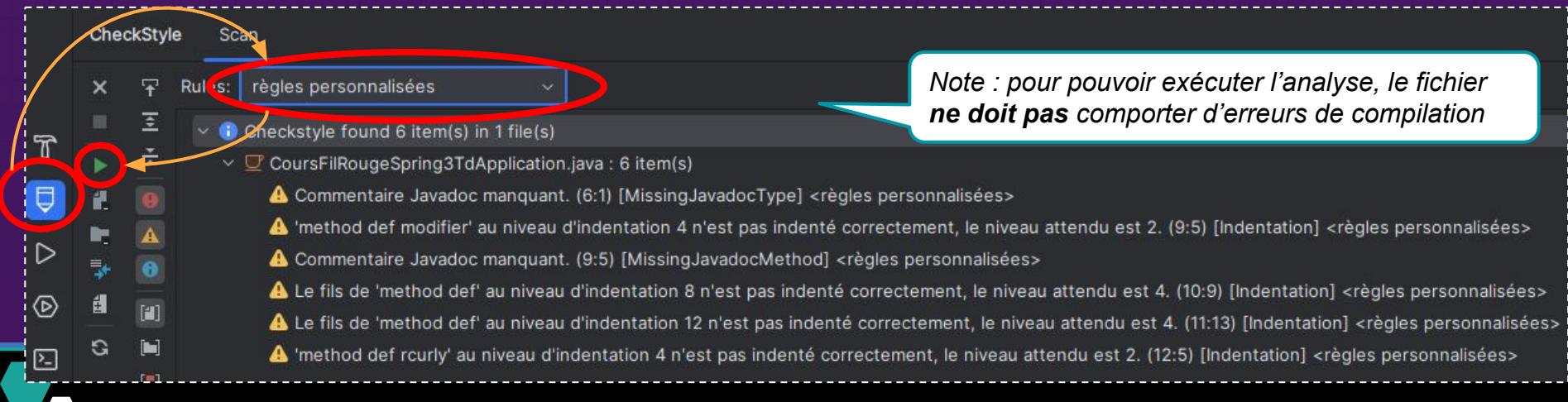


## &gt;&gt; Tester localement

Dans le fichier principal de votre application (`src / main / java / vos packages / VotreApplication`)

```
@SpringBootApplication  
public class CoursFilRougeSpring3TdApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(CoursFilRougeSpring3TdApplication.class, args);  
        System.out.println("instruction mal indentée");  
    }  
}
```

Introduire une instruction mal indentée



## >> Corriger le fichier de la classe Principale

Afin de respecter les règles, nous allons ajouter des commentaires javadoc sur la classe et la méthode, et corriger l'indentation. Il reste un problème cependant : les règles actuelles imposent des indentations de 2 espaces alors qu'IntelliJ en utilise 4 (et 8 pour les *instructions sur plusieurs lignes* : "lineWrappingIndentation")

*Fichier principal (src / main / java / vos packages / VotreApplication)*

```
/**  
 * Main class.  
 */  
@SpringBootApplication  
public class CoursFilRougeSpring3TdApplication {  
  
    /**  
     * Main method.  
     *  
     * @param args Not used.  
     */  
    public static void main(String[] args) {  
        SpringApplication.run(CoursFilRougeSpring3TdApplication.class,  
args);  
        System.out.println("instruction mal indentée");  
    }  
}
```

## >> Corriger le fichier de règles personnalisée

Dans le fichier checkstyle.xml, modifier le noeud `<module name="Indentation">` en doublant toutes les valeurs

`checkstyle.xml`

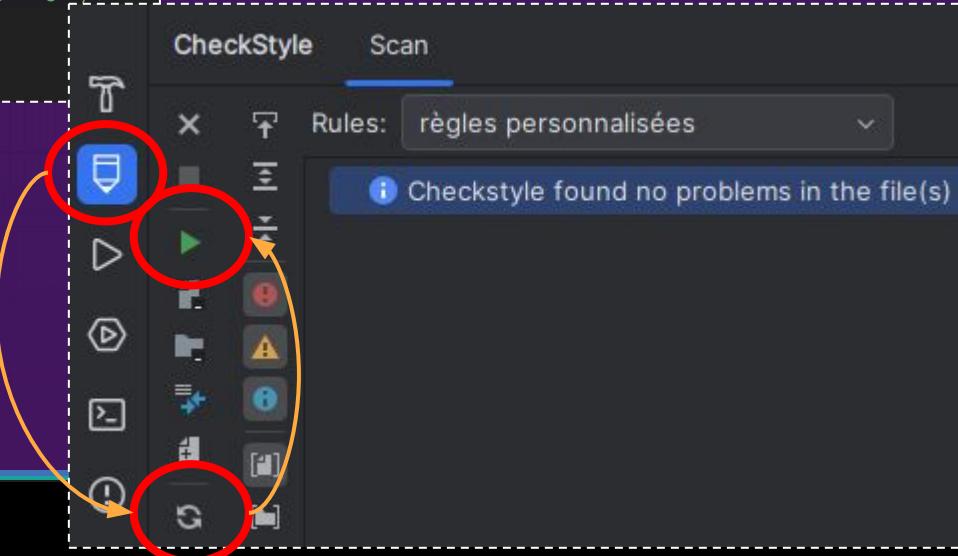
```
<module name="Indentation">
    <property name="basicOffset" value="4" />
    <property name="braceAdjustment" value="4" />
    <property name="caseIndent" value="4" />
    <property name="throwsIndent" value="8" />
    <property name="lineWrappingIndentation" value="8" />
    <property name="arrayInitIndent" value="4" />
</module>
```

Ouvrez de nouveau le fichier principal dans l'éditeur

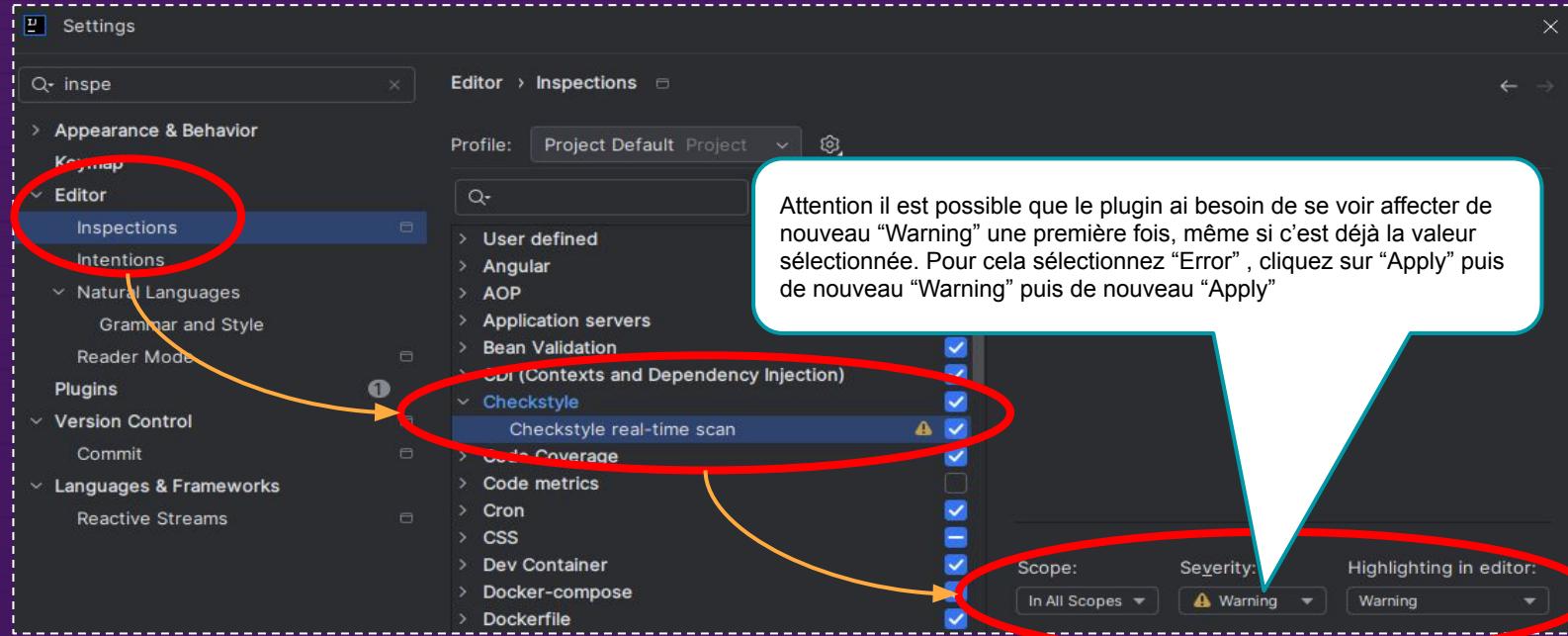
Rafraîchissez les règles

Relancez la vérification

Il ne devrait plus y avoir d'erreurs



## &gt;&gt; Activer la vérification en temps réel



The screenshot shows the IntelliJ IDEA editor with a code snippet. A red circle highlights the 'instruction mal indentée' part of the code. A callout bubble points to this red circle with the text: 'Checkstyle: Le fil d'indentation au niveau de 'method def' n'est pas correctement indiqué, le niveau attendu est 8.' Below the code, there are several status indicators: 'Suppress for Checkstyle Alt+Maj+Entrée', 'More actions... Alt+Entrée', and a small icon with a red circle.

## >> Créer un blocage en cas de non respect des règles

Dans le fichier checkstyle.xml, modifier le noeud <property name="severity" ...>

Afin de considérer un non respect des règles comme une erreur et non un avertissement

*checkstyle.xml*

```
<property name="severity" value="${org.checkstyle.google.severity}" default="error" />
```

Changer warning pour error



## &gt;&gt; Ajouter l'exécution du Linter CheckStyle dans le cycle de vie du projet

pom.xml

```
<plugin>
    <groupId>org.apache.maven.plugins </groupId>
    <artifactId>maven-checkstyle-plugin </artifactId>
    <version>3.6.0</version>
    <dependencies>
        <dependency>
            <groupId>com.puppycrawl.tools </groupId>
            <artifactId>checkstyle</artifactId>
            <version>10.21.2</version>
        </dependency>
    </dependencies>
    <executions>
        <execution>
            <phase>validate</phase>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <configLocation>${project.basedir}/checkstyle.xml </configLocation>
        <failOnViolation>true</failOnViolation>
    </configuration>
</plugin>
```

À ajouter dans le noeud  
build / plugins

puis synchronisez le projet  
maven : 



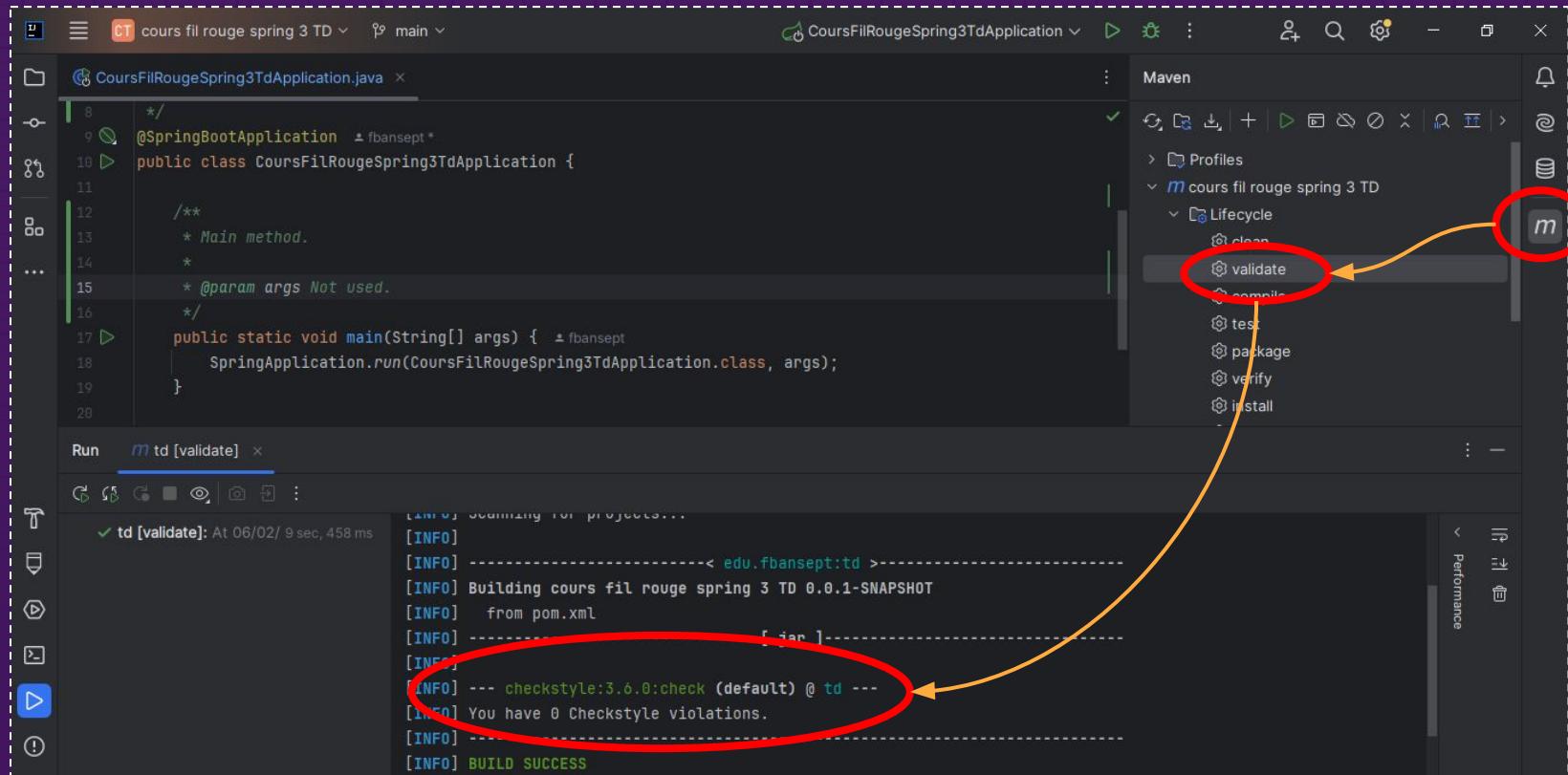
>> Améliorer son code (alternative à SonarQube)

>> Explication du fichier pom.xml

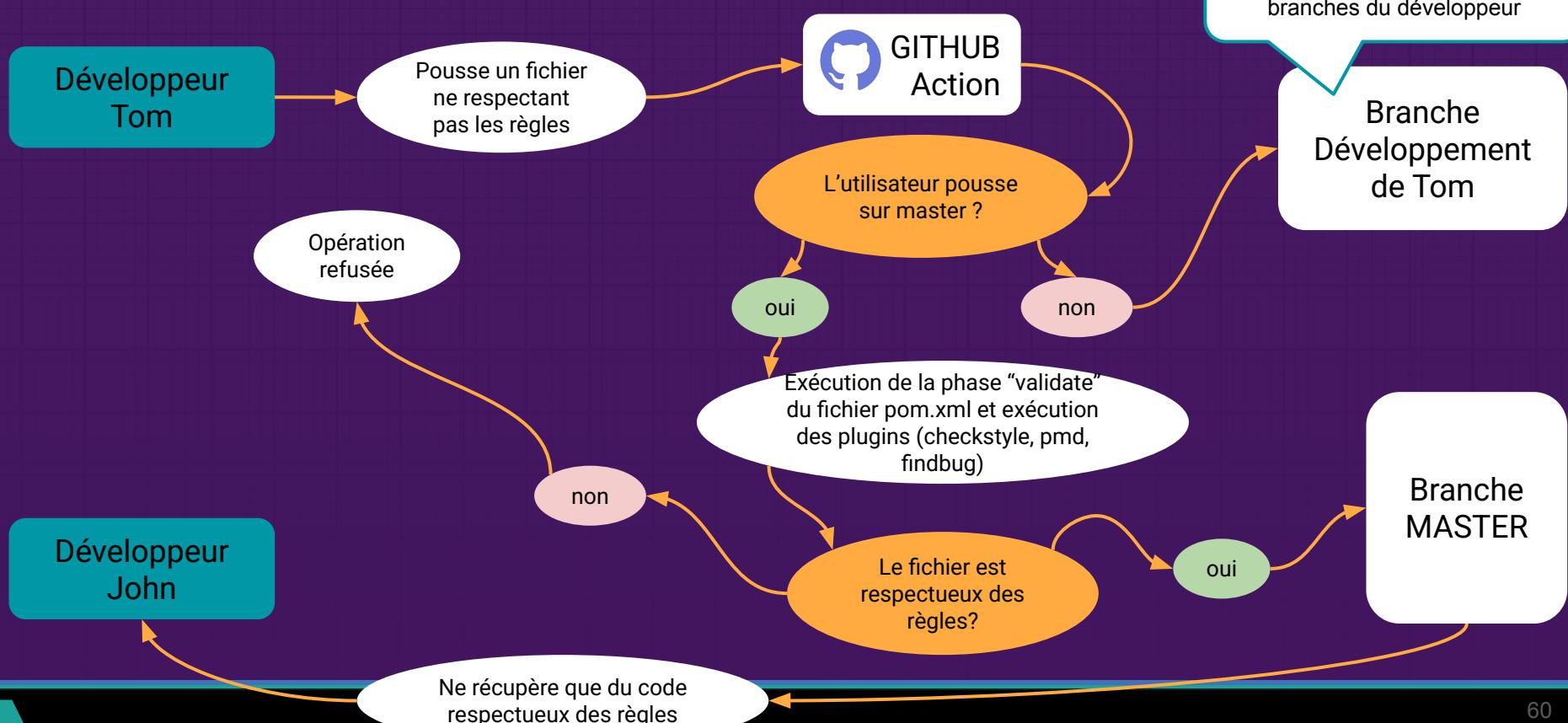
GOAL VS PHASE

DEPENDENCY



>> Vérifier localement la phase *validate* et l'exécution du plugin *Checkstyle*

## >> Ajouter la vérification des règles lors d'un push



The screenshot shows the GitHub Actions interface. At the top, there are navigation links: Code, Issues, Pull request, Actions (which is highlighted with a red circle), Projects, Wiki, Security. Below this, a large heading says "Choose a workflow". A sub-section below it says "Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want them to." A blue link "Skip this and set up a workflow yourself →" is circled in red. At the bottom, there's a file editor for "main.yml" in the ".github/workflows" directory of the repository "cours\_fil\_rouge\_spring\_3\_TD". The editor has tabs for "Edit" and "Preview". The main area contains the YAML configuration for the workflow. A green button "Commit changes..." is circled in red. A red box highlights the entire content of the "main.yml" file editor area.

Puis effectuez un pull en local pour récupérer ce fichier

```
name: Checkstyle Validation

on:
  pull_request:
  push:
    branches:
      - main

jobs:
  checkstyle:
    runs-on: ubuntu-latest

steps:
  - name: 🛠 Checkout repository
    uses: actions/checkout@v4

  - name: ☕ Setup JDK 17
    uses: actions/setup-java@v3
    with:
      distribution: 'temurin'
      java-version: '17'

  - name: 🚀 Build & Run Checkstyle
    run: mvn checkstyle:check
```

The screenshot shows the GitHub Settings page for a repository. A red circle highlights the 'Settings' tab in the top navigation bar. A large red oval encloses the 'Branches' section in the left sidebar. A yellow arrow points from the 'Code and automation' dropdown in the sidebar to the 'Branches' section. Another yellow arrow points from the 'Branches' section to the 'main' branch name in the main content area. A red circle highlights the 'main' branch name. A yellow arrow points from the 'Branches' section to the 'Require status checks to pass before merging' checkbox. This checkbox is checked and highlighted with a red circle. A yellow arrow points from the checkbox to a search bar containing 'checkstyle'. A red circle highlights the search bar. A yellow arrow points from the search bar to the 'checkstyle' entry in the list below. A red circle highlights the 'checkstyle' entry. A yellow arrow points from the 'checkstyle' entry to the 'Do not allow bypassing the above settings' checkbox at the bottom. This checkbox is also checked and highlighted with a red circle. A blue callout bubble at the bottom left contains the text: "Cocher cette case si même les administrateurs doivent respecter les règles".

General

Access

Collaborators

Moderation options

Code and automation

Branches

Tags

Rules

Level

Access

Collaborators

Moderation options

Code and automation

Branches

Tags

Actions

Webhooks

Environments

Codespaces

Pages

Security

Code security

Deploy keys

Secrets and variables

Integrations

Branch name pattern \*

main

Protect matching branches

Require a pull request before merging

When enabled, all commits must be made to a non-protected branch and submitted via a pull request before they can be merged into a branch that matches this rule.

Require status checks to pass before merging

Choose which status checks must pass before branches can be merged into a branch that matches this rule. When enabled, commits must first be pushed to another branch, then merged or pushed directly to a branch that matches this rule after status checks have passed.

Require branches to be up to date before merging

This ensures pull requests targeting a matching branch have been tested with the latest code. This setting will not take effect unless at least one status check is enabled (see below).

Q checkstyle

Status checks that are required

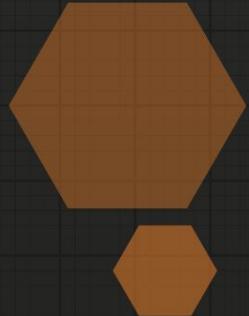
checkstyle

GitHub Actions

Do not allow bypassing the above settings

The above settings will apply to administrators and custom roles with the "bypass branch protections" permission.

Cocher cette case si même les administrateurs doivent respecter les règles



# Publier le projet sur Github

*Chapitre : Démarrer un projet spring*

>> Créer un compte sur Github

TODO

rappel push / pull / merge ...



>> Créer une branche pour chaque fonctionnalité ou correction

Nommée la branche

bugfix-but-de-la-correction ou feature-but-de-la-feature



## >> Comment ajouter ses modifications à la branche

Avant de pouvoir ajouter des commit à une branche, il est nécessaire de récupérer les éventuelles modifications qui auraient pu être effectuées depuis que l'on a créer notre branche.

Pour cela il existe plusieurs solutions :

git pull

git pull --rebase

git pull --rebase est plus propre que git pull mais il est à exclure si une personne a récupérer la branche et a effectuer un commit depuis

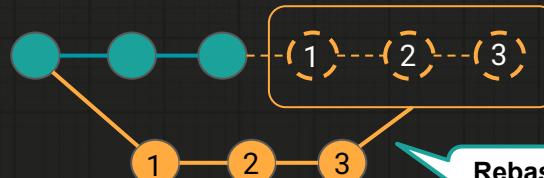
si un git pull --rebase échoue avec un problème de conflit, effectuer un simple git pull



## >> Merge VS Rebase

Rebase :

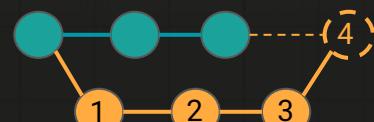
- L'historique des commits est plus lisible
- Les conflits sont plus durs à régler
- A n'utiliser que sur des branches qui ne sont pas partagées (ex, *ne pas utiliser sur : main, master, staging...*)



**Rebase** : les commit sont déplacés comme si il avait été effectués sur la **branche cible**

Merge:

- Historique des commits conservé (sur la **branche d'où provient le commit**) mais il est moins facile à suivre sur la **branche cible**
- Les conflits sont plus facile à régler
- A utiliser sur les branches partagées



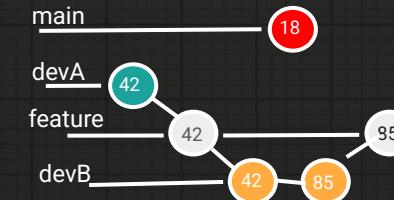
**Merge** : un nouveau commit est créé contenant toutes les modifications

Pour pouvoir lire commit après commit l'historique des changements, il faut regarder les commits situés dans la branche à l'origine du merge

## &gt;&gt; Scénario problématique avec rebase

## 1. Développeur A et B travaillent sur feature-branch :

- Le développeur A pousse un commit C1 sur feature-branch.
- Le développeur B récupère cette branche et ajoute son propre commit C2, puis pousse la branche mise à jour



Un nouvel id est  
42 et devient le  
Le commit 42 d  
va poser problème

## 2. Développeur A fait un rebase :

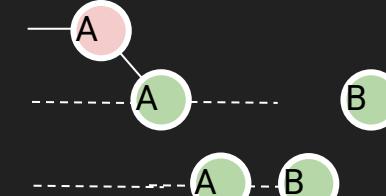
- Le développeur A exécute `git rebase main` sur feature-branch pour intégrer les changements de main.
- Cela réécrit l'historique en modifiant l'ID des commits.

3. Développeur A force le push (`git push --force`) :

- La branche sur le serveur est maintenant différente de la version que B a récupérée.
- Le commit C2 du développeur B semble disparaître du suivant distante.

## Pourquoi un nouveau commit ?

Chaque commit dans la version de Git, qui



## 4. Développeur B tente de pousser ses changements :

- Git refuse le push, indiquant que l'historique local est en冲突 avec la branche distante.
- Si le développeur B ne fait pas attention et exécute `git pull`, cela peut entraîner

- Le contenu du commit (*snapshot* des fichiers)
- L'auteur et la date
- Le message du commit
- **Le commit parent ←**

## >> Comment annuler un commit / un push ?

Avant de pouvoir ajouter des commit à une branche, il est nécessaire de récupérer les éventuelles modifications qui auraient pu être effectuées depuis que l'on a créer notre branche.

Pour cela il existe plusieurs solutions :

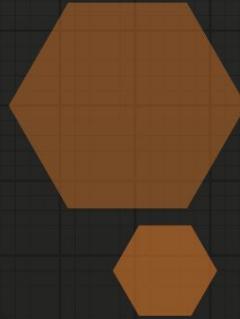
git pull

git pull --rebase

git pull --rebase est plus propre que git pull mais il est à exclure si une personne a récupérer la branche et a effectuer un commit depuis

si un git pull --rebase échoue avec un problème de conflit, effectuer un simple git pull





# Présentation du projet TD et du projet TP

*Chapitre : Démarrer un projet spring*

## >> Modèle conceptuel des données VS diagramme de classe

Afin de représenter un système, on utilise des diagrammes dont les normes d'écriture sont reconnus.

Il existe 2 diagrammes répandus pour représenter des systèmes :

- Le diagramme de classe du langage de modélisation UML (*Unified Modeling Language*)
  - Plus adapté pour représenter des classes
  - Reconnu internationalement
- Le MCD Modèle Conceptuel de Données de la méthode Merise
  - Plus adapté pour représenter les bases de données
  - Surtout utilisé en France

Afin de tirer partie du meilleurs partie des 2 diagrammes, il pourrait être plus avantageux de modéliser la base de donnée avec un MCD et nos classes avec un diagramme de classe.

Dans notre cas nous allons utiliser une façons de programmer dites "Code first" qui permettra de définir la base de donnée en fonction de nos classes dédiées à la représentation de notre structure de données (models)

Ce qui par conséquent rend le MCD optionnel puisque ce sont grâce à des classes que la base de données sera définie

Nous représenterons donc l'application avec un diagramme de classe UML.

*Note : à l'inverse, une approche "database first" rendrait un MCD plus intéressant pour la conception de la base de données*



# >> Le projet Travaux Dirigés

Le cours est orienté projet : afin d'illustrer le cours, un projet fil rouge sera utilisé.  
Il consiste en la réalisation d'une petite plateforme ecommerce

Une entreprise vend des **produits** ayant :

- une TVA (*normal, intermédiaire, réduit, particulier*)
- un *nom*, un *code*, une *description*, un *prix hors taxe*, un *prix TTC* (*calculé en fonction de la TVA et du prix hors taxes*)
- peut posséder des **étiquettes** ("en promotion", "hi-tech", "best seller" ...)

Un **utilisateur** :

- possède un *email* et un *mot de passe*
- est un simple **utilisateur** ou un **administrateur**
- peut ajouter des **produits** dans son panier (*un panier est une commande non validée*)
- peut définir une *quantité* pour chaque **produit** dans son panier (*c'est une ligne de commande*)
- actif permet de désactiver un utilisateur sans le supprimer

Une **commande** :

- à un **statut** (*Non validée, Validée, Annulée, Expédiée*)
- possède une *date* lorsqu'elle est validée

Une **ligne de commande** :

- représente une article avec une certaine quantité (comme un panier)
- un prix de vente TTC au moment de la vente (*attention à ne pas utiliser le prix de Produit qui peut évoluer dans le temps*)

Structure des données (Models)

Utilisateur
id
email
motDePasse
administrateur
actif

Diagramme de classes UML

Commande
id
date

<<ENUMERATION>>
Statut commande
PANIER
A VALIDER
ANNULEE
ENVOYEE
RECEPTIONNEE

Tva
id
designation
pourcentage

Ligne commande
id
prixVente
quantité

Etiquette
id
designation

Produit
id
code
nom
description
prixHt
prixTtc (transient)

## >> Explication TD : choisir entre une Table et une énumération

Dans le sujet du TD, le *statut de la commande* est défini comme une **énumération**, alors que la *TVA* est une **entité** à laquelle on ajoute une **association** sur *produit*.

Une **entité** sera représentée par une **table** en **base de donnée**.

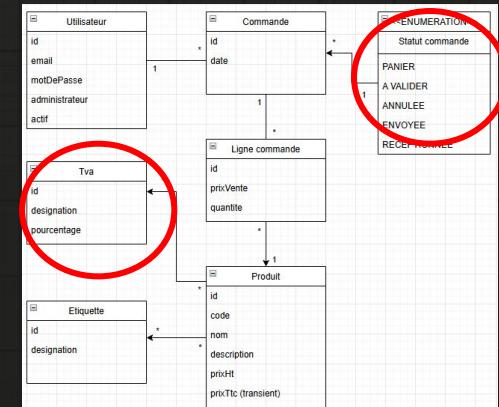
Nous pourrons y ajouter des **enregistrements** (*de nouveaux taux de TVA dans ce cas, ou des pourcentage différents*) sans avoir à modifier notre code.

Et permettre à un utilisateur d'en ajouter via un back office par exemple

Une **énumération** sera représentée par une colonne avec un type **ENUM** au niveau de la **base de données**, et d'un Enum dans le code.

L'intérêt d'un ENUM est qu'il n'y a pas besoin de jointure pour obtenir le nom de l'état dans un produit (*performance et lisibilité*)

Mais l'ajout d'une information (*un nouveau statut dans notre cas*) ce fait par une modification du code, impossible de permettre à un utilisateur d'en créer un.

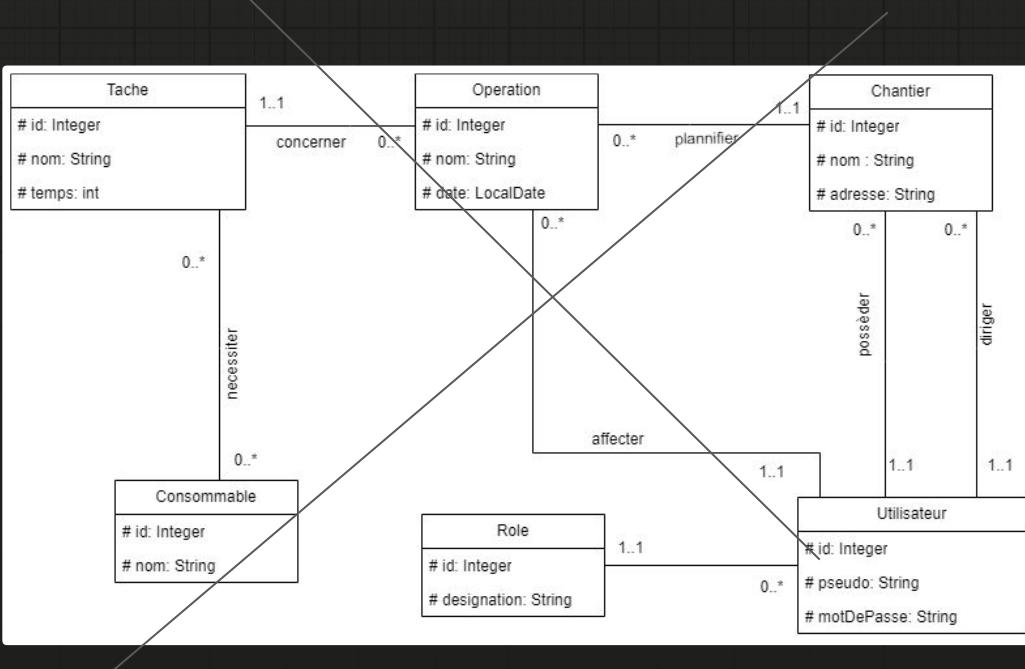


Dans notre cas ce n'est pas un problème car l'ajout d'un nouveau statut de commande entraînera forcément une logique métier supplémentaire, et donc une modification du code.



# >> Le projet Travaux Pratiques

Afin de vérifier la validation des acquis et de permettre la restitution des connaissances, un TP devra être réalisé. Il s'agit d'un projet de gestion des ressources d'une entreprise de construction



Notre entreprise se voit affecter des chantiers, le but de l'application est de gérer ses ressources de manière macroscopique

Un **utilisateur** :

- possède un *pseudo* et un *mot de passe*
- possède un **rôle** (*client, ouvrier, administrateur*)

Un **chantier** :

- possède une *adresse*
- est possédé par un client (*un utilisateur*)
- est dirigé par un ouvrier (*un utilisateur*)
- se voit planifier une liste d'**opérations**

Une **opération** :

- est planifiée pour un **chantier** à une *date* donnée
- concerne une **tâche**
- est affecté à un ouvrier (*un utilisateur*)

Une **tâche** :

- à un *nom* et un *temps* de réalisation (en minute)
- peut nécessiter une liste de **consommables**

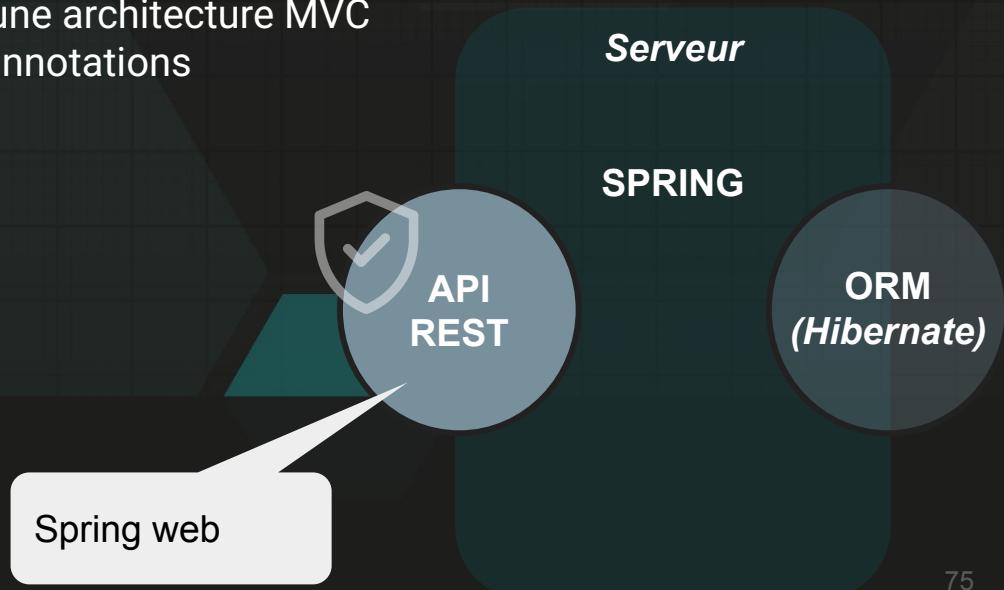
Un **consommable** :

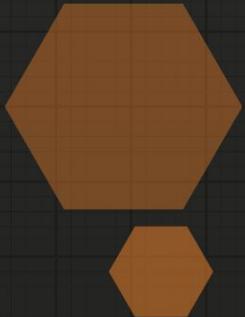
- à un *nom*
- peut être nécessaire à certaines **tâches**

# Spring web

Dans ce chapitre :

- Mettre en place une architecture MVC
- Le principe des annotations





# Le design pattern MVC

*Chapitre : Démarrer un projet spring*

## >> Principe du pattern MVC

Le pattern MVC permet de séparer le code d'une application en 3 parties distinctes :

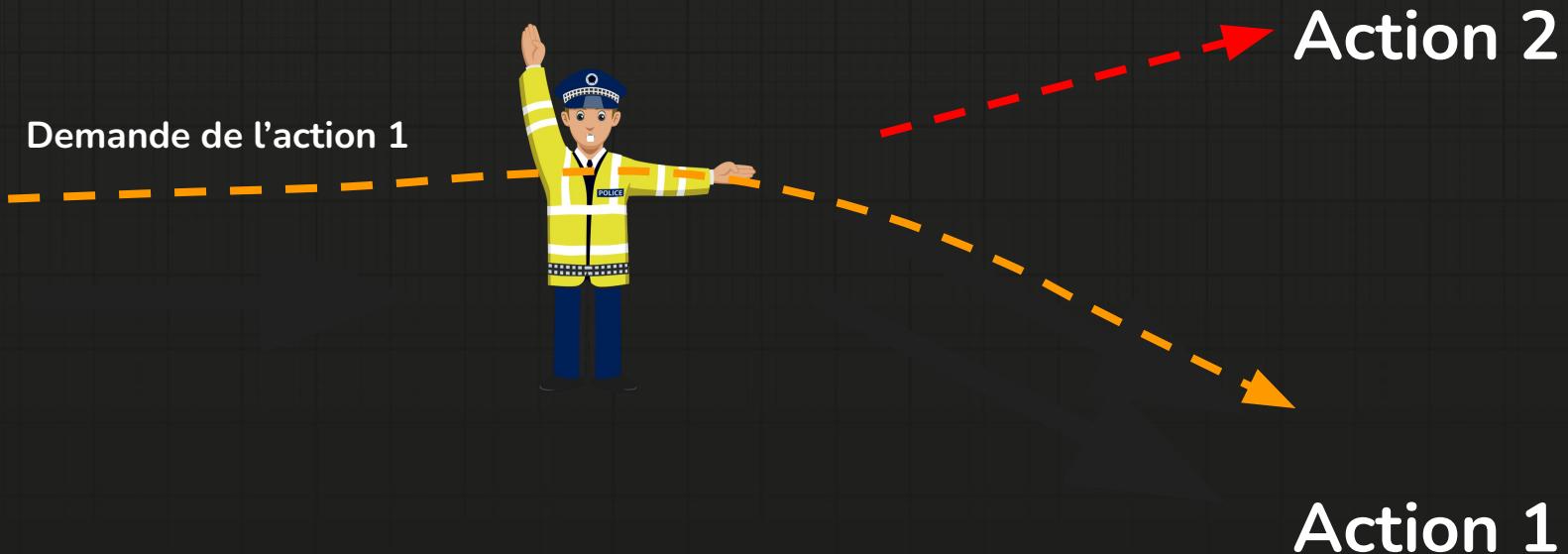
- Les **Models**, qui représentent les données de l'application (souvent les classes représentant les informations : utilisateur, client, produit, catégorie ...)
- Les **Views** qui seront les écrans visibles par l'utilisateur (pages web, sorties console, formulaires d'une application ...)
- Les **Controllers**, chargés de détecter les demandes (souvent émises par l'utilisateur) et d'effectuer l'action approprié à cette demande

C'est ce que l'on appelle le SRP (Single Responsibility Principal) le principe de responsabilité unique. Chaque composant doit avoir une tâche particulière.



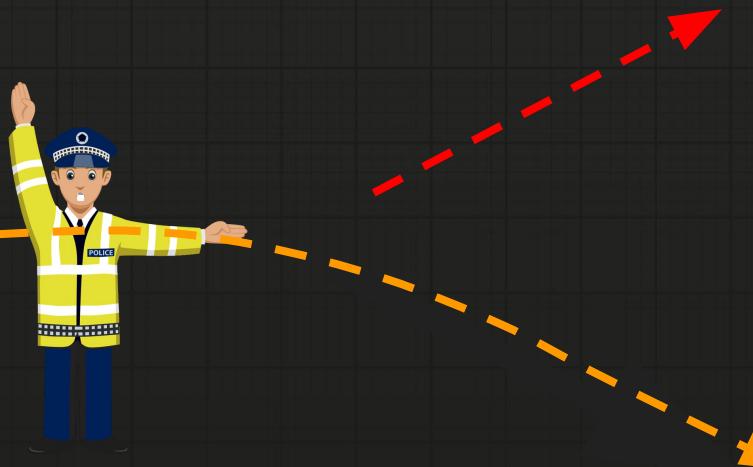
## >> Élément central : le controller

Le but du contrôleur est d'intercepter les demandes d'actions (*une URL appelée, un clic sur un bouton ...*)  
Puis de diriger vers l'action appropriée.



>> Un controller dans un site web

Affichage  
de la page  
d'un article



Je veux voir la page d'accueil  
(monsite.com/accueil)

Affichage de la  
page d'accueil

## >> Un contrôleur dans une application desktop/mobile

Dans le cas d'une application desktop ou mobile, cela pourrait être le fait de cliquer sur un bouton, swiper avec le doigt ...

Supprimer le message

Clic sur le bouton envoyer un message



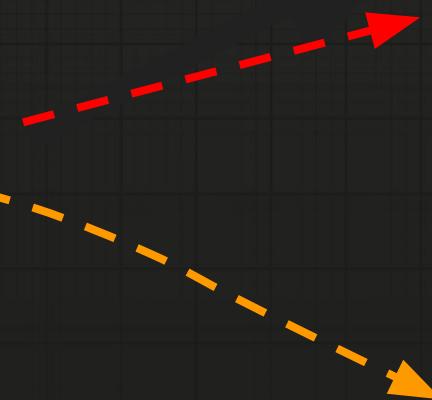
Envoyer un message

## >> Un contrôleur dans un serveur avec une API Rest

Dans notre cas, c'est l'URL qui va être interceptée par le contrôleur,  
Cette URL ne pointe pas sur une page, mais décrira la ressource  
que l'on souhaite obtenir : ici l'utilisateur avec l'id 42

Lister les utilisateurs

mon-serveur/utilisateur/42



Retourner un utilisateur

# >> Le modèle permet de stocker/transférer une information

Le **modèle** permet de structurer l'information. Ce sont principalement des **classes** qui permettent de stocker ou de transférer une donnée. Dans notre cas, le **modèle** est la **classe** permettant de représenter un utilisateur.

mon-serveur/article/42



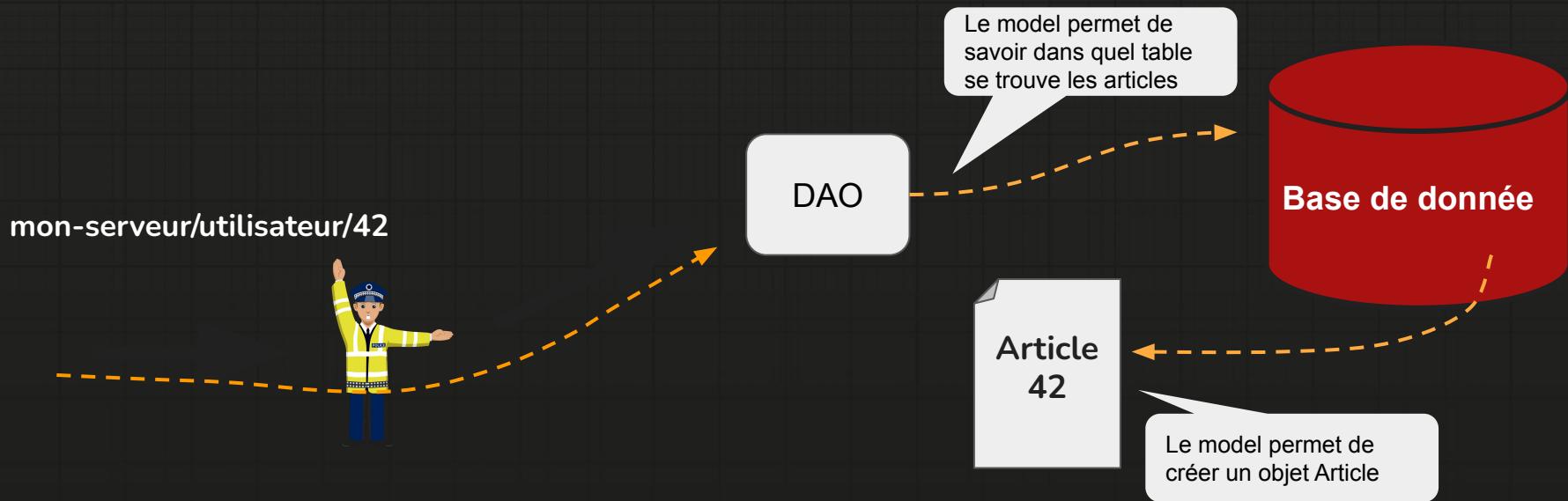
Grâce au **modèle Article**, je vais contacter ma base de données et trouver l'article 42



## >> Digression : Les DAO

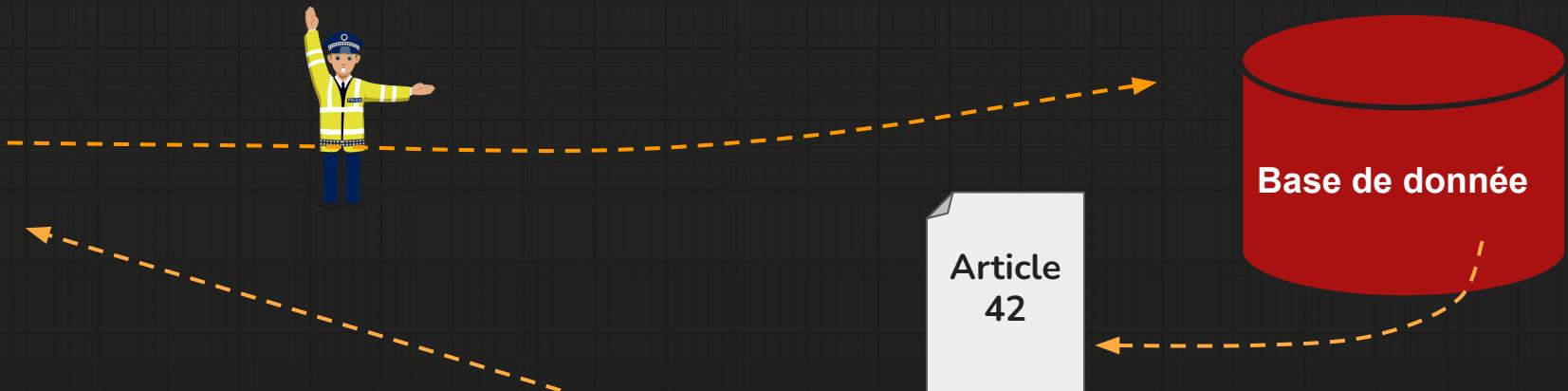
Si dans le design pattern MVC le modèle à effectivement pour rôle l'interaction entre l'application et la base de donnée, dans notre cas nous utiliserons un découpage supplémentaire : les DAO (Data Access Object)

Le modèle ne sert donc (dans notre cas) qu'à structurer les informations



>> La vue permet de créer le retour à l'utilisateur

mon-serveur/utilisateur/42



Grâce à la **View** dédiée à l'affichage d'un utilisateur, et aux informations récupérées en base de donnée je vais pouvoir renvoyer l'information à un utilisateur

## >> Note sur la vue

Dans le cas de notre serveur, nous n'allons pas avoir beaucoup de **vues**.

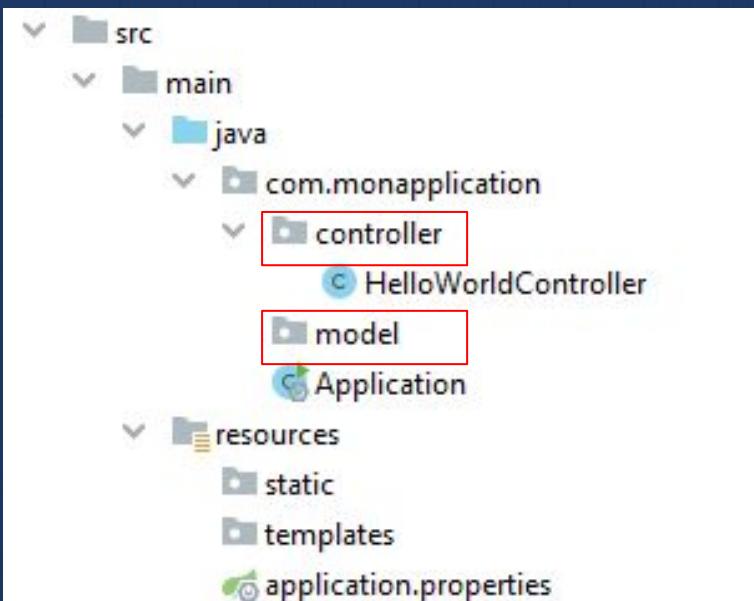
Nous allons principalement renvoyer à l'utilisateur des données au **format JSON**, et donc contrairement au site internet qui génère de l'HTML, la partie vue sera extrêmement simple.

Nous verrons dans un second temps, comment générer des pages HTML complexes, mais les exemples qui suivent se contenteront de retourner la ressource demandé au **format JSON**.

On peut également mettre en place des **vues** permettant de générer d'autre formats que de l'HTML ou du JSON : de l'XML, un document PDF, un tableur ...

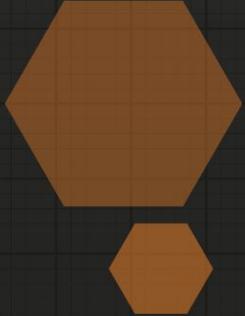


## &gt;&gt; Respecter le design pattern MVC



Comme vu précédemment, dans le cas de notre application offrant une **API Rest**, la **vue** n'est pas présente dans le serveur.

C'est pour cela que l'on aura uniquement un package **controller** et un package **model** au même niveau que la **classe** principale.



# L'inversion de contrôle

*Chapitre : Démarrer un projet spring*

## >> L'inversion de contrôle (IoC)

L'**inversion de contrôle** (inversion of control) est un **design pattern**.

Mise en place par la plupart des frameworks de développement, elle s'appuie sur un concept simple :

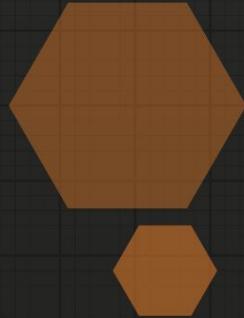
C'est le framework qui gère le flot d'exécution (*Ce n'est plus l'application qui gère les appels au framework, mais ce dernier à l'application*).

Ce **design pattern** semble au premier abord contre intuitif. Le développeur qui n'a pas encore expérimenté un tel mécanisme a plutôt pour habitude de créer un logiciel, et de demander au moment opportun une fonctionnalité du framework.

Dans le cas de l'**inversion de contrôle**, le **framework** met en place toute une série de concepts, contraignant le développeur à respecter certains **design pattern**, une architecture, une convention de nommage ...

Le **design pattern** le plus souvent mis en avant est l'**injection de dépendance**.  
(nous verrons la mise en place dans Spring plus loin dans ce cours)





# Les annotations dans Spring

*Chapitre : Démarrer un projet spring*

# >> Les annotations

Comme nous venons de le voir, **Spring** fonctionne sur le principe de l'**inversion de contrôle**

Notre application devra donc suivre les mécanismes de Spring.  
Et pour cela il existe 2 méthodes (*nous utiliserons la deuxième*).

- Une méthode historique mais peu pratique : la configuration grâce à un fichier XML. Nous ne verrons pas cette méthode car elle rend la configuration de l'application peu lisible
- La méthode plus récente : **annoter les classes, méthodes et variables** pour leur attribuer des fonctionnalités (Contrôleur, modèle, propriété spécifiques ...)

Une ou plusieurs annotations se place juste au dessus de la classe, de la méthode ou de la variable qu'elle cible.  
Certaines annotations peuvent même comporter des paramètres, nommé ou non, et pouvant avoir des valeurs par défaut.

```
@AnnotationPourMaClasse  
public class MaClasse {  
    ...  
}
```

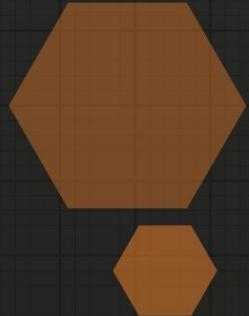
```
@AnnotationPourMaMethode  
@AutreAnnotationPourMaMethode  
public void methode() {  
    ...  
}
```

```
@AnnotationPourMaVariable("valeur paramètre")  
private int propriétéA;
```

```
@Autre(nomParametre = "valeur paramètre")  
private int propriétéB;
```

```
@Autre(  
    nomParametre = "valeur paramètre 1",  
    nomDeuxiemeParametre = "valeur paramètre 2")  
private int propriétéC;
```





# Configurer l'application

*Chapitre : Démarrer un projet spring*

## >> Configurer le fuseau horaire

Afin de ne pas poser de problème lors de l'utilisation des dates, nous allons ajouter une méthode définissant que le fuseau horaire qu'elles utilisent soient UTC (universal time coordinated)

En effet au démarrage de l'application celui-ci prendra l'heure du système, (CET en france) si l'un des composant de notre architecture finale (serveurs, base de donnée ...) ne se situe pas sur le même fuseau horaire, alors il risque de provoquer un décalage dans les dates

*fichier principal*

```
import jakarta.annotation.PostConstruct;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import java.util.TimeZone;

@SpringBootApplication
public class MonApplication {

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

    @PostConstruct
    public void init(){
        TimeZone.setDefault(TimeZone.getTimeZone("UTC"));
    }
}
```

L'annotation **@PostConstruct** permet de lancer la méthode juste après que le composant Spring est été initialisé

# Créer des contrôleurs

Dans ce chapitre :

- Qu'est ce qu'un contrôleur REST
- Les méthodes des requêtes (GET, POST, PUT, DELETE ...)
- Effectuer un test unitaire sur un contrôleur

## >> Le controller de Web service

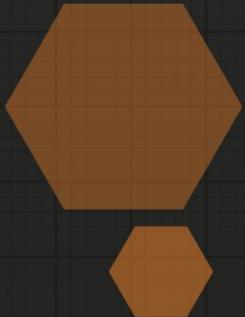
Grâce à l'**annotation** `@RestController` on déclare notre **classe** en tant que **contrôleur de web service**.

La différence entre un `@Controller` et un `@RestController` est que ce dernier n'utilise pas de système de **template**. Il ne renvoie qu'une simple **chaîne de texte**.

*controller*

```
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class HelloController {  
  
}
```





# Le mapping URL / Méthode

*Chapitre : Créer des contrôleurs*

## >> Le mapping d'URL

Grâce à l'**annotation** `@GetMapping` placée au dessus d'une méthode de notre contrôleur, nous allons pouvoir faire la **liaison** entre une **méthode** (*ici la méthode direBonjour*) et une **URL** (*ici l'url "/hello"*)

Une **annotation** peut prendre un certain nombre de **paramètre**, ici `@GetMapping` prend une chaîne de texte en tant que paramètre (*représentant l'URL associée*)

*controller*

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String direBonjour() {
        return "hello web service";
    }
}
```

Faites le test, lancez l'application, puis tapez l'adresse [localhost:8080/hello](http://localhost:8080/hello) dans un navigateur.

(attention si vous avez changez de port précédemment)

Vous devriez voir s'afficher dans le navigateur : hello web service



## >> PathVariable (1/2)

On peut gérer les **mapping** afin qu'une partie de l'url soit évaluée comme une **variable**. Dans cet exemple, le **paramètre** id vaudra le contenu situé après `/test-parametre/` dans l'**url** (encadré par des accolades)

ex : si l'on accède à l'url `localhost:8080/test-parametre/42` : parametre vaudra "42".

*controller*

```
@GetMapping("/test-parametre/{monParametre}")
public String testParametreUrl (@PathVariable int monParametre) {
    return "Le parametre de cette url est : " + monParametre;
}
```



## >> PathVariable (2/2)

De la même façons, on peut utiliser plusieurs **variables** dans une **URL**, à partir du moment où il y a autant de **paramètre** avec l'annotation **@PathVariable** que de **paramètre** dans l'**URL** :

*controller*

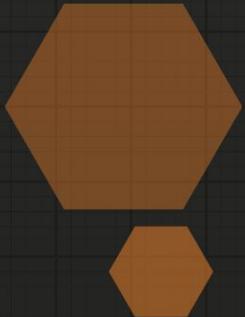
```
@GetMapping("/test-parametres/{monPremierParametre}/{monDeuxiemeParametre}")
public String testParametresUrl (
    @PathVariable int monPremierParametre,
    @PathVariable int monDeuxiemeParametre) {
    return "Le premier parametre de cette url est : " + monPremierParametre +
        " ,le deuxième parametre de cette url est : " + monDeuxiemeParametre;
}
```



>> Utiliser le debugger

TODO





# Les différentes méthodes des requêtes

*Chapitre : Créer des contrôleurs*

## >> Mapping d'URL et méthode de requête

C'est grâce à l'**annotation** `@GetMapping("/hello")` que l'on peut intercepter les requêtes ayant l'URL `"/hello"` et ayant la méthode **GET**.

Mais on peut également utiliser les annotations : `@PostMapping`, `@PutMapping`, `@DeleteMapping` et `@PatchMapping` permettant respectivement d'intercepter les méthodes Post, Put, Delete et Patch.



## >> Les méthodes des requêtes (GET, POST, DELETE, PUT ...)

Lorsque l'on envoie une **requête** via une **URL** (ex : `localhost:8080/utilisateur/42`) celle-ci possède une **méthode**.  
*(note : aucun rapport avec les méthodes des classes)*

La seule **méthode** disponible lorsque l'on utilise la **barre d'adresse d'un navigateur** est la méthode **GET**.  
Mais il en existe d'autres : **POST, DELETE, PUT, PATCH** ...

Ces **méthodes** permettent de donner un sens (*une sémantique*) à la **requête**. On peut utiliser une même URL pour notre requête (ex : `localhost:8080/utilisateur/42`) mais selon la **méthode** de la **requête**, l'action sera complètement différente :

Si ma **requête** possède l'**URL** : **localhost:8080/utilisateur/42** et la **méthode GET**, alors il serait logique que j'obtienne des **informations** sur l'utilisateur 42, mais si la **méthode** de l'**URL** est **DELETE**, alors il serait logique que l'action associée soit de **supprimer** l'utilisateur 42.



## >> Les méthodes d'URL dans une API REST (1/3)

**GET : Récupérer des données depuis le serveur (lecture).**

Exemple : GET /users (récupérer la liste des utilisateurs), GET /users/123 (récupérer l'utilisateur avec l'ID 123).

Sécurité : Cette méthode est idempotente, ce qui signifie que plusieurs appels à la même URL avec GET n'affecteront pas les données.

**DELETE : Supprimer une ressource du serveur.**

Exemple : DELETE /users/123 (supprimer l'utilisateur avec l'ID 123).

Sécurité : Idempotent, bien que la suppression soit irréversible, appeler DELETE plusieurs fois sur la même ressource n'aura pas d'effet après la première suppression.



## >> Les méthodes d'URL dans une API REST (2/3)

### **POST : Créer une nouvelle ressource sur le serveur.**

Exemple : POST /users (créer un nouvel utilisateur avec les données envoyées dans le corps de la requête).

Sécurité : Non idempotent, chaque appel peut créer une nouvelle ressource.

### **PUT : Mettre à jour une ressource existante, généralement pour remplacer complètement l'objet.**

Exemple : PUT /users/123 (mettre à jour l'utilisateur avec l'ID 123 en envoyant les nouvelles données).

Sécurité : Idempotent, une ressource mise à jour par un PUT avec les mêmes données restera la même.

### **PATCH : Mettre à jour partiellement une ressource.**

*(Contrairement à PUT, on modifie seulement certains champs de la ressource)*

Exemple : PATCH /users/123 (mettre à jour certains attributs de l'utilisateur avec l'ID 123).

Sécurité : Idempotent, mais dépend des données envoyées.



## >> Les méthodes d'URL dans une API REST (3/3)

**HEAD** : Récupérer les en-têtes de la réponse sans le corps de la réponse (*similaire à GET mais sans contenu*).

Exemple : HEAD /users/123 (vérifier l'existence de l'utilisateur 123 sans récupérer les données).

Sécurité : Idempotent.

**OPTIONS** : Obtenir les méthodes HTTP disponibles pour une ressource. Cela permet de savoir quelles actions peuvent être effectuées sur une ressource.

Exemple : OPTIONS /users/123 (vérifier quelles méthodes HTTP sont autorisées pour cette ressource).

Sécurité : Idempotent.



## >> Note pour les développeurs PHP

Si vous avez déjà construit une application avec **PHP** alors vous avez déjà certainement déjà expérimenté les formulaires avec les méthodes **POST** ou **GET**.

```
<form method="POST">          <form method="GET">  
    ...  
</form>                      ...  
                                </form>
```

C'est là exactement ce que l'on est en train de faire : **changer la méthode de la requête**.

Vous devez également savoir que PHP ajoute les champs du formulaire dans les paramètres de l'URL uniquement si on utilise la méthode GET. Et que l'on peut les retrouver dans le tableau `$_GET` (ou `$_POST` si la méthode du formulaire est POST)

**C'est un comportement propre à PHP**, la décision de traiter différemment les méthodes GET et POST vient du fonctionnement de PHP lui-même. Mais dans notre cas une méthode GET et POST (ou PUT ou DELETE) sont strictement identique. C'est à nous de décider quel traitement leur associer



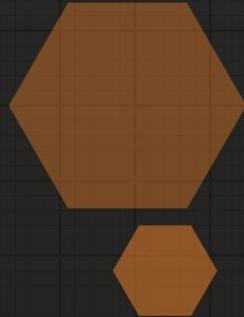
## >> Exemple avec DeleteMapping

Toutes les annotations de mapping fonctionnent de la même manière, elles agissent uniquement comme un filtre sur l'interception des requêtes. Ce sont les instructions qui décideront de l'action qu'elles déclencheront (*rien ne nous empêche de supprimer un utilisateur via un GetMapping et d'afficher ces informations via un DeleteMapping mais ça n'aurait aucun sens*)

Ajoutez le mapping suivant (*Vous ne pouvez actuellement pas le tester car un navigateur ne peut envoyer que des requêtes avec une méthode GET*)

```
@DeleteMapping("/produit/{id}")
public String supprimerProduit (@PathVariable int id) {
    return "Supprime le produit avec l'id : " + id;
}
```





# Tester une route avec thunder client

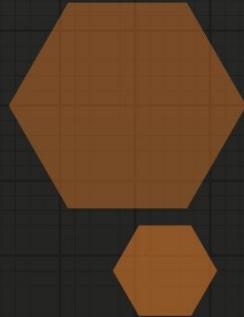


*Chapitre : Créer des contrôleurs*

>> TODO

TODO





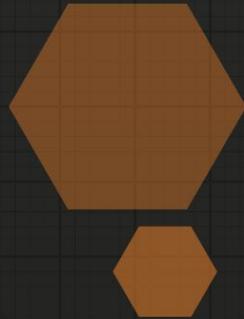
## Tester une route avec JET client (IntelliJ Ultimate)

*Chapitre : Créer des contrôleurs*

# >> Installer Jet Client

TODO

The screenshot shows the IntelliJ IDEA settings interface with the 'Marketplace' tab selected. A search bar at the top right contains the query 'jet'. Below it, a list of search results is displayed under the heading 'Search Results (452)'. The results are sorted by relevance. The first result is 'JetBrains Academy' by JetBrains s.r.o., which has 4M downloads and a rating of 4.13. The second result is 'JetBrains IDE Services' by JetBrains s.r.o., with 3.9M downloads and a rating of 4.54. The third result is 'JetForcer | The Smartest Force.com IDE' by JetForcer, with 333K downloads and a rating of 4.15. The fourth result is 'JetClient - The Ultimate REST Client' by Anton Shuvayev, which is highlighted with a blue background and a checked 'Install' button. This result has 85K downloads and a rating of 4.54, released on 2024.3.26-243. The fifth result is 'JetBrains AI Assistant' by JetBrains s.r.o., with 19.1M downloads and a rating of 1.96. The sixth result is 'IDEA Jetty Runner' by Guy Keller, with 202.9K downloads and a rating of 4.52. The seventh result is 'Jetpack Compose' by JetBrains s.r.o., with 90.7K downloads and a rating of 4.65. On the right side of the screen, a detailed view of the 'JetClient - The Ultimate REST Client' plugin is shown. It includes tabs for Overview, What's New, Reviews, and Additional Info. The Overview tab displays a screenshot of the plugin's interface, which shows a 'POST /user' request being configured with various parameters like 'username', 'password', and 'email'. Buttons for 'OK' and 'Cancel' are visible at the bottom right of this panel.



# Tester une route avec postman



*Chapitre : Créer des contrôleurs*

## >> Introduction à Postman

Sur le même principe, Postman est un logiciel qui nous servira d'application FRONT pour pouvoir tester notre serveur.

Il est utile aux développeurs back-end afin qu'ils puissent tester leur API lorsque l'application cliente n'a pas encore été créée (*ce qui est actuellement notre cas*)

Créez un compte et téléchargez le logiciel postman sur le site officiel : <https://www.postman.com/>

Une fois installé, le logiciel vous demandera de créer une collection(*collection*) ou vous pourrez stocker toutes vos requêtes.

L'avantage par rapport à thunder client, est qu'il permet de sauvegarder ses requêtes sur son compte Postman (donc entre poste de travail) et de les partager entre collaborateur, on le retrouve donc plus en entreprise



The screenshot shows the Postman application interface with several callout boxes containing French text:

- Méthode de la requête (Method of the request)
- URL de la requête (URL of the request)
- Envoyer la requête (Send the request)
- Réponse du serveur (Server response)
- Code de la réponse (404 Not found, 500 internal error ...)

The Postman interface includes:

- File Edit View Help menu
- New Request button
- Method dropdown (GET selected) and URL input field (localhost:8080/utilisateur/42)
- Params, Authorization, Headers (7), Body, Pre-request Script, Tests, Settings tabs
- Query Params table with columns: KEY, VALUE, DESCRIPTION, Bulk Edit
- Body tab selected, showing response content:

```
1 Vous avez demandé l'affichage de l'utilisateur ayant l'id : 42
```
- Headers (5), Test Results, Pretty, Raw, Preview, Visualize, Text dropdown, and a red icon
- Status bar: Status: 200 OK, Time: 17 ms, Size: 233 B, Save Response dropdown
- Bottom navigation: Find and Replace, Console, Bootcamp, Runner, Trash, and a question mark icon
- Right sidebar: Upgrade, Invite, Settings, Save, Send (blue button), Cookies, and a file icon

Changez ici la méthode de la requête

demo / Neuest

DELETE    localhost:8080/utilisateur/42

Params    Authorization    Headers (7)    Body    Pre-request Script

Query Params

KEY	VALUE
-----	-------

Body    Cookies    Headers (5)    Test Results

Pretty    Raw    Preview    Visualize    Text

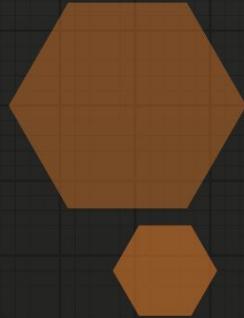
1    Vous voulez supprimer l'utilisateur avec l'id : 42

Afin de pouvoir être intercepté par l'annotation `@DeleteMapping`, vous devez changer la méthode dans la liste déroulante pour **DELETE**

Vous pouvez constater que le retour est bien celui de la méthode ayant l'annotation `@DeleteMapping` même si nous n'avons pas changé l'URL de la requête.

Les **méthodes des requêtes** ne sont que des filtres afin de donner du sens aux URL que nous allons contacter.

Le retour du serveur



# Qu'est ce qu'une API REST ?

*(REpresentational State Transfer)*

*Chapitre : Créer des contrôleurs*

## >> Qu'est ce qu'une API ?

Une API est un ensemble de définitions et de protocoles qui facilite la création et l'intégration de logiciels d'applications.

Elle est parfois considérée comme un contrat entre un fournisseur d'informations et un utilisateur d'informations, qui permet de définir le contenu demandé au consommateur (*l'appel*) et le contenu demandé au producteur (*la réponse*).

Par exemple, l'API conçue pour un service de météo peut demander à l'utilisateur de fournir un code postal et au producteur de renvoyer une réponse en deux parties : la première concernant la température maximale et la seconde la température minimale.



# >> Qu'est ce que REST ?

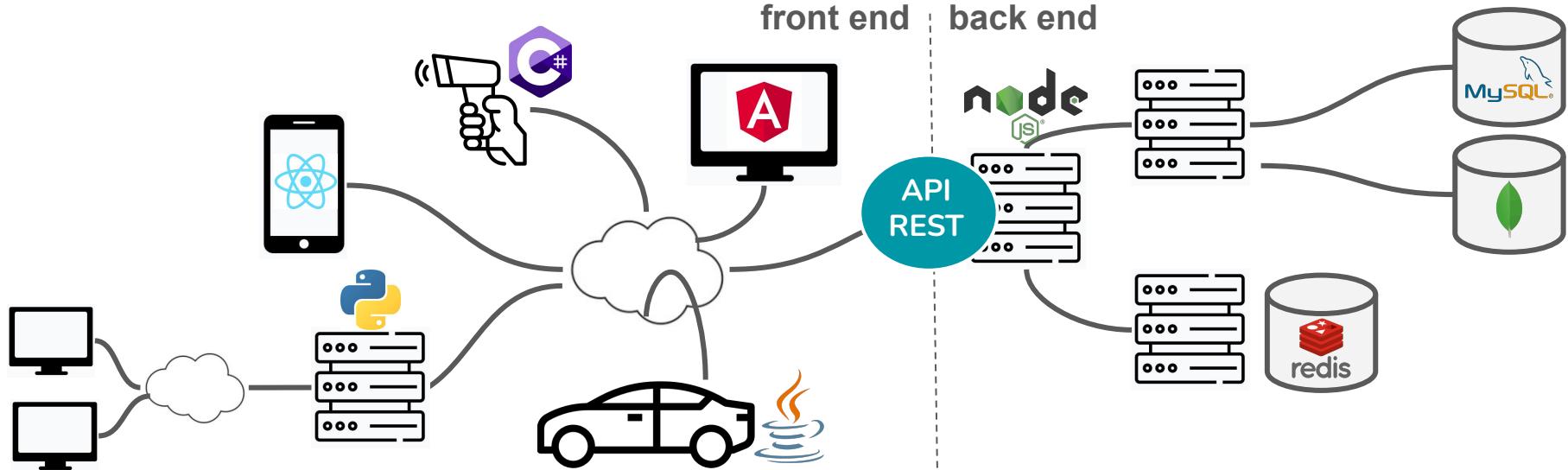
REST est un ensemble de contraintes architecturales. Il ne s'agit ni d'un protocole, ni d'une norme.

Lorsqu'un client émet une requête par le biais d'une API suivant les contraintes de REST , celle-ci transfère une représentation de l'état de la ressource au demandeur ou point de terminaison (ENDPOINT).

Ex : "<https://mon-site/api/utilisateurs>"

Cette information, ou représentation, est fournie via le protocole HTTP généralement via le format JSON (JavaScript Object Notation)

Mais il pourrait être HTML, XLT, Python, PHP ou texte brut.



# >> Quelles sont les contraintes de REST ?

- Une **architecture client-serveur** constituée de **clients**, de **serveurs** et de **ressources**, avec des **requêtes** gérées via **HTTP**
- Des communications **client-serveur stateless**, c'est-à-dire que les informations du **client** ne sont jamais stockées entre les **requêtes GET**, qui doivent être traitées séparément, de manière totalement indépendante
- La possibilité de mettre en **cache** des données afin de rationaliser les interactions **client-serveur**
- Une **interface** uniforme entre les **composants** qui permet un transfert standardisé des informations. Cela implique que :
  - les **ressources** demandées soient identifiables et séparées des représentations envoyées au **client** ;
  - les **ressources** puissent être manipulées par le **client** au moyen de la représentation reçue, qui contient suffisamment d'informations ;
  - les messages autodescriptifs renvoyés au **client** contiennent assez de détails pour décrire la manière dont celui-ci doit traiter les informations ;
  - l'**API** possède un hypertexte/hypermédia, qui permet au **client** d'utiliser des hyperliens pour connaître toutes les autres actions disponibles après avoir accédé à une **ressource**.
- Un **système à couches**, invisible pour le **client**, qui permet de hiérarchiser les différents types de **serveurs** (*pour la sécurité, l'équilibrage de charge, etc.*) impliqués dans la récupération des informations demandées
- Du code à la demande (*facultatif*), c'est-à-dire la possibilité d'envoyer du code exécutable depuis le **serveur** vers le **client** (*lorsqu'il le demande*) afin d'étendre les fonctionnalités d'un **client**



## >> Qu'est ce qu'une API RESTful ?

Une API RESTful est une API qui suit les contraintes imposées par REST. On ajoute le suffix "ful" afin d'obtenir le mot "restful" signifiant "reposant". (*Cela n'a que pour but la promotion de la technologie*)

Elle permet à des systèmes extérieurs (des clients) de bénéficier d'un point d'entrée unique dans un langage compréhensible (JSON), quelque-soit les langages utilisés ou la complexité de l'architecture back ou front



## &gt;&gt; Créer un contrôleur CRUD pour les produits

Nous pouvons donc organiser comme bon nous semble les **contrôleurs** et les **méthodes** qu'ils contiendront.

Mais nous suivrons les principes d'une architecture REST :

Un **contrôleur** contiendra toutes les **méthodes** destinées à manipuler une **entité** (*on appellera entité, les classes de notre model : Utilisateur, Article, Commande, Commentaire, Voiture ...*)

Pour l'entité Produit nous aurons donc un contrôleur ProduitController avec au moins 5 méthodes :

- **listeProduits** avec un `@GetMapping ("/produit/liste")` : *Retourne tous les produits*
- **obtenirProduit** avec un `@GetMapping ("/produit/{id}")` : *Permet d'obtenir un produit selon son ID*
- **supprimerProduit** avec `@DeleteMapping ("/produit/{id}")` : *Supprime un produit selon son ID*
- **ajouterProduit** avec un `@PostMapping ("/produit")` : *Ajoute un produit*
- **modifierProduit** avec un `@PutMapping ("/produit/{id}")` : *Modifie le produit par son ID*



# >> Créer un contrôleur CRUD pour les produits

*ProduitController (nouveau fichier)*

```
@RestController
public class ProduitController {

    @GetMapping("/produit/liste")
    public String listeProduits (){
        return "Afficher la liste des produits";
    }

    @GetMapping("/produit/{id}")
    public String obtenirProduit (@PathVariable int id){
        return "Afficher le produit avec l'id : " + id;
    }

    @DeleteMapping("/produit/{id}")
    public String supprimerProduit (@PathVariable int id){
        return "Supprime le produit avec l'id : " + id;
    }

    @PostMapping("/produit")
    public String ajouterProduit (){
        return "Ajoute un produit";
    }

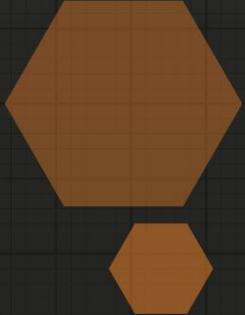
    @PutMapping("/produit/{id}")
    public String modifierProduit (@PathVariable int id){
        return "Modifie le produit avec l'id : " + id;
    }
}
```

Attention la route  
"/produit/liste"  
doit être placée avant  
"/produit/{id}"



# L'injection de dépendances

Dans ce chapitre :



A quoi sert l'injection de dépendances ?

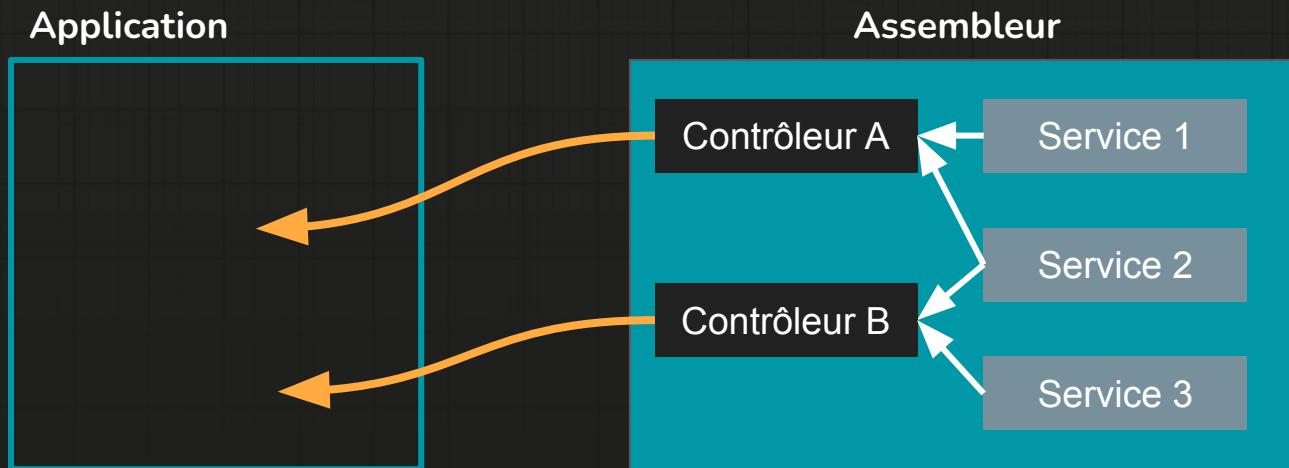
*Chapitre : Créer des contrôleurs*

# >> L'injection de dépendances : principe

L'**injection de dépendance** se base sur une classe dont le but au moment du démarrage de l'application est de créer l'intégralité des objets susceptibles d'être utilisés par d'autres classes.

On appelle cette classe **l'assembleur**.

Pendant l'exécution du programme, il suffira de récupérer ces classes dans **l'assembleur**. Celles-ci auront déjà toutes les dépendances dont elles ont besoin.



## >> Comprendre l'intérêt de ce Design pattern

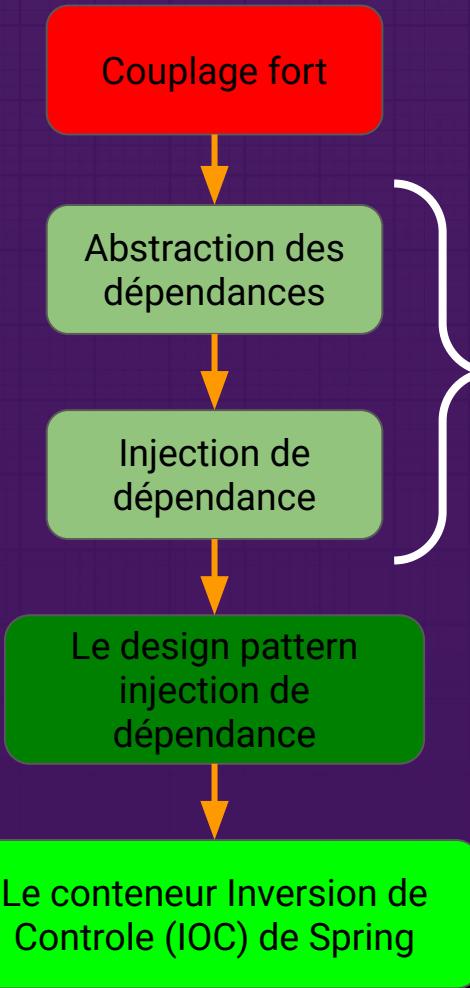
Il n'est pas nécessaire de comprendre tout ce qui se passe derrière le concept que nous allons voir pour pouvoir l'utiliser.

Mais cela permet de mieux comprendre pourquoi il a été mis en place

Pour cela nous allons partir d'un code difficilement maintenable, expliquer pourquoi il pose problème et comment l'améliorer

Nous allons devoir expliquer des concepts comme le couplage entre les classes, la notions de dépendance etc ...

maintenabilité du code



## &gt;&gt; Rappel rapide sur les liaisons des Diagrammes de classes UML



Lien de dépendance structurelle  
(la flèche de navigabilité est optionnelle)

```

public class ClasseA {
    private ClasseB instanceClasseB;
}
  
```



Lien de dépendance temporaire

```

public class ClasseA {

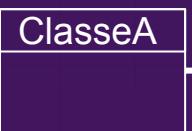
    public ClasseA() {
        ClasseB maVariable = new
        ClasseB();
    }
}
  
```



Héritage

```

public class ClasseA extends ClasseB {
}
  
```



Implémentation

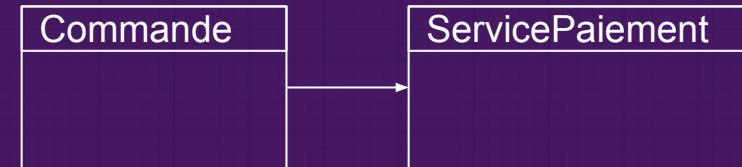
```

public class ClasseA implements InterfaceB
{
}
  
```

## >> Comprendre l'intérêt de ce Design pattern : *Qu'est ce qu'un couplage fort ? (1/2)*

On parle de **couplage fort** quand une classe A à besoin d'une classe B pour fonctionner

La classe A est dépendante de la classe B



La classe B est une **dépendance** de la classe A

```
public class Commande {  
  
    private ServicePaiement servicePaiement;  
  
    Commande() {  
        this.servicePaiement = new ServicePaiement();  
    }  
}
```

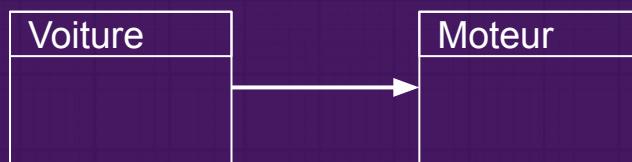
```
public class Commande {  
  
    private ServicePaiement servicePaiement;  
  
    Commande(ServicePaiement servicePaiement) {  
        this.servicePaiement = servicePaiement;  
    }  
}
```

OU

Qu'elle soit passée par constructeur, accesseur ou initialisée dans la classe, on est toujours dans le cas d'une dépendance

## >> Comprendre l'intérêt de ce Design pattern : Qu'est ce qu'un couplage fort ? (2/2)

Dans une application il est très fréquent qu'une **classe** ait besoin d'une autre **classe** pour fonctionner  
Ici la classe Moteur est une **dépendance** de la classe Voiture.



*dépendance par association*



Note : les classes prises en exemple ici sont des **models**, le design pattern **factory** est donc plus approprié que **injection de dépendance**

Cette forte **dépendance** oblige à exprimer dans le code de la classe Voiture que l'on a strictement besoin d'un moteur,

Il est également possible que la classe Voiture ait besoin d'exprimer avoir strictement besoin de la classe Cylindre.

```

public class Voiture {
    private Moteur moteur;
    Voiture (Moteur moteur) {
        this.moteur = moteur;
    }
}
  
```

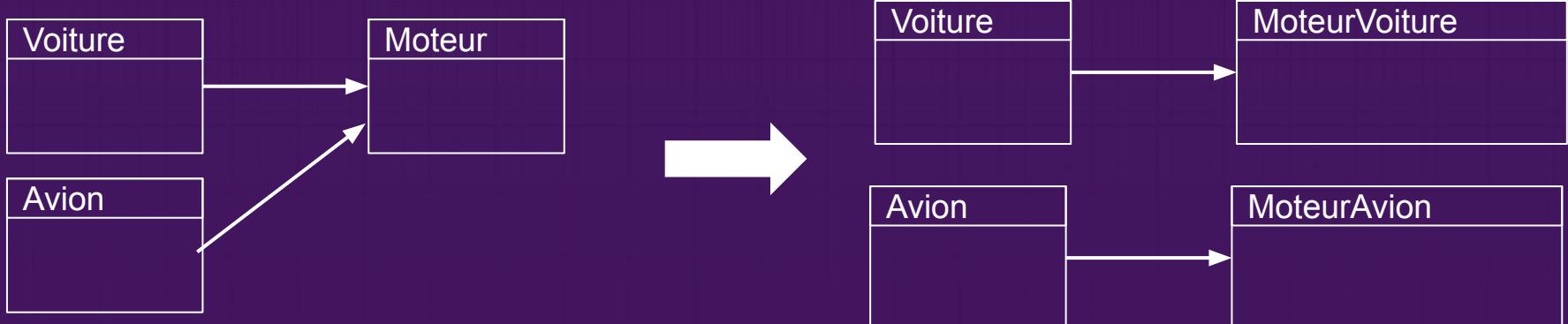


>> Le couplage fort : *Problématique*

Prenons par exemple le changement du type de la propriété “moteur” de la classe Voiture.

Cette modification intervient par exemple suite à la création d'un Moteur spécifique aux voitures et un autre spécifique aux avions.

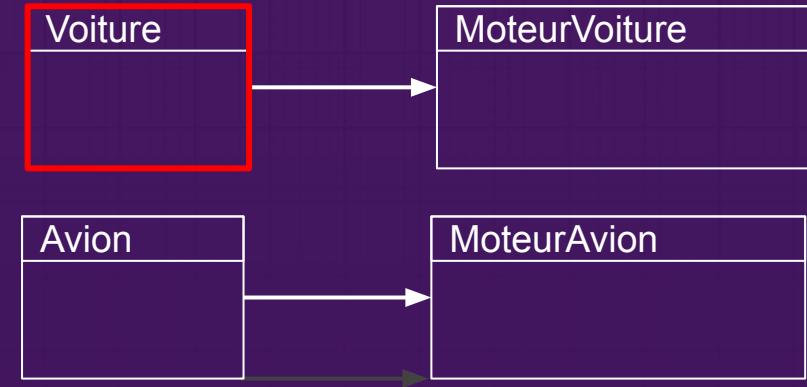
La classe Voiture n'a pas de nouvelle fonctionnalités (*donc théoriquement pas de changement dans son code, mais elle n'est plus dépendante de la même classe ...*)



Ce changement implique donc d'effectuer des modifications dans la classe Voiture (et dans la classe Avion) malgré le fait qu'elle n'est pas eu de réelles modifications.

Il en sera de même pour l'intégralité des classes qui étaient dépendantes de la classes Moteur (*par association ou simplement par ce qu'elles auraient déclarée une variable de type Moteur ou une méthode utilisant un paramètre de type Moteur*)

```
public class Voiture {  
  
    private MoteurVoiture moteur;  
  
    Voiture(MoteurVoiture moteur) {  
        this.moteur = moteur;  
    }  
}
```



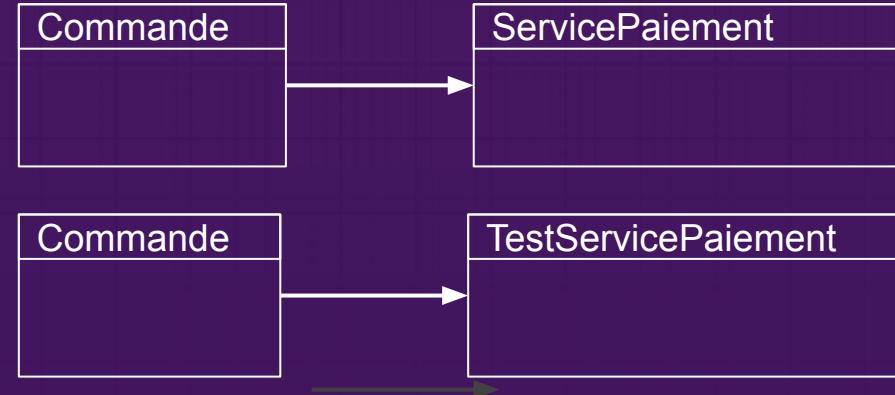
Plus généralement, il y a un cas assez fréquent qui nous oblige à être plus souple : Tester l'application (dans le cadre de tests unitaires ou de fonctionnalités)

Reprendons l'exemple de la *commande* et du *service de paiement* :

Ajoutons une **classe** *TestServicePaiement* qui permettrait de ne pas réaliser de vrais opérations de paiement mais juste effectuer un test. (*pendant la phase de développement par exemple*)

Le **couplage fort** nous obligerait à devoir modifier toutes les **références** à *ServicePaiement* et les remplacer par *TestServicePaiement* à CHAQUE FOIS que l'on veut tester l'application : C'est juste ingérable.

Ajouter des **blocs conditionnels** n'est pas non plus une solution viable sur un vrai projet



# >> Première étape du couplages faible :

## Rendre les dépendances abstraites

Le problème peut être résolu en réalisant une **interface** (ou une classe abstraite) qui contient toutes les **méthodes** provenant de la classe *Moteur*

Dans cet exemple, uniquement la **méthode allumage()**

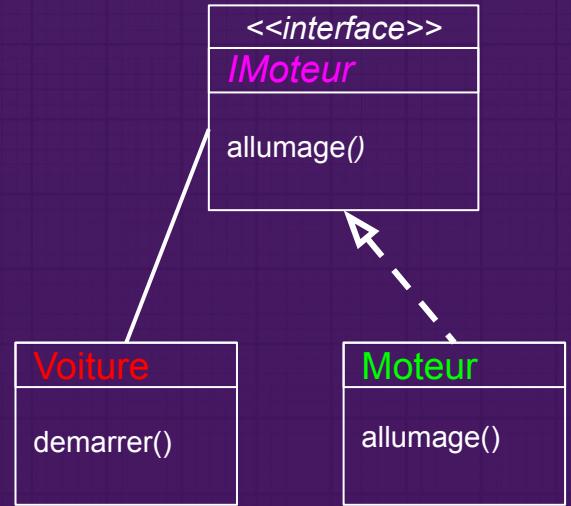
```
public class Voiture {
    private IMoteur moteur;

    Voiture(IMoteur moteur) {
        this.moteur = moteur;
    }

    public void demarrer() {
        this.moteur.allumage();
    }
}
```

```
public interface IMoteur {
    public void allumage();
}
```

```
public class Moteur implements IMoteur {
    public void allumage() {
        System.out.println("VROUM");
    }
}
```



On peut alors ajouter une **dépendance** compatible avec notre **classe Voiture** (elle doit implémenter **IMoteur**) sans modifier la **classe Voiture**.

Il suffira de passer en paramètre à notre Voiture une instance de *TestMoteur* plutôt que de *Moteur*, le reste du code restera inchangé

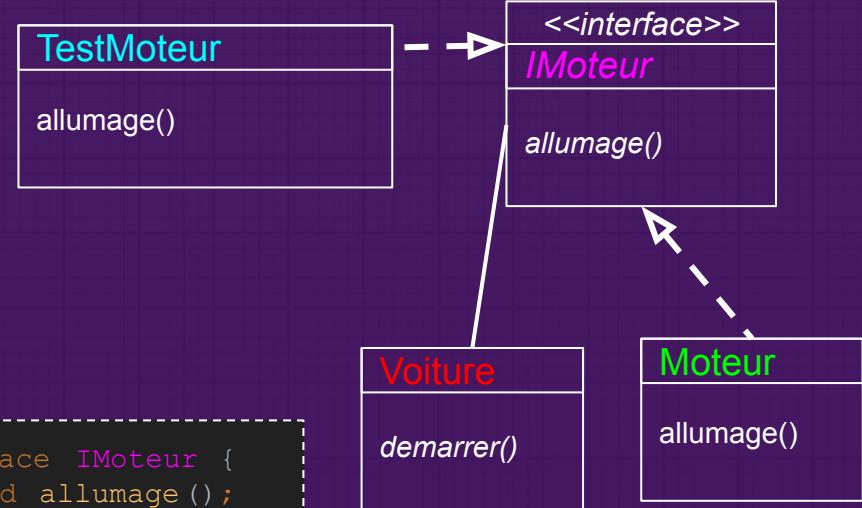
```
public class Voiture {
    private IMoteur moteur;

    Voiture(IMoteur moteur) {
        this.moteur = moteur;
    }

    public void demarrer() {
        this.moteur.allumage();
    }
}
```

```
public interface IMoteur {
    public void allumage();
}
```

```
public class TestMoteur implements IMoteur {
    public void allumage() {
        System.out.println("JUSTE UN TEST");
    }
}
```



Ajoutons 2 classes à notre exemple :

- La classe **Application** qui est l'application principale
- La classe **TestApplication** qui vérifie si tout se passe bien

Dans les 2 cas **Application** et **TestApplication** utilisent une instance de **Voiture**.

Mais **Application** affecte une instance de **Moteur**  
alors que **TestApplication** affecte une instance de **TestMoteur**



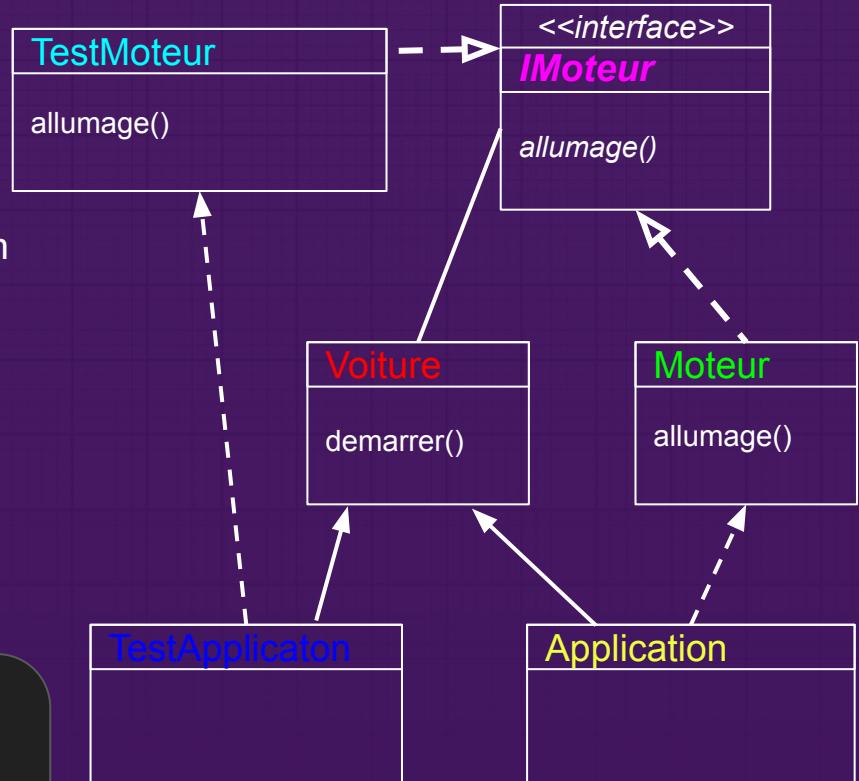
Implémentation d'une interface



Dépendance structurelle : La classe possède une propriétés du type de la classe liée



Dépendance temporaire : La classe utilise la classe liée mais ce n'est pas une de ses propriétés



Il n'y a que dans le code de la classe **Application** et **TestApplication** où l'on affectera une instance de la classe **Moteur** ou **TestMoteur**, le reste du code reste inchangé

```
public class Application {  
  
    Voiture voiture;  
  
    public Application() {  
        IMoteur moteur = new Moteur();  
        voiture = new Voiture();  
        voiture.setMoteur(moteur) ;  
        voiture.miseEnRoute() ;  
    }  
}
```

```
public class TestApplication {  
  
    Voiture voiture;  
  
    public Application() {  
        IMoteur moteur = new TestMoteur();  
        voiture = new Voiture();  
        voiture.setMoteur(moteur) ;  
        voiture.miseEnRoute() ;  
    }  
}
```

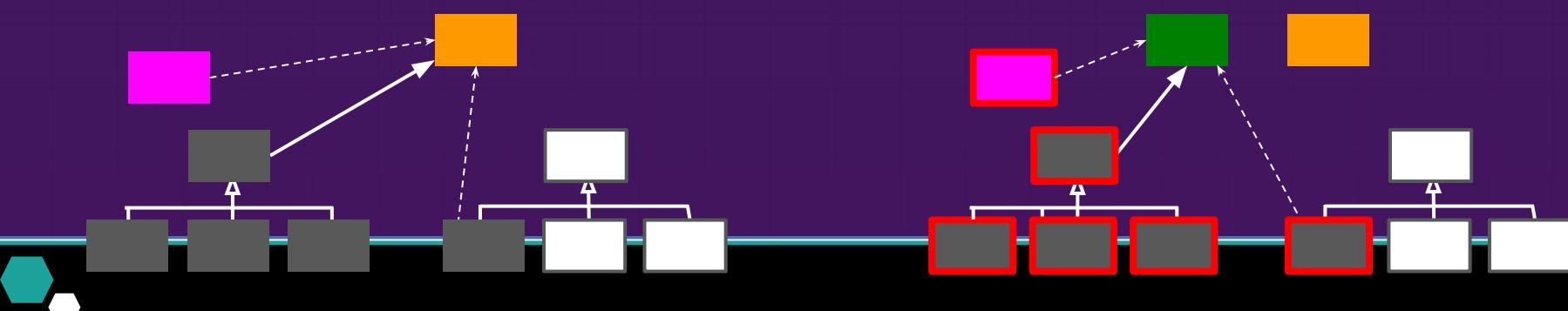


## &gt;&gt; Impact lors d'un couplage fort

Sans contexte particulier, nous allons observer ce que cela à pour conséquence de modifier une classe lorsque des **couplages forts** l'implique :

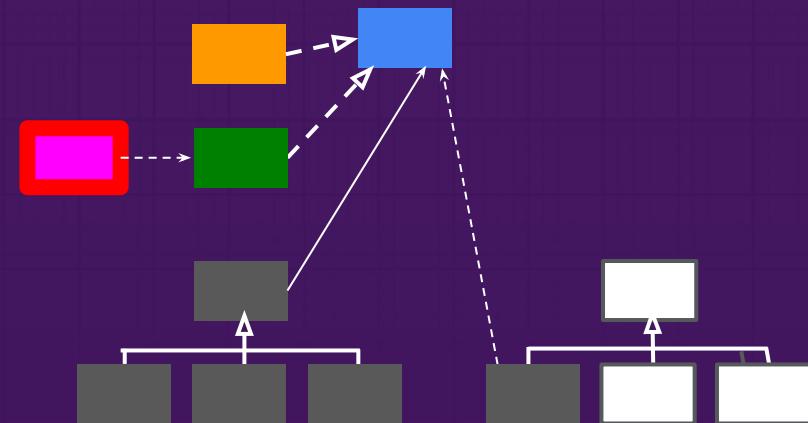
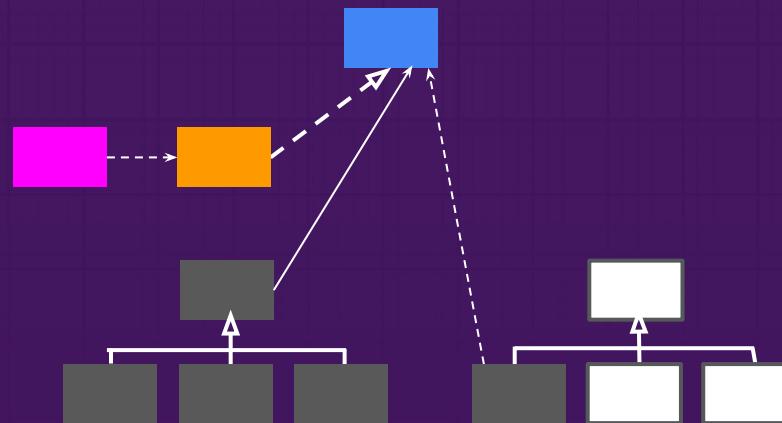
- En orange, la **classe d'origine** (*Moteur*), et en vert, la **nouvelle classe** (*TestMoteur*).
- Les rectangles **gris** représentent les **classes** qui sont **dépendantes** ou en **association** avec la **classe d'origine**. (*Un rectangle gris pourrait être Voiture*)
- Les rectangles **roses** représentent les **classes** qui sont également **dépendantes** ou en **association**, car elles utilisent le constructeur de la **classe d'origine** (*Un rectangle rose pourrait être Application*)  
(Note : Les classes grises ne sont dépendantes qu'à cause du type d'un paramètre d'une méthode ou d'une variable du type de la classe modifiée).

Dans le schéma à droite vous pouvez voir, entourée de **rouge**, les **classes** qui ont subi des **modifications**.



## &gt;&gt; Impact lors d'une abstraction des dépendances

Dans le cas où les **classes** utilisent un **couplage faible**, seules les **classes** qui utilisent explicitement des **instances** de la **classe d'origine** devront subir des **modifications**. Puisque **l'interface** n'a pas été modifiée, les **classes** qui n'ont que des **dépendances** que sur cette interface ne sont pas altérées

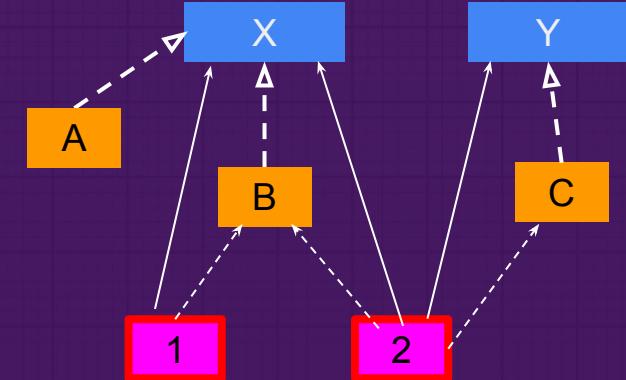


## >> Deuxième étape pour mettre en place un couplage faible : L'injection de dépendance

Nous venons de voir que mettre en place des **dépendances abstraites** améliore la **maintenabilité** de notre application.

Actuellement, le couplage est plus faible, et permet de facilement permuter les dépendances (ex : *La classe 1 peut facilement dire qu'elle a besoin d'un élément A plutôt qu'un B puisqu'ils implementent tout 2 l'interface X, et que la propriété de la classe 1 est de type X*)

Mais il est encore possible d'améliorer la maintenabilité, car la classe 1 créait ses propres instances de ses dépendances :



*Abstraction des dépendances mais sans injection de dépendance*

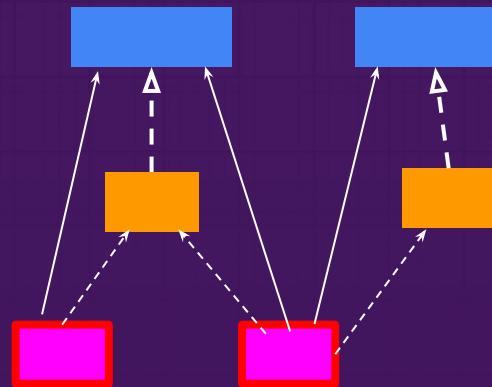
- Cela viole le principe de responsabilité (SRP Single Responsibility Principle), car on a 2 raison de la modifier : changer son code métier ou changer ses dépendances
- Il est difficile de créer une instance de la classe qui utiliserait une autre dépendance (par exemple dans le cadre d'un test)



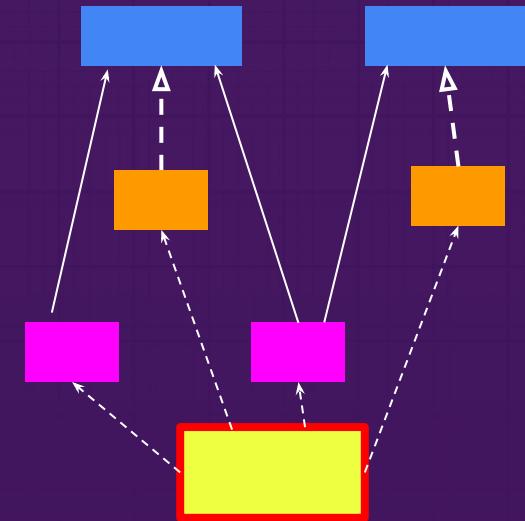
## >> Externaliser la création des dépendances

L'idée est de créer les instances des dépendances à l'**extérieur** (*dans le programme principal ou une autre classe*) et de les passer au **classe dépendante** via leur constructeur ou leurs accesseur

A noter que fondamentalement il n'est pas nécessaire d'avoir au préalable rendu les **dépendances abstraites** mais c'est fortement recommandé (*donc les types des paramètres du constructeur et des setter sont normalement les *interface* ou des *classes abstraites**)



*Sans injection de dépendance*



*Avec injection de dépendance*



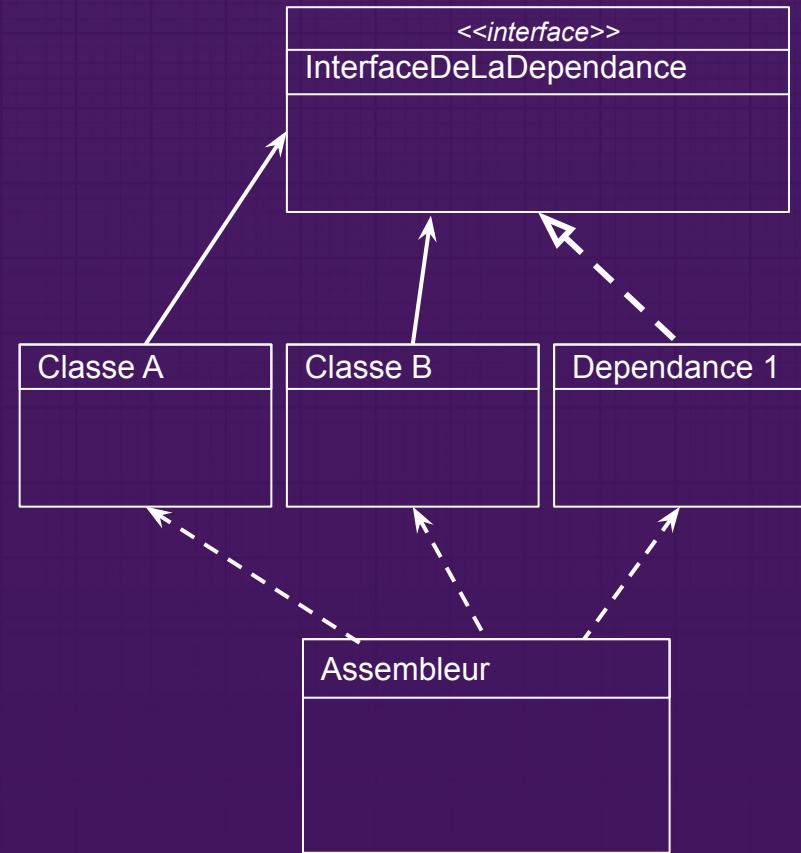
## &gt;&gt; Le design pattern : Injection de dépendance

Le **design pattern “Injection de dépendance”** est l’application des mécanismes précédents :

- *abstraction des propriété*
- *injection de dépendance*

Et centraliser la création des dépendance dans une classe souvent appelée Assembleur (*assembleur*).

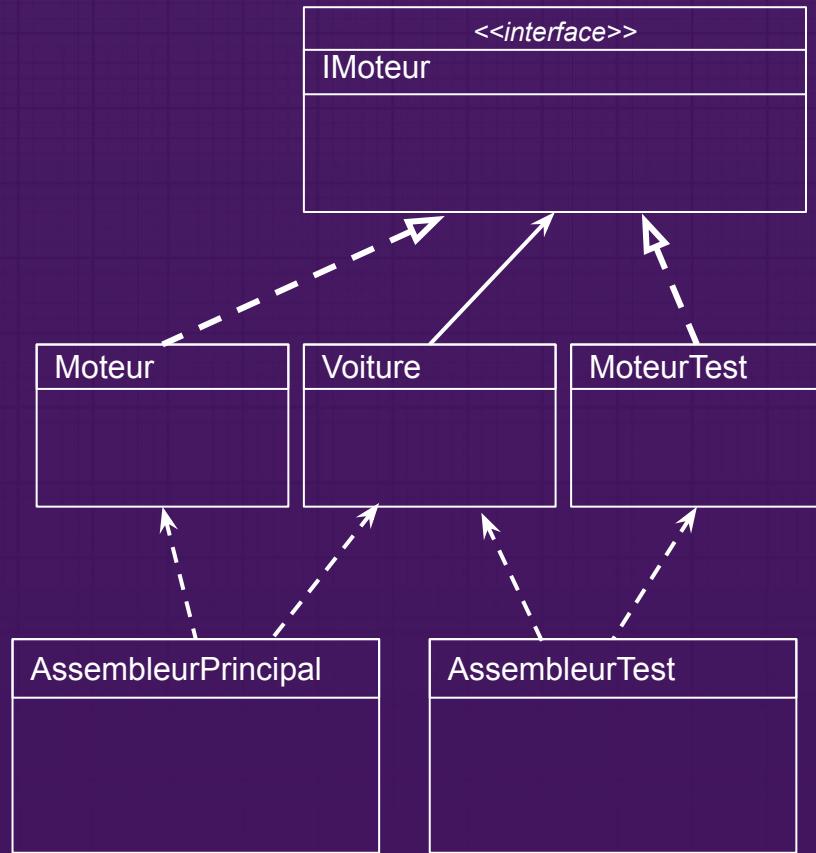
Dans cet Assembleur nous allons créer toutes les instances des classes et des dépendances, puis affecter les dépendances aux instance qui en ont besoin (A noter qu’une classe dépendante pourrait également être une dépendance d’une autre classe)



## &gt;&gt; Plusieurs configuration de dépendances avec plusieurs assembleurs

Il est non seulement possible, mais aussi recommandé d'utiliser plusieurs **assembleurs** ou **configurations d'injection** pour différents contextes d'application, comme pour la production, les tests fonctionnels, ou d'autres environnements spécifiques.

Dans notre cas, nous voulons utiliser la dépendance MoteurTest pour notre Voiture dans le cadre de nos tests, alors que dans l'application principale c'est la dépendance Moteur que nous voulons lui affecter



## &gt;&gt; Exemple plus pertinent

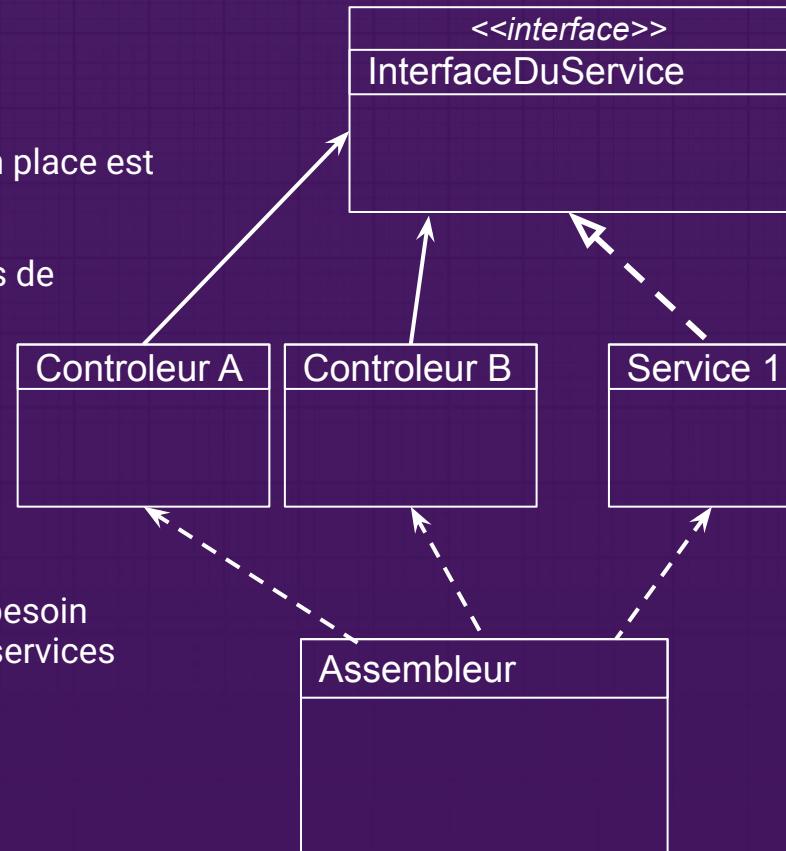
Un exemple plus proche de ce que nous allons réellement mettre en place est le suivant :

Notre application possède des contrôleurs qui vont être dépendants de services (*pour effectuer des requêtes, réaliser des calculs ...*)

Nos Contrôleurs ont donc des propriétés abstraites

L'assembleur va :

- créer les instances des contrôleurs,
- créer les instances des Services
- Affecter les instance des services aux contrôleurs qui en on besoin
- Potentiellement affecter des instances de service à d'autres services



## &gt;&gt; Exemple sans Framework

L'utilisation concrète de cet assembleur peut se faire différemment, mais le principe global sera toujours le même :

- Toutes les classes sont créées dans l'assembleur (*ControleurA et Service1 dans cet exemple, mais il pourrait y en avoir d'autres*)
- L'assembleur lie les classes entre elles (*Service1 est une dépendance de Controleur A, Service 1 est donc injectée dans Controleur A via son accesseur*)
- On accède aux classes via l'assembleur

```
public class Assembleur {  
  
    ControleurA controleurA;  
  
    public Assembleur () {  
        controleurA = new ControleurA();  
        Servicel servicel = new Servicel();  
        controleurA.setService1(servicel);  
    }  
  
    public ControleurA getControleurA () {  
        return controleurA;  
    }  
}
```

```
public class Main {  
  
    public static void main(String[] args) {  
        Assembleur assembleur = new Assembleur();  
        ControleurA controleurA =  
        assembleur.getControleurA();  
    }  
}
```



## &gt;&gt; Injection de dépendance par constructeur

Une approche légèrement différente consiste à **injecter** la dépendance via le **constructeur** plutôt que par un setter.

Les 2 approches sont valables, mais Spring recommande plutôt l'approche par constructeur.

```
public class Assembleur {  
  
    ControleurA controleurA;  
  
    public Assembleur () {  
        Service1 service1 = new Service1();  
        controleurA = new ControleurA(service1);  
    }  
  
    public ControleurA getControleurA () {  
        return controleurA;  
    }  
}
```



## >> Le conteneur IOC de Spring Boot (Inversion Of Control)

On arrive finalement à la dernière étape : utiliser le système mise en place par Spring Boot

Plutôt que de créer nous même l'assembleur, nous allons :

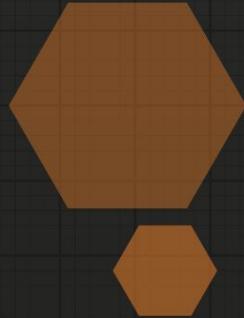
- Utiliser des Annotations sur les classes ou les interface afin de signaler que ce sont des dépendances (Appelée "Bean" en JAVA)
- Utiliser des Annotations sur les propriétés des classes afin de signaler que la classe en est dépendante (Injection par accesseur)
- Ou bien sur le constructeur, dans ce cas ses paramètres seront les dépendances de la classes (Injection par constructeur)

Cet assembleur virtuel sera alors "généré" à la compilation de l'application.

Cette approche permet de rendre la structure de l'application plus lisible, plutôt que de devoir parcourir un unique fichier Assembleur afin de savoir quelle classe est dépendante de quelle autre classe

*A noter que Spring (pas Spring boot) utilise un fichier XML plutôt que des annotations, ce qui rend la maintenance plus lourde, mais elle utilise bien l'inversion de control*





# Créer des dépendances avec Spring via des classes ou des interfaces

*Chapitre : Créer des contrôleurs*

## >> Créer des dépendances à partir d'une classe

Il est possible d'implémenter ce **design pattern**, mais comme nous l'avons vu, **Spring** suit le principe d'**inversion de contrôle**.

C'est donc **Spring** qui fournira **l'assembleur**, et c'est à nous de lui signaler quelles classes devront être des dépendances, et quelles classes doivent recevoir ces dépendances.

Pour Spring une dépendance est un **bean**.

Pour indiquer à Spring qu'elle classe est un bean on utilise des **annotations**.

On ajoute au dessus de la classe concerné l'une des Annotations suivantes : `@Repository` `@Service` ou `@Component`  
Exemple :

```
import org.springframework.stereotype.Service;

@Service
public class UnBeanService {
    public String direHello(){
        return "Hello from " + this.toString();
    }
}
```



## >> Créer des dépendances à partir d'une classe

Les annotations `@Repository` `@Service` et `@Component` permettent toutes 3 de créer une **dépendance** mais il n'ont pas les même buts.

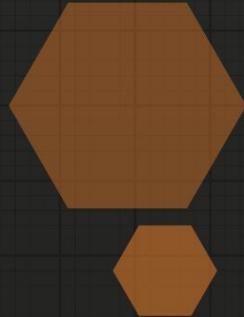
`@Service` est la plus commune, il s'agit d'une **classe** qui a pour but de stocker des méthodes utilitaires

`@Component` est identique, mais il donne un sens différent à la classe / interface : il ne contient pas des méthodes utilitaires, mais à plutôt pour but le fonctionnement d'un autre composant.

`@Repository` désigne une classe destinée à avoir des méthodes d'accès aux données (*DAO*) nous la verrons un peu plus tard en détails

Une **classe** ou une **interface** annotée avec l'une d'elle deviendra une **dépendance** prête à être **injectée**.





# Créer des dépendances avec Spring via des méthodes

*Chapitre : Créer des contrôleurs*

## >> Créer une dépendance via une méthode

Il est également possible de créer des **dépendances** via des **méthodes**

Il faut pour cela être dans une classe qui a reçu l'annotation **@Configuration** (ou d'une annotation héritant de celle ci)

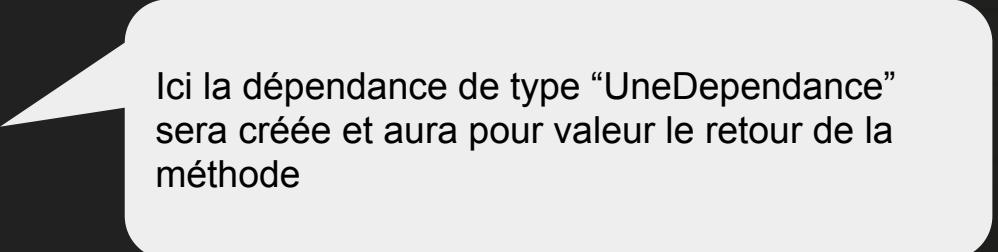
La dépendance créée sera du type de retour de la méthode

Cette méthode devra posséder l'annotation **@Bean** (le nom de la méthode et de la importe pe

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class UneConfiguration {

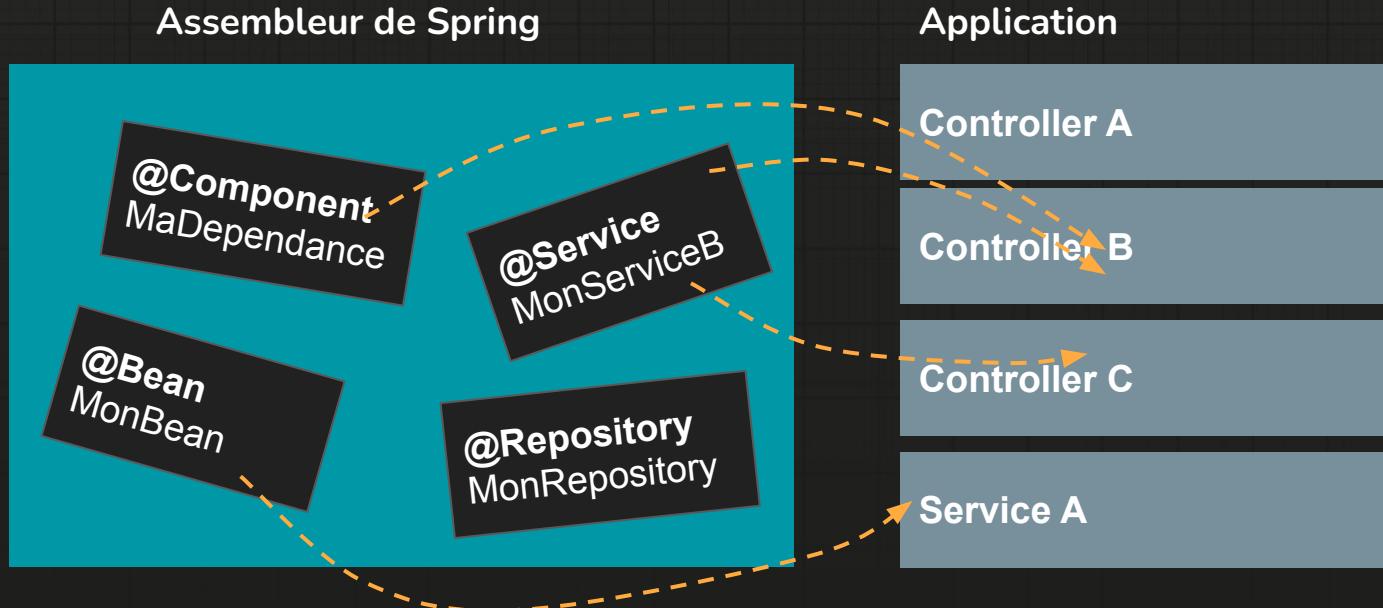
    @Bean
    public UneDependance methode(){
        return new UneDependance();
    }
}
```



Ici la dépendance de type “UneDependance” sera créée et aura pour valeur le retour de la méthode

# >> L'injection de dépendances : Création de Bean

Au démarrage de l'application, **Spring** va dans un premier temps, parcourir toutes les **classes** à la recherche des **beans** que l'on a déclaré. Nous pourrons alors les utiliser dans d'autres **classes** et réaliser ainsi une **injection de dépendance**.



# >> L'injection de dépendances : Injection par propriété

On peut réaliser une injection de dépendance de 3 façons avec Spring : **par propriété**, **par accesseur** ou **par constructeur**.

Les 3 utilisent l'annotation **@Autowired** mais la troisième est plus recommandé.

Pour utiliser **l'injection par propriété**, il suffit d'ajouter l'annotation **@Autowired** sur la propriété qui devra recevoir l'injection de dépendance. Ici on injecte une instance de la classe **UnBeanService** dans le propriété **injectionDeUnBeanService**. Cette instance a été créé par Spring au démarrage de l'application et sera injecté lors de l'instanciation de **TestDependanceController**

```
@RestController
public class TestDependanceController {

    @Autowired
    private UnBeanService injectionDeUnBeanService;

    @GetMapping("/test/dependance")
    public String testDependance() {
        return unBeanService.direHello();
    }
}
```

```
@Service
public class UnBeanService {
    public String direHello(){
        return "Hello from " + this.toString();
    }
}
```



# >> L'injection de dépendances : Injection par constructeur

L'**injection par constructeur** a une syntaxe un peu plus lourde, mais elle est recommandé par l'équipe de développement de Spring.

(Car elle permet de rendre le code plus lisible, d'être sûr que l'injection est bien réalisée, de pouvoir effectuer des test unitaires plus rigoureux, et de rendre une instance immuable puisqu'elle ne peut se faire que par le constructeur).

C'est le **constructeur** qui reçoit l'annotation **@Autowired**. Tous ces **paramètres** sont alors des **dépendances**.

Si la dépendance doit être utilisé dans ses méthodes (ce qui est souvent le cas) il faut alors affecter la valeur de ces paramètres dans ses propriétés

```
@RestController  
public class TestDependanceController {  
  
    private UnBeanService unBeanService;  
  
    @Autowired  
    public TestDependanceController(UnBeanService unBeanService) {  
        this.unBeanService = unBeanService;  
    }  
  
    @GetMapping("/test/dependance")  
    public String testDependance() {  
        return unBeanService.direHello();  
    }  
}
```

```
@Service  
public class UnBeanService {  
    public String direHello(){  
        return "Hello from " + this.toString();  
    }  
}
```

## >> Ordre de déclaration et priorisation

TODO Abstraction des Bean `@Qualifier` ou `@Primary`  
`@Qualifier` ou `@Primary`

Injection recursive, création des ordre ...

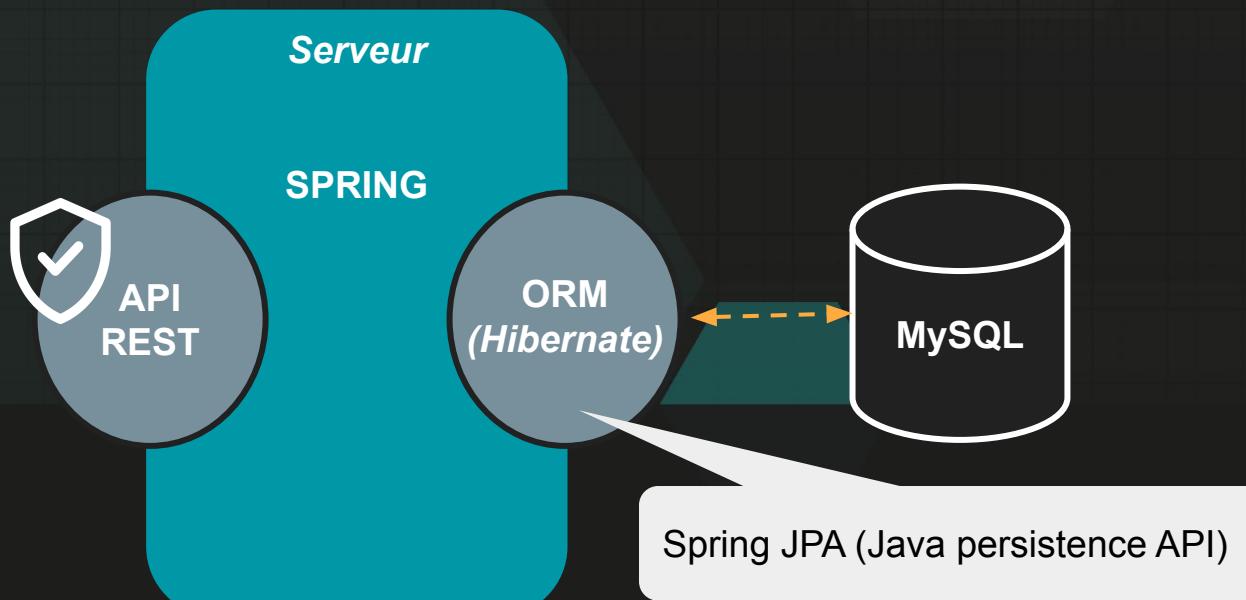


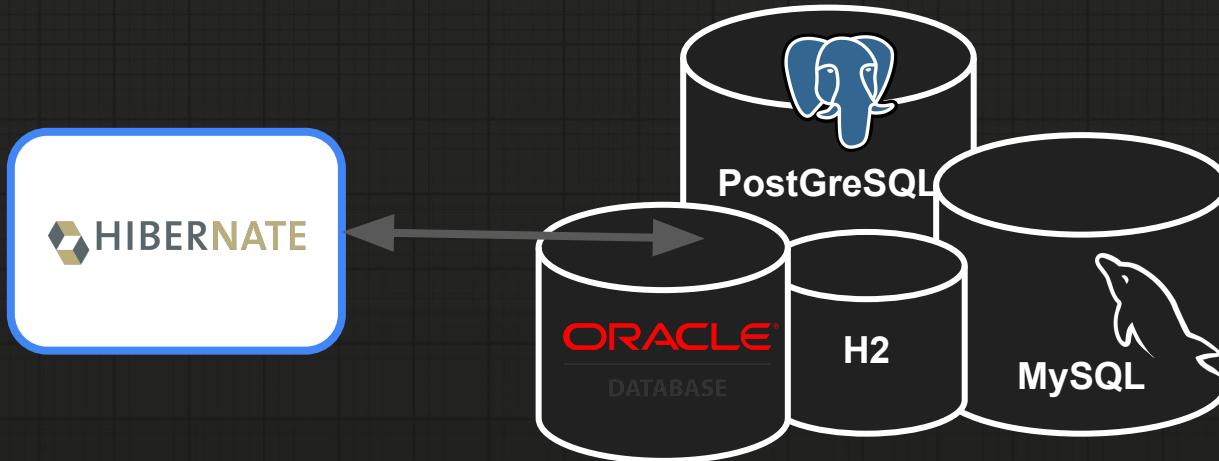
# Hibernate JPA / Spring JPA

Spring JPA : Une aide pour manipuler l'ORM Hibernate

*ORM : Object Relational Mapping*

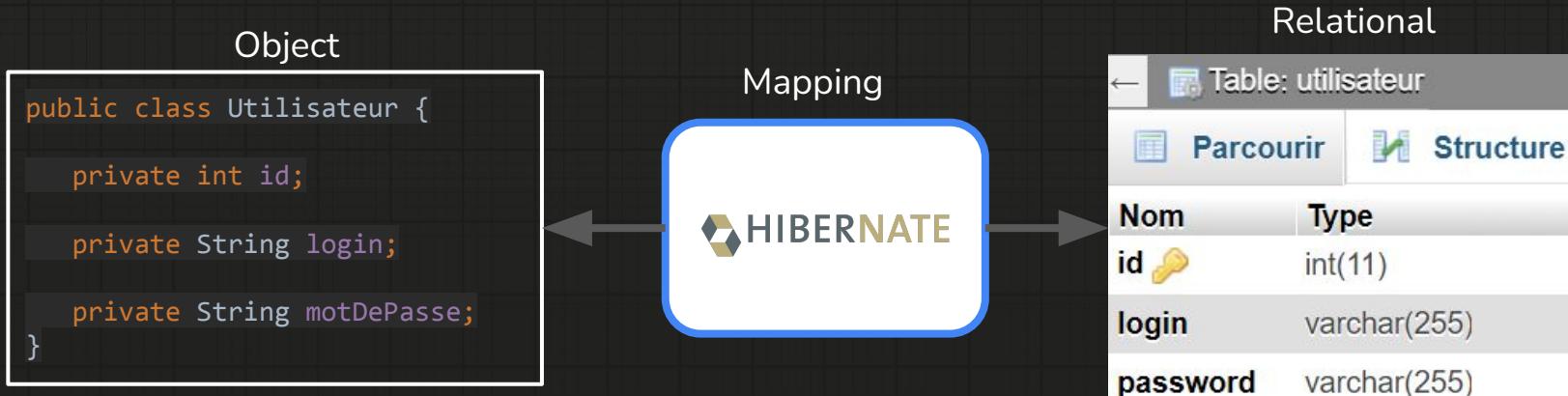
Dans ce chapitre :





**Hibernate** est un **ORM** (*Object Relational Mapping*). En se basant sur notre **modèle**, il sera capable de traduire des instructions dans le langage de la base de donnée relationnelle de notre choix. Il nous permet ainsi de facilement migrer d'un moteur de base de donnée à un autre (pas besoin de réécrire des requêtes). Et d'augmenter la productivité des développeurs en leur permettant de ne pas avoir à réécrire en permanence des requêtes SQL pour chaque **entité**.

## >> Fonctionnement d'un ORM



En indiquant assez d'informations à notre **ORM**, il sera capable de déduire la base de donnée à partir des classes de notre modèle. Chaque **classe** de notre **modèle** (que l'on nommera entité) deviendront des tables dans la base de donnée. Et chaque **propriété** de la **classe** deviendront des **champs** dans cette **table**.



## >> Fonctionnement d'un ORM : persistance

```
Utilisateur subject3910 = new Utilisateur();
subject3910.setId(3910);
subject3910.setPseudo("subject");
subject3910.setMotDePasse("th€C@k€i$@£i€");
fairePersister(subject3910);
```



Chaque **instance** d'une **entité** (une classe de notre modèle) pourra être enregistrée **dans la base de données**.

On emploiera le terme “faire **persister** l'objet dans la base de données”.

Donnant ainsi un nouvel **enregistrement** (*ou tuple*) dans la **table**.

La **valeur** des **propriétés** de l'**objet** devenant ainsi la **valeur** des **colonnes** de l'**enregistrement** dans la **table**.

## >> Fonctionnement d'un ORM : récupération

```
Utilisateur subject3910;  
  
subject3910 = recupererUtilisateurParId(3910);  
  
System.out.println(subject3910.getMotDePasse());  
  
//affiche dans la console : th€C@k€i$@Fie
```



id	login	password
1	franck	\$2a\$10\$uz5dB8k
2	john	\$2a\$10\$uz5dB8k
3910	subject	th€C@k€i\$@Fie

Naturellement l'opération inverse sera possible, et on pourra effectuer des opérations permettant de récupérer des informations en base de donnée afin qu'elles soient affectés à des instances de nos modèles

## >> Exemple d'opérations d'Hibernate

Donne moi l'utilisateur  
avec l'id 3910



1 : On regarde à quoi ressemble la classe **Utilisateur**

2 : On construit une requête SQL grâce au modèle

--> *SELECT id, pseudo, motDePasse FROM utilisateur WHERE id = 3910*

3 : On exécute la requête et on récupère le résultat

4 : On crée une instance de la classe **Utilisateur**

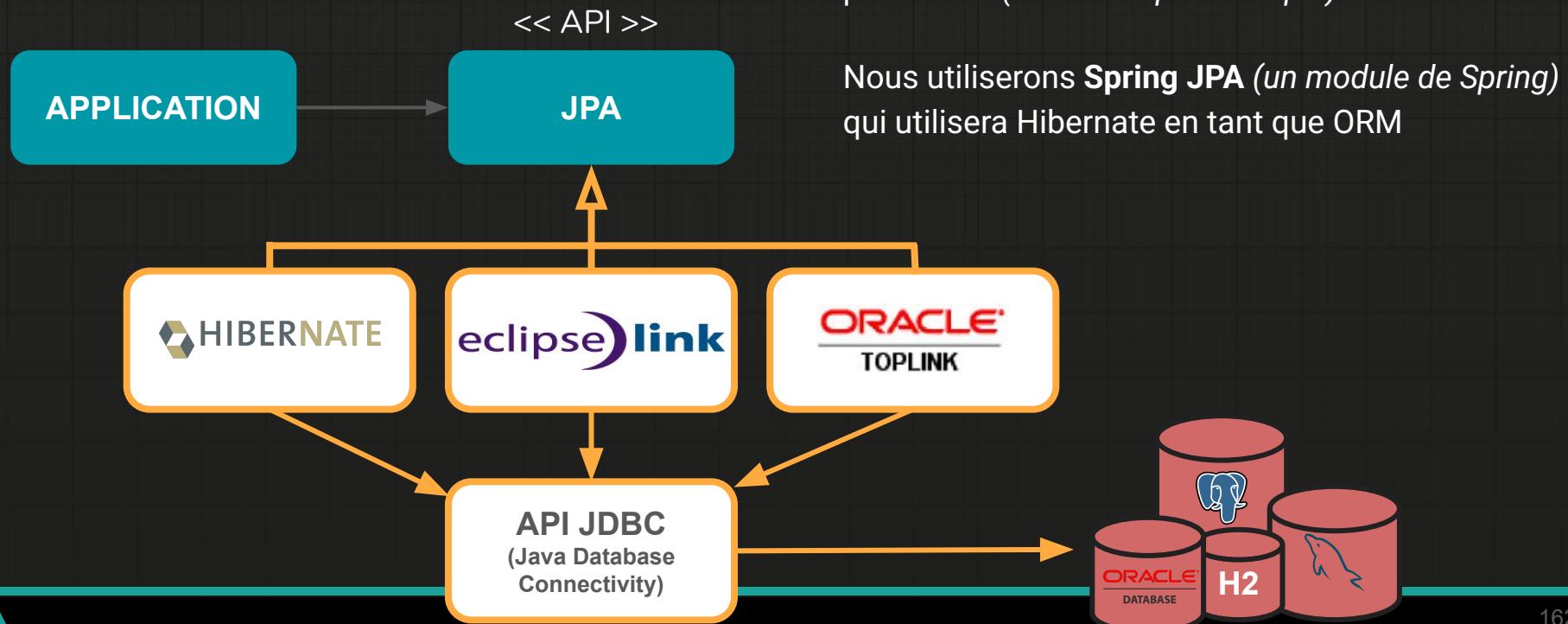
5 : On affecte les valeurs des colonnes à l'instance crééé

```
public class Utilisateur {  
  
    private int id;  
  
    private String pseudo;  
  
    private String motDePasse;  
}
```

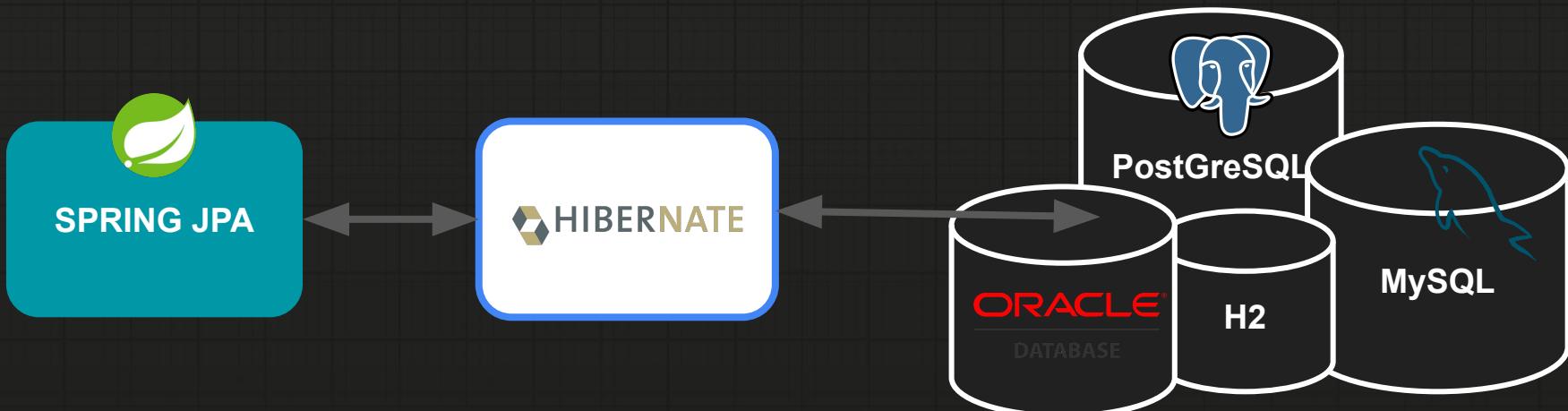
## >> JPA (Java Persistence API)

JPA est une spécification.

Elle permet de **standardiser** les ORM disponibles pour JAVA (*Hibernate par exemple*).



## >> SPRING JPA (Java Persistence API)

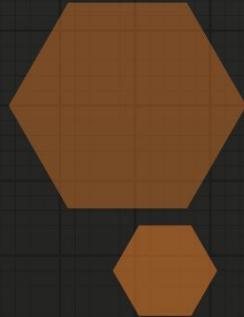


Spring JPA va nous permettre de manipuler facilement **Hibernate** afin d'effectuer d'augmenter notre productivité.

Il nous permettra entre autre d'effectuer facilement des **requêtes CRUD** (*Create Read Update Delete*) ou des requêtes plus personnalisées. Mais également de générer automatiquement le schéma de la base de donnée à partir du modèle de notre application.

# Se connecter à une base de données

Dans ce chapitre :



# Installer une base de donnée MySQL

*Chapitre : Créer des contrôleurs*

## >> Recommandé : Docker Desktop

Installez Docker desktop (*si votre configuration ne le permet pas, des méthodes alternatives sont présentées par la suite*)

<https://www.docker.com/products/docker-desktop/>

A la racine du projet, créez un fichier `docker-compose.yml`

Collez-y le code ci-contre

Lancer la commande suivante dans un terminal (ex : *le terminal d'IntelliJ*) :  
`docker-compose up -d`

Si un message vous avertit qu'un port est déjà utilisé, vous pouvez changer le port **3306** du fichier et/ou le port **8181**

```
Error response from daemon: Ports are not available: exposing port TCP 0.0.0.0:3306
-> 0.0.0.0:0: listen tcp 0.0.0.0:3306: bind: Only one usage of each socket address
(protocol/network address/port) is normally permitted.
```

Une fois lancée vous pouvez administrer votre base de donnée mysql à l'adresse : <http://localhost:8181>

`docker-compose.yml (nouveau fichier)`

```
version: '3'

services:
  db:
    image: mysql:latest
    container_name: db
    environment:
      MYSQL_ROOT_PASSWORD: root
    ports:
      - "3306:3306"
    volumes:
      - dbdata:/var/lib/mysql
  phpmyadmin:
    image: phpmyadmin/phpmyadmin
    container_name: pma
    links:
      - db
    environment:
      PMA_HOST: db
      PMA_PORT: 3306
    restart: always
    ports:
      - 8181:80
  volumes:
    dbdata:
```



## >> Méthode alternatives

Vous pouvez également utiliser le programme Xampp disponible sur tous les OS (*vous pouvez n'installer que apache, php, phpmyadmin et mysql afin de bénéficier de l'interface web phpmyadmin*) :

<https://www.apachefriends.org/fr/index.html>

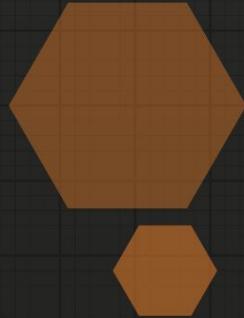
Une alternative plus légère sur windows :

<https://www.uwamp.com/fr/>

Sur linux, dans cet exemple Ubuntu 22, vous pouvez également simplement installer le package mysql :

<https://www.digitalocean.com/community/tutorials/how-to-install-mysql-on-ubuntu-22-04>





# Installer les dépendances

*Chapitre : Créer des contrôleurs*

## &gt;&gt; Dépendances maven

pluriel

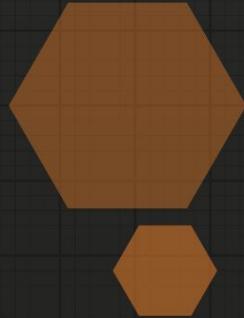
Ajoutez les dépendance suivantes dans le noeud <**dependencie**s> du fichier pom.xml

pom.xml

```
<dependency>
  <groupId>org.springframework.boot </groupId>
  <artifactId>spring-boot-starter-data-jpa </artifactId>
</dependency>

<dependency>
  <groupId>com.mysql </groupId>
  <artifactId>mysql-connector-j </artifactId>
  <scope>runtime</scope>
</dependency>
```

N'oubliez pas de mettre à jour l'application via l'icône  qui apparaît alors en haut à droite.



# Configurer la connexion

*Chapitre : Créer des contrôleurs*

## >> Configuration Spring

Dans le fichier `src/main/resources/application.properties` ajoutez le code suivant

```
spring.datasource.url=jdbc:mysql://localhost:3306/maBaseDeDonnee?serverTimezone=UTC&createDatabaseIfNotExist=true  
spring.datasource.username=utilisateurMySql  
spring.datasource.password=motDePasseMySql
```

Remplacer `maBaseDeDonnee` par le **nom de la base de donnée** que vous désirez créer

Normalement vous n'avez pas à changer le **port** par défaut de MySql (`3306`)

Remplacez `utilisateurMySql` et `motDePasseMySql` par les identifiants d'un utilisateur de votre base de données (*qui aura tous les priviléges sur votre base de donnée*)

Si vous n'avez pas de mot de passe, ajouter tout de même la propriété

```
spring.datasource.password=
```

Mais n'ajoutez pas de valeur



## &gt;&gt; Configuration Spring

Normalement votre configuration devrait être :

```
spring.datasource.url=jdbc:mysql://localhost:3306/spring_td?serverTimezone=UTC&createDatabaseIfNotExist=true  
spring.datasource.username=root  
spring.datasource.password=root
```

Si vous avez suivi exactement la configuration docker votre mot de passe doit être **root**

Si vous avez installé Xampp vous ne devriez pas avoir de mot de passe

Si vous avez installé UwAmp vous devriez pas avoir **root** comme mot de passe



# >> Lire une stacktrace

Une stacktrace JAVA contient un grand nombre d'information et 99% des informations sont inutiles pour résoudre le problème rencontré.

Elle est composée d'une succession d'erreurs, généralement causées par l'erreur précédente.

Ainsi qu'un grand nombre de ligne commençant par "at" ne servant qu'à expliquer quelle succession de méthodes ont été appelée jusqu'à ce que l'erreur soit apparue.

Chaque erreurs est également composé d'une succession de ligne commençant par "Caused by" qui résulte d'un englobage de l'erreur initial qui est généralement plus générique.

Donc en résumé :

- les "at" sont inutiles
- les erreurs qui surviennent après la première doivent être ignoré
- **C'est le dernier des "Caused by" de la première des erreurs qui apporte le plus d'informations**

*Stacktrace provoquée lorsque l'utilisateur ou le mot de passe est erroné (ou manquant)*

```
org.hibernate.exception.GenericJDBCException: unable to obtain isolated JDBC connection ...
  at org.hibernate.exception.internal.StandardSQLExceptionConverter.co ...
  at org.hibernate.engine.jdbc.spi.SqlExceptionHelper.convert(SqlExceptionHelper ...
Caused by: java.sql.SQLException: Access denied for user 'root'@'172.17.0.1' (using password: NO)
  at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java: ...
  at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(S ...
```

```
org.springframework.beans.factory.BeanCreationException: Error creating bean with name
'entityManagerFactory' ...
  at org.springframework.beans.factory.support.AbstractAutowireCapableBe ...
  at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactor ...
Caused by: org.hibernate.service.spi.ServiceException: Unable to create requested service [org.hiber ...
  at org.hibernate.service.internal.AbstractServiceRegistrationImpl.createService(AbstractSer ...
  at org.hibernate.service.internal.AbstractServiceRegistrationImpl.initializeService(Abst ...
Caused by: org.hibernate.HibernateException: Unable to determine JDBC dialect
  at org.hibernate.engine.jdbc.dialect.internal.DialectFactoryImpl.determineDial ...
  at org.hibernate.engine.jdbc.dialect.internal.DialectFactoryImpl.buildDialect( ...
```

Process finished with exit code 1

Ici cette erreur n'importe aucune information.  
Elle est simplement due à la première de nos erreurs : la base de données est inaccessible



# >> Lire une stacktrace

Un autre exemple plus parlant, provoqué lorsque la base de données n'est pas accessible (à cause d'une mauvaise URL ou si la base de données est arrêtée)

4 erreurs sont affichées, seule la première est importante, et la dernière ligne "Caused by" est assez explicite sur le problème.

Au final sur une centaine de ligne d'erreur, seul une nous intéresse

Pensez donc à vérifier la **console de sortie** est de la redimensionnée assez pour pouvoir détecter la **première erreur** et son **dernier "Caused by"**

*Stacktrace provoquée lorsque l'URL est erronée (hôte ou port), ou la base de donnée stoppée*

```
The last packet sent successfully to the server was 0 milliseconds ago. The driver has not received any packets from the server.] [n/a]
at org.hibernate.exception.internal.SQLExceptionConversionDelegate...
at org.hibernate.exception.internal.StandardSQLExceptionConverter...
Caused by: com.mysql.cj.jdbc.exceptions.CommunicationsException: Communications link failure

The last packet sent successfully to the server was 0 milliseconds ago. The driver has not received any packets from the server.
at com.mysql.cj.jdbc.exceptions.SQLException.createCommunicationsException...
at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException...
Caused by: com.mysql.cj.exceptions.CJCommunicationsException: Communications link failure

The last packet sent successfully to the server was 0 milliseconds ago...
at java.base/jdk.internal.ref...
at java.base/java.lang.reflec...
...
Caused by: java.net.ConnectException: Connection refused: getsockopt
at java.base/sun.nio.ch.Net.pollCon...
at java.base/sun.nio.ch.Net.pollConn...
...
2025-03-13T20:59:24.711+01:00 ERROR 26500 Failed to initialize ...
Error starting ApplicationContext. To display the condition...
2025-03-13T20:59:24.741+01:00 ERROR 26500 ...
...
Error creating bean with name 'entityManagerFactory' defined in cl...
Dialect implementation must be provided)
at org.springframework.beans.factory.suppor...
at edu.fbansept.td.CoursFilRougeSpring3TdA...
...
Caused by: org.hibernate.service.spi.ServiceException: Unable to create requested service ...
at org.hibernate.service.internal.AbstractSe...
...
Caused by: org.hibernate.HibernateException: Unable to determine Dialect without JDBC metadata ...
at org.hibernate.engine.jdbc.dialect.interna...
```

Process finished with exit code 1

# Créer le model

Dans ce chapitre :

## >> Créer des JavaBean

Toutes les classes de notre model devront respecter la notre JavaBean :

- Posséder le constructeur par défaut (sans paramètre) ou ne pas avoir de constructeur
- Posséder un getter et un setter pour chacune des propriété
- Respecter les règle de nommage des getter et des setter
  - Pour tous les propriété à l'exception des booléens les getter devront être nommé avec le le mot "get" puis le nom de la propriété commençant par une majuscule (exemple pour une propriété nom : getNom() ) et retournant le type de la propriété
  - Idem pour les propriétés de type booléen, mais commencer par is à la place de get (exemple pour une propriété admin : isAdmin() )
  - Les setters prennent en paramètre le type de la propriété et commencent par "set" puis le nom de la propriété commeçant par une majuscule (exemple : setNom(String nom) , setAdmin(Boolean admin))



## &gt;&gt; Créer des entités

models/Produit (nouveau fichier)

```
import lombok.Getter;  
import lombok.Setter;  
  
@Getter  
@Setter  
public class Produit {  
  
    protected Integer id;  
    protected String code;  
    protected String nom;  
    protected String description;  
    protected float prixHt;  
    protected float prixTtc;  
    //Pas besoin d'accesseurs avec les annotation @Getter et @Setter  
}
```

Les annotations **@Getter** et **@Setter** génèrent automatiquement nos accesseurs

Chaque **classe** de notre **model** correspondra à une **table**, et chaque **propriété** correspondra à un **champ** de cette table



Pour créer une entité, il faut au minimum lui ajouter l'annotation `@Entity`.

Il faut également ajouter l'annotation `@Id` sur le champs représentant la clé primaire

*models/Produit*

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
@Entity
public class Produit {

    @Id
    protected Integer id;
    protected String code;
    protected String nom;
    protected String description;
    protected float prixHt;
    protected float prixTtc;
}
```



## >> Les propriétés à ne pas intégrer dans la table

Il est possible d'avoir dans notre entité des propriétés qui ne doivent pas être gérées par JPA (*il ne faut pas que la propriété soit mappé avec un champs*) il faut pour cela ajouter l'annotation `@Transient`

C'est le cas d'une données calculée par exemple : le prix TTC sera calculé en fonction de la TVA et le prix Hors Taxe  
*models/Produit*

```
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
@Entity
public class Produit {

    @Id
    protected Integer id;
    protected String code;
    protected String nom;
    protected String description;
    protected float prixHt;

    @Transient
    protected float prixTtc;
}
```

Il n'est pas encore possible de calculer le prix TTC car il n'y a pas encore le TVA, mais cette propriété n'aura pas de colonne correspondante dans la table produit

Cette propriété apparaîtra tout de même dans le JSON qui sera envoyé au FRONT (*avec en valeur la données calculée*)



## >> Les valeurs par défaut des propriétés

Pour rappel la valeur initiale d'une propriété dépend de son type. (Si son constructeur ne défini pas déjà la valeur de cette propriété.

Si c'est une classe (Integer, String, Float ...) cette propriété n'a pas de référence, elle a donc la valeur "null"

Si c'est un type primitif, cela dépendra de ce type, un numérique donnera *0*, alors qu'un booléen donnera *false*

Dans le cas de l'id, il est donc préférable de ne pas le définir avec un type primitif (comme int), puisqu'une entité non persistée n'a tout simplement pas d'id, plutôt qu'un id à 0

Dans tous les cas si une valeur est initialisé, c'est celle-là qui sera affectée

```
public class Article {  
  
    protected Integer id; //initialisé à NULL  
  
    protected String nom; //initialisé à NULL  
  
    protected float prix; //initialisé à 0f  
  
    protected Float promotion; //initialisé à NULL  
  
    protected float tva = 5.5f; //initialisé à 5.5f  
  
    protected boolean ruptureDeStock; //initialisé à false  
}
```

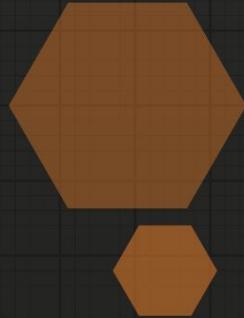


>> Fil rouge

>> Connecter DATAGRIP

TODO





Auto générer la base de donnée à partir du model

*Chapitre : Créer des contrôleurs*

## >> Auto générer la base de donnée

Une fonctionnalité très intéressante de JPA est de permettent de construire la base de donnée à partir de notre model.

Bien entendu cela reste **OPTIONNEL**. Mais cela nous permettra de tester les effets de la modification de notre model sur la base de donnée et d'éviter les problèmes de mapping en créant nous même la base de donnée



## >> Auto générer la base de donnée

A partir des classes de notre modèle, on peut autogénérer notre base de donnée. Cette autogénération peut se faire de 4 manière différentes :

- validate : valide le schéma, mais n'effectue aucune opération (utile en cas d'approche database first)
- update: met à jour les informations manquantes, mais ne supprime pas les champs existant s'ils ne sont pas mappé (garde les enregistrement des tables)
- create: supprime puis recréé le schéma (supprime tous les enregistrements)
- create-drop: comme create, mais supprime le schéma lors de l'arrêt de l'application (utile pour des scénarios de test)
- none : n'effectue aucune modification/validation (à utiliser en production)

Exemple :

```
spring.jpa.hibernate.ddl-auto = create
```

Note 2 : DDL = Data Description Language (C'est à dire toutes les instructions qui concerne la structure de la BDD)



## &gt;&gt; Auto générer la base de donnée

Ajouter les lignes suivantes dans resources/application.properties

```
spring.jpa.hibernate.ddl-auto = create
```

Nous préférons la propriété create car même si la base de données est supprimée à chaque fois que l'application est relancée, nous ajouterons un fichier de données de test qui insérera les enregistrement dans la base de données après sa reconstruction.

Ainsi, même après différents tests (suppressions, modifications ...) les données seront toujours réinitialisée et il n'y aura pas de colonnes orpheline comme avec l'utilisation du mode update.



## >> Tester

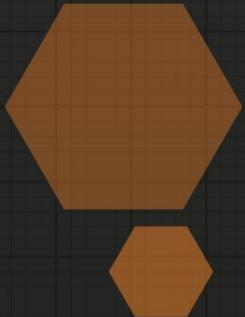
Vous pouvez tester en démarrant l'application.

Attention : en mode update JPA ne supprimera aucune information. Ainsi renommer une propriété ne fera que rajouter un champs sans supprimer l'ancien.

Certaines fonctionnalités comme les jointures ne seront pas non plus supprimée, ce qui peut amener à supprimer à la main certains élément de la base de donnée.

Arrivé à un certain niveau de complexité il est préférable d'exporter un jeu d'essai et de supprimer la base lors de ce type de modification avant de relancer l'application.





# Customiser le model

*Chapitre : Créer des contrôleurs*

## >> Personnaliser le matching

Via l'annotation `@Table` et `@Column` il est possible de personnaliser le matching

```
@Getter  
@Setter  
@Entity  
@Table(name = "product")  
public class Produit {  
  
    @Id  
    protected Integer id;  
  
    protected String code;  
  
    @Column(name = "name")  
    protected String nom;  
  
    protected String description;  
  
    protected float prix;  
}
```

Via l'annotation `@Table` le mapping se fera sur la table `product`

Via l'annotation `@Column` la propriété `nom` sera finalement mappé avec le champs `name` de la table `product`

## &gt;&gt; Déclarer la clé primaire comme étant auto incrémentée

Il est possible d'utiliser plusieurs méthodes pour auto-incrémente une clé primaire, mais nous ne verrons que la plus commune via l'annotation `@GeneratedValue(strategy = GenerationType.IDENTITY)`

*models/Produit*

```
@Getter  
@Setter  
@Entity  
public class Produit {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    protected Integer id;  
  
    protected String code;  
  
    protected String nom;  
  
    protected String description;  
  
    protected float prix;  
}
```



## &gt;&gt; Autres propriétés de l'annotation @Column

Par défaut toutes les propriétés qui ne sont pas des type primitif peuvent recevoir la valeur NULL en base de donnée. Via l'annotation `@Column`, il est possible de les rendre obligatoire, ou d'affecter d'autres contraintes.

*models/Produit*

```
...
public class Produit {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    protected Integer id;

    @Column(nullable = false, unique = true)
    protected String code;                                Un code est obligatoire et doit être unique

    @Column(nullable = false, length = 100)
    protected String nom;                                Un nom est obligatoire et ne peut pas dépasser 100 caractères

    @Column(columnDefinition = "TEXT")
    protected String description;                         Le type par défaut en MySQL pour un String est VARCHAR(255). Puisque notre
                                                        description peut dépasser 255 caractères, on force le type de la colonne à TEXT

    protected float prix;                               Les types primitifs ne sont pas nullable
}
```



## >> Les propriétés Dates

Il existe des formats de date qui sont plus appropriés selon l'usage. Les dates "Locale" gèrent les fuseaux horaires, alors que les timestamp non. Certaines dates gèrent l'heure d'autres non.

```
public class Utilisateur {
```

```
    //Date  
    LocalDate dateDeNaissance;
```

Ne stock que la date. Favoriser ce type lorsque la date ne doit pas subir les fuseaux horaires mondiaux.

```
    //heure  
    LocalTime heureDeReveil;
```

Ne stock que l'heure

```
    //Date + heure  
    LocalDateTime derniereConnexion;
```

Stock les 2 (datetime sur mysql)

```
    //Timestamp en seconde  
    Long dureeDeConnexion;
```

Pour un timestamp (*nombre de seconde depuis le 1er janvier 1970*)  
Stocké en nombre de seconde

```
    //Timestamp représenté en date  
    Instant dateDeCreation;
```

Pour un timestamp (*nombre de seconde depuis le 1er janvier 1970*)  
Stocké au format date (datetime 6 )

```
}
```

## >> PrePersist et PreUpdate

L'annotation `@PrePersist` permet de déclencher une méthode avant l'ajout de l'enregistrement en base de données, et `@PreUpdate` de la déclencher avant la mise à jour, les traitements seront réalisés à chaque insertion / mise à jour de l'entité

Attention toutefois à ne pas rendre ces méthodes trop complexes. Cela violerait le principe de responsabilité de la classe : ce n'est pas le rôle d'un model de posséder de la logique métier. Préférer placer les traitements plus lourd dans le contrôleur, voir dans un service. (*exemple de traitement trop lourd : à la création d'un utilisateur, créer et hasher son mot de passe*)

```
@Entity
public class Utilisateur {

    @Id
    @GeneratedValue (strategy = GenerationType.IDENTITY)
    private Integer id;

    private String nom;

    @PrePersist
    @PreUpdate
    private void convertirNomEnMajuscules () {
        if (nom != null) {
            nom = nom.toUpperCase(); // Convertir le nom en majuscules
        }
    }
}
```

## >> Les propriétés Enum

Les énumérations peuvent être stockées de 2 manières : par leur valeur textuelle ou ordinal (*leur index*).

Par valeur textuel, le texte est affiché en base de donnée, même si c'est l'index qui est utilisé pour stocker l'information (*même consommation mémoire qu'en ordinal*)

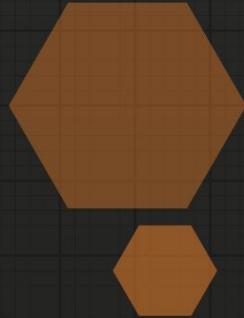
	id	statut	date
1	1	PANIER	2026-07-01 08:31:18
2	2	ENVOYEE	2027-06-06 10:02:11

Par valeur ordinal, on perd le texte dans la base de donnée, ce qui rend la maintenance plus difficile, mais il est alors inutile de lister toutes les valeur possible

	id	status	date
1	1	1	2026-06-30 13:48:44...
2	2	3	2027-01-31 13:49:53...

```
...
public class Commande {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Integer id;  
  
    @Enumerated(EnumType.STRING)  
    @Column(columnDefinition =  
"ENUM('PANIER', 'A_VALIDER', 'ENVOYE',  
'ANNULER')")  
    private Status status;  
  
    private LocalDateTime date;  
  
    public enum Status {  
        PANIER,  
        A_VALIDER,  
        ENVOYE,  
        ANNULER  
    }  
}
```

Doit être identique



# Créer des données de test

*Chapitre : Créer des contrôleurs*

## &gt;&gt; Configurer l'application

Le fichier chargé sera la concaténation de “data-”, de la propriété `spring.sql.init.platform` et de “.sql”

Dans le fichier application.properties, ajouter les configurations suivantes, afin que Spring exécute le fichier `data-donnees-de-test.sql`

```
spring.sql.init.platform = donnees-de-test  
spring.jpa.defer-datasource-initialization=true  
spring.sql.init.mode = always
```

data-donnees-de-test.sql

INSERT INTO .....

INSERT INTO .....

## &gt;&gt; Ajouter des produits de test

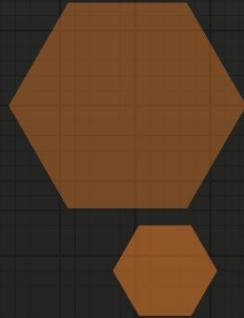
*resources/data-donnees-de-test.sql*

```
INSERT INTO produit (prix, nom, code, description) VALUES
(20.99, 'Ecouteurs Bluetooth sans Fil, AVOCE Casque Bluetooth 5.3 avec 4 ENC Réduction Bruit', 'AOVOCEA60B',
'la nouvelle génération de puce Bluetooth 5.3 personnalisée du écouteurs Bluetooth sans fil A60Pro est deux
fois plus rapide que la génération précédente (Bluetooth 5.2) et a une distance de transmission quatre fois
plus longue. Audio de haute qualité plus rapide et plus stable (portée de transmission jusqu'à 15 m). Grâce
à la fonction mémoire, le casque se connecte automatiquement à votre téléphone (après la première
connexion).'),
(11.99, 'Chargeur USB C 20W Rapide Prise', 'ChrgUSBC20WAnlikool', 'L\'adaptateur de multiple prise USB est
fabriqué en matériau ignifuge PC et fournit des protections multiples, y compris un court-circuit
anti-sortie, une surintensité intégrée, une surtension et une protection contre la surchauffe. Assure votre
sécurité et celle de vos appareils. Portables Telephone Caricatore usb c rapide et sûr.');
```



# Créer les DAO

Dans ce chapitre :



# Rappel sur les classes anonymes

*Chapitre : Créer les DAO*

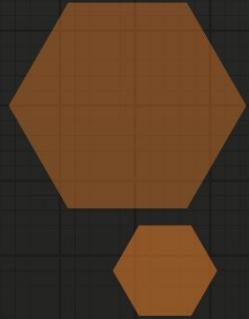
## >> Rappel sur les classes anonymes

Pour rappel : JAVA à la capacité de créer des classes anonymes à partir de l'**héritage** d'une classe ou de l'**implémentation** d'une **interface**.

```
public class Principale {  
    public static void main(String[] args) {  
  
        UneInterface objetAnonyme = new UneInterface() {  
            @Override  
            public String methodeAimplémenter() {  
                return "Ca marche";  
            }  
        };  
  
        System.out.println(objetAnonyme.methodeAimplémenter());  
    }  
}
```

```
interface UneInterface {  
    String methodeAimplémenter();  
}
```





# Ajouter un DAO

*Chapitre : Créer les DAO*

## &gt;&gt; Créer un DAO : ProduitDao

Un DAO JPA est une **interface** héritant de **JpaRepository** avec 2 **types génériques** :

une de nos **entités**, et le **type** de la **clé primaire** de cette **entité**.

*dao/ProduitDao*

```
import edu.fbansept.td.models.Produit;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ProduitDao extends JpaRepository<Produit, Integer> {

}
```

Une entité du  
package model

Le type de la clé  
 primaire



## &gt;&gt; Utilisation du DAO dans le contrôleur

Dans cet exemple on injecte notre dao, puis lors de l'appel de la route "/produit/liste", on appelle la méthode findAll() du dao, qui va chercher dans la base de données, l'intégralité des utilisateurs.

```
@RestController
public class ProduitController {

    @Autowired
    protected ProduitDao produitDao;

    @GetMapping("/produit/liste")
    public List<Produit> listeProduits (){
        return produitDao.findAll();
    }
    ...
}
```

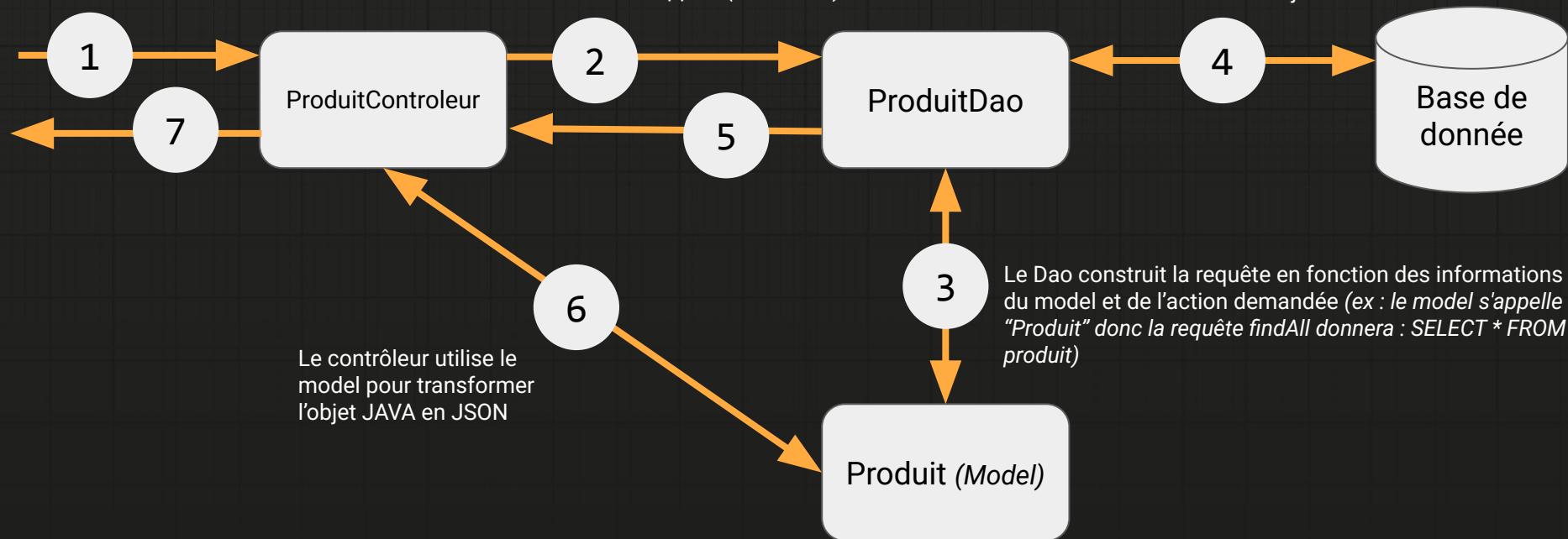


## >> Résumé de l'architecture en place

La requête est interceptée par le bon contrôleur  
en fonction de la route et sa méthode

Une méthode du DAO  
est appelé (ex : *findAll*)

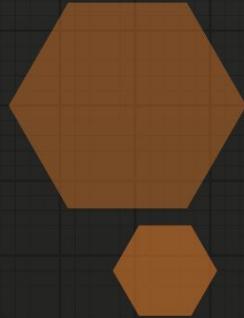
La requête est effectuée et le résultat  
est transformé en objet JAVA



# Créer un CRUD

*(Create Read Update Delete)*

Dans ce chapitre :



# Le code de retour des requêtes

*Chapitre : Créer un CRUD*

## >> Les codes de retour en cas de succès

Vous connaissez très certainement ce code de retour : 404, dont le message est “Page not Found”

Il en existe d'autres que nous allons utiliser si il n'y a pas d'erreurs :

- **200 OK** : Requête traitée avec succès. La réponse dépendra de la méthode de requête utilisée.
- **201 Created** : Requête traitée avec succès et création d'un document.
- **204 No Content** : Requête traitée avec succès mais pas d'information à renvoyer.

Notez que ces codes de retour sont inférieurs à 400. C'est le cas de tous les codes qui ne renvoient pas d'erreur

Vous pouvez regarder la liste exhaustive des codes de retour sur le site wikipedia :

[https://fr.wikipedia.org/wiki/Liste\\_des\\_codes\\_HTTP](https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP)



## >> Les codes de retour en cas d'erreur

Puis nous utiliserons certains code en cas d'erreur, certains seront directement renvoyé par Spring comme le code 500 ou 404, mais nous pouvons en être volontairement à l'origine :

- **400 Bad Request** : La syntaxe de la requête est erronée (ex : l'utilisateur n'a pas fourni de JSON dans le corps de la requête)
- **401 Unauthorized** : Une authentification est nécessaire pour accéder à la ressource. (ex : l'utilisateur n'est pas connecté alors qu'il devrait l'être)
- **403 Forbidden** : La requête ne peut pas être effectuée par cet utilisateur (ex : L'utilisateur n'a pas les droits nécessaires, ou les règles CORS sont enfreintes)
- **405 Method Not Allowed** : Méthode de requête non autorisée. (ex : l'URL existe bien en méthode GET, mais vous êtes en train de l'effectuer via la méthode POST)
- **500 Internal Server Error** : Une erreur est survenue sur le serveur (ex : vous avez déclenché une exception : vérifiez la console de Spring)



## >> Ajouter un code de retour : ResponseEntity

Dans un RestController il est optionnel de renvoyer un code de retour (*jusqu'ici on ne l'a pas fait*).

Afin 'ajouter ce dernier, il est nécessaire que la méthode utilisé ne renvoie pas une entité (ex : *Produit*) ou une valeur (ex : *Integer, String*), mais une instance de **ResponseEntity**

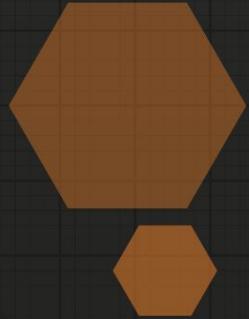
Dans l'exemple suivant, le type de retour de la méthode est changé.

La valeur renournée sera toujours une liste de produite, mais le code de retour pourra être modifié

Dans le cas présent nous retournons un code 200 signifiant que la requête a bien été effectuée.

```
@GetMapping("/produit/liste")
public ResponseEntity<List<Produit>> listeProduits (){
    return new ResponseEntity<>(produitDao.findAll, HttpStatus.OK);
}
```

Dans ce cas précis il n'y aura aucune conséquence puisque cette méthode ne peut renvoyer qu'un code 200 (si il n'y a pas de produit, alors il y aura un tableau vide, mais pas d'erreur 404)



# Read

*Chapitre : Créer un CRUD*

## >> Récupérer une entité par son identifiant

Comme nous l'avons déjà vu, l'annotation `@PathVariable` permet de récupérer un paramètre dans l'URL. Nous récupérons l'identifiant du produit, afin de le passer à la méthode `findById` et récupérer le produit correspondant.

`findById` ne renvoie pas un objet `Produit` mais un `Optional<Produit>`. La classe `Optional` permet de contenir un objet qui n'existe potentiellement pas (*ce qui serait par exemple le cas si l'on tente d'accéder à l'URL "/produit/99"*)

Si le produit n'est pas trouvé, notre variable `optionalProduit` serait vide, et nous devons donc retourner une erreur 404

```
@GetMapping("/produit/{id}")
public ResponseEntity<Produit> obtenirProduit (@PathVariable int id){

    Optional<Produit> optionalProduit = produitDao.findById(id);

    if(optionalProduit.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    return new ResponseEntity<>(optionalProduit.get(), HttpStatus.OK);
}
```



# >> Tests

TC New Request X

GET  http://localhost:8080/produit/1

Query Headers 2 Auth Body Tests Pre Run

Query Parameters

parameter value

Status: 200 OK Size: 603 Bytes Time: 382 ms

```
1 {  
2   "id": 1,  
3   "code": "AOVOCEA60B",  
4   "nom": "Ecouteurs Bluetooth sans Fil, AVOCE Casque"  
5 }
```

TC New Request X

GET  http://localhost:8080/produit/2

Query Headers 2 Auth Body Tests Pre Run

Query Parameters

parameter value

Status: 200 OK Size: 447 Bytes Time: 22 ms

```
1 {  
2   "id": 2,  
3   "code": "ChrgUSBC20WAnlikool",  
4   "nom": "Chargeur USB C 20W Rapide Prise",  
5   "description": "L'adaptateur de multiple prise USB e  
6 }
```

GET  http://localhost:8080/produit/3

Query Headers 2 Auth Body Tests Pre Run

Query Parameters

parameter value

Status: 404 Not Found Size: 0 Bytes Time: 30 ms

# Delete

*Chapitre : Créer un CRUD*

## >> Supprimer une entité par son identifiant

Sur le même principe, nous allons créer une méthode permettant de supprimer un utilisateur par son identifiant. Celle-ci sera disponible via une méthode DELETE.

```
@DeleteMapping ("/produit/{id}")
public String delete (@PathVariable int id) {

    produitDao.deleteById(id);

    return "Element supprimé";
}
```



## &gt;&gt; Améliorer le retour du service

Toujours sur le même principe, et afin d'améliorer notre service, nous renverrons un code 200 si l'utilisateur existe bien, et un code 404 dans le cas contraire.

```
@DeleteMapping ("/produit/{id}")
public ResponseEntity<Produit> delete(@PathVariable int id) {

    Optional<Produit> optionalProduit = produitDao.findById(id);

    if (optionalProduit.isEmpty()) {
        return new ResponseEntity<>(HttpStatus. NOT_FOUND);
    }

    produitDao.deleteById(id);

    return new ResponseEntity<>(HttpStatus. NO_CONTENT);
}
```

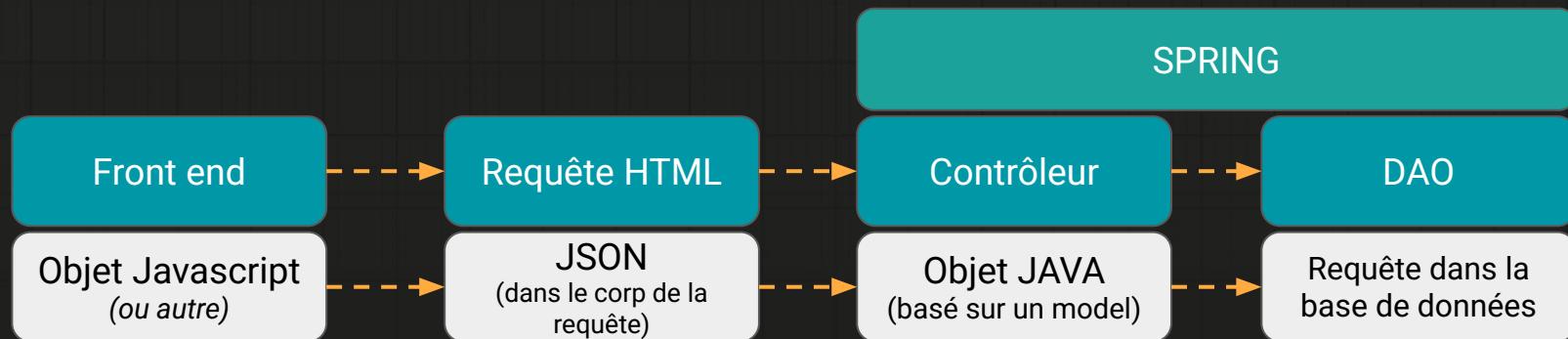


# Create

*Chapitre : Créer un CRUD*

Lors de l'envoi d'une requête de création :

- La partie front-end transforme un de ses objets en texte au format JSON
- La partie front-end ajoute ce texte au format JSON, au corps de la requête
- la requête est interceptée par l'un des contrôleurs de Spring
- Un objet JAVA est créé à partir du texte au format JSON du corps de la requête
- Cet objet sera utilisé par le DAO pour créer la requête en base de donnée



Lorsque l'on effectue une requête de création, le JSON représentant l'objet à créer est contenu dans le corps de la requête. La **méthode** doit alors posséder un **paramètre du type de l'objet** à créer avec l'annotation `@RequestBody`. Un **objet** Java est alors reconstitué en fonction des **propriétés** dans le JSON et de l'existence des setters correspondant dans la classe de l'objet à créer (`setId`, `setNom ...`)

*Corps de la requête (JSON)*

```
{  
    "nom" : "nom produit",  
    "code" : "code produit",  
    "description" : "description",  
    "prix" : 42,  
    "autre" : "autre"  
}
```

```
public class Produit {  
    protected Integer id;  
    protected String nom;  
    protected String code;  
    protected String description;  
    protected float prix;  
    // setter / getter ...  
}
```

L'absence de la propriété **id** dans le JSON fera qu'aucune valeur ne sera affectée à la propriété **id** de l'**objet Produit**. La propriété **id** de **produit** aura donc la valeur *null* (*comme n'importe quel objet non initialisé*)

Comme **Produit** n'a pas de setter "`setAutre`" la propriété **autre** du JSON sera ignorée.

```
@PostMapping("/produit")  
public ResponseEntity<Produit> ajout(@RequestBody Produit produit) {  
  
    produitDao.save(produit);  
  
    return new ResponseEntity<>(produit, HttpStatus.CREATED);  
}
```

## >> Ajouter une entité

La création et la mise à jour d'une entité peut se faire via la même méthode. Si l'entité envoyée possède un identifiant et que celui-ci existe déjà dans la table, alors ce sera une mise à jour, sinon se sera une insertion.

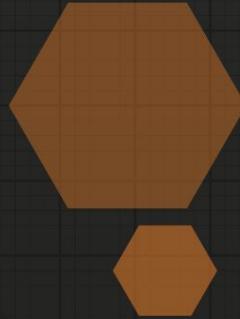
Afin de s'assurer d'effectuer une insertion et non une modification.

Et éviter le cas où le front-end aurait ajouté l'identifiant de l'objet dans le JSON (*la propriété id*), il peut être judicieux d'affecter la valeur null à l'objet à insérer pour être sur que c'est bien une insertion qui sera effectuée

```
@PostMapping("/produit")
public ResponseEntity<Produit> save(@RequestBody @Valid Produit produit) {
    produit.setId(null);
    produitDao.save(produit);
    return new ResponseEntity<>(produit, HttpStatus.CREATED);
}
```

Cela reste optionnel, mais il est intéressant de renvoyer l'objet initialement envoyé par le front, car cet objet aura sa propriété id affectée par l'auto-incrémentation (*via la méthode save*). Le front pourra alors utiliser cette information.

Le code à renvoyer est un code 201 “created” expliquant au front-end que l'opération est un succès et qu'il y a eu une création de ressource



# Update via PUT

*Chapitre : Créer un CRUD*

## >> PUT : mise à jour complète

Pour rappel, une opération d'update peut être réalisé via une méthode PUT en passant un objet entier, ou via une méthode PATCH en passant un objet partiel.

```
@PutMapping("/produit/{id}")
public ResponseEntity<Produit> update(
    @PathVariable int id,
    @RequestBody Produit produit) {

    Optional<Produit> optionalProduit = produitDao.findById(id);

    if (optionalProduit.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    produit.setId(id);

    produitDao.save(produit);

    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}
```

Par convention PUT et PATCH nécessite que l'identifiant de l'objet à mettre à jour fasse partie de l'URL

On affecte donc l'identifiant fourni dans l'URL à l'identifiant de l'objet passé en JSON

Inutile d'afficher du contenu, un code 204 "no content" suffit

## >> Ne pas mettre à jour certains champs

On peut empêcher la mise à jour de certains champs. cela ne transforme pas forcément la méthode PUT en méthode PATCH, mais permet de restreindre les champs qui peuvent être mise à jour.

```
@PutMapping("/produit/{id}")
public ResponseEntity<Produit> update(
    @PathVariable int id,
    @RequestBody Produit produit) {

    Optional<Produit> optionalProduit = produitDao.findById(id);

    if (optionalProduit.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    produit.setId(id);
produit.setCode(optionalProduit.get().getCode());

    produitDao.save(produit);

    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}
```

Une première méthode consiste à affecter à la propriété à conserver, l'ancienne valeur de l'objet

## >> Ne pas mettre à jour certains champs via le model

Ce type de restriction peut également être appliquée via les paramètres insertable et updatable de la l'annotation `@Column`.

Mais la restriction sera globale, alors que la méthode précédente permet de l'appliquer par contrôleur, ou selon une logique basé sur les droits de l'utilisateur (ex : *un administrateur peut modifier le code d'un article lors qu'un gestionnaire non*)

```
@Entity  
public class Produit {  
  
    ...  
  
    @Column(updatable = false)  
    protected String code;  
    ...  
}
```

On indique ici que la méthode `save` n'affectera jamais le champs lors d'une mise à jour.

Il reste néanmoins possible de lui affecter une valeur lors de la création (*sauf dans le cas où l'on ajouterait le paramètre `insertable = false`*)



## >> Cas où l'objet envoyé est incomplet

Une méthode de type **PUT** prend en paramètre un objet entier.

L'absence de propriété dans cet objet (*donc dans le JSON puisque l'objet est créé à partir de ce dernier*) entraîne une initialisation à sa valeur par défaut selon son type (*null pour un objet, 0 pour un nombre, false pour un boolean...*).

Sauf dans les cas suivants :

Une valeur par défaut est affectée à la propriété de l'objet :

```
protected String nom = "Sans nom";
```

Si le JSON ne possède pas du tout la propriété concernée alors c'est "Sans nom" qui sera affecté

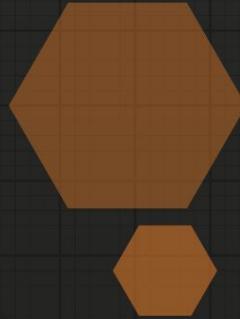
Si le JSON possède la propriété concernée est qu'elle a la valeur null, alors c'est bien null qui sera affecté

Le JSON possède la propriété concernée mais avec une valeur null et une valeur par défaut est définie en base de donnée

```
@Column(columnDefinition = "varchar(255) default 'Sans nom'")  
protected String nom;
```

Que le JSON possède la propriété et qu'elle est null, ou qu'elle n'existe pas,  
fera que l'objet aura cette valeur par défaut ('Sans nom') une fois inséré dans la base de donnée





# Update via PATCH

*Chapitre : Créer un CRUD*

## >> Note sur ce type de mise à jour

Ce type de requête n'est pas obligatoire et nécessite une bonne compréhension de comment les objets s'initialisent, de l'interaction entre le JSON et l'objet créé.

Il soulève des questions de conception :

Que se passe t-il quand l'utilisateur envoie des données partiel, ou nulles ?

Est ce que je ne rend pas l'utilisation de mon API plus compliquées inutilement ?

**Si vous voulez opter pour une API simple à utiliser, il est conseillé de se contenter d'une mise à jour via PUT.**

Nous allons quand même explorer une solution "optimale" , mais qui nécessite un certains nombre de digression pour pouvoir être mise en place, au final la requête devra :

- garder l'ancienne valeur si le JSON envoyé ne possède pas une certaine propriété
- affecter null à la valeur si la propriété vaut null (et envoyer une erreur si cela est impossible)
- permettre la validation des données (voir chapitre suivant) et retourner une erreur 400 en cas d'invalidation
- respecter toutes les contraintes du model



## >> Mise à jour partiel

Ce type de traitement est adapté dans le cas où l'on souhaite permettre à l'utilisateur de ne mettre à jour que certains champs via une méthode **PATCH**.

Si le JSON ne contient pas la propriété à mettre à jour, ou si celle-ci est null, l'ancienne valeur en base de données est utilisée

Mais il est important de faire la distinction entre une propriété que l'on désire ignorer, et une propriété à laquelle on veut volontairement affecter la valeur null (ex : on désire supprimer le code d'un produit)

La méthode PATCH peut donc difficilement faire la différence entre les 2 cas

```
@PatchMapping ("/produit/{id}")
public ResponseEntity<Produit> partialUpdate (
    @PathVariable int id,
    @RequestBody Produit produit) {

    Optional<Produit> optionalProduit = produitDao.findById(id);

    if (optionalProduit.isEmpty()) {
        return new ResponseEntity<>(HttpStatus. NOT_FOUND);
    }

    Produit produitBdd = optionalProduit.get();

    if(produit.getNom() != null)
        produitBdd.setNom(produit.getNom());
    if(produit.getCode() != null)
        produitBdd.setCode(produit.getCode());
    if(produit.getDescription() != null)
        produitBdd.setDescription(produit.getDescription());
    if(produit.getPrix() != 0)
        produitBdd.setPrix(produit.getPrix());

    produitDao.save(produitBdd);

    return new ResponseEntity<>(HttpStatus. NO_CONTENT);
}
```

Dans le JSON, que la propriété est été absente ou null le résultat est le même : on garde l'ancienne valeur



## >> Mise à jour partiel non basée sur un model

La solution au problème précédent peut être résolu en ne transférant pas les informations du JSON dans un model (*qui aura par nature ses propriétés initialisées, et par conséquent, on ne distinguera donc pas une propriété qui était inexistante d'une propriété null*)

On utilisera à la place un objet *Map<String, Object>* qui permet de récupérer les propriétés d'un Objet JSON

*String* étant le nom de la propriété (*nom, code, prix...*)  
Et *Object* sa valeur (*un texte, un nombre, un object, un booléen, un tableau*)

Dans ce cas si la propriété n'existe pas dans le JSON c'est l'ancienne valeur qui est utilisée, mais si le front envoie un objet avec une propriété volontairement nulle, c'est bien la valeur nulle qui sera affectée (*sauf en cas de valeur par défaut de la base de donnée sur la colonne correspondante*)

```
@PatchMapping("/produit/{id}")
public ResponseEntity<Produit> partialUpdate(
    @PathVariable int id,
    @RequestBody Map<String, Object> updates) {

    Optional<Produit> optionalProduit = produitDao.findById(id);

    if (optionalProduit.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    Produit produitBdd = optionalProduit.get();

    if(updates.containsKey("nom"))
        produit.setNom((String) updates.get("nom"));
    if(updates.containsKey("prix"))
        produit.setPrix((float) updates.get("prix"));
    ...

    produitDao.save(produitBdd);

    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}
```



## >> Ajouter la validation des données

La solution précédente fonctionne mais n'est pas compatible avec la validation des donnée et l'annotation `@Valid` ou `@Validated` (*voir chapitre suivant*) puisque l'on n'utilise pas un model

Une solution consiste donc à utiliser l'objet construit via la méthode précédente et le repasser paramètre d'une méthode avec l'annotation `@Valid` ou `@Validated`

```
@PatchMapping("/produit/{id}")
public ResponseEntity<Produit> partialUpdate(@PathVariable int id, @RequestBody Map<String, Object> updates) {
    ...
    if(updates.containsKey("nom"))
        produit.setNom((String) updates.get("nom"));
    if(updates.containsKey("prix"))
        produit.setPrix((float) updates.get("prix"));
    ...

    return partialUpdateValidation(produit);
}

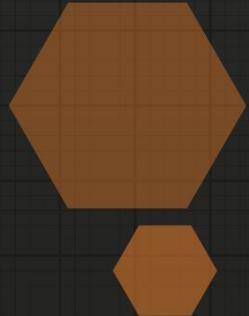
private ResponseEntity<Produit> partialUpdateValidation(@Valid Produit produit) {
    produitDao.save(produit);
    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}
```

## >> Changer l'erreur 500 pour une erreur 400

Contrairement à la validation faite dans une action du contrôleur (PostMapping, PutMapping, PatchMapping) qui aurait retourné une erreur 400, le fait que la validation est été faite dans une méthode, fait que l'on obtient une erreur 500. Pour modifier ce comportement, il est nécessaire de créer une classe capable d'intercepter les erreurs de validation. Ce concept est abordé un peu plus loin, mais pour plus de compréhension la méthode à y rajouter ce trouve ci-dessous (*mais elle sera redonner dans le chapitre correspondant*)

*IntercepteurExceptionGlobal (nouveau fichier)*

```
@ControllerAdvice
public class IntercepteurExceptionGlobal {
    ...
    // Intercepte les erreurs de validation des contraintes
    @ExceptionHandler(ConstraintViolationException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ResponseBody
    public Map<String, String>
    handleConstraintViolationException(ConstraintViolationException ex) {
        Map<String, String> errors =new HashMap<>();
        for (ConstraintViolation<?> violation : ex.getConstraintViolations()) {
            errors.put(violation.getPropertyPath().toString(), violation.getMessage());
        }
        return errors;
    }
}
```



# Tester avec javascript

*Chapitre : Créer un CRUD*

## &gt;&gt; Erreur CORS

Avant de pouvoir effectuer des requêtes avec le navigateur via Javascript, il est nécessaire d'avertir le navigateur que le serveur autorise les reqête provenant d'une origine différente (cad un domaine différent)

Si vous testez le script ci-dessous vous obtiendrez dans la console du navigateur une erreur ressemblant à :

Access to fetch at 'http://localhost:8080/produit/liste' from origin 'null' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.

*test-javascript.html (nouveau fichier)*

```
<script>
  fetch("http://localhost:8080/produit/liste")
    .then((res) => res.json())
    .then((listeProduit) => {
      listeProduit.forEach((produit) => {
        console.log(produit.nom);
      });
    })
</script>
```

Pour régler ce problème, ajoutez l'annotation `@CrossOrigin` au dessus de vos RestController :

```
@RestController
@CrossOrigin
public class ProduitController {

  @Autowired
  protected ProduitDao produitDao;

  @GetMapping("/produit/liste")
  public ResponseEntity<List<Produit>> listeProduits (){
    return new ResponseEntity<>(produitDao.findAll(), HttpStatus.OK);
  }
  ...
}
```



## >> Tester l'ajout avec javascript

Cet autre script ajoute un produit via la route POST du contrôleur de l'entité produit (*lorsque le bouton est cliqué*)

On précise que la méthode est de type POST (*elle est de type GET par défaut*)

Que le contenu est du JSON

On ajoute dans le corps de la requête un objet javascript transformé en texte au format JSON

On attend le retour du serveur Spring

En cas de retour avec un code  $\geq 400$  la console affiche un message d'erreur avec le statut

Sinon le json est transformé en objet Javascript

Puis il affiché dans la console

Si une erreur survient (*par exemple un problème de connexion ou d'url*), un autre message d'erreur est affiché

```
<script>
    function ajoutProduit() {
        const produit = {
            code: "nouvcode",
            nom: "nouveau produit",
            description: "nouvelle description",
            prix: 42,
        };

        fetch("http://localhost:8080/produit", {
            method: "POST",
            headers: { "Content-type": "application/json" },
            body: JSON.stringify(produit),
        })
            .then((response) => {
                if (!response.ok) {
                    throw new Error("La requête a échouée avec le statut " + response.status);
                }
                return response.json();
            })
            .then((produit) => {
                console.log(produit);
            })
            .catch((error) => {
                console.error("Il y a eu un problème avec l'opération fetch: " + error.message);
            });
    }
</script>

<button onclick="ajoutProduit()">Ajouter un produit</button>
```

Attention : lancer une erreur ici est important afin de passer directement au bloc catch (*sans tenter de convertir en JSON un corps de requête surement vide*)

## >> Tester la suppression avec javascript

L'exemple ci-contre montre comment supprimer un produit.

La différence avec l'exemple précédent est le type de la méthode (DELETE)

La présence de l'identifiant de l'entité à supprimer dans l'URL

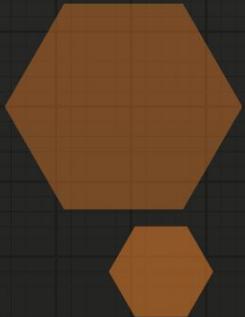
Et le fait qu'il n'y a pas besoin de corps ni de renseigner le type de contenu

Note : appuyer 2 fois sur le bouton supprimer, entraîne automatiquement une erreur 404, puisque le produit n'existe plus

```
<script>
    function supprimerProduit() {
        fetch("http://localhost:8080/produit/2", {
            method: "DELETE",
        })
        .then((response) => {
            if (!response.ok) {
                throw new Error(
                    "La requête a échouée avec le statut " + response.status
                );
            }
            return response.json();
        })
        .then((produit) => {
            console.log(produit);
        })
        .catch((error) => {
            console.error(
                "Il y a eu un problème avec l'opération fetch: " + error.message
            );
        });
    }
</script>

<button onclick="supprimerProduit()">Supprimer un produit</button>
```





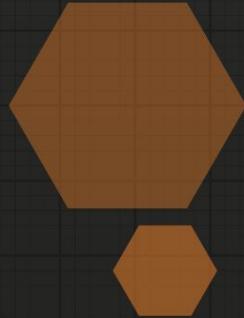
# Tester avec SWAGGER

*Chapitre : Créer un CRUD*

>> TODO

<https://medium.com/@f.s.a.kuzman/using-swagger-3-in-spring-boot-3-c11a483ea6dc>





## Gérer les dates de création et de modification automatiquement

*Chapitre : Validation des données*

## &gt;&gt; Laisser Spring gérer la date de création et de modification

```
@Entity  
//important pour @CreatedDate et @LastModifiedDate  
@EntityListeners(AuditingEntityListener.class)  
public class Produit {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    private String nom;  
  
    @CreatedDate  
    @Column(updatable = false, nullable = false)  
    private LocalDateTime createdAt;  
  
    @LastModifiedDate  
    private LocalDateTime updatedAt;  
}
```

*Fichier principal (src/main/java/vos packages/VotreApplication)*

```
@SpringBootApplication  
@EnableJpaAuditing//important pour  
@CreatedDate et @LastModifiedDate  
public class  
CoursFilRougeSpring3TdApplication{  
    ...  
}
```



## >> Empêcher l'utilisateur de modifier la date de création

Dans le contrôleur correspondant, modifiez la méthode PUT en ajoutant une instruction qui réaffecte l'ancienne date de création

Ainsi si l'utilisateur envoie un objet avec une date de création différente, elle n'affectera pas l'entité

*Note : il pourrait être possible de vérifier par la suite si l'utilisateur est administrateur et ainsi lui permettre de modifier cette date de création*

*controllers/ProduitController*

```
@PutMapping("/produit/{id}")
public ResponseEntity<?> modifierProduit (
    @RequestBody Produit produit,
    @PathVariable int id){

    produit.setId(id);

    Optional<Produit> optionalAncienProduit = produitDao.findById(id);

    if(optionalAncienProduit.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }

    produit.setDateCreation(optionalAncienProduit.get().getDateCreation());

    produitDao.save(produit);

    return new ResponseEntity<>(produit, HttpStatus.OK);
}
```



## &gt;&gt; @CreationTimestamp et @UpdateTimestamp

Il est également possible d'utiliser un Timestamp (nombre de secondes écoulée depuis le 01/01/1970), seulement ce n'est pas recommandé car l'horodatage est absent. Mais il peut être nécessaire sur des projets existants.

*models/Produit*

```
import org.hibernate.annotations.CreationTimestamp;
import org.hibernate.annotations.UpdateTimestamp;

import java.time.Instant;

@Getter
@Setter
@Entity
@EntityListeners(AuditingEntityListener.class)
public class Produit {

    ...

    @CreationTimestamp
    private Instant dateCreation;

    @UpdateTimestamp
    private Instant dateModification;

}
```

Dorénavant, ajouter un produit permet d'affecter une date dans les champs date\_creation et date\_modification de la base de données.

Et la mise à jour de ce produit entraînera la modification automatique du champ date\_modification de la base de données.

## >> Modifier le fichier de jeu de donnée

Afin de ne pas provoquer d'erreur il est nécessaire de modifier le fichier de jeu de donnée afin de remplir ces 2 champs.

Vous pouvez y ajouter des valeur fixe au format : '1987-06-23 22:30:42.000000' ou utiliser la méthode CURRENT\_TIMESTAMP() de Mysql qui retourne le timestamp de la date actuelle.

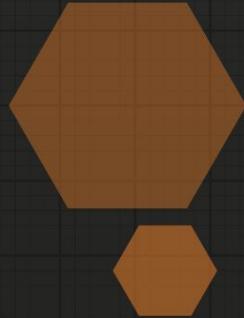
*data-donnees-de-test.sql*

```
INSERT INTO produit ( ...ancien champs... date_creation, date_modification) VALUES  
(...anciennes valeurs..., CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP(),  
(...anciennes valeurs..., CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP());
```



# Validation des données

Dans ce chapitre :



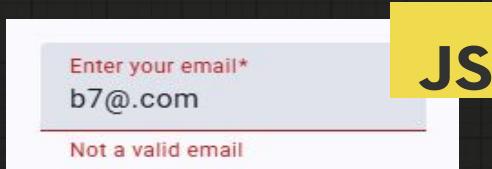
# Principe de la validation des données

*Chapitre : Validation des données*

## >> Validation des données côté frontend VS backend

La validation des données peut se faire côté Front-end (avec *des validations html intégrées ou en javascript généralement pour un site web*) ou côté Backend (Avec Spring dans notre cas)

Mais si une **validation** est fait du **côté Front-end** il ne s'agit que d'ergonomie et d'expérience utilisateur (UI/UX) **et non de sécurité**



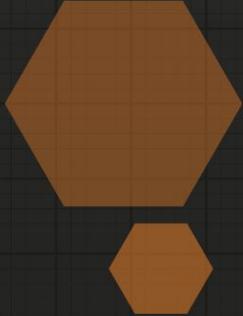
Il est toujours possible de désactiver javascript, ou encore d'utiliser les même logiciel que nous utilisons pour nos tests (JetClient, Thunder client, Postman ...)

Il est donc toujours nécessaire de valider les données côté Backend, car on ne fait jamais confiance aux données envoyée par un utilisateur. Il n'a pas accès au backend alors qu'il peut toujours modifier le front end. Mais la validation côté Front-end est un plus

Certaines contraintes sur la base de données garantissent des données plus cohérentes, mais la validation des données que nous allons aborder concernent les valeurs fournies en JSON à nos modèles

Ces contraintes peuvent entrer en doublon avec certaines contraintes de la base de données, et ce n'est pas un problème.

Exemple : un champs de la base de données qui n'est pas nullable. La validation des données envoyés en JSON peut déjà vérifier cette contrainte avant son envoi en base de données. Le doublon est intéressant : un autre backend connecté à cette base de donnée n'a peut être pas de validation, et il est plus intéressant d'envoyer une erreur explicite générée par Spring qu'une erreur SQL difficilement lisible (et soulager la base de donnée d'une requête inutile)



# Valider les données avec Spring Validation

*Chapitre : Validation des données*

## &gt;&gt; Ajouter la dépendance

*pom.xml*

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

N'oubliez pas de mettre à jour l'application via l'icône  qui apparaît alors en haut à droite.

# >> Ajouter les règles de validation sur le model

Il existe un certain nombre de validateurs fournis par Spring Validation.

models/Produit

```
...
public class Produit {
    ...
    @NotNull(message = "Le code ne peut pas être null.")
    @NotBlank(message = "Le code ne peut pas être vide.")
    @Column(nullable = false, unique = true)
    protected String code;

    @NotNull(message = "Le nom ne peut pas être null.")
    @NotBlank(message = "Le nom ne peut pas être vide.")
    @Size(max = 100, message = "La longueur du nom ne peut pas dépasser 100 caractères.")
    @Column(nullable = false, length = 100)
    protected String nom;

    @Column(columnDefinition = "TEXT")
    protected String description;

    @DecimalMin(value = "0", message = "Le prix doit être supérieur à 0")
    //Note 1 : la contrainte impose un nombre supérieur à 0 (et non supérieur ou égale à 0)
    //Note 2: float étant un primitif, si il n'est pas renseigné, sa valeur sera de 0, donc @NotNull n'aura aucun effet
    protected float prix;
    ...
}
```

La propriété **message** des annotations n'est pas obligatoire, mais elle permet au client de comprendre quelle est l'erreur

*Note : dans le cas où le backend n'est censé communiquer qu'avec un frontend bien défini c'est inutile puisque le client est censé valider de son côté. La validation côté serveur ne servant qu'à bloquer une tentative malveillante*

*Mais dans le cas d'une API plus globale cela reste intéressant à implémenter, voir gérer une application front end problématique*



## &gt;&gt; Activer la validation dans le contrôleur

Dans le contrôleur, ajouter dans la rout POST et PUT l'annotation `@Valid`. Si l'un des validateurs n'est pas respecté la requête retournera un code 400

*controllers/ProduitController*

```
...
@PostMapping("/produit")
@JsonView(VueProduit.class)
public ResponseEntity<?> ajouterProduit (@Valid @RequestBody Produit produit){
    produitDao.save(produit);
    return new ResponseEntity<?>(produit, HttpStatus.CREATED);
}

@GetMapping("/produit/{id}")
public ResponseEntity<?> modifierProduit (
    @Valid @RequestBody Produit produit,
    @PathVariable int id){
    ...
}
...
```



## >> Valider par expression régulière (regex)

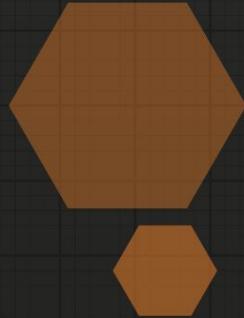
Même si le validateur `@Email` existe, il est très basique et ne vérifie que la présence de l'@ entre 2 caractères alphanumérique.

Le validateur `@Pattern` permet de rendre ce test plus précis

*models/Utilisateur (nouveau fichier)*

```
@Getter  
@Setter  
@Entity  
public class Utilisateur {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    protected Integer id;  
  
    @NotNull(message = "L'email ne peut pas être null.")  
    @NotBlank(message = "L'email ne peut pas être vide.")  
  
    //@Email(message = "L'adresse email n'est pas valide.")  
    //note : le validateur par défaut ne prend pas en compte les extensions (ex : a@a est valide)  
    //Pour un validateur plus précis :  
    //      invalide : "a", "a@", "a@a", "a@a.", "a@a.a", "a@a.aaaaaaaa", "a@a_.aaaaaaaa"  
    //      valide  : "a@a.aa", "a@a.aaaaaa", "a_-.@a.-.aaa"  
    @Pattern(  
        regexp = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.\w{2,6}$",  
        message = "L'adresse email n'est pas valide."  
  
    private String email;  
}
```





# Gérer globalement les retours des exceptions

*Chapitre : Validation des données*

## &gt;&gt; Ajouter les messages d'erreur de validation à la réponse

Actuellement les messages ne sont pas ajouté à la réponse.

Pour cela nous allons ajouter une nouvelle classe (*par exemple au même niveau que la classe principale*)

Cette classe aura pour but d'attraper les exceptions de type `MethodArgumentNotValidException` et de modifier le corps de la réponse

Elle parcourt toutes les erreurs de l'exception et crée une Map qui sera transformée en JSON

*IntercepteurExceptionGlobal (nouveau fichier)*

```
@ControllerAdvice
public class IntercepteurExceptionGlobal {

    //Intercepte toutes les erreurs de validation du model (@NotNull, @NotBlank, @Size ...)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ResponseBody // Assure que la réponse est envoyée au format JSON (dans le corps de la réponse)
    public Map<String, Object> handleMethodArgumentNotValidException(MethodArgumentNotValidException ex) {
        Map<String, Object> errors = new HashMap<>();
        ex.getBindingResult().getAllErrors().forEach((error) -> {
            String fieldName = ((FieldError) error).getField();
            String errorMessage = error.getDefaultMessage();
            errors.put(fieldName, errorMessage);
        });
        return errors;
    }
}
```

## >> Ajouter les messages d'erreur de validation à la réponse dans le cas d'une validation hors requête

Il est également possible que les exceptions soient levée à cause d'une validation manuelle. (*via une annotation @Valid ou @Validated sur un objet dans une méthode de service ou de composant en dehors d'une action d'un contrôleur*) c'est le cas par exemple pour la solution proposée précédemment pour la méthode PATCH.

Dans de cas l'exception n'est pas une MethodArgumentNotValidException mais une ConstraintViolationException, qui se gère différemment

*IntercepteurExceptionGlobal (nouveau fichier)*

```
@ControllerAdvice
public class IntercepteurExceptionGlobal {
    ...
    // Intercepte les erreurs de validation des contraintes
    @ExceptionHandler(ConstraintViolationException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ResponseBody
    public Map<String, String>
    handleConstraintViolationException(ConstraintViolationException ex) {
        Map<String, String> errors =new HashMap<>();
        for (ConstraintViolation<?> violation : ex.getConstraintViolations()) {
            errors.put(violation.getPropertyPath().toString(), violation.getMessage());
        }
        return errors;
    }
}
```



## >> Attraper les erreurs d'intégrité des données

Sur le même principe mais sur un autre sujet, nous pouvons également attraper les erreur de type : **DataIntegrityViolationException** qui sont levé lorsqu'une contrainte de la base de données n'est pas respecté (*champs unique en doublon, suppression d'un enregistrement lié à une contrainte de clé étrangère ...*)  
Dans ce cas le code d'erreur sera modifié pour **409 conflict** plus précise que l'erreur **500 Internal Server Error** actuelle  
Vous pouvez tester en insérant en produit avec un code identique à un autre produit par exemple

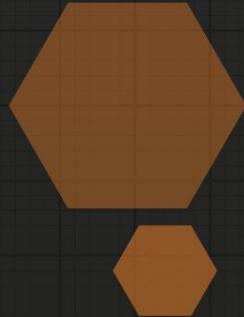
### *IntercepteurExceptionGlobal*

```
@ControllerAdvice
public class IntercepteurExceptionGlobal {

    ...

    //Intercepte toutes les erreurs de contraintes dans la base de données
    // (contrainte de clé étrangère, champs unique ...)
    // Note : attention attrape également les contraintes nullable = false
    // si l'annotation @NotNull est manquantes dans le modèle par exemple
    @ExceptionHandler(DataIntegrityViolationException.class)
    @ResponseStatus(HttpStatus.CONFLICT) // Code de retour
    @ResponseBody // Assure que la réponse est envoyée au format JSON (dans le corps de la réponse)
    public Map<String, Object> handleDataIntegrityViolationException(DataIntegrityViolationException ex) {
        return Map.of("message", "Une violation de contrainte d'intégrité de données a été détectée");
    }
}
```





# Valider partiellement des données

*Chapitre : Validation des données*

# >> L'annotation @Validated

Il est fréquent qu'une partie des données ne soient pas les même à valider lors de la création, la mise à jour d'un objet, ou tout autre opérations.

Par exemple, le mot de passe d'un utilisateur ne devrait pas être null lors de la création d'un utilisateur, mais pourrait l'être lors de la modification d'un utilisateur car on ne souhaite pas le fournir à chaque fois.

Dans ce cas il est possible d'ajouter une propriété groups à nos annotation de validation. C'est groupes prennent en paramètre une [liste de nom de classe ou d'interface](#), qui ne serviront qu'à filtrer les contraintes à respecter selon la validation partielle choisie

Dans cet exemple, lors de la création d'un utilisateur, son mot de passe ne doit pas être vide ou null, mais lors la modification d'un utilisateur, le mot de passe n'est pas vérifiée (*car la modification du mot de passe se fait dans une autre action par exemple*)

## controller

```
@PutMapping("/produit/{id}")
public ResponseEntity<Produit> modification(
    @PathVariable Integer id,
    @RequestBody @Validated(Produit.OnUpdate.class) Produit produitEnvoye) {
    ...
}

@PostMapping("/produit")
public ResponseEntity<Produit> modification(
    @RequestBody @Validated(Produit.OnCreate.class) Produit produitEnvoye) {
    ...
}
```

## model

```
public class Utilisateur {

    public interface OnUpdate { }
    public interface OnCreate { }

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    protected Integer id;

    @Column(nullable = false)
    @NotBlank(groups = {OnUpdate.class, OnCreate.class})
    @Email(groups = {OnUpdate.class, OnCreate.class})
    protected String email;

    @Column
    @NotBlank(groups = {OnCreate.class})
    protected String password;
}
```

# Créer les jointures

Dans ce chapitre :

## >> Les jointures

Il est important de comprendre les diagrammes de classes UML et/ou les MCD Merise afin de réaliser cette partie.

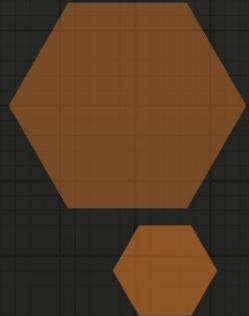
En effet, une fois comprise, les liaisons entre les entités ne seront plus qu'une simple retranscription du schéma en annotations.

Les jointure permettent de compléter un objet A avec l'objets B dont il est composé. (Par exemple compléter un Utilisateur avec un Statut). "Un Utilisateur ne possède qu'un statut", la relation est @ManyToOne vers Statut

A l'inverse du point de vue du statut sa relation est "Un Statut est possédé par plusieurs Utilisateur" la relation est @OneToMany vers Utilisateur.

Au niveau de la base de donnée, les jointure permettent de préciser les clés étrangères et crée des contraintes. Ce mécanisme est non modifiable et les règle bien précises.





# Les Jointure ManyToOne

*Chapitre : Créer les jointures*

## >> Les jointures ManyToOne

Une jointure **ManyToOne** est la représentation de la **liaison** suivante en **diagramme de classe UML** :  
*(Attention : c'est une relation ManyToOne uniquement du point de vue d'Utilisateur)*



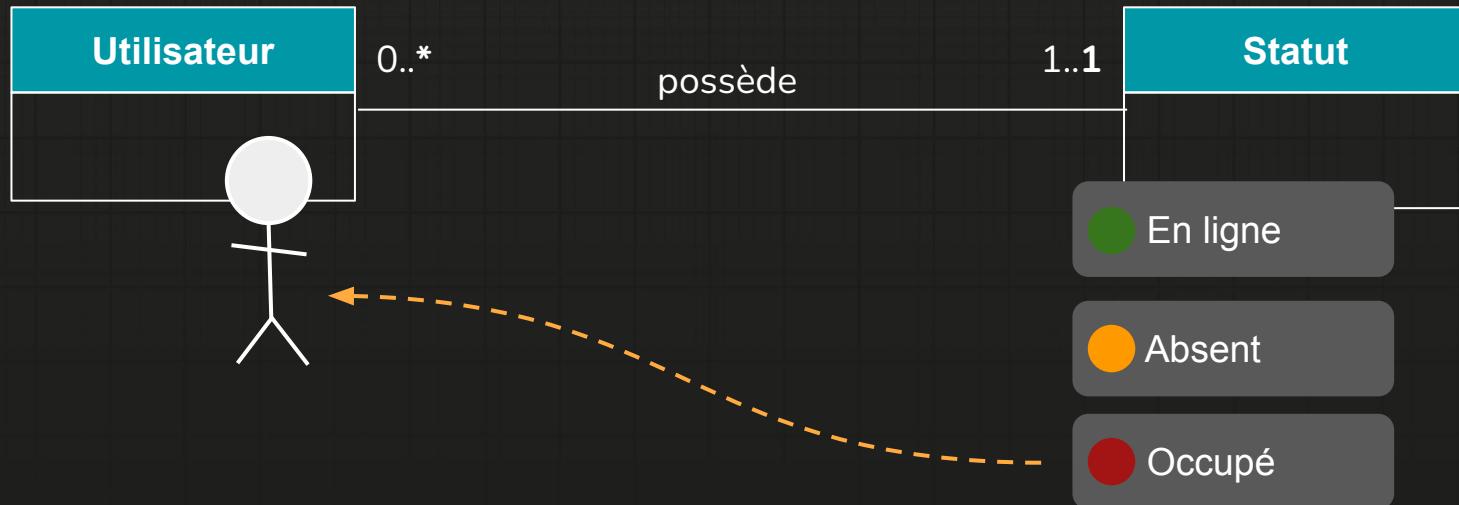
Ou en **Modèle Conceptuel de Données** (merise) :



## >> Lecture de la liaison

Pour rappel : la liaison **du point de vue d'utilisateur** se lit ainsi :

**“Un utilisateur (*n’importe lequel*) possède 1 et 1 seul statut”**  
(il en a au moins 1 et au maximum 1)

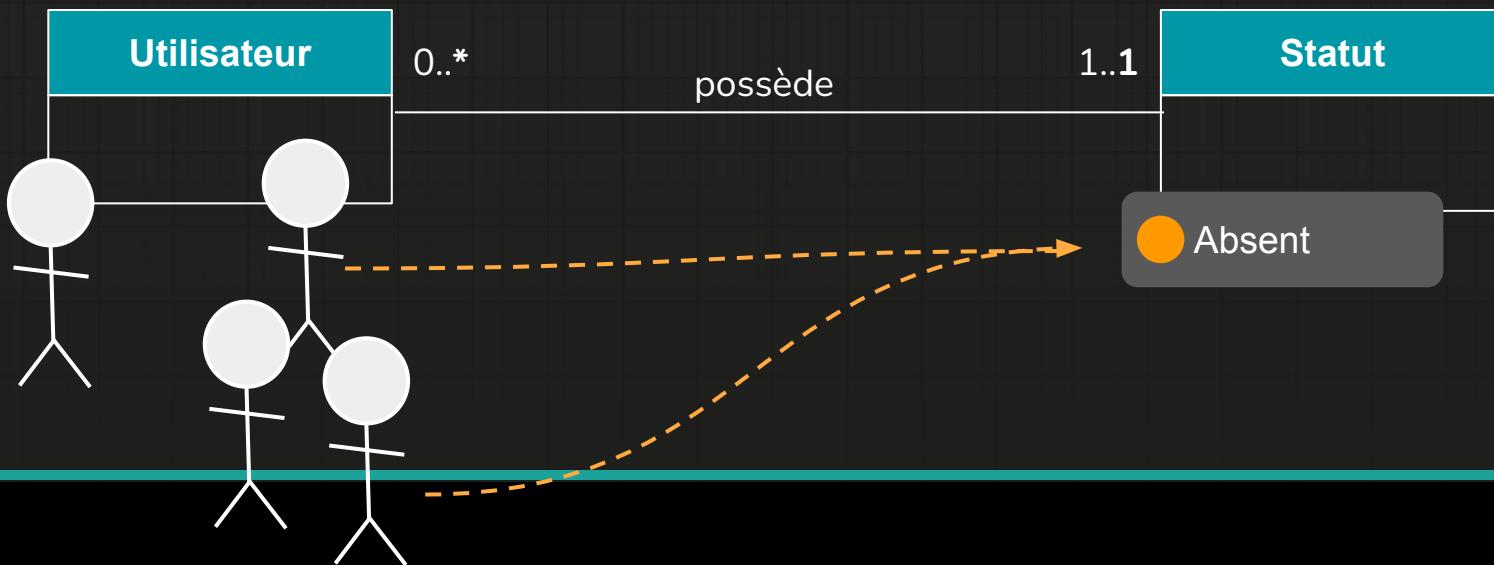


## >> Lecture de la liaison inverse (OneToMany)

Pour rappel : cette même liaison mais du point de vue de statut se lit ainsi (mais c'est un OneToMany)

**“Un statut (*n’importe lequel*) possède 0 ou plusieurs utilisateurs”**

(il peut ne pas en avoir, n’en avoir qu’un ou plusieurs)



## >> Les jointures ManyToOne

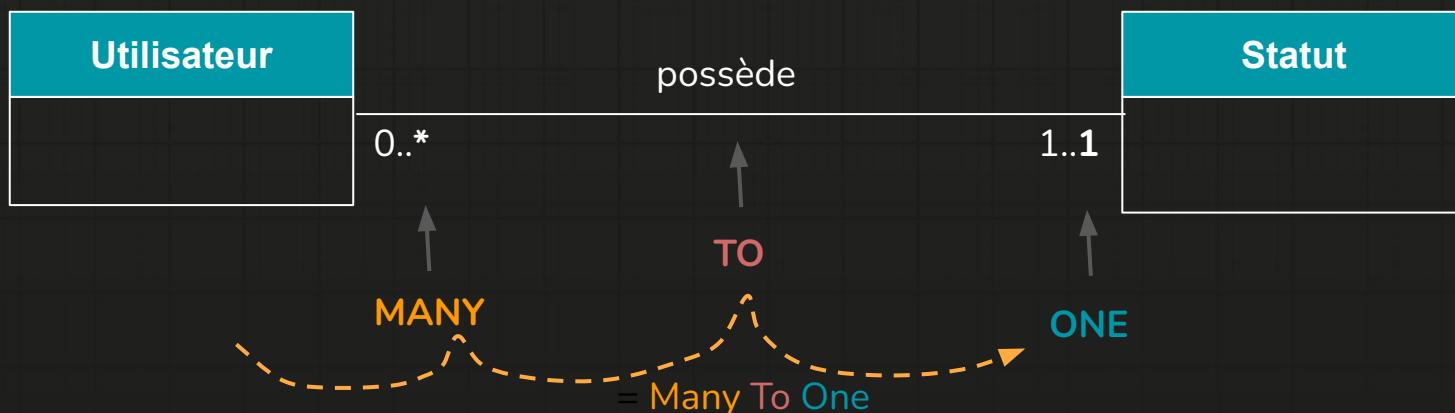
Pour rappel : les **cardinalités minimum** n'ont **pas d'importances** pour définir le **type de liaison**.  
Seules les **cardinalités maximum** permettent de la définir.

Les 2 liaisons ci-dessous sont toutes deux des relation **ManyToOne** pour l'**Utilisateur**



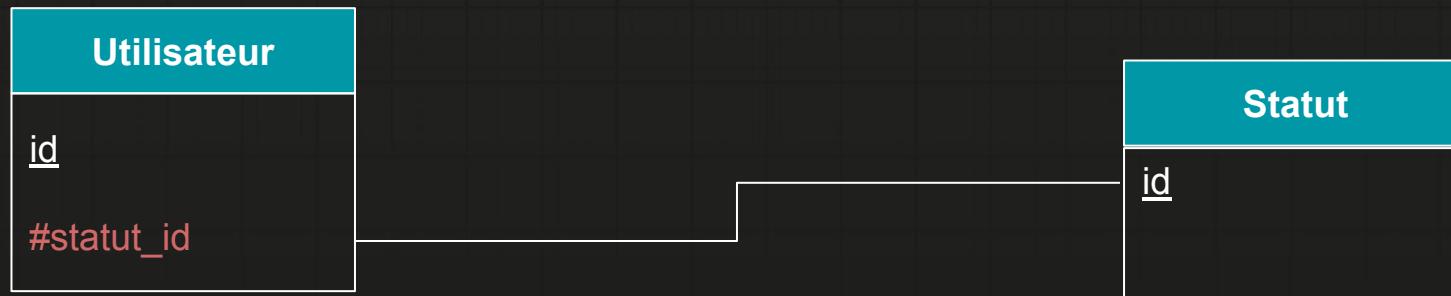
## >> Moyen mnémotechnique

Pour facilement définir le type de liaison, il suffit de lire les cardinalités maximum de la relation, de la table concernée vers la table liée :



## >> Conséquence sur la base de donnée

Pour rappel : au niveau de la base de donnée, cela se traduira par la création d'une clé étrangère **statut\_id** dans la table utilisateur.



## >> Créer un responsable par produit

Dans notre cas nous voulons que chaque produit possède un responsable

Et donc par conséquent qu'un utilisateur peut être responsable de 0, 1 ou plusieurs produits



## >> ManyToOne en Spring

Spring utilise l'annotation `@ManyToOne` sur une propriété ayant pour type l'**entité ciblée**.

Dans l'exemple suivant, l'entité **Utilisateur** possède une propriété **Statut** ayant l'annotation `@ManyToOne` afin de représenter la relation précédente.

```
public class Produit {  
    ...  
    @ManyToOne  
    private Utilisateur responsable;
```

Si on ne précise rien, l'ORM considérera que la relation sera représentée par une clé étrangère **“responsable\_id”** dans la base de donnée

```
public class Produit {  
    ...  
    @ManyToOne  
    @JoinColumn(name =  
    "id_utilisateur_responsable")  
    private Utilisateur responsable;
```

Via l'annotation `@JoinColumn`, il est possible de définir un autre nom pour la clé étrangère.



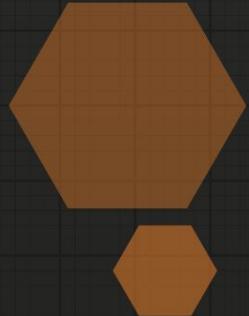
## &gt;&gt; ManyToOne obligatoire

Comme notre relation est 1..1, il ne peut pas y avoir de produit sans responsable,.  
Il faut alors préciser dans notre relation qu'elle n'accepte pas de valeur null

*models/Produit*

```
public class Produit {  
    ...  
    @ManyToOne(optional = false)  
    private Utilisateur responsable;
```





# Les Jointure OneToMany

*Chapitre : Créer les jointures*

## >> Les jointures OneToMany

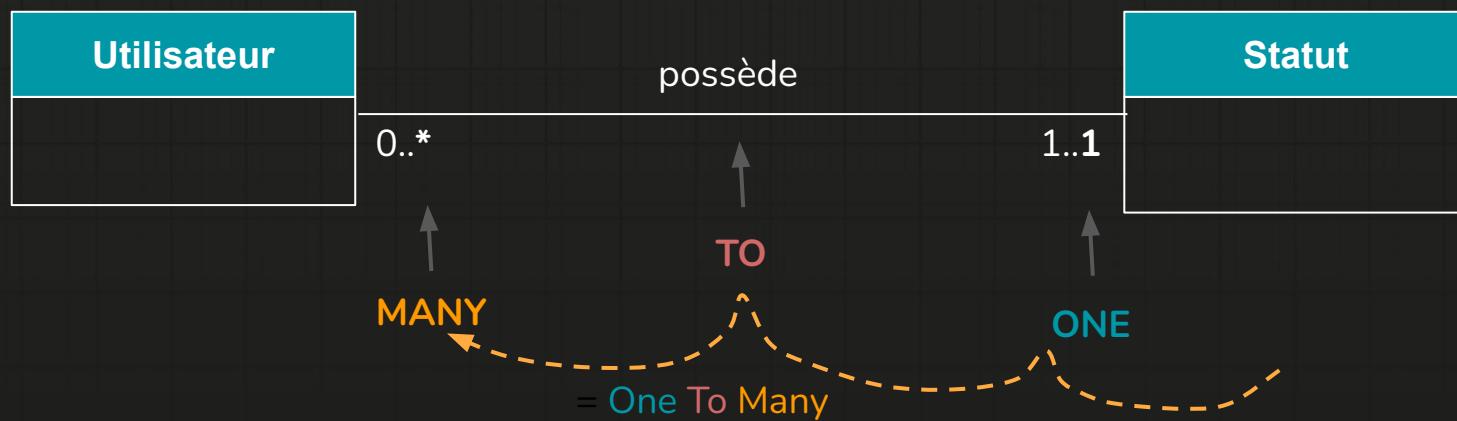
Comme nous venons de le voir, les jointures **OneToMany** et **ManyToOne** sont la représentation d'une seul et même relation. Tout dépend depuis qu'elle entité on l'observe.

La relation précédente, du point de vue de l'entité **Statut** est un **OneToMany**



## >> Moyen mnémotechnique

Encore une fois,  
pour facilement définir le type de liaison, il suffit de lire les cardinalités maximum de la relation,  
de la table concernée vers la table liée (Statut vers Utilisateur) :



## &gt;&gt; Les jointures OneToMany

Les jointures **OneToMany** sont des **relations faibles**. C'est à dire qu'elles sont **optionnels**, et qu'elles ne peuvent exister sans une jointure **ManyToOne** déjà définie dans l'entité ciblée.

Par opposition les relation **ManyToOne** sont des **relation fortes** (*cela aura un impact sur la sauvegarde et la récupération des données dans la base*)

Les relations **OneToMany** possèdent donc un paramètre “`mappedBy`” qui indique le nom de la propriété qui possède la relation `ManyToOne` dans l'entité ciblé (*ici responsable*).

*models/Utilisateur*

```
public class Utilisateur {  
    ...  
    @OneToMany(mappedBy="responsable")  
    Set<Produit> listeProduitSurveille = new HashSet<>();  
}
```

*models/Produit (pas de modification)*

```
public class Produit {  
    ...  
    @ManyToOne(optional = false)  
    private Utilisateur responsable;
```



## >> Les types des propriété OneToMany

Les jointures **OneToMany** sont obligatoirement placées sur des **propriétés** qui héritent de la classe **Collection**. Il est recommandé de les instancier afin d'éviter un NullPointerException en les manipulant. Le type générique de la collection est le type de l'entité liée (c'est dans cette dernière que Spring cherchera la propriété décrite dans le paramètre mappedBy)

```
@OneToMany(mappedBy="responsable")
Set<Produit> listeProduitSurveille = new HashSet<>();
```

Collection avec éléments uniques  
qui ne gardent pas l'ordre

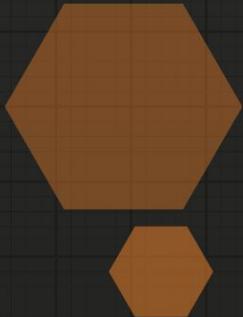
```
@OneToMany(mappedBy="responsable")
Set<Produit> listeProduitSurveille = new LinkedHashSet<>();
```

Collection avec éléments  
uniques qui gardent l'ordre

```
@OneToMany(mappedBy="responsable")
List<Produit> listeProduitSurveille = new ArrayList<>();
```

Collection avec éléments non  
uniques qui gardent l'ordre  
mais rapide en lecture





# Les relations récursives 1/\*

*Chapitre : Créer les jointures*

## >> Les relations récursives 1 / \*

Une relation récursive 1 / \* est une relation

ManyToOne/OneToMany un peu particulière, où la liaison se fait sur l'entité elle-même.

**Utilisateur**

0..1

0..\*

est le père biologique

*Exemple : Un utilisateur à 0 ou 1 utilisateur étant son père biologique. (ManyToOne)*

*Et un utilisateur est le père biologique de 0 ou plusieurs autres utilisateurs. (OneToMany)*

Il n'y a aucune différence avec une relation ManyToOne ou OneToMany classique.

*(Excepté que les 2 relations se trouvent dans la même classe)*

```
@Entity  
public class Utilisateur {  
  
    @ManyToOne  
    private Utilisateur pereBiologique;  
  
    @OneToMany(mappedBy = "pereBiologique")  
    private Set<Utilisateur> listeEnfantsBiologiques;
```

Toujours optionnel



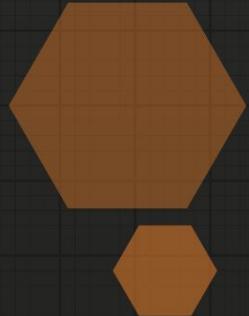
## >> Les jointures ManyToMany

Tout comme les **OneToMany**, les annotation **@ManyToMany** doivent être placées sur des propriété qui héritent de la classe Collection, et qui possèdent en type générique le Type de l'entité ciblée :

```
@Getter  
@Setter  
@Entity  
public class Produit {  
  
    @ManyToOne  
    @JoinTable(  
        name = "etiquette_produit",  
        joinColumns = @JoinColumn(name = "produit_id"),  
        inverseJoinColumns = @JoinColumn(name = "etiquette_id"))  
    Set<Etiquette> listeEtiquette;  
}
```

Si l'annotation **@JoinTable** n'est pas utilisé le nom de la table de liaison sera “utilisateur\_liste\_categorie” et les clés étrangères seront “utilisateur\_id” et “categorie\_id”. Cette notation permet de personnaliser cette table de liaison.





# Les Jointures ManyToMany

*Chapitre : Créer les jointures*

## >> Les jointures ManyToMany

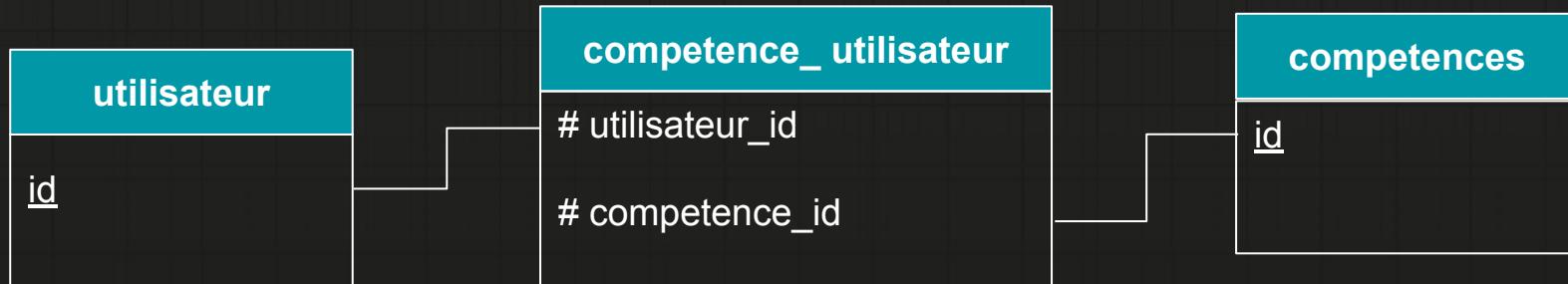
Une jointure ManyToMany représente une liaison ayant des cardinalités maximum à étoile (ou N pour merise) des 2 côtés de la relation :



## >> Conséquence sur la base de donnée

En base de données, cela se traduira toujours par la création d'une nouvelle table (que l'on appelle table de liaison, table de jointure ou table d'association).

Cette table n'est composée que de 2 champs : les **clé étrangères** des 2 **clés primaires** des tables concernées.



## >> Représentation en base de données

Pour rappel : un exemple d'enregistrements permettant de représenter les informations suivantes :

- Arthur Dent est **chanceux** et **voyageur**
- John Doe est également **chanceux** (*Donc chanceux est lié à la fois à Arthur Dent et John Doe*)
- Luci n'est ni l'un ni l'autre



## >> Les jointures ManyToMany

Tout comme les **OneToMany**, les annotation **@ManyToMany** doivent être placées sur des propriété qui héritent de la classe Collection, et qui possèdent en type générique le Type de l'entité ciblée :

```
@Getter  
@Setter  
@Entity  
public class Utilisateur {
```

```
...  
@ManyToMany  
Set<Competence> listeCompetence;
```

OU

```
@ManyToMany  
@JoinTable(  
    name = "utilisateur_competence",  
    joinColumns = @JoinColumn(name = "id_utilisateur"),  
    inverseJoinColumns = @JoinColumn(name = "id_competence"))  
Set<Competence> listeCompetence;
```

```
}
```

Si l'annotation **@JoinTable** n'est pas utilisé le nom de la table de liaison sera “utilisateur\_liste\_competence” et les clés étrangères seront “utilisateur\_id” et “competence\_id”.

Cette notation permet de personnaliser cette table de liaison.

## >> Les jointures ManyToMany

Il est également possible de définir la relation inverse dans la classe ciblé (à l'instar d'un *OneToMany* vis à vis d'un *ManyToOne*). Cette déclaration est également **optionnelle** et sera considérée comme **faible**.

Tout comme un **OneToMany**, il est alors obligatoire de renseigner le nom de la propriété possédant l'annotation **@ManyToMany** dans l'entité ciblée, via le paramètre `mappedBy`.

```
@Entity  
public class Competence {  
  
    ...  
  
    @ManyToMany(mappedBy="listeCompetence")  
    Set<Utilisateur> listeUtilisateur;
```



## &gt;&gt; Ajouter des étiquettes à nos produits

*models/Etiquette (nouveau fichier)*

```
@Getter  
@Setter  
@Entity  
public class Etiquette {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    protected Integer id;  
  
    @NotNull(message = "Le nom ne peut pas être null.")  
    @NotBlank(message = "Le nom ne peut pas être vide.")  
    @Size(max = 30, message = "La longueur du nom ne peut pas dépasser 30 caractères.")  
    @Column(nullable = false, length = 30)  
    protected String nom;  
}
```

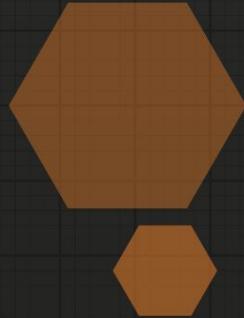


## &gt;&gt; Ajouter des étiquettes à nos produits

*models/Produit*

```
@Entity
public class Produit {
    ...
    @ManyToMany
    @JoinTable(
        name = "etiquette_produit",
        joinColumns = @JoinColumn(name = "produit_id"),
        inverseJoinColumns = @JoinColumn(name = "etiquette_id"))
    Set<Etiquette> listeEtiquette;
}
```





## Les ManyToMany avec clés primaires

*Chapitre : Créer les jointures*

## >> Les relations ManyToMany avec clés primaires

Par défaut Spring **n'ajoute pas de clé primaires** à la **table de liaison**. La représentation en base de donnée devrait normalement nous contraindre à ajouter sur la table de liaison **une clé primaire recomposée des 2 champs**. Les champs sont alors clés primaires et clés étrangères.

Si cette contrainte est nécessaire, alors il n'est pas possible de représenter la relation via un ManyToMany. Il est nécessaire de la créer via une entité à part.



## >> Créer une classe représentant la clé composée

La première étape consiste à créer une classe représentant la clé composée de la table de liaison.  
Ici la clé est composé de 2 propriétés : utilisateurId et competenceld.

Cette classe doit implémenter l'interface `Serializable` et avoir l'annotation `@Embeddable`

*models/CleCompetenceUtilisateur (nouveau fichier)*

```
@Embeddable
public class CleCompetenceUtilisateur implements Serializable {

    @Column(name = "utilisateur_id")
    Integer utilisateurId;

    @Column(name = "competence_id")
    Integer competenceId;

    //+ getter / setter
}
```

Les annotations `@Column` sont facultatives mais permettent de personnaliser le nom des champs dans la base de donnée



## >> Créer l'entité représentant la table de liaison

La deuxième étape consiste à créer l'entité représentant la table de liaison.

Elle est composée de 2 ManyToOne (*l'un effectuant une liaison sur Utilisateur et l'autre sur Competence*)

Mais également d'une annotation **@IdClass** qui définit la class de la clé composée (celle créée précédemment)

Ainsi que 2 annotation **@Id**

*models/CompetenceUtilisateur*

```
@Entity
@Table (name = "competence utilisateur")
@IdClass (CleCompetenceUtilisateur. class)
public class CompetenceUtilisateur {

    @Id
    private Integer utilisateurId;

    @Id
    private Integer competenceId;

    @ManyToOne
    @MapsId ("utilisateur_id")
    @JoinColumn (name = "utilisateur_id")
    private Utilisateur utilisateur;

    @ManyToOne
    @MapsId ("competence_id")
    @JoinColumn (name = "competence_id")
    private Competence competence;

}
```

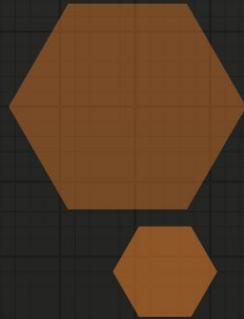


## >> Créer l'entité représentant la table de liaison

```
@Entity  
@Table(name = "competence_utilisateur")  
@IdClass(CompetenceUtilisateur.CleCompetenceUtilisateur.class)  
public class CompetenceUtilisateur {  
  
    @Id  
    private Integer utilisateurId;  
  
    ...  
  
    @Embeddable  
    public static class CleCompetenceUtilisateur implements Serializable  
    {  
        ...  
    }  
}
```

Afin de ne pas avoir à créer un fichier séparé, il est possible de déclarer la classe de la clé composée dans la classe de l'entité concernée

Ce qui implique que celle-ci soit **static** (*noté également la présence du nom de la classe de l'entité dans l'annotation `@IdClass`*)



# Les relations récursives \*/\*

*Chapitre : Créer les jointures*

## >> Les relations récursives \* / \*

```
@Entity  
@Table(name = "amitie")  
@IdClass(CleAmitie.class)  
public class Amitie {  
  
    @Id  
    private Integer utilisateur1Id;  
  
    @Id  
    private Integer utilisateur2Id;  
  
    @ManyToOne  
    @MapsId("utilisateur_1_id")  
    @JoinColumn(name = "utilisateur_1_id")  
    private Utilisateur utilisateur1;  
  
    @ManyToOne  
    @MapsId("utilisateur_2_id")  
    @JoinColumn(name = "utilisateur_2_id")  
    private Utilisateur utilisateur2;  
}
```

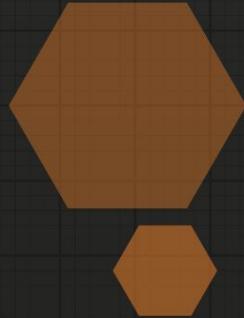


De la même manière il est tout à fait possible de définir une relation récursive \* / \*

(Ex : Un utilisateur peut être ami avec 0 ou plusieurs utilisateurs)

```
@Embeddable  
public class CleAmitie implements Serializable {  
  
    @Column(name = "utilisateur_1_id")  
    Integer utilisateur1Id;  
  
    @Column(name = "utilisateur_2_id")  
    Integer utilisateur2Id;  
  
    //GETTER + SETTER !!!!
```



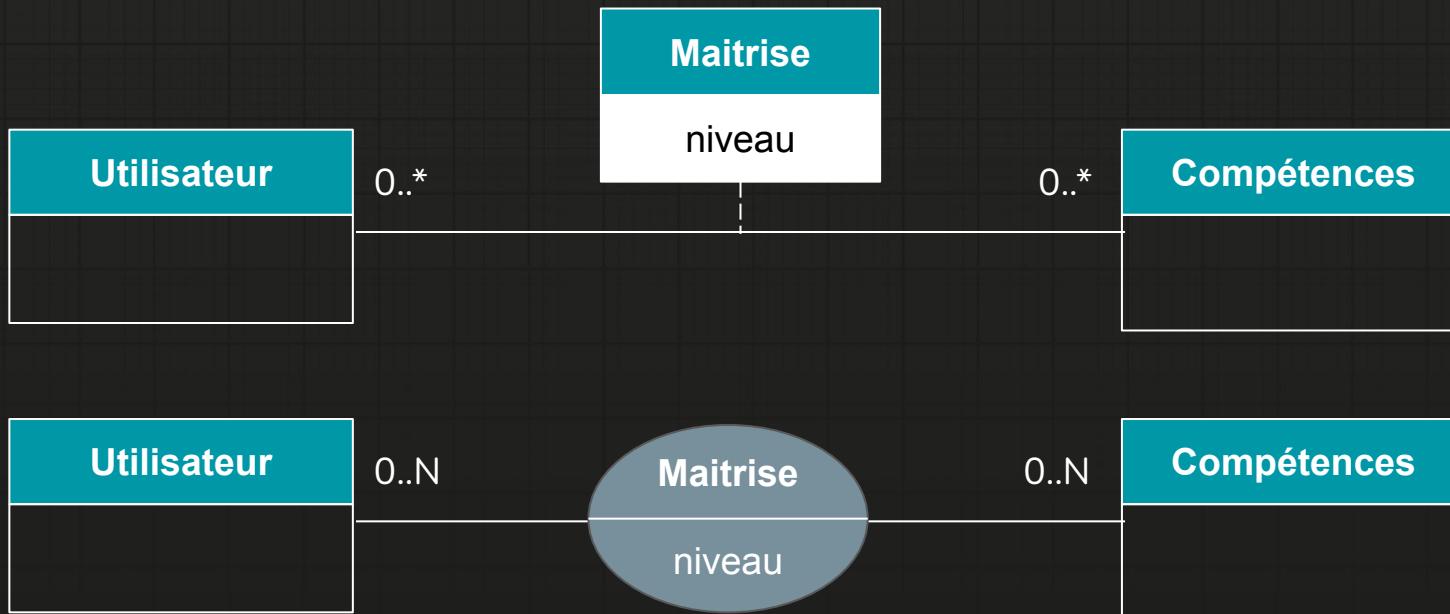


# Les relations /\* avec attributs portés

*Chapitre : Créer les jointures*

## >> Les relations récursives \* / \* avec attribut(s) porté(s)

Pour rappel, une relation \* / \* avec attribut porté consiste à ajouter une information à chaque relation qui se feront entre les 2 entités (ex : un utilisateur maîtrise 0 ou plusieurs compétence, mais avec des niveaux différents, comme : expert, débutant ... )



## >> Les relations récursives \* / \* avec attribut(s) porté(s)

Au niveau de la base de donnée, cela se représente en ajoutant un champs dans la table de liaison.  
Dans cet exemple un utilisateur peut avoir plusieurs compétence, mais une seul fois la même.



utilisateur		maitrise			competence	
id	nom	utilisateur_id	competence_id	niveau	id	designation
1	Prof	1	118	“expert”	118	JAVA
2	Simplet	1	218	“confirmé”	218	Spring
		2	118	“débutant”		

## >> Les relations \* / \* avec attribut(s) porté(s)

Il n'y a pas de changement par rapport à la relation \* / \*  
avec clés primaires décrite plus tôt, à l'exception de la  
déclaration du/des propriété(s) portées

```
@Embeddable
public class CleCompetenceUtilisateur
implements Serializable {

    @Column(name = "utilisateur_id")
    Integer utilisateurId;

    @Column(name = "competence_id")
    Integer competenceId;

    //+ getter / setter
}
```

```
@Entity
@Table(name = "maitrise")
@IdClass(CleCompetenceUtilisateur.class)
public class CompetenceUtilisateur {

    @Id
    private Integer utilisateurId;

    @Id
    private Integer competenceId;

    @ManyToOne
    @MapsId("utilisateurId")
    @JoinColumn(name = "utilisateur_id")
    private Utilisateur utilisateur;

    @ManyToOne
    @MapsId("competenceId")
    @JoinColumn(name = "competence_id")
    private Competence competence;

    private String niveau;
}
```



## >> Les relations récursives \* / \* avec attribut porté

```
@Entity  
@Table(name = "rendez vous")  
@IdClass(CleRendezVous.class)  
public class RendezVous {  
  
    @Id  
    private Integer utilisateur1Id;  
  
    @Id  
    private Integer utilisateur2Id;  
  
    @ManyToOne  
    @MapsId("utilisateur 1 id")  
    @JoinColumn(name = "utilisateur 1_id")  
    private Utilisateur utilisateur1;  
  
    @ManyToOne  
    @MapsId("utilisateur 2 id")  
    @JoinColumn(name = "utilisateur 2_id")  
    private Utilisateur utilisateur2;  
  
    private boolean accepte;  
}
```



Le même exemple avec une relation récursive \* / \*

```
@Embeddable  
public class CleRendezVous implements  
Serializable {  
  
    @Column(name = "utilisateur_1_id")  
    Integer utilisateur1Id;  
  
    @Column(name = "utilisateur_2_id")  
    Integer utilisateur2Id;  
  
    //GETTER + SETTER !!!!
```

On ajoute les attributs portés

>> Les relations \* / \*  
avec attribut porté qui est  
une partie de la clé primaire

```
@Entity
@IdClass(CleConnexion.class)
public class Connexion {

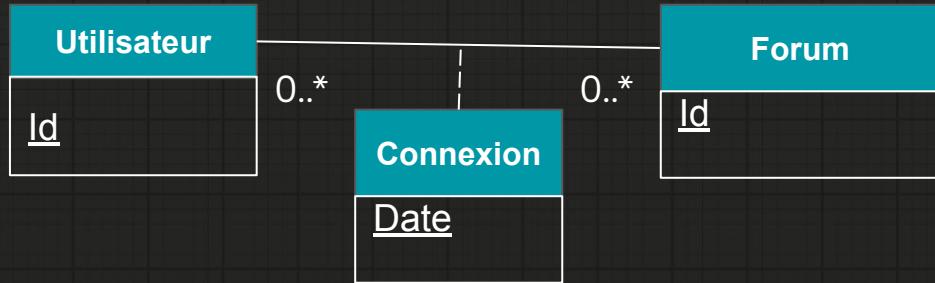
    @Id
    private Integer utilisateurId;

    @Id
    private Integer forumId;

    @Id
    private Date date;

    @ManyToOne
    @MapsId("utilisateur id")
    @JoinColumn(name = "utilisateur_id")
    private Utilisateur utilisateur;

    @ManyToOne
    @MapsId("forum id")
    @JoinColumn(name = "forum_id")
    private Forum forum;
```



```
@Embeddable
public class CleConnexion implements Serializable {

    Integer utilisateurId;

    Integer forumId;

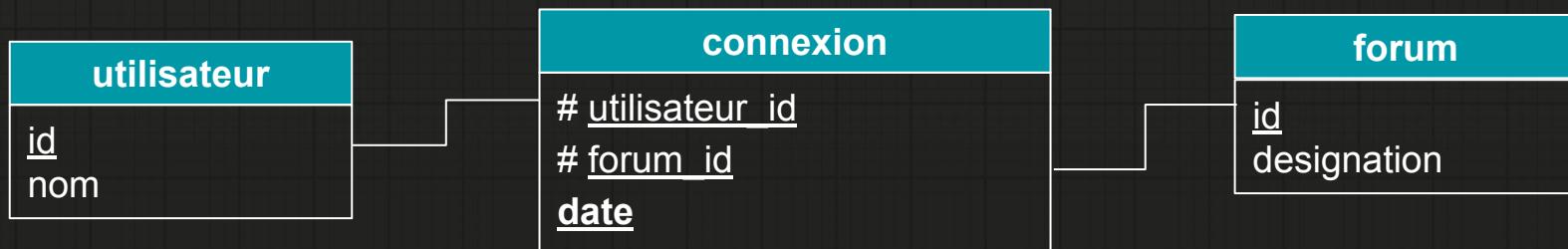
    Date date;

    //GETTER + SETTER !!!!
```

## >> Les relations \* / \*avec attribut porté qui est une partie de la clé primaire

La différence au niveau de la base de donnée, sera que l'attribut fera partie de la clé primaire.

Dans cet exemple, une personne peut se connecter plusieurs fois à un forum à des dates différentes



utilisateur		competence_utilisateur			forum
id	nom	utilisateur_id	forum_id	date	id
1	Ramsay	1	4	2021-01-01 00:00:03	
2	Linus	1	4	2021-01-01 07:30:00	
		2	5	2020-01-01 11:00:00	

>> Les relations récursives \* / \*  
avec attribut porté qui est  
une partie de la clé primaire

```
@Entity
@Table(name = "rendez vous")
@IdClass(CleRendezVous.class)
public class RendezVous {

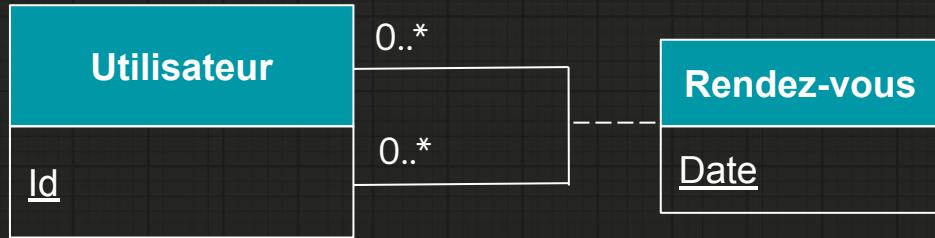
    @Id
    private Integer utilisateur1Id;

    @Id
    private Integer utilisateur2Id;

    @Id
    private Date date;

    @ManyToOne
    @MapsId("utilisateur 1 id")
    @JoinColumn(name = "utilisateur 1_id")
    private Utilisateur utilisateur1;

    @ManyToOne
    @MapsId("utilisateur 2 id")
    @JoinColumn(name = "utilisateur 2_id")
    private Utilisateur utilisateur2;
```



Le même exemple avec une relation récursive \* / \*

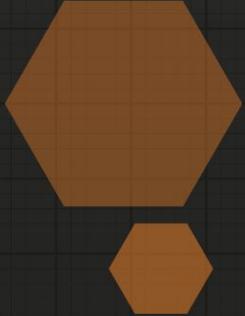
```
@Embeddable
public class CleRendezVous implements Serializable {

    @Column(name = "utilisateur_1_id")
    Integer utilisateur1Id;

    @Column(name = "utilisateur_2_id")
    Integer utilisateur2Id;

    Date date;

    //GETTER + SETTER !!!!
```



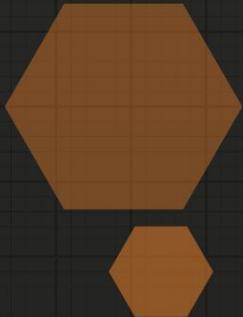
# Many to Many VS Entité

*Chapitre : Créer les jointures*

>> TODO

TODO





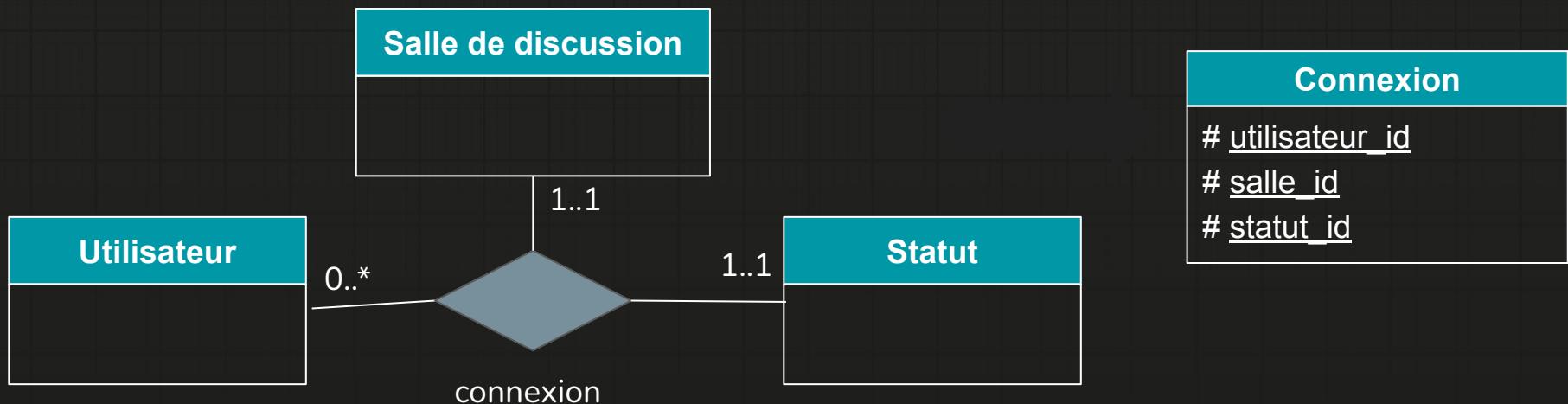
## Les ternaires (et N-aires)

Avec ou sans attribut portés ...

*Chapitre : Créer les jointures*

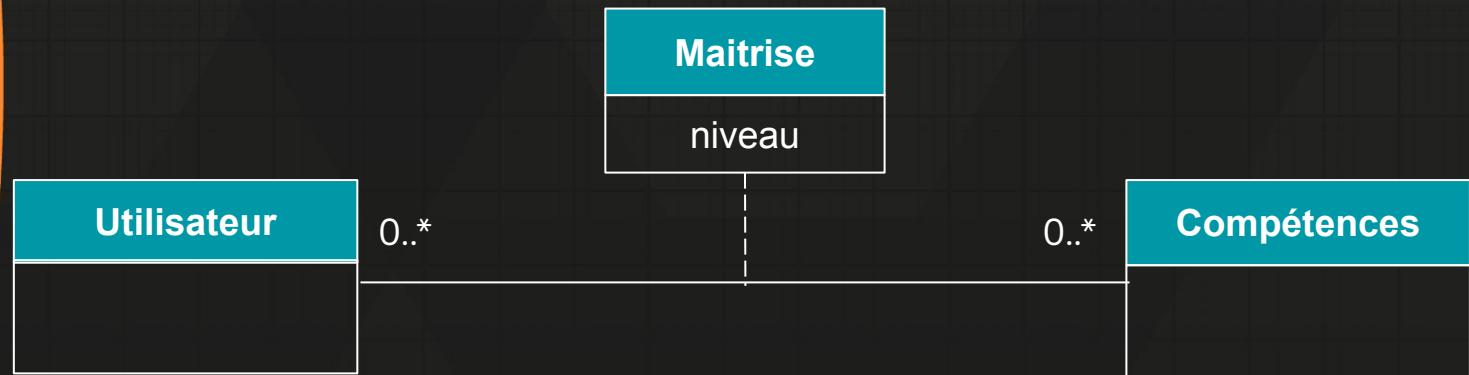
## >> Les relations N-aires

Tous les relations impliquant plus de 2 entités, avec ou sans attribut porté suivront la même logique, il suffira de déclarer plus de clés primaires dans la classes représentant la relation N-aire et dans la classe représentant sa clé



# Exemple d'opérations d'Hibernate

*Les DAO des entités avec clé composée*



*Chapitre : Créer les jointures*

## >> Les DAO des entité à clé composée

La première étape consiste à créer une classe représentant la clé composée de la table de liaison.

```
@Repository  
public interface MaitriseDao extends JpaRepository<Maitrise, CleMaitrise>  
{  
}
```

**OU (Si la classe @Embeddable de la clé est déclarée dans la classe de l'entité)**

```
@Repository  
public interface MaitriseDao extends JpaRepository<Maitrise, Maitrise.CleMaitrise> {  
}
```



## >> Effectuer des opérations CRUD avec des entités à clé composée

Pour récupérer un enregistrement, il faudra créer au préalable une méthode se basant sur les id de l'entité.  
Idem pour une suppression.

```
Optionnal<Maitrise> findByUtilisateurIdAndNoteId (int utilisateurId, int noteId);
```

Définition dans le Dao

```
maitriseDao.findByUtilisateurIdAndCompetenceId (42, 118)
```

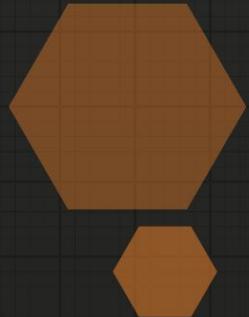
Appel dans le Controller

Pour un ajout il sera nécessaire de définir la valeur de toutes les clés qui composent l'entité.  
A la différence des clés étrangères des entités, c'est la propriété avec l'annotation @Id que l'on change (*ici utilisateurId*) et non l'id de l'entité liée par un @ManyToOne (ici ce serait utilisateur.id)

```
Maitrise maitrise = new Maitrise();
maitrise.setNiveau (5);
maitrise.setUtilisateurId (42);
maitrise.setCompetenceId (118);

maitriseDao.save (maitrise);
```

Il sera néanmoins impossible d'effectuer un update d'un des id de l'entité via la méthode save (*les clés permettant justement de définir l'enregistrement à modifier*), supprimez au préalable l'enregistrement concerné, ou effectuez une **requête HQL** (*voir chapitre correspondant*)



# Héritage

*Chapitre : Créer les jointures*

## >> Héritage

Un héritage implique que la classe enfant, récupère toutes les propriétés de la classe parent.

En base de données cela peut se représenter de 3 façons différentes, mais nous ne verrons que la représentation la plus commune :

La clé primaire de la table enfant (vendeur) est également une clé étrangère de la table parent (utilisateur)

Ce qui signifie qu'un vendeur ayant l'id 42 doit obligatoirement exister comme utilisateur avec l'id 42

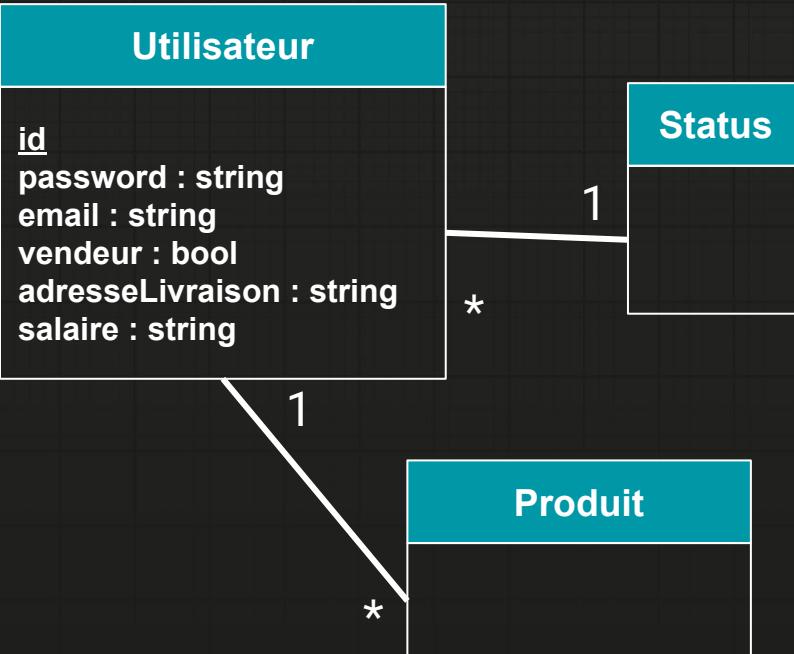


Utilisateur

Vendeur

# >> Pourquoi mettre en place un héritage ?

Prenons un exemple simple : un utilisateur peut être un client ou un vendeur. C'est le booléen vendeur qui le définit. Dans la table Utilisateur on retrouve mélangé les 2 type d'utilisateur



Un produit est vendu par un utilisateur

Mais comme les vendeur et les clients sont dans la même table, rien n'empêche que l'on désigne un client comme vendeur d'un produit (en affectant à la clé étrangère de la table produit, l'id d'un client)

De plus, dans le cas où l'utilisateur est un vendeur, la propriété adresse sera null, de même pour salaire dans le cas d'un client.

Ce qui nous empêche d'appliquer des contraintes "not null" sur ces champs si nous le souhaitons.

Une solution pourrait consister à créer 2 tables (vendeur et client), mais cela doublerait les propriétés (de la table et des models) ainsi que les liaisons communes : password, email, et la liaison avec status.

Dans le cas où il y aurait encore plus de type d'utilisateur et de liaison / propriété, cela rendrait l'application difficilement maintenable



## >> Représentation en base de donnée sans heritage

plusieurs incohérences peuvent être insérées :  
Des vendeur sans salaire ou avec une adresse  
Des client sans adresse ou avec un salaire

Utilisateur					
<b>id</b>	<b>email</b>	<b>password</b>	<b>vendeur</b>	<b>adresse</b>	<b>salaire</b>
1	a@a	root	false	ville A	null
2	b@b	azerty	false	null	1000
3	c@c	qsdfgh	true	ville C	2000
4	d@d	toto	true	null	null

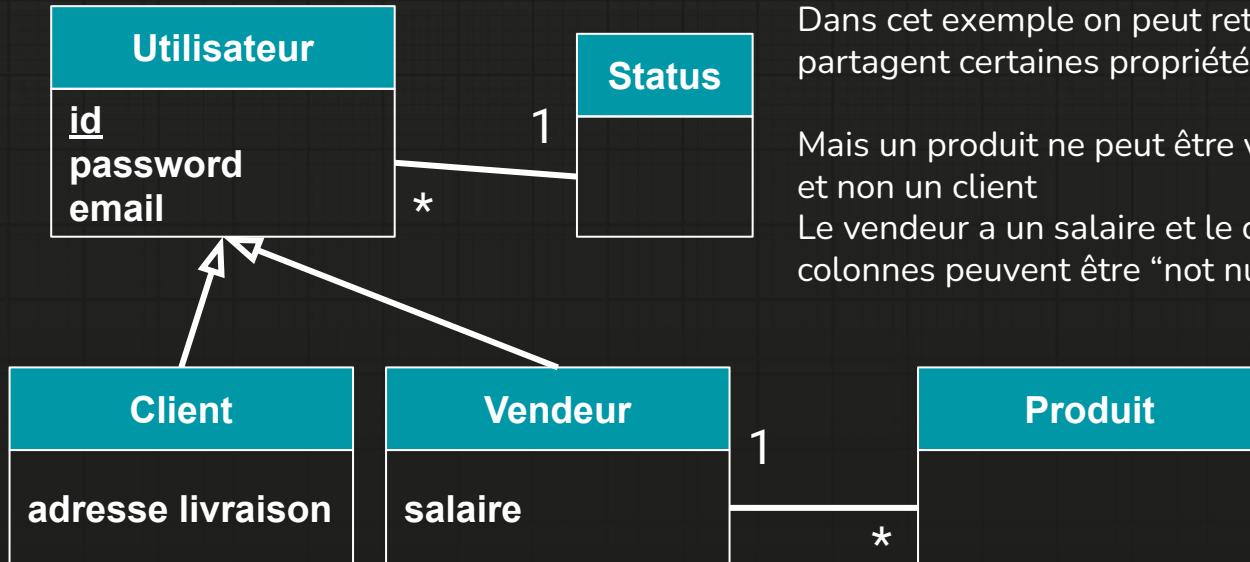
Produit	
<b>id</b>	<b>vendeur_id</b>
147	4
895	1

Ici c'est l'id d'un client qui a été renseigné comme vendeur du produit



## >> Solution avec un héritage

L'intérêt d'un héritage dans une base de données, est de faire en sorte que certaines entités partagent des propriétés ou des liaisons avec d'autres entités, mais elles auraient leur propres propriétés/liaisons



Dans cet exemple on peut retrouver les 2 types d'utilisateurs, qui partagent certaines propriétés et liaisons

Mais un produit ne peut être vendu que par un vendeur et non un client

Le vendeur a un salaire et le client une adresse de livraison, ces colonnes peuvent être “not null” sans que cela ne pose de problèmes

## >> Représentation en base de donnée avec héritage

Utilisateur			
<b>id</b>	<b>email</b>	<b>password</b>	<b>status_id</b>
1	a@a	root	2
2	b@b	azerty	1
3	c@c	qsdfgh	2
4	d@d	toto	3

Client	
<b>id</b>	<b>adresse</b>
2	ville A
1	ville B

Vendeur	
<b>id</b>	<b>salaire</b>
3	1500
4	2000

Produit	
<b>id</b>	<b>vendeur_id</b>
147	4
895	4

On peut définir les adresses et les salaires à "not null"

Impossible de définir un client comme vendeur, puisque la contrainte se situe sur la colonne id de la table vendeur

## >> Héritage

```
@Entity  
@Inheritance (strategy = InheritanceType.JOINED)  
public class Utilisateur {  
  
    @Id  
    private Integer id;  
  
}
```

Ici on indique comment l'héritage doit être effectué : JOINED signifie que chaque table à ses propre colonnes et qu'une jointure entre le parent et l'enfant est nécessaire pour récupérer toutes les informations

```
@Entity  
public class Client extends Utilisateur{  
  
    ...  
}
```

```
@Entity  
public class Vendeur extends Utilisateur{  
  
    ...  
}
```

Pas de propriété id dans Client ou Vendeur puisqu'ils héritent de la propriété id de Utilisateur

## >> Dao / contrôleur

Il sera sûrement nécessaire de créer autant de DAO / contrôleur que d'entité enfant. Chacun retournera un objet composé de ses propriétés et celle de son parent

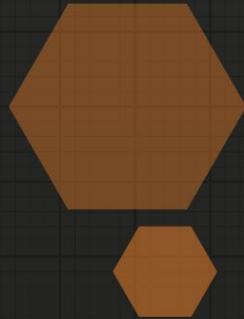
Vous pouvez considérer les entités enfants comme des entités complètement à part, qui se gère comme n'importe quelle autre entité.

```
@Repository  
public interface UtilisateurDao extends JpaRepository<Chef, Integer> {  
}
```

```
@Repository  
public interface ClientDao extends JpaRepository<Client, Integer> {  
}
```

```
@Repository  
public interface VendeurDao extends JpaRepository<Vendeur, Integer> {  
}
```





# Gérer le retour au format JSON

*Chapitre : Créer les jointures*

## >> Liaison automatique

Si une entité est lié à une autre, cette entité sera ajoutée lors des requêtes et par conséquent transmises dans la chaîne JSON

```
@GetMapping({"/listeUtilisateur"})
public List<Utilisateur> listeUtilisateur () {
    return utilisateurDao.findAll();
}
```

```
[{"id": 1, "nom": "Snow", "prenom": "Jon", "voiture": {"id": 1, "nom": "206"}}, {"id": 2, "nom": "Stark", "prenom": "Sansa", "voiture": {"id": 2, "nom": "Twingo"}}]
```

## >> Les erreurs de redondance

Si un utilisateur à une voiture, et une voiture des utilisateurs. Les utilisateurs de ces voitures, on une référence à une voiture etc .... nous sommes dans une référence cyclique. Le JSON sera complété à l'infini.

*model/Utilisateur*

```
@ManyToOne  
protected Voiture voiture;
```

*model/Voiture*

```
@OneToMany (mappedBy="voiture")  
List<Utilisateur> listeUtilisateur = new ArrayList<>();
```



## >> Solution 1 : JSON IGNORE (*qui apporte beaucoup d'inconvénients*)

L'annotation `@JsonIgnore` permet de palier à ce problème.

La propriété possédant l'annotation `@JsonIgnore` n'apparaîtra pas dans JSON.

Cela permet de "casser la boucle" en n'affichant plus l'une ou l'autre des propriétés.

Il y a tout de même 2 problèmes majeurs à cette solution :

- Une fonctionnalité pourrait avoir besoin d'afficher une propriété et une autre fonctionnalité afficher l'autre propriété, mais celle annotée `@JsonIgnore` ne sera jamais affiché
- Plus important l'annotation `@JsonIgnore` empêche l'affichage mais également la lecture de la propriété, ainsi il n'est pas possible de créer un objet via `@RequestBody` qui permettrait de définir la clé étrangère en lien avec la propriété annotée avec `@JsonIgnore`

*model/Utilisateur*

```
@ManyToOne  
@JsonIgnore  
protected Voiture voiture;
```

*model/Voiture*

```
@OneToMany(mappedBy="voiture")  
List<Utilisateur> listeUtilisateur = new ArrayList<>();
```

## >> Solution 2 : JSON VIEW

Il existe une solution un peu fastidieuse, mais qui peut s'adapter à tous les cas de figure : les JsonView, ils permettent :

- De pouvoir régler le problème des JSON infinis,
- De ne pas afficher l'intégralité des informations du JSON, pour des raisons de sécurité (ex : le mot de passe ou l'email d'un utilisateur),
- De ne pas afficher des informations inutiles dans le but de simplifier l'utilisation de l'API ainsi que le poids des requête HTTP

Ils permettent de définir un système de “white list” en créant des filtres sur chaque propriété des models, et d'utiliser l'un de ces filtres sur les requêtes

Un filtre d'un JSON View se base simplement sur le nom d'une classe. Il n'y a aucun restriction sur le nom de cette classe et il est possible d'en créer autant que nécessaire, on les place généralement dans un package “view”

*view/Categorie*

```
public class Categorie{};
```

*view/AffichageProfil*

```
public class AffichageProfil{};
```



## >> Solution 2 : JSON VIEW

Il existe une solution un peu fastidieuse, mais qui peut s'adapter à tous les cas de figure : les JsonView

Ils permettent de définir un système de “white list” en créant des filtres sur chaque propriété des models, et d'utiliser l'un de ces filtres sur les requêtes

```
@GetMapping("/utilisateurs")
@JsonView(Utilisateur.class)
public List<Utilisateur> getUtilisateurs() {
    return utilisateurDao.findAll();
}
```

```
@GetMapping("/categories")
@JsonView(AffichageCategorie.class)
public List<Categorie> getCategories() {
    return categorieDao.findAll();
}
```



## >> Solution 2 : JSON VIEW 2/2

Enfin pour chaque propriété utiliser l'annotation `JsonView` prenant en paramètre un tableau de toutes les vues utilisant ce paramètre.

```
@Entity
public class Utilisateur {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @JsonView({Utilisateur.class})
    private int id;

    @JsonView({Utilisateur.class, AffichageCategorie.class})
    private String prenom;

    @JsonView({Utilisateur.class, AffichageCategorie.class})
    private String nom;

    @JsonView({Utilisateur.class})
    private Set<Categorie> listeCategorie;
```

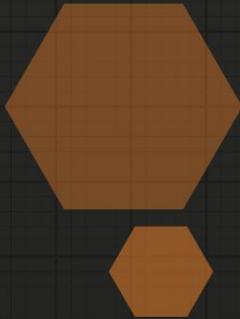
```
@Entity
public class Categorie {

    @Id
    @JsonView({AffichageCategorie.class})
    private int id;

    @JsonView({Utilisateur.class, AffichageCategorie.class})
    private String nom;

    @JsonView({AffichageCategorie.class})
    private Set<Utilisateur> listeUtilisateur;
```





# Ajouter des suppressions d'enregistrements orphelins

*Chapitre : Créer les jointures*

## &gt;&gt; Ajouter des suppressions d'enregistrements orphelins

Dans le cas des commandes, il n'y a aucun intérêt à garder les lignes de commande lorsque la commande lié ou le produit sont supprimés de la base de donnée. On peut donc ajouter une annotation `@OnDelete(action = OnDeleteAction.CASCADE)` qui ajoutera l'action DELETE sur la contrainte (*et donc supprimera la ligne de commande si l'on supprime la commande ou le produit lié, plutôt que de générer une erreur d'intégrité*)

*models/LigneCommande*

```
@Getter  
@Setter  
@Entity  
public class LigneCommande {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    protected Integer id;  
  
    @ManyToOne(optional = false)  
    @OnDelete(action = OnDeleteAction.CASCADE)  
    protected Produit produit;  
  
    @ManyToOne(optional = false)  
    @OnDelete(action = OnDeleteAction.CASCADE)  
    protected Commande commande;  
  
    protected int quantite;  
}
```



# Mettre à jour une entité

Dans ce chapitre :

## >> La méthode Save des DAO

La méthode pour faire **ajouter** ou **modifier** un objet en base de donnée est la même dans les DAO.

Si l'objet a sa **clé primaire existante dans la table**, il sera considéré comme devant subir un **UPDATE** ou un **INSERT** si elle n'est pas renseignée ou si elle n'existe pas dans la table.

```
ClasseDuModel objetASauvegarder = new ClasseDuModel();
objetASauvegarder.setPropriete("une valeur");
ClasseDuModelDao.save(objetASauvegarder);
```

L'objet créé n'a pas sa clé primaire renseignée, il sera donc ajouter en base de donnée (INSERT)

```
Utilisateur john = new Utilisateur();
toto.setId(42);
toto.setNom("Doe");
UtilisateurDao.save(toto);
```

L'objet créé a sa clé primaire renseignée, ce sera donc l'enregistrement avec l'id 42 qui sera mis à jour (UPDATE)

*Note : uniquement si l'id 42 existait déjà dans la base de donnée*



## >> La méthode Save des DAO

Attention lorsque sauvegarde une entité dans la base de donnée via les DAO ce sont tous ses champs qui sont mis à jour.  
Ainsi, si vous exécutez les instructions suivantes :

```
Utilisateur utilisateur = new Utilisateur();
utilisateur.setId(42);
utilisateur.setPseudo("jdoe");
utilisateurDao.save(utilisateur);
```

L'enregistrement aura la valeur de sa colonne mot\_de\_passe effacé.

Le cas d'usage normal est de récupérer l'enregistrement en base de donnée, de l'éditer et finalement de l'enregistrer. (*exemple slide suivante*)

Ou bien d'utiliser une **requête HQL** (*que l'on verra plus loin dans le cours*)



## >> Persistance via un DAO Exemple 1

L'exemple suivant illustre une méthode permettant de sauvegarder un statut sans aucune vérification. Dans ce cas, selon l'objet JSON envoyé, le statut sera ajouté ou modifié (présence de la clé étrangère ou non), et les champs mis à jour (ou vidés) dépendent totalement des propriétés JSON renseignés.

```
@PostMapping("/statut")
public int editStatut (@RequestBody Statut statut) {

    statut = statutDao.save(statut);
    return statut.getId();
}
```



## >> Persistance via un DAO Exemple 2

Un autre exemple de sauvegarde qui permet de ne pouvoir mettre à jour que certaines informations.

L'utilisateur est récupéré dans la base de donnée, et on change le mot de passe par l'ancien de manière à ce qu'il ne puisse pas être changé.

```
@PostMapping("/user/utilisateur")
public int editUser(@RequestBody Utilisateur utilisateur){

    Optional<Utilisateur> ancienUtilisateur =
        utilisateurDao.findById(utilisateur.getId());

    if(ancienUtilisateur.isPresent()) {
        Utilisateur utilisateurBdd = ancienUtilisateur.get();
        utilisateur.setMotDePasse(utilisateurBdd.getMotDePasse());
    }
    return utilisateurDao.saveAndFlush(utilisateur).getId();
}
```

## >> Aparté : Un problème de performance ?

Vous avez peut être remarqué que l'on a volontairement effectué 2 requêtes pour mettre à jour l'utilisateur.

La première pour le récupérer (correspondant à un **select**)

```
Optional<Utilisateur> ancienUtilisateur = utilisateurDao.findById(utilisateur.getId());
```

La deuxième pour le mettre à jour (correspondant à un **update**)

```
utilisateurDao.save(utilisateurBdd);
```

Il est légitime de se demander si il n'y a pas là un problème de performance ? Alors qu'un simple update aurait pu suffir. Effectivement un ORM n'a généralement pas pour première vocation la performance. **Il vise avant tout la productivité.**

Si nous devions réaliser la requête SQL la mieux optimisée pour la mise à jour de chaque élément, il faudrait alors sélectionner rigoureusement les champs à ajouter (et cela à chaque changement de la structure de chaque entité). Ce qui conduit à du temps de développement, de test unitaire de validation ... C'est une vision qui prend en compte l'aspect financier d'un projet et qui n'est pas partagée par tous les développeurs (loin de là).

Il reste tout de même possible d'effectuer des **requêtes SQL (ou HQL) plus performantes**, mais cela demande un travail supplémentaire. (*voir slide Personnaliser et optimiser les requêtes*)



## >> Persistance via un DAO Exemple 3

Un autre exemple de sauvegarde qui cette fois ne permet que de mettre à jour un utilisateur déjà existant. L'utilisateur est récupéré dans la base de donnée, et on ne change que les propriétés qu'on lui "autorise" à mettre à jour. Le reste du JSON est totalement ignoré. Si l'id de l'utilisateur n'existe pas, aucune insertion n'est effectuée. On notera qu'ici l'utilisation de 2 requêtes est tout à fait justifié.

```
@PostMapping("/user/utilisateur")
public boolean editUser(@RequestBody Utilisateur utilisateur){

    Optional<Utilisateur> ancienUtilisateur =
        utilisateurDao.findById(utilisateur.getId());

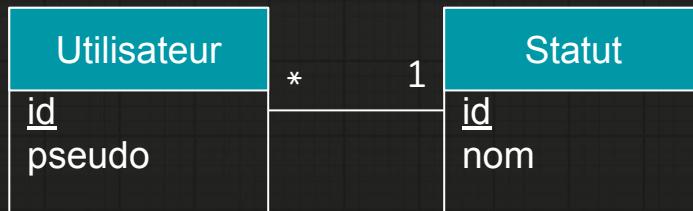
    if(ancienUtilisateur.isPresent()) {
        Utilisateur utilisateurBdd = ancienUtilisateur.get();
        utilisateurBdd.setPseudo(utilisateur.getPseudo());
        utilisateurDao.save(utilisateurBdd);
        return true;
    }
    return false;
}
```



## >> Persistance via un DAO : Jointure

Lorsqu'une entité est en relation avec une autre entité, il y a différent scénario pour mettre à jour cette liaison.

Prenons l'exemple du diagramme de classe suivant : Un utilisateur possède un statut, et un statut peut être possédé par 0 ou plusieurs utilisateurs.



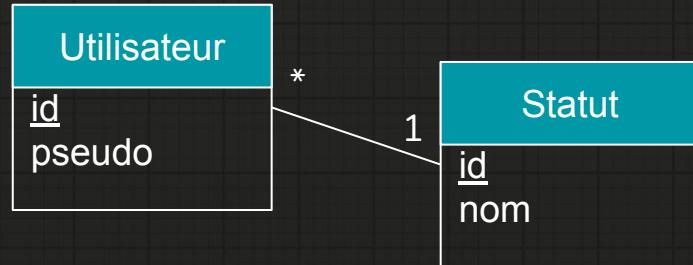
Ce qui donnera en base de donnée 2 tables, où utilisateur aura la clé étrangère **statut\_id**



## >> Persistance via un DAO : Jointure

Nous avons donc 2 entités : Utilisateur et Statut.

Utilisateur possède une **ManyToOne** vers Status  
Et Statut possède une relation **OneToMany** vers Utilisateur



Une relation **OneToMany** est une **relation faible**.  
Elle subit la relation imposé par le **ManyToOne** présent dans la classe utilisateur : **la relation forte**

```
public class Utilisateur {
    @Id
    private int id;

    private String pseudo;

    @ManyToOne
    private Statut statut;
}
```

```
public class Statut {
    @Id
    private int id;

    private String denomination;

    @OneToMany(mappedBy = "statut")
    private List<Utilisateur> listeUtilisateur;
}
```

## >> Exemple d'opérations d'Hibernate

### Persistante via un DAO : Jointure

Puisque c'est **Utilisateur** qui détient la **relation forte**, c'est l'enregistrement d'un objet **Utilisateur** qui peut mettre à jour la relation dans la base de données (la clé étrangère). Pour cela l'objet **Utilisateur** que l'on souhaite enregistrer doit posséder une instance de l'objet **Statut** (dans sa propriété **statut**), dont l'id est correctement affecté.

Autrement dit : la clé étrangère **statut\_id** dans la table **utilisateur** prendra la valeur de l'id de l'objet **statut**.

Dans l'exemple ci-contre :

On créait une nouvelle instance d'**Utilisateur** avec un l'id 42.

(L'utilisateur peut exister ou non dans la BDD)

On lui affecte une nouvelle instance de **statut** qui a l'id 3.

Le **statut** avec l'id 3 doit par contre bien exister en BDD

*(notez que seul le renseignement de la clé primaire est obligatoire)*

Une fois enregistré, le champs **statut\_id** de l'**utilisateur** 42 aura la valeur 3

```
Statut statut = new Statut();
statut.setId(3);
```

```
Utilisateur utilisateur = new
Utilisateur();
utilisateur.setId(42);
utilisateur.setPseudo("toto");
utilisateur.setStatut(statut);
```

```
utilisateurDao.save(utilisateur);
```



## >> Aparté : Objet issu de JSON

Une erreur courante est de confondre les **propriétés** de l'objet JAVA avec les **champs** de la table associée.

Si vous souhaitez réaliser l'opération précédente en envoyant des données au format JSON (qui sera alors transformé en objet JAVA), vous devrez envoyer cet objet :

```
{  
    "id": 42,  
    "pseudo": "toto",  
    "statut": {  
        "id": 10  
    }  
}
```

**Et non cet objet**

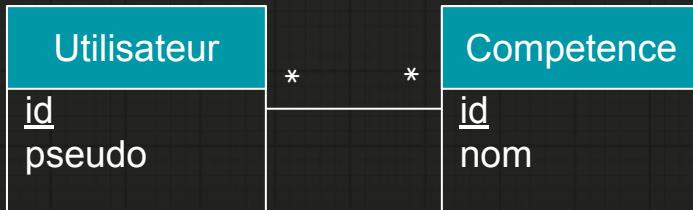
*(qui ne correspond pas à la classe Utilisateur en JAVA)*

```
{  
    "id": 42,  
    "pseudo": "toto",  
    "statut_id": 10  
}
```

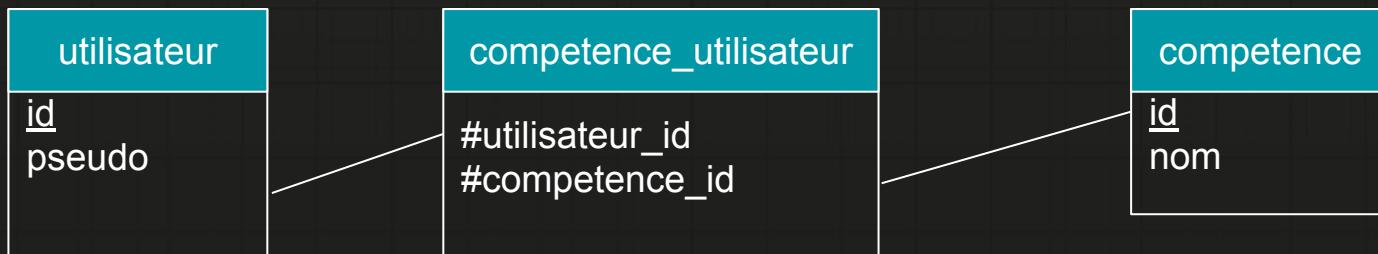


## >> Persistance via un DAO : Jointure

De la même manière, on peut imaginer une relation avec une entité via un **ManyToMany**



Ce qui donnera en base de donnée 3 tables :



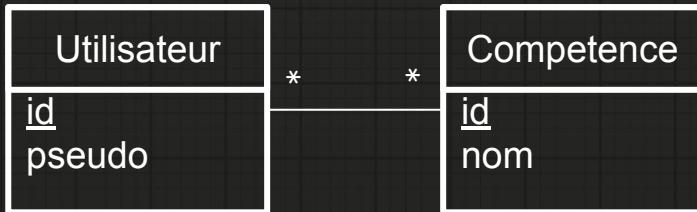
## >> Persistance via un DAO : Jointure

Nous avons donc 2 entités : Utilisateur et Competence.

**Utilisateur** possède une relation **ManyToMany** vers **Competence**

Et **Competence** possède également une relation **ManyToMany** vers **Utilisateur**

Ce n'est pas obligatoire (à l'instar de la relation **OneToMany**), mais cela peut être pratique dans certains cas.



L'un des 2 **ManyToMany** doit être désigné comme **relation faible** (*celui qui aura la propriété mappedBy dans ses paramètres*)

```
public class Utilisateur {
    ...
    @ManyToMany
    private Set<Role> listeCompetence;
    ...
}
```

```
public class Competence {
    ...
    @ManyToMany(mappedBy = "listeCompetence")
    private Set<Utilisateur> listeUtilisateur;
    ...
}
```



## >> Persistance via un DAO : Jointure

De la même manière que pour la relation ManyToOne, c'est l'entité qui a la relation forte qui va pouvoir mettre à jour la table de jointure **competence\_utilisateur**.

Dans notre cas c'est l'objet **Utilisateur** qui devra posséder une liste de **Compétence** avec leur id affecté.

```
Competence java = new Competence();
java.setId(1);
Competence spring = new Competence();
java.setId(5);

Utilisateur utilisateur = new Utilisateur();
utilisateur.setId(42);
utilisateur.setPseudo("toto");
utilisateur.getListeCompetence().add(java);
utilisateur.getListeCompetence().add(spring);

utilisateurDao.save(utilisateur);
```



## >> Aparté : Objet issu de JSON

Toujours afin d'éviter une erreur courante (confondre les **propriétés** de l'objet JAVA avec les **champs** de la table associée).

Si vous souhaitez réaliser l'opération précédente en envoyant des données au format JSON (qui sera alors transformé en objet JAVA), vous devrez envoyer cet objet :

```
{  
    "id": 42,  
    "pseudo": "toto",  
    "listeCompetence": [  
        { "id": 1 },  
        { "id": 5 },  
    ]  
}
```

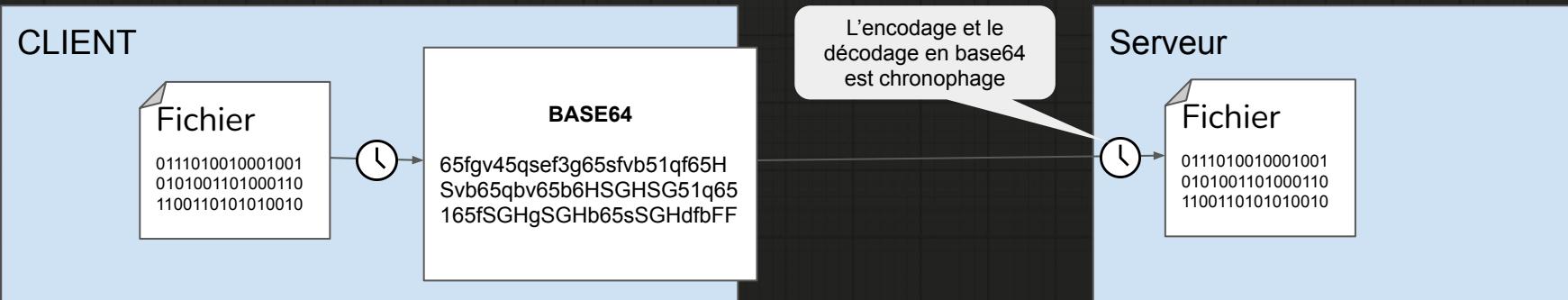


# Upload de fichier

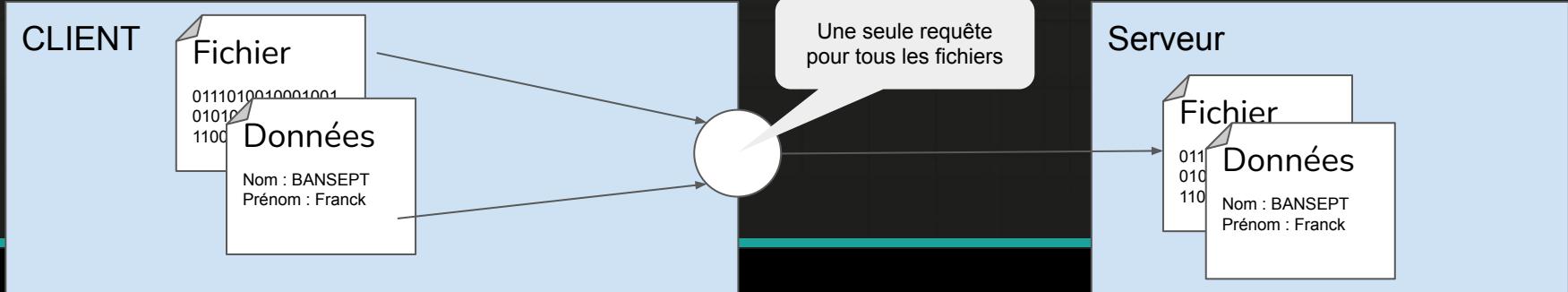
Dans ce chapitre :

## >> Transfert Multipart vs Base64

### Base64



### Multipart



## >> Transfert Multipart vs Base64

Dans la plupart des cas, le format multipart/form-data est plus optimisé car il permet d'envoyer à la fois des données textuelles (ex : *champs de formulaire, json ...*) et binaire (ex : *image, pdf ....*)

Il est largement utilisé et pris en charge pour envoyer des données à partir de formulaires HTML.

Mais il peut y avoir des cas où il n'est pas possible d'utiliser le format multipart/form-data pour envoyer des données :

- Restrictions du navigateur : certains navigateurs, tels que les navigateurs mobiles plus anciens, peuvent ne pas prendre en charge le format multipart/form-data.
- Restrictions du protocole : certaines implémentations de protocoles, telles que les API REST, ne prennent pas en charge le format multipart/form-data.
- Restrictions du serveur : certaines configurations de serveur, telles que les pare-feu, peuvent bloquer les requêtes HTTP en format multipart/form-data.
- Performance : dans des environnements avec des ressources limitées, l'utilisation de la codification base64 peut être plus appropriée en raison de sa simplicité et de ses performances accrues.



## >> Création d'un service d'upload

```
@Service  
public class FichierService {  
  
    @Value("${dossier.upload}")  
    private String dossierUpload;  
  
    public void uploadToLocalFileSystem(MultipartFile multipartFile, String fileName) throws IOException {  
        uploadToLocalFileSystem(multipartFile.getInputStream(), fileName);  
    }  
}
```

Pour faciliter le développement entre développeur il est préférable de créer une propriété qui sera configurable pour chaque développeur avec un profil maven

Pour effectuer un simple test il est possible de mettre un chemin comme :

```
private String dossierUpload = "C://upload-spring"
```



## >> Méthode d'écriture d'un fichier

```
public void uploadToLocalFileSystem(InputStream inputStream, String fileName) throws IOException {  
  
    Path storageDirectory = Paths.get(dossierUpload);  
  
    if(!Files.exists(storageDirectory)){  
        try {  
            Files.createDirectories(storageDirectory);  
        }catch (Exception e){  
            e.printStackTrace();  
        }  
    }  
  
    Path destination = Paths.get(storageDirectory.toString() + "/" + fileName);  
  
    Files.copy(inputStream, destination, StandardCopyOption.REPLACE_EXISTING);  
}
```



## >> Création d'une route d'upload

```
@RestController  
@CrossOrigin  
public class FichierController {  
  
    @Autowired  
    private FichierService fichierService;  
  
    @PostMapping("/test/upload-fichier")  
    public ResponseEntity<String> uploadMultipartFile(@RequestParam("file") MultipartFile file) {  
        try {  
  
            String extension = FilenameUtils.getExtension(file.getOriginalFilename());  
  
            StringBuffer fileName = new StringBuffer();  
            fileName.append(UUID.randomUUID().toString().replaceAll("-", ""));  
            On crée un nom de fichier unique  
  
            fichierService.uploadToLocalFileSystem(file, fileName + "." + extension);  
  
            return ResponseEntity.status(HttpStatus.OK).body("Sauvegarde réussie");  
        } catch (Exception e) {  
            return ResponseEntity.status(HttpStatus.EXPECTATION_FAILED).body("Sauvegarde échouée");  
        }  
    }  
}
```

## >> Fichier public VS Fichier privé

En production, si le fichier est accessible auprès de n'importe quel utilisateur, alors il suffit que le dossier upload soit accessible en lecture et exposé sur le réseau. (*Via un serveur Nginx par exemple*)

Ce serait généralement le cas pour un avatar ou l'image d'un produit par exemple.

Si la **ressource est privée** (*par exemple une image partagée à une personne ou un groupe restreint*)

Il ne faut surtout pas qu'elle soit accessible depuis un dossier public.

Pour pouvoir la récupérer, Il faut faire en sorte que l'utilisateur soit connecté et transmet ses informations de connexion (*cookie, jwt ...* )

La méthode permettant de récupérer un fichier privé sera expliquée après le chapitre sur les JWT.



>> Slides manquantes

TODO

working project :

[https://github.com/fbansept/spring\\_mns\\_23/blob/master/src/main/java/edu/fbansept/demo/controller/UtilisateurController.java](https://github.com/fbansept/spring_mns_23/blob/master/src/main/java/edu/fbansept/demo/controller/UtilisateurController.java)



# Utiliser des query methods

Dans ce chapitre :

## >> Créer ses propres requêtes “Query method”

Spring JPA intègre un mécanisme de création automatique de requête en fonction du nom d'une méthode que l'on déclare. En effet vous avez peut être vu que l'interface JpaRepository possède une méthode `findById` eh bien la requête de cette méthode est auto générée via son nom.

Explication : En analysant le nom de la méthode jpa remarque l'utilisation de plusieurs mot clé :

- `findBy` : il va essayer d'autogénérer une requête permettant de trouver un élément unique
- `Id` : cette recherche se fera sur la propriété `Id`
- Par conséquent la méthode doit prendre en paramètre une valeur du même type que la propriété qui est recherchée.

Ainsi la requête autogénérée si le type générique du JPA repository est Utilisateur sera:  
`select u from Utilisateur u where u.id = ?`



## >> Créer ses propres requêtes “Query method”

Vous avez sûrement remarqué que cette syntaxe SQL est assez particulière, car c'est une requête JPQL qui a été générée (Une adaptation du HQL d'hibernate, lui même une adaptation du SQL mais optimisé pour les ORM)

```
select u from Utilisateur u where u.id = ?
```

En JPQL on ne  
retourne pas un champs  
mais un objet entier

Nous n'accédons pas aux  
champs de la table, mais aux  
propriétés de l'objet



## >> Créer ses propres requêtes “Query method”

findById existe déjà dans JpaRepository, mais il est possible de créer ses propres méthodes, par exemple :

```
@Repository  
public interface UtilisateurDao extends JpaRepository<Utilisateur, Integer> {  
  
    Optional<Utilisateur> findByPseudo(String pseudo);  
  
    List<Utilisateur> findAllByCivilite(Civilite civilite);  
}
```

Cette méthode retournera une liste d'utilisateurs dont la propriété civilité est égale à l'enum de Civilité passé en paramètre

Cette méthode retournera un Optionnal contenant un Utilisateur dont le pseudo est identique au pseudo passé en paramètre



## >> Mécanisme “Query method”

Toutes les méthodes commençant par “find”, “read”, “query”, “count”, “get”, et finissant par “By” sont des méthodes qui peuvent autogénérer des requête jpql.  
(find...By, read...By, query...By, count...By, and get...By)

Chaque méthode peut être complétée avec différents mots clés (voir diapo suivante) et des propriétés du type générique du JpaRepository .



## >> Mot clés “Query method” 1/2

Mot clé	Exemple	Méthode générée
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstname, findByFirstnameIs	... where x.firstname = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	... where x.age <= ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
GreaterThanOrEqual	findByAgeGreaterThanOrEqual	... where x.age >= ?1
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1
IsNull, Null	findByAge(Is)Null	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null



## >> Mot clés “Query method” 2/2

Mot clé	Exemple	Méthode générée
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	... where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	... where x.age not in ?1
True	findByActiveTrue()	... where x.active = true
False	findByActiveFalse()	... where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	... where UPPER(x.firstname) = UPPER(?1)



## >> Query method delete

Les query méthodes commençant par delete doivent obligatoirement être précédée de l'annotation  
exemple :

```
@Transactional  
void deleteByUtilisateurId(int utilisateurId);
```

Sans cette annotation, l'erreur suivante peut survenir :

No EntityManager with actual transaction available for current thread - cannot  
reliably process 'remove' call

Spring boot utilisant un PersistenceContext de type TRANSACTION (*une configuration de l'ORM*) les  
méthodes qui ajoutent ou suppriment des données doivent être réalisées dans des transactions.



## >> Utilisation des dates comme paramètre de requête ou comme données

Attention lors de l'utilisation de date, les fuseaux horaires peuvent impacter le résultat des requête.

Si vous avez définie la timezone de votre base de donnée (par exemple avec le fuseau horaire UTC )  
Mysql : SET time\_zone = 'UTC' ;

Ou que vous vous connectez à la base de donnée via une chaîne de connexion contenant le paramètre **serverTimezone**, ex : jdbc:mysql://localhost:3306/maBaseDeDonnee?serverTimezone=UTC&createDatabaseIfNotExist=true

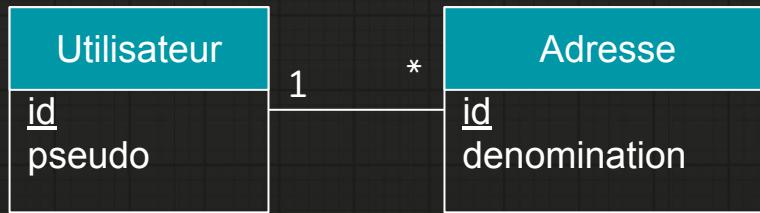
Il est nécessaire de préciser la **timezone** lors de la création de la date, car la valeur par défaut sera le fuseau horaire local (*UTC + 2 en France*) exemple :

```
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
simpleDateFormat.setTimeZone(TimeZone.getTimeZone("UTC"));

entiteDao.deleteByDateEquals(simpleDateFormat.parse("2022-00-00 00:00:00"));
```



>> Query method avec condition impliquant des objets



## >> Query method avec condition impliquant des objets

```
@GetMapping("/utilisateur/{idUtilisateur}/adresses")
public List<Adresse> getAdresses(
    @PathVariable int idUtilisateur) {

    Utilisateur utilisateur = new Utilisateur();
    utilisateur.setId(idUtilisateur);
    return adresseDao.findByUtilisateur(utilisateur);
}
```

```
@Entity
@Getter
@Setter
public class Adresse {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String denomination;

    private boolean principale;

    @ManyToOne
    @JsonIgnore
    private Utilisateur utilisateur;
}
```

```
@Repository
public interface AdresseDao extends JpaRepository<Adresse, Integer> {

    List<Adresse> findByUtilisateur(Utilisateur email);
}
```



# Pagination et tri des requêtes

Dans ce chapitre :

## >> Pagination

Afin de paginer les résultats il est possible de passer un paramètre de type **Pageable** à n'importe quelle **query method**. Ici un exemple de service paramétré permettant de regrouper les résultats par pages et de choisir la liste de résultat à afficher.

`elementPerPage` représente le nombre de résultat regroupé par “page” et `indexPage` le numéro de la page à récupérer (0 étant la première page).

```
@GetMapping("/{}/test/utilisateur-pagination/{elementPerPage}/{indexPage}")  
@JsonView(CustomJsonView.VueUtilisateurIgnoreCompetence.class)  
public List<Utilisateur> testUtilisateurPagination(  
    @PathVariable("elementPerPage") int elementPerPage,  
    @PathVariable("indexPage") int indexPage){  
  
    Pageable pagination = PageRequest.of(indexPage, elementPerPage);  
  
    return utilisateurDao.findAll(pagination).toList();  
}
```



## >> Pagination

Dans l'exemple précédent, la méthode a déjà été déclarée dans l'interface PagingAndSortingRepository dont hérite l'interface JpaRepository que nous avons étendue :

```
Page<T> findAll(Pageable var1);
```

Pour utiliser une **query method** paginable qui ne serait pas encore déclarée, il suffit de l'ajouter dans son propre DAO et d'ajouter un paramètre Pageable.

```
@Repository  
public interface SimpleDao extends JpaRepository<Utilisateur,  
Integer> {  
  
    ...  
  
    Page<Utilisateur> findByAgeGreaterThan (int age, Pageable  
pagination);  
}
```



## >> Tri Ascendant et descendant

Tout comme pour la pagination il est possible de d'instancier un objet de tri (Sort) et de le passer en paramètre d'une **query method**.

```
@GetMapping({"/test/utilisateur-tri-par-pseudo"})
public List<Utilisateur> listeUtilisateurTriParPseudo(){
    Sort triParPseudo = Sort.by("pseudo");
    return simpleDao.findAll(triParPseudo);
}
```

Par défaut le tri est **descendant**, il suffit d'ajouter la méthode .ascending() pour l'inverser

```
@GetMapping({"/test/utilisateur-tri-par-age-ascendant"})
public List<Utilisateur> listeUtilisateurTriParAgeAscendant(){
    Sort triParAge = Sort.by("age").ascending();
    return simpleDao.findAll(triParAge);
}
```



## >> Tri multiple

Il est possible de regrouper plusieurs tri dans un même objet Sort

```
@GetMapping({"/test/utilisateur-tri-par-pseudo-et-age"})
public List<Utilisateur> listeUtilisateurTriParPseudoEtAge(){

    Sort triParPseudo = Sort.by("pseudo");
    Sort triParAgeAscendant = Sort.by("age").ascending();
    Sort regroupementTri = triParPseudo.and(triParAgeAscendant);

    return simpleDao.findAll(regroupementTri);
}
```



## >> Tri et Pagination combinés

Si vous souhaitez effectuer une pagination en plus du tri, c'est l'objet de pagination qui recevra l'objet de tri en 3ème paramètre.

```
@GetMapping({"/test/page-utilisateur-tri-par-pseudo-et-age"})
@JsonView(CustomJsonView.VueUtilisateurIgnoreCompetence.class)
public List<Utilisateur> paginationUtilisateurTriParPseudoEtAge(){

    Sort triParPseudo = Sort.by("pseudo");
    Sort triParAgeAscendant = Sort.by("age").ascending();
    Sort regroupementTri = triParPseudo.and(triParAgeAscendant);

    Pageable pagination = PageRequest.of(0, 5, regroupementTri);

    return simpleDao.findAll(pagination).toList();
}
```

# Déboguer les requêtes créées par Hibernate

Dans ce chapitre :

## >> Activer le mode debug d'Hibernate

En ajoutant les 2 lignes suivantes dans le fichier application.properties, les requêtes effectuée dans la console s'afficheront dans la console

```
spring.jpa.show-sql=true  
spring.jpa.properties.hibernate.format_sql=true
```

En effectuant un appel à un DAO, on peut voir la translation en SQL

```
Optional<Utilisateur> utilisateur = utilisateurDao.findByPseudo(username);
```

```
Hibernate:  
  select  
    utilisateur0_.id as id1_3_,  
    utilisateur0_.pseudo as pseudo2_3_,  
    utilisateur0_.password as password3_3_  
  from  
    utilisateur utilisateur0_  
  where  
    utilisateur0_.id=?
```



# Le mode LAZY et FETCH

Dans ce chapitre :

## >> FETCH vs LAZY Mode

Afin de préserver ne pas effectuer des jointures inutiles, il peut être intéressant de mettre les jointure de nos entités en mode **LAZY**.

```
@ManyToMany(fetch = FetchType.LAZY)
```

```
@ManyToOne(fetch = FetchType.LAZY)
```

Une jointure en mode **LAZY** n'est jamais effectuée sans que l'on ne le demande explicitement.

A l'inverse, si l'association est en mode **EAGER**, une jointure sera toujours effectuée sur la table correspondante au moment de la requête.

**Attention cependant, cela n'est valable que si on applique la configuration adaptée  
(Voir slides suivantes)**



## >> OSIV un "anti design pattern"

L'explication concernant **FETCH** et **LAZY** de la slide précédente, ne peut s'appliquer à Spring Boot uniquement si on désactive un **design pattern** pourtant bien pratique :  
**OSIV (Open Session In View)**

Sans entrer dans les détails techniques, c'est ce pattern qui nous permet de récupérer des objets dont les relations `@ManyToMany` sont toujours complétée (*même lorsque l'on ne l'a pas spécifiquement demandé*).

Cette option, **activée par défaut sur Spring Boot**, divise la communauté en 2 camps.

Le premier le défendant car il permet pour un développeur d'**augmenter sa productivité** car il n'aura pas à se préoccuper des associations (ce qui est un argument tout à fait recevable)

Le deuxième l'accusant de favoriser la création de **requêtes inutilement gourmandes**, le reléguant ainsi au rang de "**anti design pattern**" (une solution inefficace à un problème, voir contre productive)



## >> Désactiver le pattern OSIV

Le simple ajout de la ligne suivante dans application.properties désactive le modèle **OSIV**

```
spring.jpa.open-in-view = false
```

C'est tout à fait normal, le JSON ne peut pas être créé avec des jointures **LAZY**.

Notez que si même si vous ne mettez pas l'option `@ManyToMany(fetch = FetchType.LAZY)` cela reste tout de même l'option par défaut des associations `@ManyToMany` (tout comme les associations `@OneToMany`) à l'inverse des associations `@ManyToOne` qui sont par défaut en mode **EAGER**



## >> La problématique des associations LAZY

Si l'on tente de récupérer un objet avec une association **LAZY**, puis de tenter d'y accéder, on générera une erreur de type **LazyInitializationException**. Exemple :

```
Utilisateur utilisateur = utilisateurDao.findById(1).orElse(null);  
Set<Competence> listeCompetence = utilisateur.getListeCompetence();  
return utilisateur;
```

Également en tentant de renvoyer l'objet via un controller REST (Spring Web tentera à un moment donné de récupérer la liste de compétence afin de construire le JSON), vous tomberez sur une erreur :

...Could not write JSON: failed to lazily initialize a collection of role...

```
@GetMapping({"/utilisateur"})  
public Utilisateur testListeUtilisateur() {  
    Utilisateur utilisateur = utilisateurDao.findById(1).orElse(null);  
    return utilisateur;  
}
```



# Utiliser le langage de Hibernate : HQL

Dans ce chapitre :

## >> Créer ses propres requêtes en HQL

Hibernate nous propose plusieurs manières pour booster notre productivité, comme nous avons pu le voir. Mais il est possible d'écrire directement dans le langage de requêtage d'Hibernate : **Hibernate Query Language**. Il est lui même un complément du **JPQL** (*Java Persistence Query Language*), le langage de requêtage de **JPA** (*le standard de JAVA pour l'implémentation des ORM*). Toute requête **JPQL** est donc valide en **HQL** mais pas l'inverse.

Les **requêtes** exécutées en HQL seront directement transformées dans le langage de notre base de donnée (Mysql, H2, Oracle, PostGreSQL ...) Sa syntaxe se rapproche très fortement du langage SQL standard, exemple :

**HQL** : `FROM Utilisateur U WHERE U.id = 42` → **Mysql** : `SELECT * FROM utilisateur u WHERE u.id = 42`

On peut déjà remarquer ici que l'on n'effectue pas de requête sur une **table**, mais bien sur une **entité** (une classe de notre modèle), ici l'entité **Utilisateur** mappé avec la table **utilisateur**



## >> Créer des requêtes HQL dans un DAO

Grâce à l'annotation @Query il est assez simple de créer ses propres requêtes **HQL**. Il suffit pour cela d'ajouter une méthode dans un DAO et de lui ajouter l'annotation @Query

```
@Query("FROM Utilisateur u WHERE  
u.age >= 18")  
List<Utilisateur>  
trouverUtilisateursMajeurs();
```

Comme pour les autres méthodes de DAO le type de retour peut être modifié afin de répondre au mieux au besoin :

```
@Query("FROM Utilisateur u WHERE u.age >= 18")  
Utilisateur trouverUnUtilisateurMajeur();  
  
@Query("FROM Utilisateur u WHERE u.age >= 18")  
Optional<Utilisateur> trouverOptionnellementUnUtilisateurMajeur();
```

*Note : attention dans les exemples précédents à ne pas retourner + de 1 utilisateur sous peine de retourner une javax.persistence.NonUniqueResultException*



## >> Paramétriser les requêtes HQL

L'utilisation de l'annotation `@Param` sur les paramètres de la méthode, permet de paramétriser la requête. En utilisant dans la requête la notation `:nomParamètre` via la valeur de l'annotation `@Param("nomParamètre")`

```
@Query("FROM Utilisateur u WHERE u.pseudo = :pseudo AND u.motDePasse = :motDePasse")
Utilisateur trouveParMotDePasseEtPseudo(
    @Param("motDePasse") String motDePasse,
    @Param("pseudo") String pseudo);
```

Exemple d'utilisation dans un controller REST:

```
@GetMapping("/{test/utilisateur/{pseudo}/{motDePasse}}")
public Utilisateur testUtilisateurMotDePasse(
    @PathVariable String pseudo, @PathVariable String motDePasse){

    return testHalDao.trouveParMotDePasseEtPseudo(motDePasse, pseudo);
}
```

## >> Paramétriser les requêtes HQL

Il est également possible de passer une collection en paramètre lorsque l'on effectue un filtre via l'opérateur IN

```
@Query(value = "FROM Utilisateur u WHERE u.pseudo IN :pseudos")
List<Utilisateur> trouveUtilisateursParPseudos(@Param("pseudos") Collection<String>
listePseudo);
```

Exemple d'utilisation dans un controller REST:

```
@GetMapping({"/test/utilisateur-par-pseudos"})
@JsonView(CustomJsonView.VueUtilisateurIgnoreCompetence.class)
public List<Utilisateur> trouveUtilisateursParPseudos(){
    return testHqlDao.trouveUtilisateursParPseudos(Arrays.asList("franck", "john"));
}
```



## >> Obtenir des champs personnalisés

Via la clause SELECT, il est possible de limiter le nombre de champs retournés (afin de réduire la taille de la requête par exemple), il sera par contre nécessaire de créer un constructeur compatible.

```
@Query("SELECT new Utilisateur(u.pseudo, u.age) FROM Utilisateur u WHERE u.id = 1")
List<Utilisateur> trouveUtilisateur1AvecMinimumInformation();
```

Il est également possible de récupérer des champs sous forme d'une List de tableau d'objet, plutôt que des entités.

Chaque élément de la liste est une ligne, et chaque index du tableau contient une colonne

```
@Query("SELECT u.pseudo, u.age FROM Utilisateur u WHERE age >= 18")
List<Object[]> trouveInformationUtilisateur1(String motDePasse);
```



## >> Les instructions d'agrégation

Ne pas retourner une instance d'un modèle, permet également d'utiliser les méthodes d'agrégations présente dans le langage HQL (également présente dans les langages SQL en général).

<b>avg</b> (nom de la propriété)	<b>Moyenne</b> des valeurs de la propriété
<b>count</b> (nom de la propriété OU *)	<b>Nombre de fois</b> où la propriété apparaît dans les résultats
<b>max</b> (nom de la propriété)	<b>Maximum</b> de la valeur apparue dans les résultats
<b>min</b> (nom de la propriété)	<b>Minimum</b> de la valeur apparue dans les résultats
<b>sum</b> (nom de la propriété)	<b>Somme</b> de la valeur apparue dans les résultats



## >> Exemple de l'utilisation des instructions d'agrégation

Puisqu'il n'y a qu'un seul champs de type numérique dans notre requête, il est possible de retourner le résultat sous forme d'un entier

```
@Query("SELECT COUNT(*) FROM Utilisateur")
Integer compteUtilisateur();
```

Exemple d'utilisation :

```
@GetMapping={"/test/nombre-utilisateur"})
public Integer compteUtilisateur(){
    return testHqlDao.compteUtilisateur();
}
```



>> Retourner d'autre structure que les entités

TODO DTO + Map<String, Object> exemple dashboard



## >> Exemple de l'utilisation de GROUP BY

Il est plutôt fréquent d'utiliser une fonction d'agrégation avec une clause GROUP BY.

Le type de retour sera alors une liste d'objet où chaque élément de la liste est une ligne, et chaque index du tableau contient une colonne.

```
@Query("SELECT u.age, COUNT(*) FROM Utilisateur u GROUP BY u.age")
List<Object[]> nombreUtilisateurParAge();
```

Au format JSON la requête suivante retournerait :

```
[  
    [ 15, 2 ],  
    [ 32, 1 ]  
]
```

Si 2 utilisateurs avaient 15 ans, et 1 utilisateur avait 32 ans



## >> Fonctions disponibles dans la clause WHERE

Il est plutôt fréquent d'utiliser une fonction d'agrégation avec une clause GROUP BY.

Le type de retour sera alors une liste d'objet où chaque élément de la liste est une ligne, et chaque index du tableau contient une colonne.

```
substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length(), mod()
```



## >> JOINTURES

En plus des traditionnel **JOIN** (*jointure fermée*) et **LEFT JOIN** (*jointure ouverte par la gauche*)

HQL intègre un autre type de jointure : **JOIN FETCH** et **LEFT JOIN FETCH**

Il permet entre autre de charger l'intégralité des champs de la tables jointe de manière à initialiser les données dans les associations **LAZY**. (*Voir slide LAZY vs FETCH*)

```
@Query("FROM Utilisateur u JOIN FETCH u.listeCompetence")
List<Utilisateur> getListeUtilisateurAvecCompetence();
```



## >> CRUD avec HQL

Il est possible d'exécuter des requête **DELETE** et **UPDATE** avec HQL. Il est simplement nécessaire d'ajouter l'annotation `@Modifying` à la méthode.

```
@Modifying  
@Query("UPDATE Utilisateur u SET u.statut.id = 2")  
void updateUtilisateurAvecStatutOccupe();
```

```
@Modifying  
@Query("DELETE Utilisateur u WHERE u.age < 18")  
int supprimeUtilisateursMineurs();
```

Il n'est par contre pas possible d'effectuer des insertions autrement qu'en utilisant les méthode `save`, `saveAll` et `saveAndFlush` vu précédemment. Ou bien via des **requêtes SQL natives** (*voir slide suivante*)



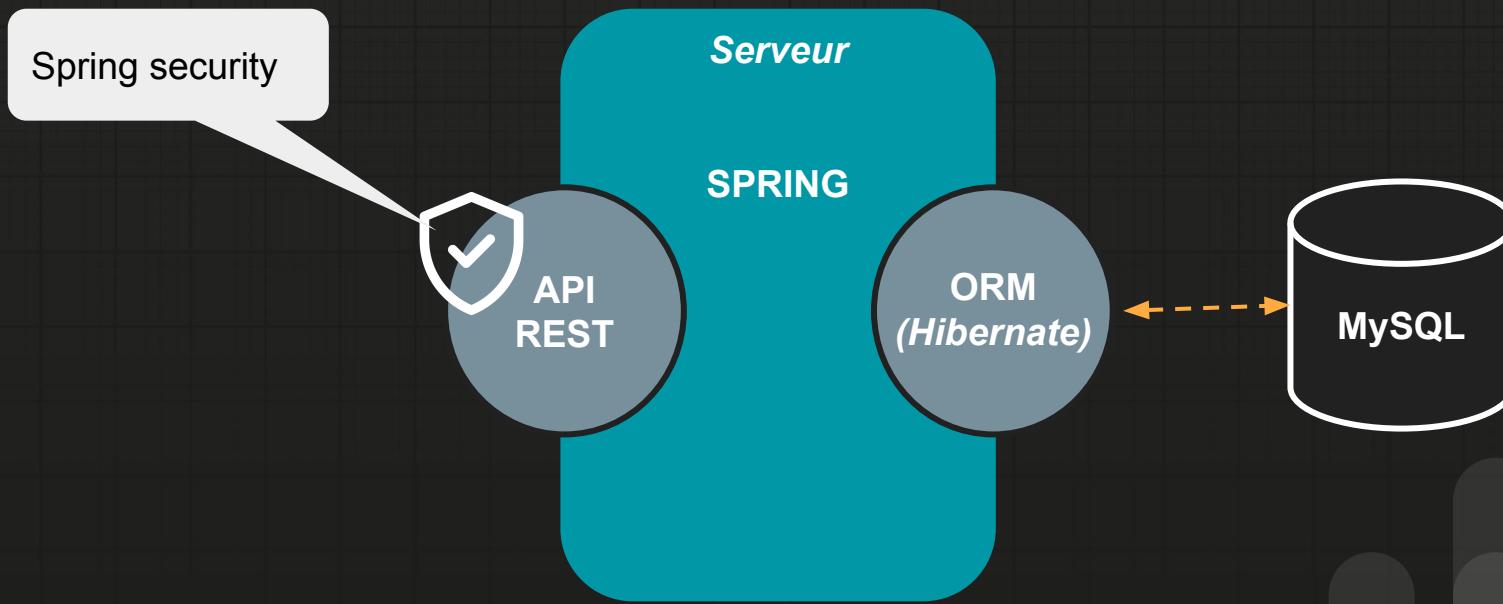
## >> Requêtes SQL Natives

Il est possible d'utiliser des requête SQL native (en MySQL, Oracle SQL ...) selon la base de donnée que nous sommes en train de requêter. Cela permet donc de profiter de fonctionnalité qu'il n'est pas possible d'effectuer en HQL. Il est par contre important de limiter ce genre de méthode afin de favoriser la migration des bases de données.

```
@Modifying
@Query(value = "INSERT INTO utilisateur (pseudo, age) values (:pseudo, :age)",
       nativeQuery = true)
void ajoutUtilisateur(@Param("pseudo") String name, @Param("age") Integer age);
```



# Spring security



# SPRING SECURITY

Dans ce chapitre :

## >> Les différents mot clés de Spring security

**Authentification**

Permettre à l'utilisateur de s'identifier

**Authorization**

Savoir si cet utilisateur a accès aux ressources qu'il demande

**Principal**

C'est l'utilisateur qui s'est identifié grâce à l'authorization

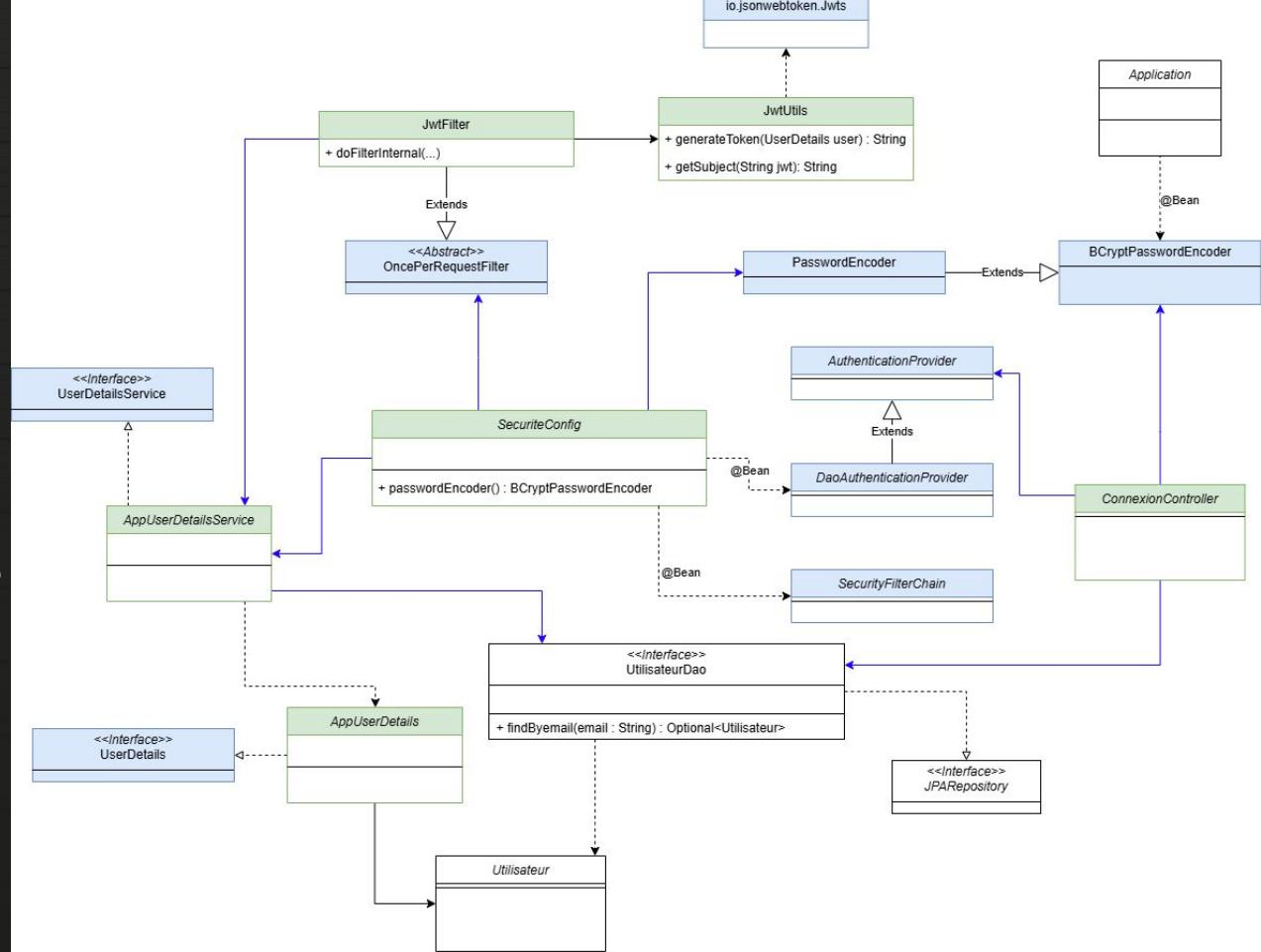
**Granted Authority**

Qu'est ce que cet utilisateur peut faire comme actions ?



Pour information voilà le diagramme de classe de ce que nous devons mettre en place :

- Les entités en Verts sont les classes que nous allons ajouter (ex : *SecurityConfig*)
- Les entités en bleu représentent les classes de Spring Security que nous allons utiliser (ex : *PasswordEncoder*)
- Les entités en blanc représentent les classes déjà existante (ex: l'*interface UtilisateurDao* ou le modèle *Utilisateur*)
- Les liens de dépendance bleu seront des injections (ex : *SecurityConfig* dépendant de *PasswordEncoder* et devra en ajouter un via l'annotation *@Autowired*)
- Les liens avec *@Bean* définissent quelle entité est à l'origine de la création de la dépendance (ex:*SecurityConfig* va créer un dépendance *DaoAuthenticationProvider*)



## >> Dépendances maven de Spring security

Ajoutez les dépendance suivantes dans le noeud <**dependencies**> du fichier pom.xml

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

N'oubliez pas de mettre à jour l'application via l'icône  qui apparaît alors en haut à droite.



## >> La sécurité de base de spring security

Le simple fait d'ajouter cette dépendance fait que vous devez vous dorénavant vous connecter pour obtenir une ressource :

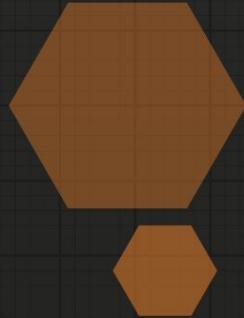
- Un formulaire à été ajouté à votre application et apparaîtra tant que vous ne serez pas connecté
- Le nom d'utilisateur est "user"
- Le mot de passe vous est communiqué dans la console de l'application (attention il est différent à chaque fois que l'application est démarrée)
- L'identification se fait par Session ID
- Il est possible de customiser le nom de l'utilisateur et le mot de passe via les propriétés à ajouter dans application.properties

`spring.security.user.name=fbansept`

`spring.security.user.password=root`

Après ce test vous pouvez supprimer ces informations de votre configuration pour la suite





# Configuration de spring security

*Chapitre : Spring security*

## >> Authentification VS Autorisation

Bien que les deux concepts sont liés, il ne représente pas la même action en terme de sécurité.

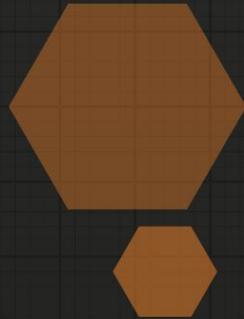
L'**authentification** désigne les instructions à réaliser pour reconnaître une personne dans notre système.  
*Actuellement celui-ci consiste simplement à renseigner le pseudo "user" et saisir le mot de passe fourni dans la console. Un système de Session permet alors de se rappeler qu'il est connecté.*

Nous verrons par la suite des systèmes reposant sur une base de donnée et sur des tokens

L'**autorisation** désigne les instructions à mettre en place pour savoir si une personne peut accéder à une certaine ressource (une URL dans notre cas).

*Par exemple, un simple utilisateur même authentifié ne peut pas exécuter des actions réservées aux administrateurs (comme supprimer un autre utilisateur)*





# Configurer l'authentification

*Chapitre : Spring security*

## &gt;&gt; Configurer l'authentification

Afin de définir le système d'authentification, il nous faut créer un nouveau fichier dans un nouveau package : security

Ce fichier sera la classe principale de toute la sécurité concernant notre application

Spring Security attend une dépendance de type **AuthenticationProvider** nous allons donc créer une méthode avec l'annotation **@Bean** pour lui fournir

Nous allons spécialiser l'objet retournant en instantiant un type **DaoAuthenticationProvider** (qui implémente l'interface **AuthenticationProvider**)

Cet objet a besoin de savoir quel système de hashage notre base de donnée va utiliser : ici **BCryptPasswordEncoder**

Il a également besoin d'une instance de **UserDetailsService** qui nous permettra d'indiquer de quel façons on identifie l'utilisateur

*security/ConfigurationSecurite (nouveau fichier)*

```
@EnableWebSecurity
@Configuration
public class ConfigurationSecurite{

    @Autowired
    private UserDetailsService uds;

    @Autowired
    private BCryptPasswordEncoder encoder;

    @Bean
    public AuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvider();
        authenticationProvider.setUserDetailsService(uds);
        authenticationProvider.setPasswordEncoder(encoder);
        return authenticationProvider;
    }
}
```

## &gt;&gt; Créer la dépendance BCryptPasswordEncoder

Dans le fichier précédent, nous avons besoin d'un Bean BCryptPasswordEncoder, il serait tentant de le définir sur le même fichier, mais cela donnerait lieu à une dépendance circulaire : pour être créé, BCryptPasswordEncoder a besoin que l'instance de ConfigurationSecurite soit initialisée, mais ConfigurationSecurite à besoin que BCryptPasswordEncoder soit créé.

On va donc créer notre Bean dans un autre fichier, soit dans le fichier principal de l'application, soit dans un fichier de configuration à part

src/main/MonApplication

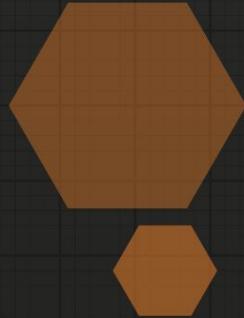
```
@SpringBootApplication  
public class MonApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(MonApplication.class, args);  
    }  
  
    @Bean  
    public BCryptPasswordEncoder getBCryptPasswordEncoder ()  
    {  
        return new BCryptPasswordEncoder();  
    }  
}
```

OU

src/main/ConfigurationMonApplication (nouveau fichier)

```
@Configuration  
public class ConfigurationMonApplication {  
  
    @Bean  
    public BCryptPasswordEncoder getBCryptPasswordEncoder ()  
    {  
        return new BCryptPasswordEncoder();  
    }  
}
```





## Digression : Hasher les mots de passe

*Chapitre : Spring security*

## >> Utiliser les bons termes

En cryptographie les termes sont importants, ainsi en Français on utilise le mot “cryptage” pour tout opération qui consisterait à rendre le mot de passe invisible... Sauf que le mot “cryptage” n'existe pas.

Voici les termes corrects à employer :

**Encoder** : Processus de modification d'une valeur (texte, fichier, ...) pour un autre. (exemple : compresser un fichier en .zip, encoder une vidéo en mp4 ...)

**Hacher** : utiliser un algorithme permettant de modifier un texte. Il est impossible d'effectuer l'opération inverse (exemples : MD5, BCrypt, SHA1)

**Chiffrer** : encoder un texte de telle sorte que seuls des personnes autorisées puissent le déchiffrer.

Il peut être :

Symétrique : la même clé de chiffrement est utilisée pour chiffrer/déchiffrer un message.

Asymétrique : Les clés pour chiffrer et déchiffrer sont différentes (exemple HTTPS)



## >> Digression : MD5

L'algorithme MD5 est un algorithme destructif très utilisé durant les premières décennies d'Internet.  
Il permet de transformer une chaîne de caractère en une suite de lettres et de nombres incompréhensibles  
Vous pouvez le tester sur [md5.fr](http://md5.fr)

ex :

- “root” = “63a9f0ea7bb98050796b649e85481845”
- “azety” = “ab4f63f9ac65152575886860dde480a1”

Il est dit “destructif” car il n'est pas possible d'effectuer les opérations inverse pour obtenir la valeur originale.

**Seulement avec le temps, comme chaque chaîne originale correspond toujours à la même chaîne haché, des tables d'associations ont été créées (rainbow table) permettant de vite trouver qu'elle chaîne haché correspond à quel chaîne originale**

L'algorithme MD5 est donc à éviter, même si il existe une technique consistant à le rendre moins prévisible (“ajout de sel” pour les + curieux)



## >> Digression : Bcrypt

L'algorithme **Bcrypt** est également **destructif** (comme MD5).

Néanmoins, il ne donne pas le même résultat lorsque l'on hache 2 fois la même chaîne originale.

root :

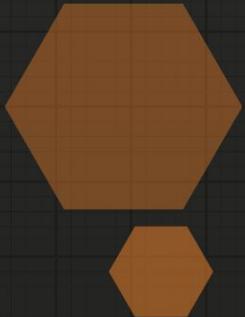
- \$2y\$10\$aGL6dmKPTPRuebshOOQseOqOF.pKkEPXOsSd442mMhJHsDQyFolPy
- \$2y\$10\$nrFtwy0gXQY/T259/O/M1eDVhucB2DNGgP5Lnm85v3XdGIZzxCh1K
- \$2y\$10\$X7gp5c/dsRUpZi7Vf/h31OOnqk4aObwkKOeiXNSii88Ar3wHlQxZu ...

Il n'est donc **pas possible de réaliser des tables d'associations** avec un tel algorithme.

Pour vérifier si un mot de passe original correspond à un mot de passe hashé, on ne peut pas tester l'égalité du mot de passe hashé stocké en base de donnée avec le mot de passe fourni par l'utilisateur que l'on hachera de nouveau (puisque celui-ci nous donnerait un hash différent)

Mais on peut en revanche tester **la compatibilité** d'un mot de passe en clair avec sa version hashé.  
Ainsi root **est compatible** avec \$2y\$10\$aGL6dmKPTPRuebshOOQseOqOF.pKkEPXOsSd442mMhJHsDQyFolPy





# User details service

*Chapitre : Spring security*

## >> Principe du user details service

Cette technique d'authentification se veut générique.

Nous pouvons l'adapter à n'importe quel besoin pour notre application.

Nous devrons simplement utiliser des classes compréhensible par Spring security :

Un **utilisateur** sera stocké dans une classe implémentant **UserDetails**

Un **rôle** sera une **Authorities**

L'information permettant d'**identifier un utilisateur** sera un **Username** (nom, email, pseudo ...)

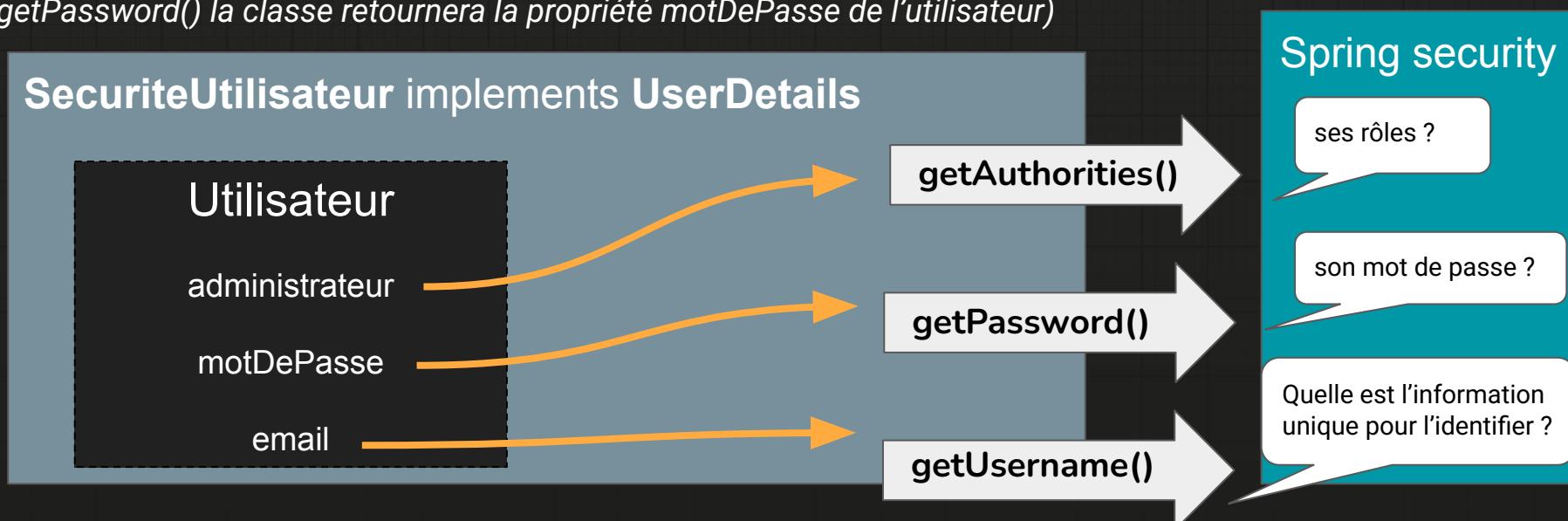
La classe permettant de **récupérer un utilisateur** par son Username devra implémenter **UserDetailsService**



## >> A quoi sert l'interface user details ?

Spring security ne pas comprendre ce que c'est qu'un "rôle" ou un "motDePasse", nous allons devoir utiliser une interface pour pouvoir communiquer avec lui.

Cette interface c'est UserDetails, elle sera implémentée dans une classe contenant une instance d'utilisateur et les méthodes permettant de communiquer avec Spring Security (ex : en appelant `getPassword()` la classe retournera la propriété `motDePasse` de l'utilisateur)



## &gt;&gt; Implémentation de UserDetails

Nous allons créer une classe `AppUserDetails` implémentant l'interface `UserDetails` (*il y a 3 méthodes à implémenter obligatoirement et 3 optionnelles*).

Le constructeur de cette classe prendra en paramètre un utilisateur.

*security/AppUserDetails (nouveau fichier)*

```
public class SecuriteUtilisateur implements UserDetails {  
  
    private Utilisateur utilisateur;  
  
    public SecuriteUtilisateur (Utilisateur utilisateur) {  
        this.utilisateur = utilisateur;  
    }  
  
    public Utilisateur getUtilisateur() {  
        return utilisateur;  
    }  
}
```



## &gt;&gt; Implémentation de UserDetails

Les 2 premières méthodes que nous allons implémenter, vont permettre de récupérer le mot de passe, et l'information permettant d'identifier l'utilisateur (*ici son email*)

*security/AppUserDetails*

```
@Override  
public String getPassword() {  
    return utilisateur.getMotDePasse();  
}  
  
@Override  
public String getUsername() {  
    return utilisateur.getEmail();  
}
```



## &gt;&gt; Implémentation de UserDetails

La troisième méthode consiste à renvoyer la liste des rôles de l'utilisateur. Dans notre cas, la gestion des rôles se limite à un booléen indiquant si l'utilisateur est un administrateur. Le cas échéant, une instance de *SimpleGrantedAuthority* (représentant un rôle pour Spring) sera ajouté à la liste retournée

security/AppUserDetails

```
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {

    ArrayList<SimpleGrantedAuthority> listeAuthority = new ArrayList<>();
    if(this.utilisateur.isAdministre()) {
        listeAuthority.add(new SimpleGrantedAuthority("ROLE_ADMIN"));
    }
    return listeAuthority;
}
```



>> Fil rouge

>> Ajouter la désactivation de l'utilisateur

TODO



## &gt;&gt; Implémentation de UserDetails

Les 4 dernières méthodes n'ont pas besoin d'être obligatoirement implémentée

Elles permettent de définir l'utilisateur comme étant incapable de se connecter pour 4 raisons différentes

Dans notre cas nous allons seulement définir la dernière :  
isEnabled

*security/AppUserDetails*

```
@Override  
public boolean isAccountNonExpired() {  
    return true;  
}  
@Override  
public boolean isAccountNonLocked() {  
    return true;  
}  
@Override  
public boolean isCredentialsNonExpired() {  
    return true;  
}  
@Override  
public boolean isEnabled() {  
    return this.utilisateur.isAdministrateur();  
}
```



## &gt;&gt; Création d'un Dao personnalisé

Afin de pouvoir récupérer l'utilisateur correspondant à l'identifiant fourni (email, pseudo ...) nous allons ajouter une query method dans UtilisateurDao

*dao/UtilisateurDao*

```
@Repository  
public interface UtilisateurDao extends JpaRepository<Utilisateur, Integer> {  
    Optional<Utilisateur> findByEmail(String email);  
}
```



## &gt;&gt; Implementation de UserDetailsService

L'implementation de UserDetailsService est la classe que nous allons passer à l'authentification manager.

A l'aide de la requête permettant de trouver une personne par son pseudo, si l'utilisateur n'existe pas, on lance une exception, sinon on renvoie une instance de AppUserDetails à partir de cet utilisateur

*security/UtilisateurService (nouveau fichier)*

```
@Service
public class AppUserDetailsService implements UserDetailsService {

    @Autowired
    UtilisateurDao utilisateurDao;

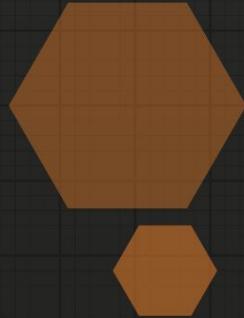
    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException
    {

        Optional<Utilisateur> optionalUtilisateur = utilisateurDao.findByEmail(email);

        if(optionalUtilisateur.isPresent()) {
            return new AppUserDetails(optionalUtilisateur.get());
        }

        throw new UsernameNotFoundException("Email introuvable");
    }
}
```





## Digression : Le chaînage de méthode

*Chapitre : Spring security*

## >> Le chaînage de méthode

Le **chaînage de méthode** est une façons de construire des classe afin de faciliter l'utilisation de ces dernières, de rendre le code plus concis, plus lisible.

Note : Ce concept est également appelé par abus de langage “interface fluide” (luent interface), mais il n'est qu'un des composants de ce qu'est réellement une interface fluide (*que nous ne verrons pas dans ce cours*)

Un exemple de classes utilisant ce concept est l'**API stream** de JAVA.

```
Arrays.asList("franck", "tom", "theo").stream()
    .filter(o -> o.startsWith("t"))
    .map(o -> o.toUpperCase())
    .findFirst()
    .orElse(null);
```

Ici chaque méthode retourne un objet dont une méthode sera appelée. L'opération est renouvelé jusqu'à obtenir le résultat final



## >> Exemple

Appliquons ce concept à une classe simple.

Analysez les accesseurs (*setNom* et *setPrenom*) de la classe Utilisateur.

Il retourne l'instance de la classe elle même plutôt que de ne rien retourner (*void*)

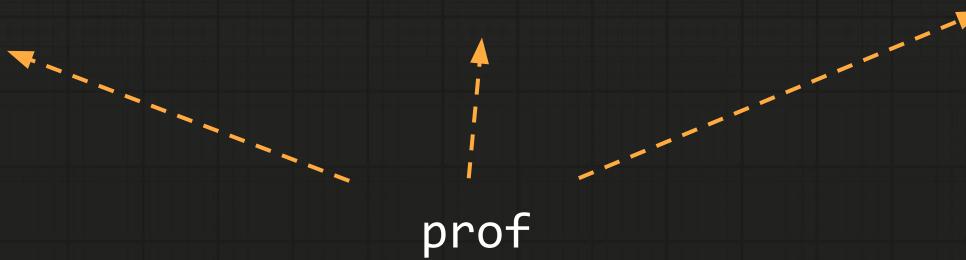
Ce simple ajout va nous permettre après l'appel de l'un de ces accesseurs d'appeler une autre méthode de l'instance de la classe retournée (qui sera la même que l'instance à l'origine de l'appel)

```
public class Utilisateur {  
  
    private String nom;  
    private String prenom;  
  
    public Utilisateur setNom(String nom) {  
        this.nom = nom;  
        return this;  
    }  
  
    public Utilisateur setPrenom(String prenom) {  
        this.prenom = prenom;  
        return this;  
    }  
  
    public String nomComplet () {  
        return this.prenom + " " + this.nom;  
    }  
}
```



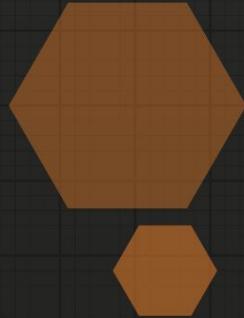
## >> Exemple

```
Utilisateur prof = new Utilisateur();  
  
System.out.println(  
    prof.setNom("BANSEPT").setPrenom("Franck").nomComplet()  
);
```



Puisque “setNom” retourne l’instance de la classe qui l’a appelé (*c’est à dire prof*) alors c’est encore `prof` qui est retourné, on peut donc enchaîner un appel de la méthode `setPrenom` qui retourne également `prof`. On finit en appelant la méthode `nomComplet` qui retourne un `String`





# Configurer l'authorization

*Chapitre : Spring security*

## &gt;&gt; Configurer les Règles CORS

*security/ConfigurationSecurite*

```
private CorsConfigurationSource corsConfigurationSource() {  
    CorsConfiguration corsConfiguration =new CorsConfiguration();  
    corsConfiguration.setAllowedOrigins(Listof(""));  
    corsConfiguration.setAllowedMethods(Listof("GET", "POST", "DELETE", "PUT", "PATCH"));  
    corsConfiguration.setAllowedHeaders(Listof(  
        "Authorization", // Utilisé pour envoyer un JWT  
        "Content-Type", // Envoie vers serveur d'un format particulier (ex: Content-Type: application/json)  
        "Accept", //Réception depuis le serveur d'un format particulier (ex: Accept: application/json)  
        "Origin", //Permet d'indiquer d'où la requête provient (quelle IP / domaine ...)  
        "X-Requested-With" //Permet d'indiquer quel a été l'action à l'origine de la requête, souvent dans le  
but de préciser que c'est javascript qui l'exécute et non une page (ex : X-Requested-With: XMLHttpRequest)  
    ));  
    corsConfiguration.setExposedHeaders(Listof("Authorization"));  
  
    // Permettre au client d'accéder au token (ex:JWT) si il envoyé en réponse (ex: pour un a fraîchissement)  
    UrlBasedCorsConfigurationSource source =new UrlBasedCorsConfigurationSource();  
  
    //Attribue la configuration à l'ensemble des requêtes  
    //(ex : il serait possible de ne l'appliquer qu'au requêtes commençant par "/api" avec: "/api/**")  
    source.registerCorsConfiguration(/**, corsConfiguration);  
    return source;  
}
```



## &gt;&gt; Configurer l'autorisation

security/ConfigurationSecurite

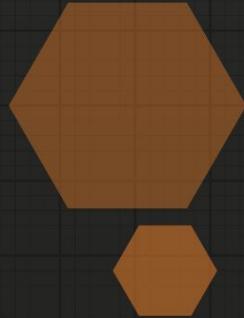
```
@Bean  
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {  
  
    return http  
  
        .csrf(config -> config.disable())  
  
        .cors(config -> config.configurationSource( corsConfigurationSource () ))  
  
        .sessionManagement(config -> config.sessionCreationPolicy(SessionCreationPolicy. STATELESS) )  
  
        .build()  
}
```

On désactive la protection contre les failles CSRF (protection par jeton de formulaire)

On configure la protection CORS via la méthode que nous venons de créer

On indique que l'authentification ne se fait pas par cookies





# Gestion de droit avec un booléen

*Chapitre : Spring security*

### Utilisateur

- id : entier
- pseudo : chaîne
- mot de passe : chaîne
- administrateur : booléen
- actif : booléen



## &gt;&gt; Configuration du modèle

La gestion de droit peut se faire avec un booléen qu'il n'y a que 2 droits possibles  
Dans le cas suivant : l'utilisateur est administrateur, ou ne l'est pas.

model/Utilisateur

```
@Entity
public class Utilisateur {

    ...
    private boolean administrateur;

    ...
}
```

model/Utilisateur

```
@Entity
public class Utilisateur {

    ...
    private boolean correcteur;
    private boolean redacteur;
    private boolean validateur;

    ...
}
```

Il est également possible d'avoir plusieurs booléens à condition que la totalité des combinaisons qu'ils représentent soient possibles (*ex : un utilisateur qui serait correcteur / validateur mais qui ne serait pas redacteur, ou un validateur qui ne serait ni correcteur, ni rédacteur ....*)

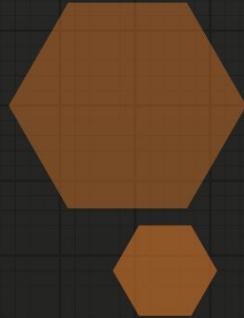
>> Configuration de la méthode *getAuthorities* de l'implémentation de *UserDetails*

Avec cette approche, le rôle retourné ne dépend que de la valeur du booléen *administrateur* de l'utilisateur

*security/AppUserDetails*

```
@Override  
public Collection<? extends GrantedAuthority> getAuthorities () {  
  
    return List.of(new SimpleGrantedAuthority(  
        utilisateur.isAdministrateur()  
    ? "ROLE_ADMIN"  
    : "ROLE_USER"));  
}
```





Gestion de droit avec 1 rôle parmi  
une liste

*Chapitre : Spring security*

## >> UML

Dans cette gestion de rôle, une liste de rôles est définie dans une table. L'utilisateur se voit affecter l'un de ces rôles.

*Note : cette gestion de rôle est particulièrement adaptée pour les applications dont les utilisateurs ont un rôle particulier, et qu'il y a plus de 2 rôles possibles (ex : lecteur, rédacteur, administrateur)*



## >> Configuration du model

model/Utilisateur

```
@Entity  
public class Utilisateur {  
  
    ...  
  
    @ManyToOne  
    @JoinColumn(nullable = false)  
    protected Role role;  
  
    ...  
}
```

model/Role

```
public class Role {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    protected Integer id;  
  
    protected String nom;  
}
```



## >> Configuration de la méthode *getAuthorities* de l'implémentation de *UserDetails*

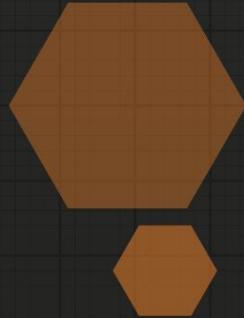
*security/AppUserDetails*

```
@Override  
public Collection<? extends GrantedAuthority> getAuthorities () {  
  
    return List.of(new SimpleGrantedAuthority( this.utilisateur.getRole().getNom() ));  
  
}
```

Dans le cas où l'on fait le choix de ne pas stocker le préfixe “ROLE\_” dans la base de donnée, il est nécessaire de l'ajouter dans le nom du droit :

```
@Override  
public Collection<? extends GrantedAuthority> getAuthorities () {  
  
    return List.of(new SimpleGrantedAuthority( "ROLE_" + this.utilisateur.getRole().getNom() ));  
  
}
```





## Gestion de droit avec 1 rôle parmi une ENUMERATION

*Chapitre : Spring security*

## &gt;&gt; UML

Cette gestion de rôle est une alternative intéressante à la méthode précédente. Plutôt que de lier l'utilisateur à une liste de rôle, on lui attribue le rôle via une énumération.

Les rôles n'ayant pas pour vocation à être changés sans modification dans la logique métier, ils sont de très bon candidats à l'utilisation d'un ENUM



## &gt;&gt; Configuration du model

model/Utilisateur

```
@Entity  
public class Utilisateur {  
  
    ...  
  
    @Enumerated (EnumType.STRING)  
    @Column (columnDefinition = "ENUM('CLIENT', 'VENDEUR', 'ADMIN')")  
    protected Role role;  
  
    ...  
}
```

model/Role

```
public enum Role {  
  
    CLIENT,  
    VENDEUR,  
    ADMIN  
}
```

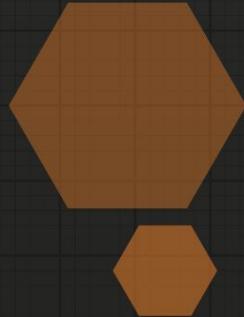


>> Configuration de la méthode *getAuthorities* de l'implémentation de *UserDetails*

*security/AppUserDetails*

```
@Override  
public Collection<? extends GrantedAuthority> getAuthorities () {  
  
    return List.of(new SimpleGrantedAuthority( utilisateur.getRole().name ()));  
  
}
```





## Gestion de droit avec plusieurs rôles parmi une liste

*Chapitre : Spring security*

## >> UML

Dans cette gestion de rôle, comme dans l'exemple précédent, une liste de rôles est définie dans une table. L'utilisateur se voit affecté un ou plusieurs rôles.

*Note : cette gestion de rôle est particulièrement adaptée pour les applications qui auraient plusieurs "modules" (ex : comptabilité, marketing, stock ...) l'utilisateur pourrait se voir attribuer tout ou partie des rôles proposés (ex : il peut accéder au module comptabilité et stock, mais non au module marketing)*



## >> Configuration du model

model/Utilisateur

```
@Entity  
public class Utilisateur {  
  
    ...  
  
    @ManyToMany  
    protected List<Role> roles = new  
    ArrayList<>();  
  
    ...  
}
```

model/Role

```
public class Role {  
  
    @Id  
    @GeneratedValue (strategy = GenerationType. IDENTITY)  
    protected Integer id;  
  
    protected String nom;  
}
```



## >> Configuration du Dao

Puisque la liaison entre *Role* et *Utilisateur* est de type *ManyToMany*, la méthode `findById` de *UtilisateurDao* chargera les rôles en mode *LAZY* (voir le chapitre correspondant)

Le pattern OSIV (*permettant de charger automatiquement les liaisons OneToMany et ManyToMany*) ne serait pas appliqué ici, puisque l'on n'utilise pas de *RestContrôleur* (*on passe simplement à Spring Security notre utilisateur*).

Il est donc nécessaire d'ajouter une requête JQL permettant de récupérer un utilisateur par son email, avec une jointure *FETCH* sur ses rôles

*model/UtilisateurDao*

```
@Repository
public interface UtilisateurDao extends JpaRepository<Utilisateur, Integer> {

    @Query("SELECT u FROM Utilisateur u LEFT JOIN FETCH u.roles WHERE u.email = :email")
    Optional<Utilisateur> findByEmailWithRoles(@Param("email") String email);
}
```



## >> Configuration de l'implémentation de *UserDetailsService*

*security/AppUserDetailsService*

```
@Override
public AppUserDetails loadUserByUsername (String email) throws UsernameNotFoundException {
    Utilisateur utilisateur = utilisateurDao
        .findByEmailWithRoles (email)
        .orElseThrow(() -> new UsernameNotFoundException( "Mauvais pseudo / mot de passe" ));

    return new AppUserDetails(utilisateur);
}
```



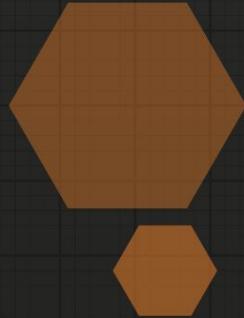
## >> Configuration de la méthode `getAuthorities` de l'implémentation de `UserDetails`

`security/AppUserDetails`

```
@Override  
public Collection<? extends GrantedAuthority> getAuthorities () {  
  
    ArrayList<SimpleGrantedAuthority> listeAuthority = new ArrayList<>();  
  
    for (Role role : this.utilisateur.getRoles ()) {  
        listeAuthority.add( new SimpleGrantedAuthority(role.getNom ()));  
    }  
  
    return listeAuthority;  
}
```

Identique mais en utilisant l'api stream

```
@Override  
public Collection<? extends GrantedAuthority> getAuthorities () {  
    return this.utilisateur.getRoles () .stream()  
        .map(role -> new SimpleGrantedAuthority(role.getNom ()))  
        .toList();  
}
```



# Gestion de droit avec des rôles obtenus par héritage

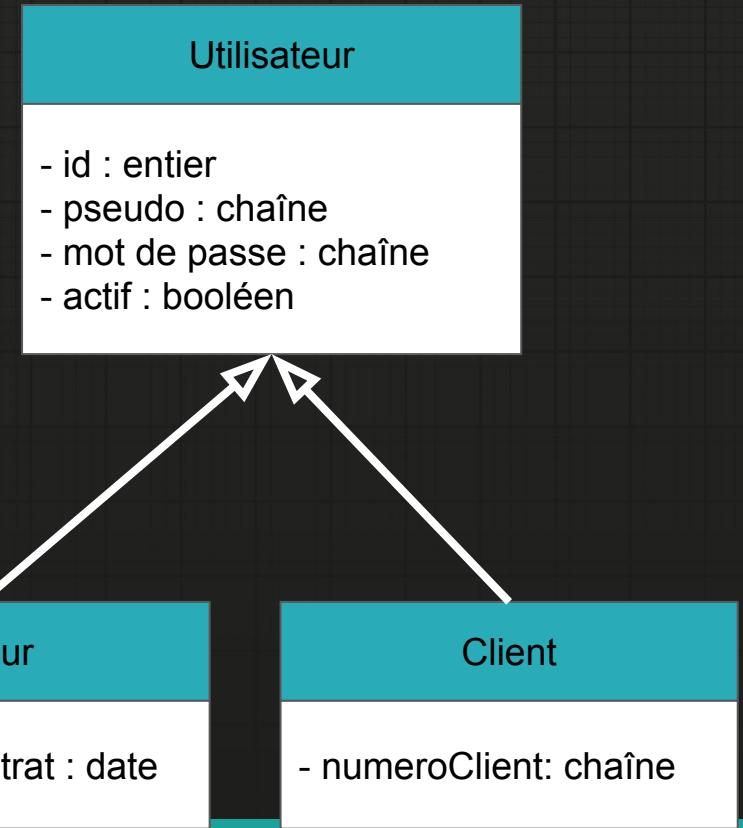
*Chapitre : Spring security*

## >> UML

Cette gestion de rôle se base sur un héritage qui se ferait sur la table utilisateur.

Ici, il est inutile d'ajouter une table rôle puisque c'est l'existence de l'utilisateur dans les table hérité qui font que celui-ci possède tel ou tel rôle.

Dans cet exemple, si l'id de l'utilisateur existe dans la table "vendeur" alors l'utilisateur doit avoir le rôle "vendeur"



## >> Configuration du model

*model/Utilisateur*

```
@Inheritance(strategy = InheritanceType.JOINED)
@Entity
public class Utilisateur {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    protected Integer id;

    ...
}
```

*model/Client*

```
@Entity
public class Client extends Utilisateur {

    protected String numero;

}
```

*model/Vendeur*

```
@Entity
public class Vendeur extends Utilisateur {

    protected LocalDate dateDebutContrat;

}
```



## >> Configuration UserDetails

```
public class AppUserDetails implements UserDetails {  
  
    protected Client client;  
    protected Vendeur vendeur;  
    protected String email;  
    protected String password;  
  
    public AppUserDetails(Client client) {  
        this.client = client;  
        this.email = client.getEmail();  
        this.password = client.getPassword();  
    }  
  
    public AppUserDetails(Vendeur vendeur) {  
        this.vendeur = vendeur;  
        this.email = vendeur.getEmail();  
        this.password = vendeur.getPassword();  
    }  
}
```

Dans cet exemple, il est possible de créer un AppUserDetails englobant un Client ou un Vendeur (*l'une des propriétés client ou vendeur sera null et l'autre non*)

```
@Override  
public Collection<? extends GrantedAuthority> getAuthorities() {  
  
    return List.of(new SimpleGrantedAuthority(  
        client != null  
            ? "ROLE_CLIENT"  
            : "ROLE_VENDEUR"));  
}  
  
@Override  
public String getPassword() {  
    return password;  
}  
  
@Override  
public String getUsername() {  
    return email;  
}
```



## >> Configuration UserDetailsService

```
@Service
public class AppUserDetailsService implements UserDetailsService {

    protected VendeurDao vendeurDao;
    protected ClientDao clientDao;

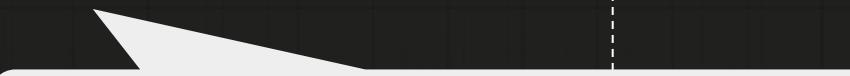
    public AppUserDetailsService(VendeurDao vendeurDao, ClientDao clientDao) {
        this.vendeurDao = vendeurDao;
        this.clientDao = clientDao;
    }

    @Override
    public AppUserDetails loadUserByUsername(String email) throws UsernameNotFoundException {

        Optional<Vendeur> vendeur = vendeurDao.findByEmail(email);
        Optional<Client> client = clientDao.findByEmail(email);

        if (client.isPresent()) {
            return new AppUserDetails(client.get());
        } else if (vendeur.isPresent()) {
            return new AppUserDetails(vendeur.get());
        }

        throw new UsernameNotFoundException(email);
    }
}
```

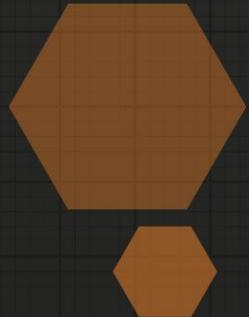


Selon le cas où l'on récupère un client ou un vendeur, c'est l'un ou l'autre des constructeurs de AppUserDetails qui sera utilisé



# Gérer les jetons d'authentification *(JWT : Json Web Token)*

Dans ce chapitre :



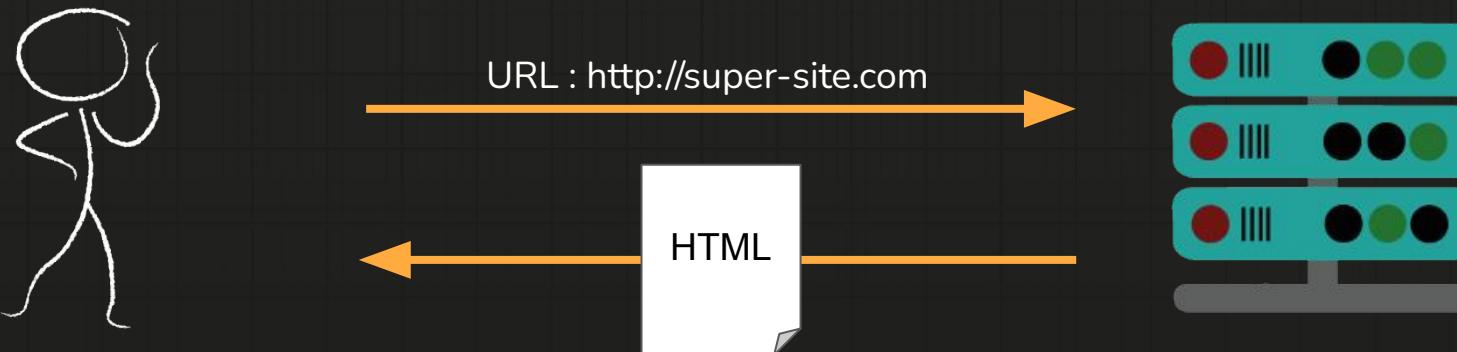
# La problématique HTTP

*Chapitre : Echange serveur*

## >> La problématique HTTP

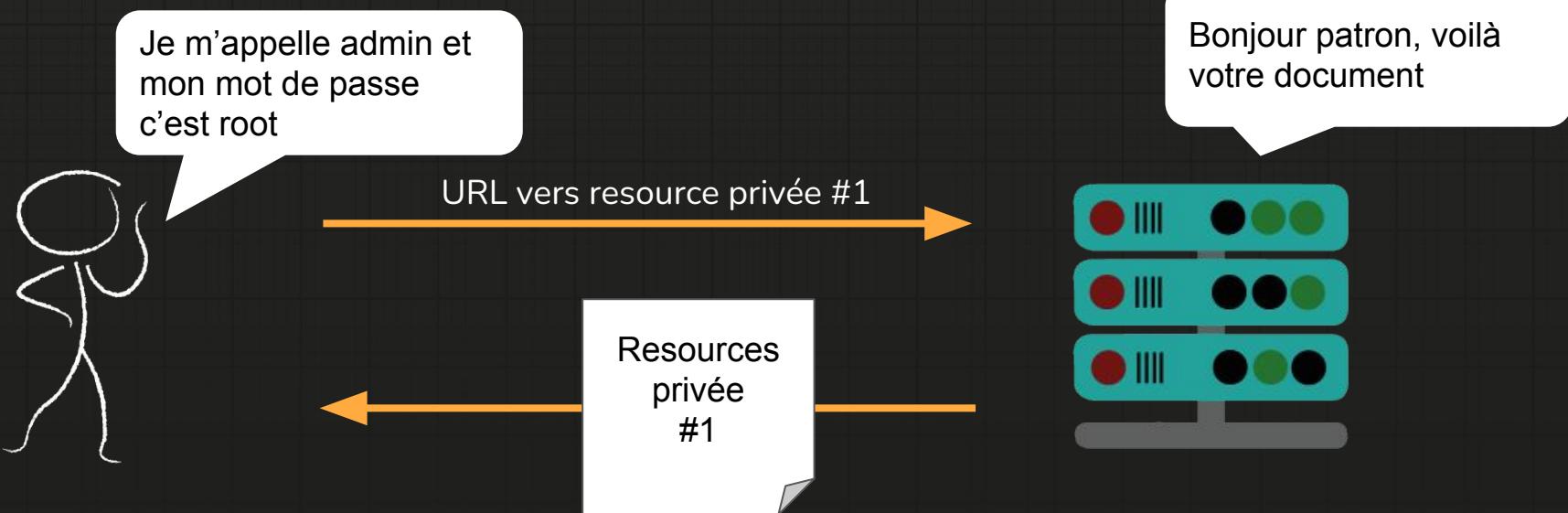
La problématique majeure avec le protocole http est qu'il doit contenir toutes les informations nécessaires à la communication entre le client et le destinataire, rien n'est sauvegardé. (stateless).

Quand il s'agit d'obtenir une ressource que n'importe qui peut avoir accès, ce n'est pas un problème. Un utilisateur demande une ressource via une url, et le serveur lui renvoie (Une page html, une image, un service web public....)



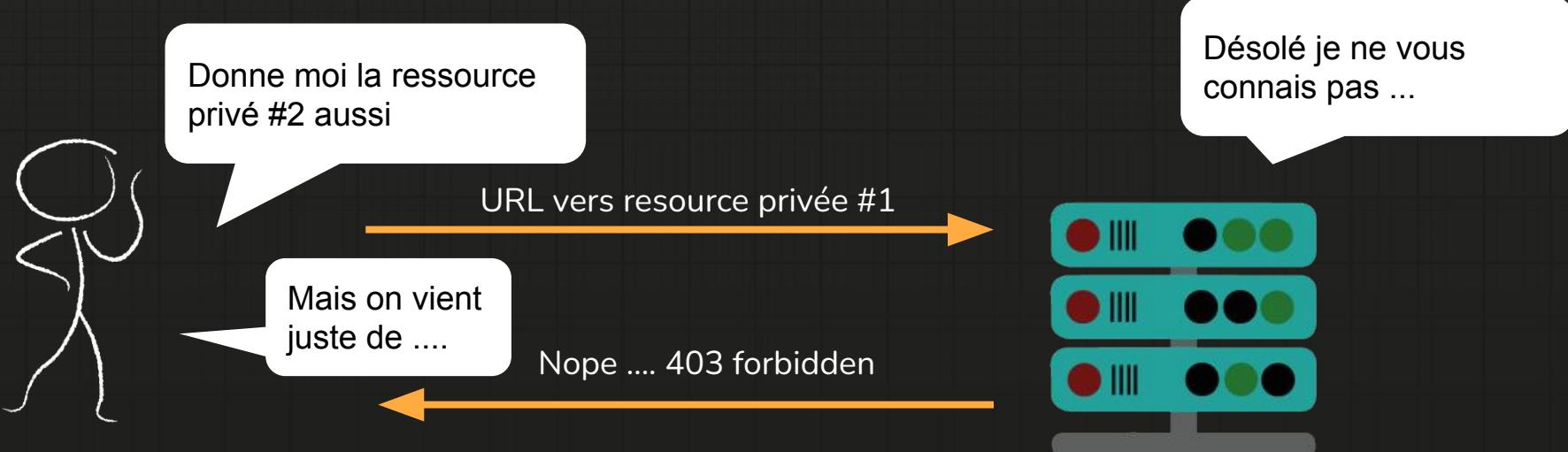
## >> Le serveur ne se souvient de rien avec le protocole http

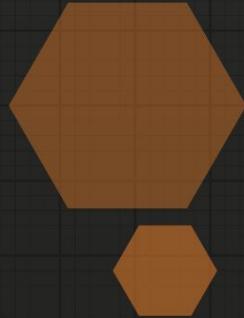
Si une ressource nécessite que l'utilisateur s'identifie pour être obtenue, alors on peut mettre en place un système de mot de passe afin de vérifier si il a accès à cette information



## >> Le serveur ne se souvient de rien avec le protocole http

Même si la personne est connectée, il n'y a pas de mécanisme natif en http pour se souvenir de cette personne lors de la requête suivante ...





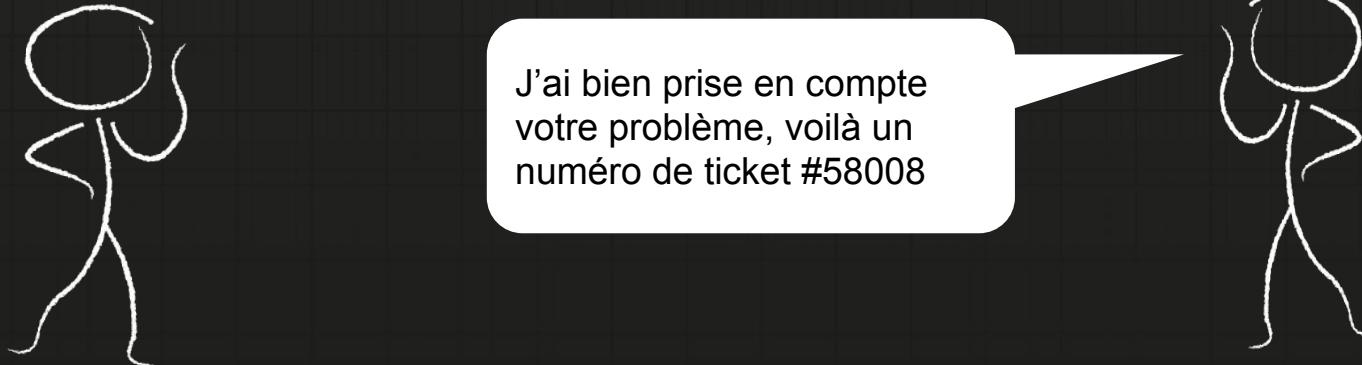
# La solution historique : Session ID

*Chapitre : Echange serveur*

## >>La solution historique : session ID

Lorsque vous avez un problème technique et que vous appelez le service client, vous expliquez votre problème en incluant tous les détails de manière à ce que la personne puisse vous identifier et connaître la raison de votre appel.

Mais il est possible que cette personne vous demande de rappeler ultérieurement car votre problème nécessite un technicien



## >> La solution historique : session ID

Grâce à ce numéro de ticket, la personne connaît votre problème, votre identité.  
Pas besoin de lui rappeler.

Il lui suffit de consulter quel client est lié à quel ticket sur son registre



Bonjour je vous ai appelé et  
mon numéro de ticket est le  
#58008

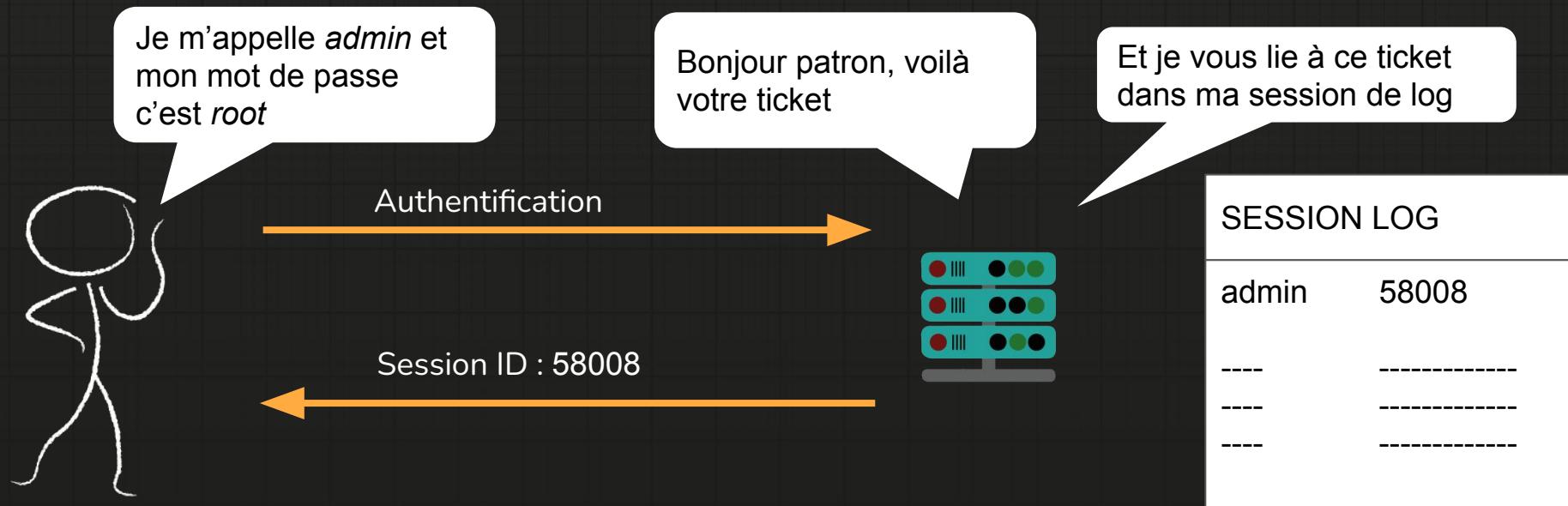
Bonjour monsieur, votre  
problème nécessite de  
redémarrer 5 fois votre box



Ticket # 58008  
  
Problème de  
box

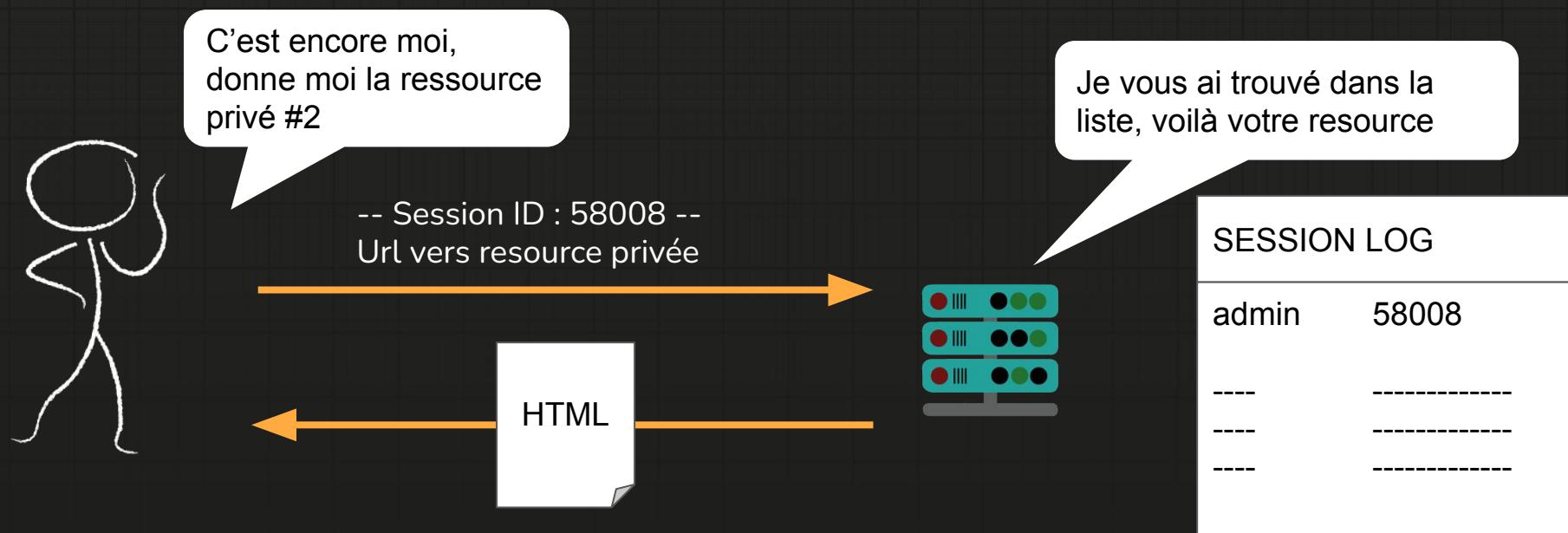
## >> La solution historique : session ID

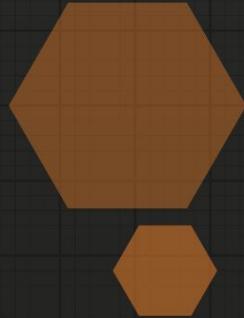
Cette solution existe depuis les débuts d'internet et est intégrée nativement aux navigateurs.



## >> La solution historique : session ID

Chaque requête se verra ajouter une information dans son en-tête : le session ID précédemment obtenu. Le serveur reconnaît l'utilisateur tant que le cookie n'a pas été supprimé.





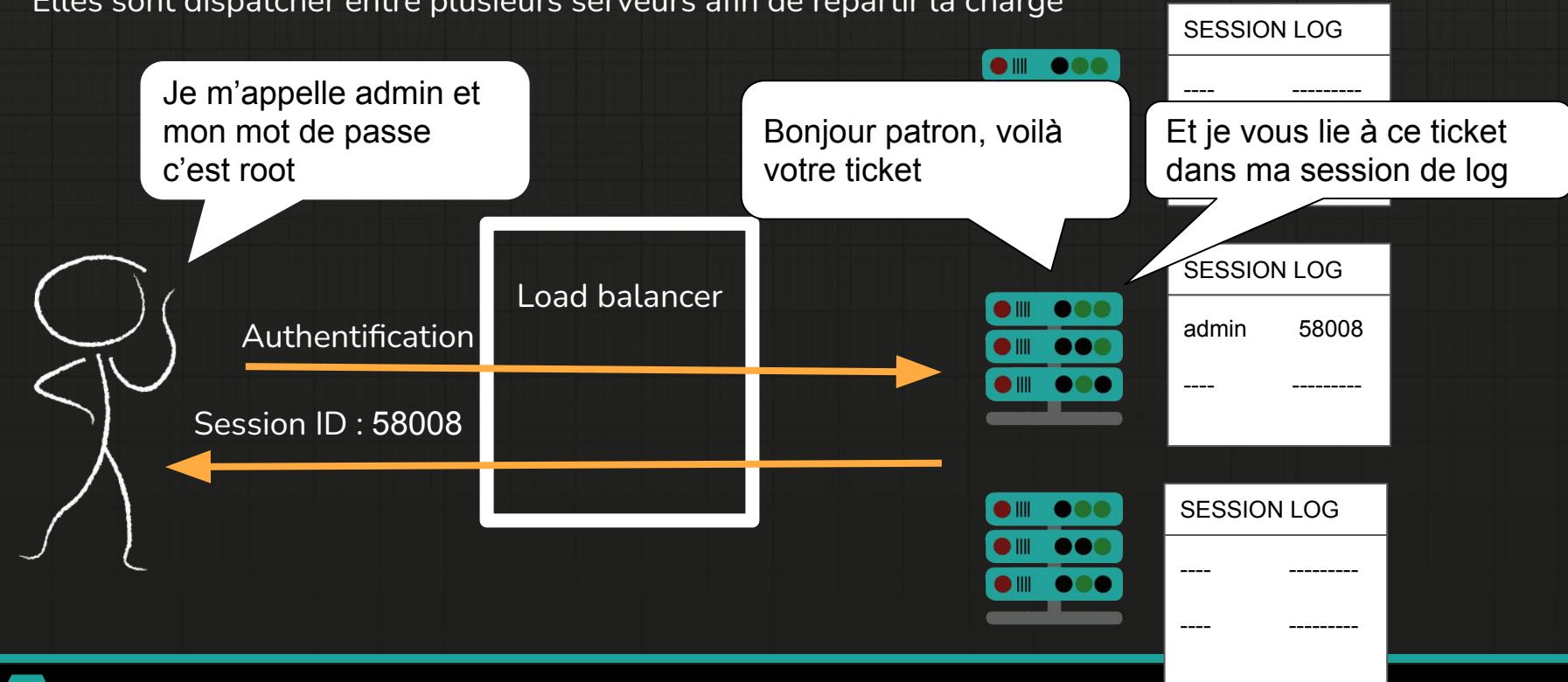
# La problématique des session ID

*Chapitre : Echange serveur*

## >> Problématique de cette approche

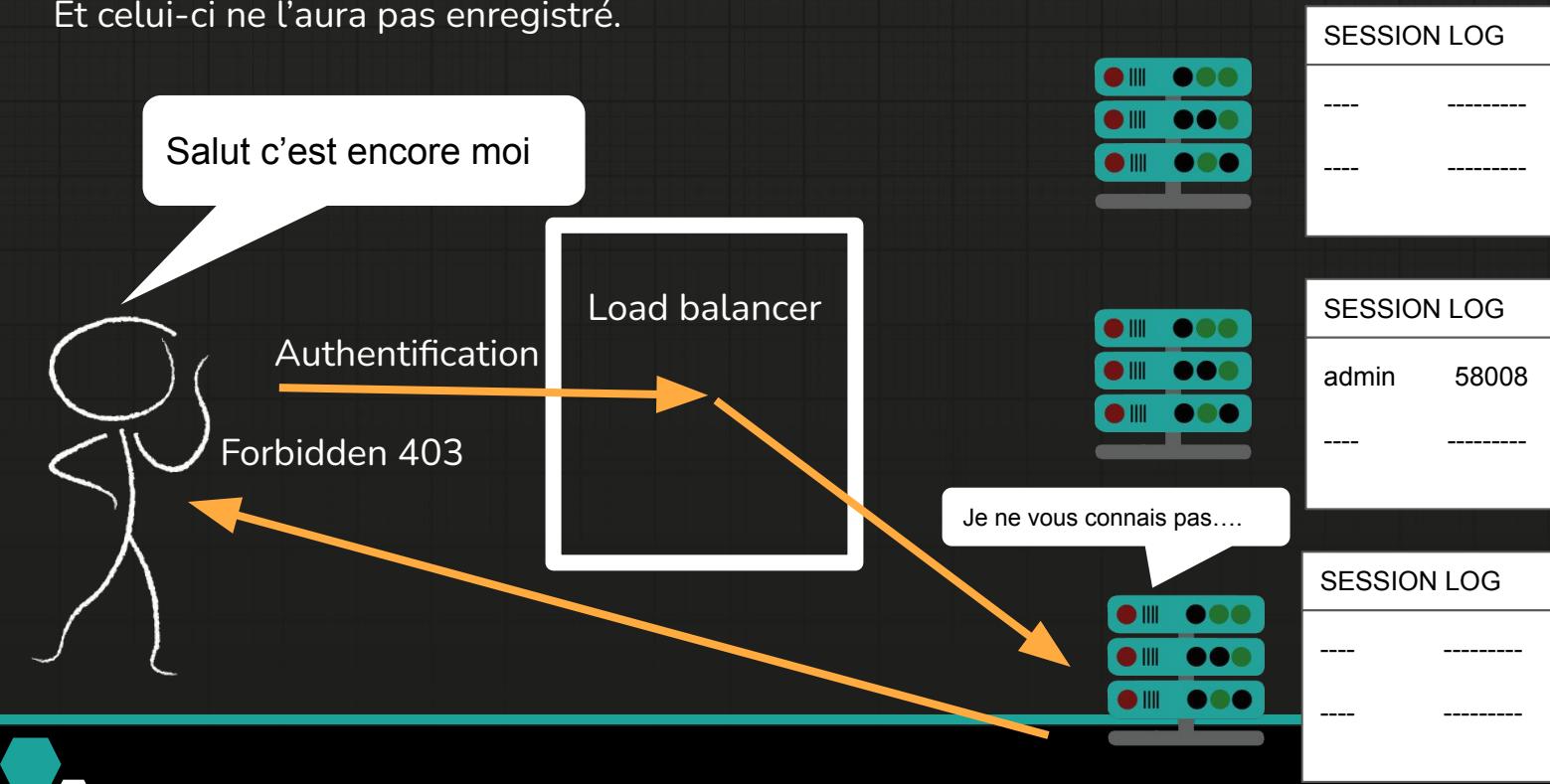
De nos jours, les applications sont rarement monolithiques.

Elles sont dispatcher entre plusieurs serveurs afin de répartir la charge



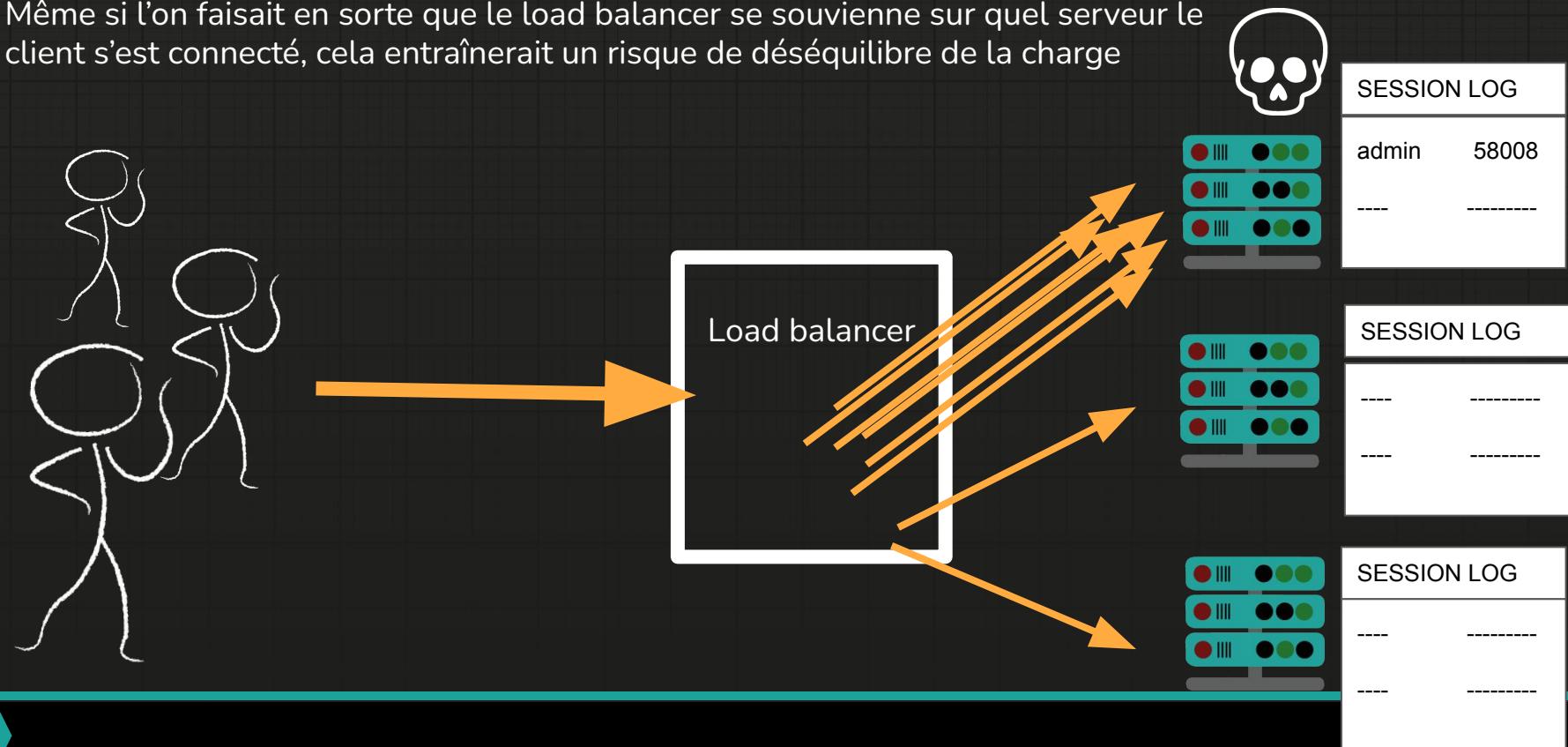
## >> Problématique de cette approche

Chaque serveur possède son propre registre, et si un utilisateur se connecte sur un serveur il ne tombera pas toujours sur le même.  
Et celui-ci ne l'aura pas enregistré.



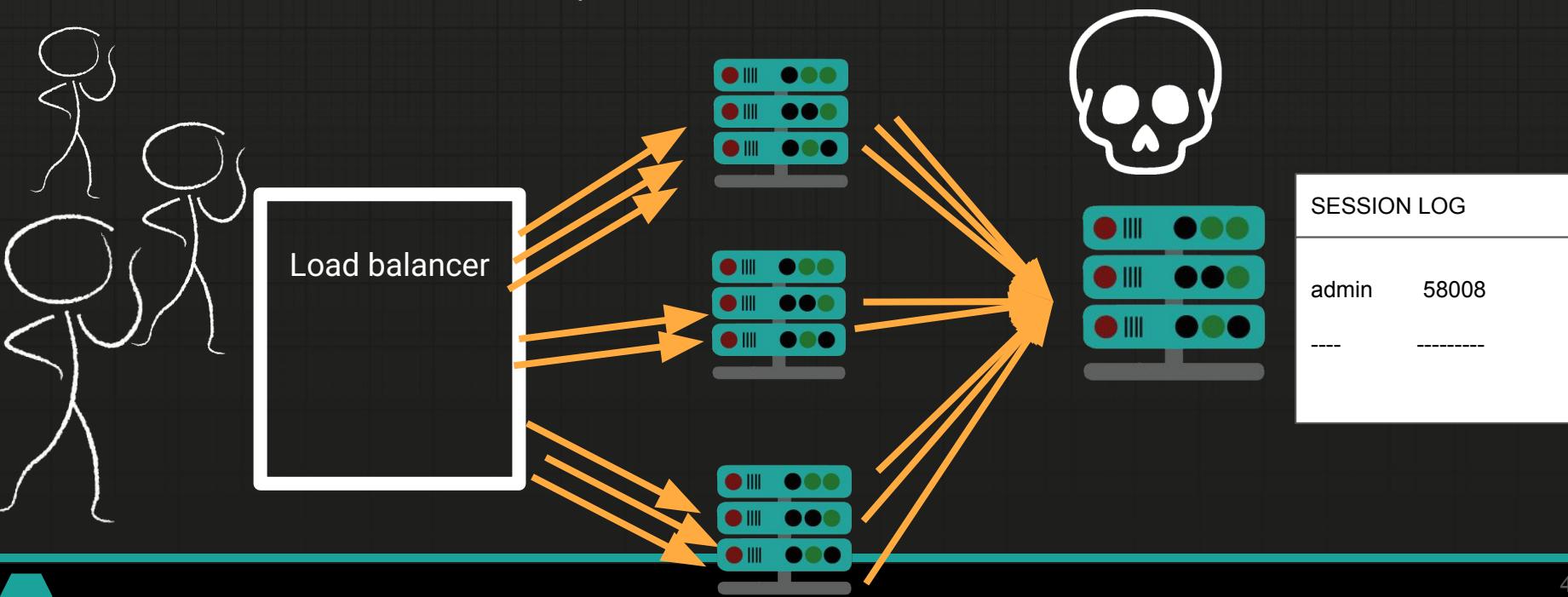
## >> Problématique de cette approche

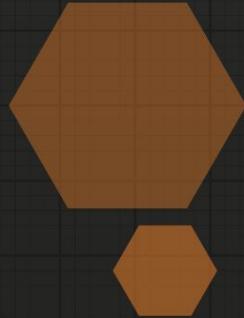
Même si l'on faisait en sorte que le load balancer se souvienne sur quel serveur le client s'est connecté, cela entraînerait un risque de déséquilibre de la charge



## >> Problématique de cette approche

On pourrait stocker le registre sur un seul et même serveur, mais si ce server venait à tomber, il entraînerait tous les autres serveurs avec lui (ce qui reviendrait à une architecture monolithique)





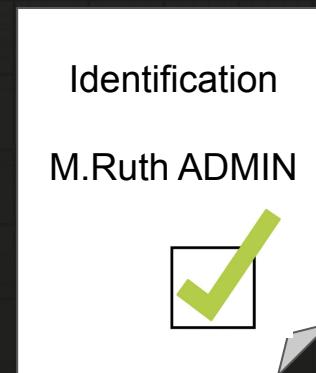
# La solution des Json Web Token

*Chapitre : Echange serveur*

# Principe

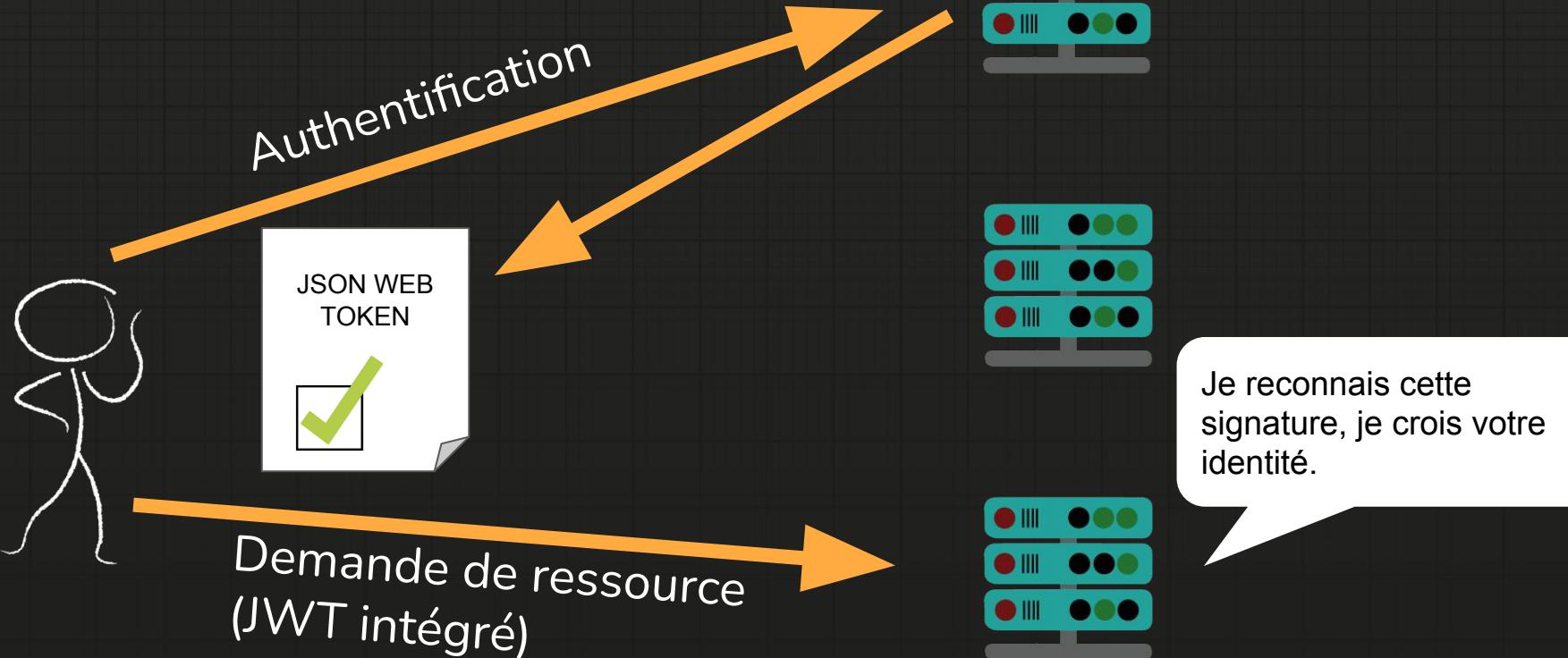
Bonjour je m'appelle M.ADMIN Ruth,  
j'ai un compte chez vous...

Oui tout à fait voilà dossier signé  
avec votre identification



Dans le cas présent, le service client ne stock aucune information sur l'identité de l'utilisateur,  
se sera a lui de donner le dossier (Comme une carte de fidélité à tampon)

## >> Principe



# A quoi ressemble un JWT ?

Prononcez "jawt", ou "JiDeubeuliouTi" (*ou à la française, personne ne vous en voudra...*)

Voilà le contenu d'un JWT, à première vu ça ne ressemble pas à grand chose :

```
eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9.eyJzdWliOilxMjM0NTY3ODkwliwibmFtZSI6Ikpvag  
4gRG9IliwiaWF0IjoxNTE2MjM5MDlyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQss  
w5c
```



## >> Structure d'un JWT

Mais on peut distinguer 3 parties, chacune séparée par un point.

la première est l'**en-tête (header)**

la deuxième se sont les **données (payload cad “charge utile”)**

la troisième partie est la signature

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpv  
aG4gRG9lIiwiZWFOIjoxNTE2MjM5MDIyfQ.Sf1KxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_ad  
Qssw5c
```



# Le Compteau d'opérations d'Hibernate haché !!!

Il est très important de comprendre qu'un JWT contrairement à ce que l'on pourrait croire, n'est pas haché (c'est la signature qui est haché).

Vous pouvez d'ailleurs aller sur les site <https://jwt.io/> afin de pouvoir lire le contenu de n'importe quel JWT.

Ou effectuer l'opération inverse

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpvag4gRG9lIiwiawF0IjoxNTE2MjM5MDIyfQ.cThIIoDvwdueQB468K5xDc5633seEFoqwxjf_xSJyQQ
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

PAYOUT: DATA

```
"sub": "1234567890",  
"name": "John Doe",  
"iat": 1516239022  
}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  your-256-bit-secret  
)  secret base64 encoded
```

# Comment est obtenuen la signature ?

```
{  
    "alg": "HS256",  
    "typ": "JWT"  
}
```

PAYOUT: DATA

```
{  
    "sub": "1234567890",  
    "name": "John Doe",  
    "iat": 1516239022  
}
```

VERIFY SIGNATURE

```
HMACSHA256(  
    base64UrlEncode(header) + "." +  
    base64UrlEncode(payload),  
    ma-signature-cryptée  
) secret_bienf_encoded
```

La signature dépend de 3 choses :

- l'algorithme de hachage employé (ici HS256)
- le contenu à hacher (la partie violette)
- le mot de passe secret que seul votre serveur connaît.

Imaginons que votre mot de passe secret sur votre serveur (que l'on appelle "secret") est "azerty-root-123"

Le contenu en violet, haché avec l'algorithme en rouge, en utilisant le secret de votre serveur à donné : "ma-signature-haché" (bien sur en réalité cela donne plutôt une très longue suite de chiffre)

Ici "ma-signature-haché" est la signature obtenue grâce au mot de passe secret de votre serveur

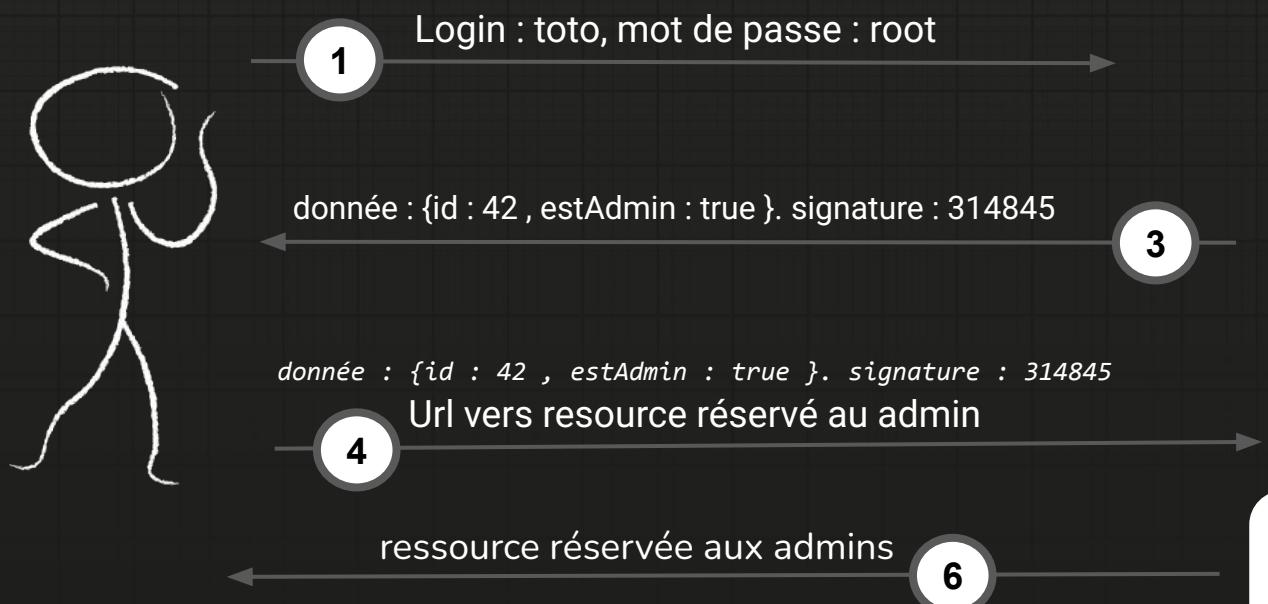
# >> Exemple concret

Un administrateur se connecte, on lui retourne un token, avec en donnée : son id et si il est administrateur.

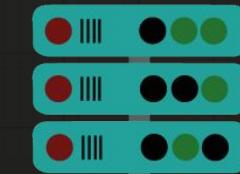
Il tente ensuite d'accéder à une ressource réservée au administrateur.

Je connais bien un "toto" qui a le mot de passe "root" dans ma bdd, il est admin et a l'id 42.

Grâce à mon secret j'obtiens la signature 314845. Voilà votre token



2  
secret :  
"azerty-root-123"



Base de données

5  
Tu prétend être admin et avoir l'id 42 ?  
laisse moi vérifier le fait que si je hache ces  
données avec mon secret, cela donne bien  
la même signature ... oui c'est bien 314845

## >> Que peut-on stocker dans les données d'un JWT ?

On peut stocker toutes les informations que l'on souhaite (nom, prénom, id, email, liste des modules auquel l'utilisateur a accès, liste des droits qu'il possède, son panier ...)

Ce qu'il ne faut pas stocker dans un JWT :

toutes les informations permettant la connexion de l'utilisateur (secrets, mot de passe ...)

Au minimum, il faut stocker les informations nécessaires pour retrouver l'utilisateur dans la base de donnée, par exemple sa clé primaire.



## >> Faisons un petit résumé

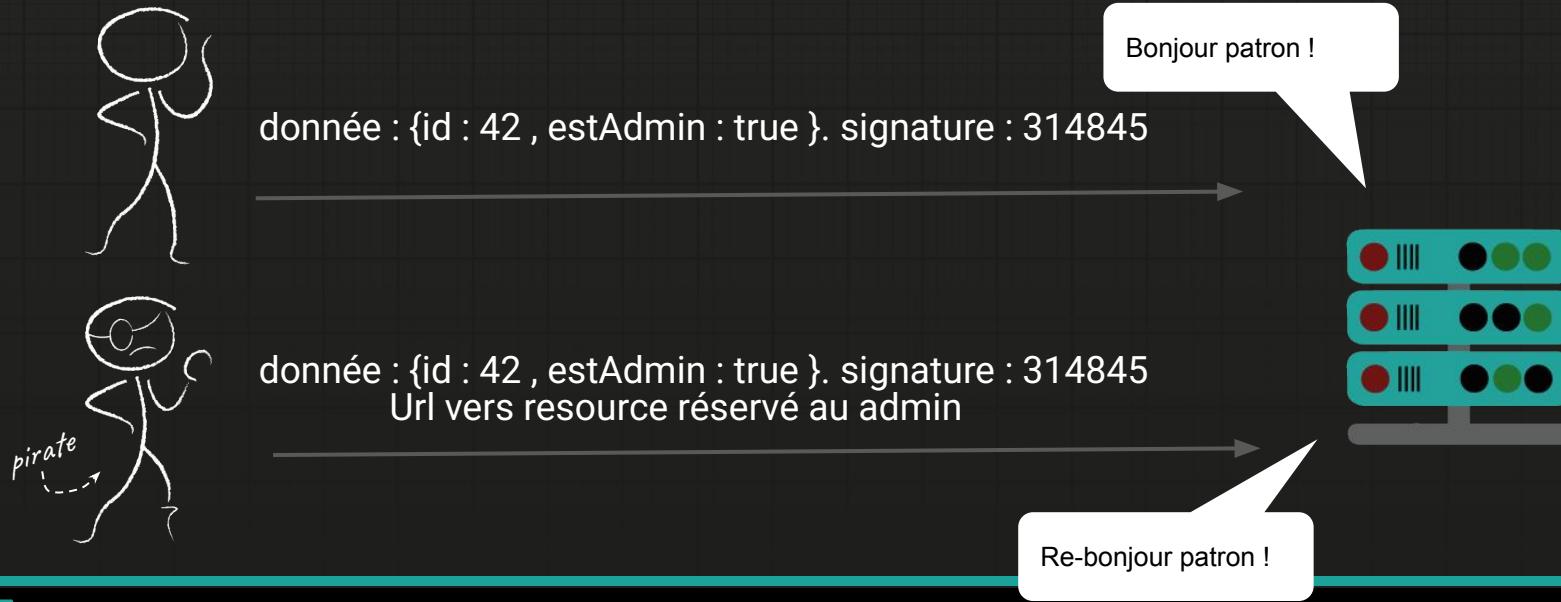
- La seule information qui est stocké sur le serveur est son secret.
- Il est utilisé pour créer n'importe quels signatures
- Tous les serveurs possèdent le même secret, ce qui permet le load balancing
- On peut stocker n'importe quel information dans le JWT à partir du moment où ce n'est pas une information de connexion (ex le mot de passe)
- Si la signature correspond aux données du token : on doit croire l'utilisateur
- C'est L'UNIQUE moyen de vérifier si l'utilisateur est bien celui qu'il prétend être



## >> Le vol de token

Il est donc très important que ce token reste confidentiel pour l'utilisateur.

Si il est volé par un autre utilisateur, nous ne pouvons faire aucune différence entre les 2 utilisateurs.



## >> Eviter le vol de token par faille XSS

Les token peuvent se stocker de plusieurs façons. Via un cookie, le localStorage, une base de données SQLite ...

Il est recommandé de ne pas utiliser les cookies. Il est assez facile de subir une faille XSS et de se faire dérober.

Pour rappel les failles XSS consistent à faire lire un script js à un utilisateur (via un forum, un livre d'or, un set d'emoticon gratuit) ce qui peut avoir pour but la divulgation des cookies.

On peut préconiser la sauvegarde dans le localStorage, plus difficile à atteindre.



## >> Eviter le vol de token par sniffeur réseau

Si vous avez bien compris, les tokens sont ajoutés dans l'en-tête des requêtes.

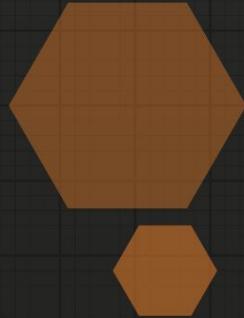
Si vous utilisez le protocole HTTP les requêtes sont visibles sur le réseau (donc leur en-tête également, c'est à dire le token).

Il existe des dizaines de façons d'intercepter une requête sur le réseau : Spyware, équipement réseau corrompu, wifi public, un hacker au sein de votre entreprise ....

Une seule solution : le protocole HTTPS.

Voyez le protocole HTTPS comme un train blindé dont vous seul et le destinataire pouvez lire le contenu.  
*(Attention le train est blindé mais le destinataire peut être le hacker)*





# Les token de connexion et les failles CSRF

*(Cross Site Request Forgery)*

*Chapitre : Echange serveur*

## >> Rappel des attaques CSRF

Les attaques CSRF consistent à faire exécuter des instructions à une personne ayant des privilèges élevés (ex : à un administrateur).

Elle peut prendre différentes formes : un lien trompeur dans un email ou un sms, ou bien dans un QRcode.

L'administrateur en cliquant sur ce lien va être redirigé vers notre propre application et comme il possède les droits suffisant pour effectuer l'action, va la réaliser contre son gré.

Ex : l'administrateur flash un QRCode le dirigeant vers le lien : <http://mon-application/supprimer/utilisateur/42>  
L'utilisateur 42 serait alors effectivement supprimé.



## >> Se protéger des failles CSRF dans une application utilisant une API

Dans le cas d'une application utilisant une API (comme la nôtre) l'utilisation d'un token de connexion suffit pour se protéger.

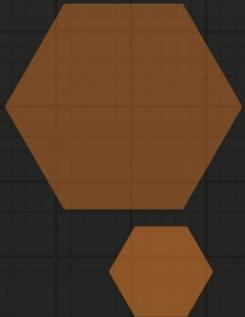
En effet, comme nous devons obligatoirement ajouter le JWT dans l'entête de la requête afin de vérifier si l'utilisateur possède bien les droits pour effectuer l'action, la requête sera considérée comme étant effectuée par un utilisateur non connecté

Car il n'est pas possible de modifier un en-tête en cliquant sur un simple lien.

Si l'utilisateur nous redirige vers un site qui aurait pour but d'ajouter le token dans la requête avant d'envoyer cette dernière, celui-ci ne pourrait pas avoir accès au localstorage qui est cloisonné par domaine.

Note : Une extension de navigateur malicieuse pourrait néanmoins modifier le comportement de notre application et faire exécuter des requêtes indésirables à l'administrateur qui l'aurait installée





# Le vol de token

*Chapitre : Echange serveur*

## >> Et si le vol de token arrive ?

Impossible pour l'utilisateur de se "déconnecter" comme pour les Session ID (le serveur ne disposant pas de tables de sessions).

Puisque l'on ne stock aucune information sur le serveur, il n'est pas possible de dire : cette personne est désormais déconnectée.

On peut néanmoins indiquer une information dans les données du token qui ferait que celui-ci est invalidé.

Voir le chapitre "Mettre en place un système d'invalidation d'un token"



## >> Limiter l'impact d'un vol de token

Afin de limiter l'impact d'un vol de token, on peut mettre plusieurs action en place (outre le système de deconnexion que l'on verra par la suite) :

Inclure une date d'expiration dans les données du token. Le token est effectivement confirmé, mais on ne renverra aucune information, si la date indiquée est expiré.

Demander à l'utilisateur de confirmer son mot de passe pour les opérations les plus sensibles (suppression / modification de données, mouvement d'argent, changement de mot de passe ...)

Confirmer son identité grâce à un code ou un lien à cliquer via un email lorsque l'utilisateur change d'ordinateur, de pays ...



## >> Résumé

- Un JWT n'est pas haché, seule sa signature l'est
- C'est une authentification compatible avec le load balancing
- Il ne doit pas être stocké dans les cookies du client (favorisez le localstorage par exemple)
- Il est nécessaire de mettre en place un protocole HTTPS
- Ajoutez une date d'expiration adaptée
- Demandez de nouveau la confirmation par login pour les opérations sensibles
- Demandez de nouveau la confirmation par login en cas de changement d'ordinateur / pays (à stocker dans les données du token, comme pour la date d'expiration)
- En cas de vol, changez une données de l'utilisateur afin d'invalider le token volé



## >> Eviter le vol de token par sniffeur réseau

Si vous avez bien compris, les tokens sont ajoutés dans l'en-tête des requêtes.

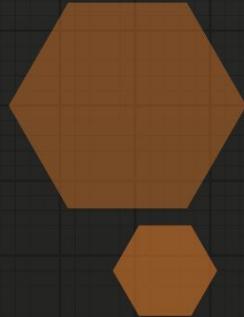
Si vous utilisez le protocole HTTP les requêtes sont visibles sur le réseau (donc leur en-tête également, c'est à dire le token).

Il existe des dizaines de façons d'intercepter une requête sur le réseau : Spyware, équipement réseau corrompu, wifi public, un hacker au sein de votre entreprise ....

**Une seul solution : le protocole HTTPS.**

Voyez le protocole HTTPS comme un train blindé dont vous seul et le destinataire pouvez lire le contenu.  
(Attention le train est blindé mais le destinataire peut être le hacker)





# Les token de connexion et les failles CSRF

*(Cross Site Request Forgery)*

*Chapitre : Spring security*

## >> Rappel des attaques CSRF

Les attaques CSRF consistent à faire exécuter des instructions à une personne ayant des privilèges élevés (ex : à un administrateur).

Elle peut prendre différentes forme : un lien trompeur dans un email ou un sms, ou bien dans un QRcode.

L'administrateur en cliquant sur ce lien va être redirigé vers notre propre application et comme il possède les droits suffisant pour effectuer l'action, va la réaliser contre son grés.

Ex : l'administrateur flash un QRCode le dirigeant vers le lien : <http://mon-application/supprimer/utilisateur/42>  
L'utilisateur 42 serait alors effectivement supprimé.



## >> Se protéger des failles CSRF dans une application utilisant une API

Dans le cas d'une application utilisant une API (comme la nôtre) l'utilisation d'un token de connexion suffit pour se protéger.

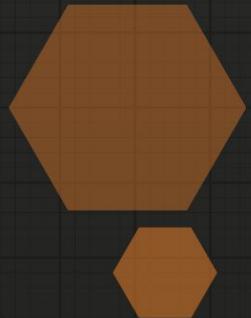
En effet, comme nous devons obligatoirement ajouter le JWT dans l'en-tête de la requête afin de vérifier si l'utilisateur possède bien les droits pour effectuer l'action, la requête sera considérée comme étant effectuée par un utilisateur non connecté

Car il n'est pas possible de modifier un en-tête en cliquant sur un simple lien.

Si l'utilisateur nous redirige vers un site qui aurait pour but d'ajouter le token dans la requête avant d'envoyer cette dernière, celui-ci ne pourrait pas avoir accès au localstorage qui est cloisonné par domaine.

**Note : Une extension de navigateur malicieuse pourrait néanmoins modifier le comportement de notre application et faire exécuter des requêtes indésirables à l'administrateur qui l'aurait installée**





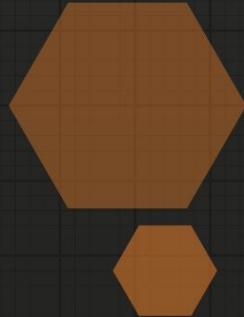
# Résumé

*Chapitre : Spring security*

## >> Résumé

- Un JWT n'est pas haché
- C'est une authentification compatible avec le load balancing
- Il ne doit pas être stocké dans les cookies du client (favorisez le localstorage par exemple)
- Il est nécessaire de mettre en place un protocole HTTPS
- Ajoutez une date d'expiration adaptée
- Demandez de nouveau la confirmation par login pour les opérations sensibles
- Demandez de nouveau la confirmation par login en cas de changement d'ordinateur / pays (à stocker dans les données du token, comme pour la date d'expiration)
- En cas de vol, changez une données de l'utilisateur afin d'invalider le token volé





# Mettre en place une identification par JWT

*Chapitre : Spring security*

## >> Configurer le pom.xml

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>

<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
</dependency>
```

Cette dépendance n'est nécessaire que si vous utilisez un jdk supérieur à la version 8



## >> Créer un service utilitaire pour les JWT

*Créer une méthode permettant de générer un JWT à partir d'un UserDetails*

*security/JwtUtils (nouveau fichier)*

```
@Service
public class JwtUtils {

    public String generateToken(AppUserDetails userDetails) {

        return Jwts.builder()
            .setSubject(userDetails.getUsername())
            .signWith(SignatureAlgorithm.HS256, "azerty")
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .compact();
    }
}
```



>> Pour aller plus loin

## >> Créer un service utilitaire pour les JWT 2/3

Nous n'allons pas l'utiliser dans ce TD, mais il est possible d'ajouter une date d'expiration au token (*qui le rendra invalide si il est testé après cette date*), ainsi que des données supplémentaires

security/JwtUtils

```
Map<String, Object> tokenData = new HashMap<>();  
  
//ici vous pouvez rajouter tout ce que vous voulez  
tokenData.put("roles", userDetailsService.getAuthorities().stream()  
    .map(GrantedAuthority::getAuthority)  
    .collect(Collectors.joining(","));  
  
return Jwts.builder()  
    .setClaims(tokenData)  
    .setSubject(userDetailsService.getUsername())  
    .setIssuedAt(new Date(System.currentTimeMillis()))  
    .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 10))  
    .signWith(SignatureAlgorithm.HS256, secret).compact();
```

Ce token se voit ajouter des données que l'on pourra exploiter dans notre front (*ici ses rôles mais ce pourrait être autre chose*)

Ce token se voit ajouter une date d'expiration (ici 10h)

Note : l'ajout de données supplémentaires peut être pratique mais nécessite d'invalider le token précédent (voir la partie sur l'invalidation d'un JWT)



## >> Différence entre JWT, JWS et JWE

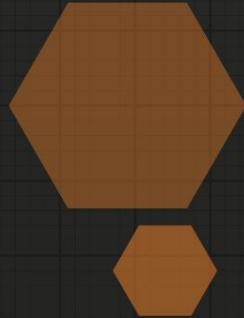
Un JWT est un **terme générique** pour un token qui peut être :

- **Signé** (JWS – JSON Web Signature)
- **Chiffré** (JWE – JSON Web Encryption)

Un JWT peut donc être un **JWS** ou un **JWE** selon l'usage.

Par abus de langage on utilise le terme “JWT” pour désigner un “JWS” mais en réalité un JWT n'aurait aucune signature (*ce qui dans l'usage n'est pas courant*)





Créer un service pour s'inscrire /  
ajouter un utilisateur

*Chapitre : Spring security*

## &gt;&gt; Injecter l'encoder

Pour enregistrer un utilisateur, il nous faudra au préalable injecter le PasswordEncoder que nous avions définis dans la classe GestionSecurite"

*controller/ConnexionController*

```
@CrossOrigin  
@RestController  
public class ConnexionController {  
  
    @Autowired  
    private PasswordEncoder encoder;
```

Par propriété

OU

```
private PasswordEncoder encoder;  
  
@Autowired  
public ConnexionController (PasswordEncoder encoder , ...)  
{  
    this.encoder = encoder;  
    ...  
}
```

Ou par constructeur



Lors de l'inscription nous utiliserons cet encoder afin de transformer le mot de passe en clair en mot de passe haché :

*controller/ConnexionController*

```
@PostMapping("/inscription")
public ResponseEntity<String> inscription(@RequestBody Utilisateur utilisateur) {
    utilisateur.setMotDePasse(encoder.encode(utilisateur.getMotDePasse()));
    utilisateurDao.save(utilisateur);
    return ResponseEntity.ok().build();
}
```



## &gt;&gt; Gérer le droit par défaut

Si l'utilisateur possède un droit par défaut (par exemple USER) il est nécessaire de lui ajouter ce droit avant son insertion. Ici un exemple où le rôle USER a l'id 1 en base de donnée.

*controller/ConnexionController*

```
@PostMapping("/inscription")
public ResponseEntity<String> inscription(@RequestBody Utilisateur utilisateur) {

    utilisateur.setMotDePasse(encoder.encode(utilisateur.getMotDePasse()));

    utilisateur.setRole(Role.USER);

    utilisateurDao.save(utilisateur);
    return ResponseEntity.ok().build();
}
```



>> Pour aller plus loin

>> Gérer le droit par défaut dans le cas d'une table liée

*controller/ConnexionController*

```
@PostMapping("/inscription")
public ResponseEntity<String> inscription(@RequestBody Utilisateur utilisateur) {

    utilisateur.setMotDePasse(encoder.encode(utilisateur.getMotDePasse())));
}
```

```
Role roleUser = new Role();
roleUser.setId(1);

utilisateur
    .setRole(roleUser);
```

Cas avec un ManyToOne  
sur une table Role

```
Role roleUser = new Role();
roleUser.setId(1);

utilisateur
    .getlisteRole()
    .add(roleUser);
```

Cas avec un ManyToMany sur  
une table Role

Dans cet exemple, l'id 1 correspond au droit en base de données, que l'on souhaite affecter.

Pour rappel, JPA n'a besoin que de connaître l'id du droit pour effectuer la liaison

>> Pour aller plus loin

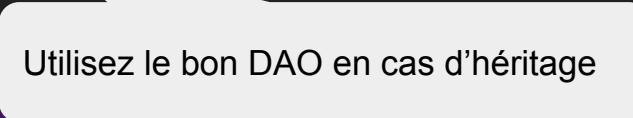
## >> Gérer le droit par défaut dans le cas d'un héritage

En cas d'héritage, utilisez le DAO approprié pour sauvegarder l'utilisateur, c'est la présence ou non de l'id de l'utilisateur dans la table qui hérite (par exemple un client qui hérite d'utilisateur) qui définira son rôle

*controller/ConnexionController*

```
@PostMapping("/inscription")
public ResponseEntity<Employe> inscription(@RequestBody Client client) {

    client.setPassword(client.encode(client.getMotDePasse()));
    clientDao.save(client);
```



Utilisez le bon DAO en cas d'héritage

## &gt;&gt; Ajouter une création d'utilisateur

Dans certain contexte, le fait qu'un utilisateur fasse lui même son "l'inscription" est juste inaprorié.  
Si l'application est une application interne à une entreprise, il y a plus de chance que ce soit une personne avec des privilèges particulier qui s'en occupe (RH, Administrateur, chef de service ....)

*controller/UtilisateurController*

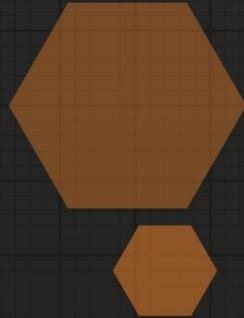
```
@PostMapping("/utilisateur")
public ResponseEntity<String> save(@RequestBody Utilisateur utilisateur) {

    utilisateur.setMotDePasse(encoder.encode(utilisateur.getMotDePasse()));
    Role roleUser = new Role();
    roleUser.setId(1);
    utilisateur.getListeRole().add(roleUser);

    utilisateurDao.save(utilisateur);

    return ResponseEntity.ok().build();
}
```





# Créer un service pour s'authentifier

*Chapitre : Spring security*

## >> Créer un service d'authentification 1/2

Le service /authentification va nous permettre de valider l'existence dans la base de donnée de l'utilisateur qui tente de se connecter et de lui renvoyer un JWT comme réponse. Nous allons autowired toutes les classes dont nous aurons besoin :



## >> Créer un service d'authentification 2/2

puis ajouter la requête d'authentification

```
@PostMapping("/authentification")
public ResponseEntity<String> authentification(@RequestBody Utilisateur utilisateur) throws Exception {

    try {
        authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                utilisateur.getPseudo(), utilisateur.getMotDePasse()));
    }
    catch (BadCredentialsException e) {
        throw new Exception("Pseudo ou mot de passe incorrect", e);
    }

    final UserDetails userDetails = userDetailsService
        .loadUserByUsername(utilisateur.getPseudo());

    return ResponseEntity.ok(jwtUtil.generateToken(userDetails));
}
```



## >> Créer un filtre de requête 1/3

La prochaine étape consiste à créer un filtre de requête. Afin d'extraire le token à chaque requête qui est transmise vers notre serveur

On autowired les classes nécessaires :

```
@Component
public class JwtRequestFilter extends OncePerRequestFilter {

    private MonUserDetailsService userDetailsService;
    private JwtUtil jwtUtil;

    @Autowired
    public JwtRequestFilter(MonUserDetailsService userDetailsService, JwtUtil jwtUtil) {
        this.userDetailsService = userDetailsService;
        this.jwtUtil = jwtUtil;
    }
}
```



## >> Créer un filtre de requête 2/3

Puis on ajoute la méthode qui sera appelé afin d'extraire l'en-tête des requête.  
On vérifie si le token commence bien par "Bearer"  
A l'aide de notre utilitaire on extrait le token et son subject.

```
@Override  
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)  
    throws ServletException, IOException {  
  
    final String authorizationHeader = request.getHeader("Authorization");  
  
    String username = null;  
    String jwt = null;  
  
    if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {  
        jwt = authorizationHeader.substring(7);  
        username = jwtUtil.extractionDuCorpDuToken(jwt).getSubject();  
    }  
}
```

...

## >> Créer un filtre de requête 3/3

L'utilisateur est récupéré via le sujet du token, et après vérification de la validité du token, on charge l'UserDetails dans le contexte de sécurité (afin de permettre les autorisations liées aux rôle notamment)

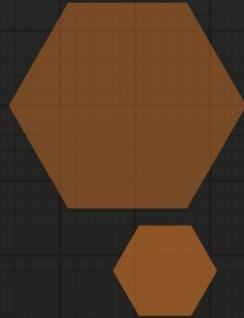
```
...  
  
if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {  
  
    UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);  
  
    if (jwtUtil.validateToken(jwt, userDetails)) {  
  
        UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken =  
            new UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());  
  
        usernamePasswordAuthenticationToken  
            .setDetails(new WebAuthenticationDetailsSource().buildDetails(request));  
  
        SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);  
    }  
}  
chain.doFilter(request, response);
```

## >> Modifier l'authorization

Nous allons ensuite modifier la méthode `configure(HttpSecurity http)` de notre classe `MaConfigurationSecurite`. Ainsi que d'ajouter un Bean `AuthenticationManager`

```
@Override  
@Bean  
public AuthenticationManager authenticationManagerBean() throws Exception {  
    return super.authenticationManagerBean();  
}
```





## Appliquer des restriction des les routes

*Chapitre : Spring security*

## >> Configurer l'authorization par pattern d'url

Il existe plusieurs façons de gérer quel route peut être utilisée par tels ou tels rôles

Il est par exemple possible de définir un pattern sur les urls dans la classe permettant de gérer la sécurité de notre application ("ConfigurationSecurite" dans nos exemples)

```
@Bean  
SecurityFilterChain filterChain (HttpSecurity http) throws Exception {  
  
    return http ...  
        .authorizeHttpRequests(conf ->  
            conf  
                .requestMatchers("/login","/sign-in").permitAll()  
                .requestMatchers("/admin/**").hasRole("ADMIN")  
                .anyRequest().authenticated()  
        )  
        ...  
}
```

Dans cet exemple, les 2 seules routes ne nécessitant pas d'être connecté sont : "/login" et "/sign-in"

Toutes les routes commençant par "/admin/" ne peuvent être accéder par un administrateur, et les autres d'être authentifié (avec *n'importe quel rôle*)



## >> Configurer l'authorization par Annotation (1/4)

L'exemple précédent permet d'affecter rapidement les différents rôles aux routes, mais ne permet pas de facilement mettre en place une gestion plus complexe des rôles nécessaires à chaque routes d'un contrôleur, à moins de lister exhaustivement toutes les routes dans le fichier.

C'est pourquoi nous allons voir une méthode alternative à base d'annotations que nous allons devoir créer.

Dans notre cas, arbitrairement appelée **@IsUser** et **@IsAdmin**

```
@RestController  
@CrossOrigin  
public class ProduitController {  
  
    @GetMapping("/produit/liste")  
    public List<Produit> liste() {  
        ...  
    }  
  
    @PostMapping("/produit")  
    @IsUser  
    public ResponseEntity<Produit> add(  
        ...  
    }  
  
    @DeleteMapping("/produit/{id}")  
    @IsAdmin  
    @JsonView(ProduitView.class)  
    public ResponseEntity<Produit> delete(@PathVariable int id) {  
        ...  
    }  
}
```

Cette route ne nécessite pas d'être authentifié

Cette route nécessite d'être authentifié en tant qu'utilisateur

Cette route nécessite d'être authentifié en tant qu'administrateur



## &gt;&gt; Configurer l'authorization par Annotation (2/4)

Dans un premier temps nous allons ajouter l'annotation `@EnableMethodSecurity` sur la classe `ConfigurationSecurite`

```
security/ConfigurationSecurite  
  
@EnableWebSecurity  
@Configuration  
@EnableMethodSecurity   
public class ConfigurationSecurite {  
  
    ...  
}
```

## &gt;&gt; Configurer l'authorization par Annotation (3/4)

Puis nous allons ajouter autant d'Annotation que de Rôles (dans notre cas )

security/IsAdmin

```
import org.springframework.security.access.prepost.PreAuthorize;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
// Définition de la cible de l'annotation, qui peut être un type (classe, interface) ou une méthode  
@Target({ElementType.METHOD, ElementType.TYPE})  
// L'annotation sera disponible au moment de l'exécution  
@Retention(RetentionPolicy.RUNTIME)  
// Restreindre l'accès aux utilisateurs ayant le rôle 'ROLE_ADMIN'  
@PreAuthorize("hasRole('ROLE_ADMIN')")  
public @interface IsAdmin {  
}
```

Ici on définit que l'annotation isAdmin oblige à être Utilisateur OU administrateur

security/IsUser

```
@Target({ElementType.METHOD, ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@PreAuthorize("hasAnyRole('ROLE_USER', 'ROLE_ADMIN')")  
public @interface IsUser {  
}
```

## &gt;&gt; Configurer l'authorization par Annotation (4/4)

Enfin placé les annotations au dessus des routes concernées, ou bien au dessus d'une classe entière (les annotations sur les méthodes auront la priorité sur l'annotation sur la classe)

```
@RestController  
@CrossOrigin  
public class Contrôleur1 {  
  
    @GetMapping("/route1")  
    public Item route1() {  
        ...  
    }  
  
    @GetMapping("/route2")  
    @IsUser  
    public Item route2(){  
        ...  
    }  
  
    @GetMapping("/route3")  
    @isAdmin  
    public Item route3(){  
        ...  
    }  
  
}
```

Cette route ne nécessite pas d'être authentifié

Cette route nécessite d'être authentifié en tant qu'administrateur

```
@RestController  
@CrossOrigin  
@IsUser  
public class Contrôleur1 {  
  
    @GetMapping("/route1")  
    public Item route1() {  
        ...  
    }  
  
    @GetMapping("/route2")  
    public Item route2(){  
        ...  
    }  
  
}
```

Cette route nécessite d'être authentifié en tant qu'utilisateur

```
@GetMapping("/route2")  
public Item route2(){  
    ...  
}
```

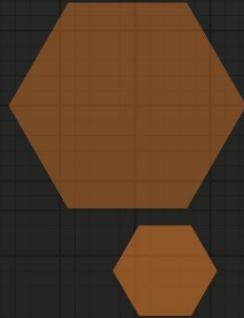
Cette route nécessite d'être authentifié en tant qu'utilisateur

```
@RestController  
@CrossOrigin  
@IsUser  
public class Contrôleur1 {  
  
    @GetMapping("/route1")  
    public Item route1() {  
        ...  
    }  
  
    @GetMapping("/route2")  
    @isAdmin  
    public Item route2(){  
        ...  
    }  
  
}
```

Cette route nécessite d'être authentifié en tant qu'utilisateur

```
@GetMapping("/route2")  
@isAdmin  
public Item route2(){  
    ...  
}
```

Cette route nécessite d'être authentifié en tant qu'administrateur



# Récupérer les informations contenues dans le JWT lors d'une requête

*Chapitre : Spring security*

## >> Récupérer les informations de l'utilisateur connecté

Maintenant que le token est envoyé dans chacune de nos requêtes, il est possible de savoir quel utilisateur tente d'effectuer cette requête.

Par exemple afin de récupérer les informations de l'utilisateur connecté :

```
@GetMapping("/get")
public List<MonEntite> get (@AuthenticationPrincipal SecuriteUtilisateur utilisateurConnecte) {

    System.out.println(utilisateurConnecte.getUsername());
    ...
}
```



L'annotation `@AuthenticationPrincipal` permet d'extraire l'utilisateur via la classe implementant `UserDetails`



## >> Extraire le token de l'entête de la requête

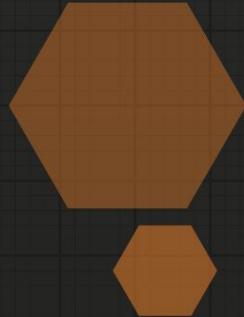
Plutôt que d'utiliser le système intégré de spring security, il est également possible d'extraire le JWT de l'en-tête de la requête

```
@GetMapping("/user/utilisateur")
public ResponseEntity<Utilisateur> getInformationUtilisateurConnecte(
    @RequestHeader(value="Authorization") String authorization){
    //la valeur du champs authorization est extraite de l'entête de la requête
    //On supprime la partie "Bearer " de la valeur de l'authorization
    String token = authorization.substring(7);
    //on extrait l'information souhaitée du token
    String username = jwtUtil.getTokenBody(token).getSubject();
    Optional<Utilisateur> utilisateur = utilisateurDao.findByPseudo(username);

    if(utilisateur.isPresent()) {
        return ResponseEntity.ok().body(utilisateur.get());
    }
    return ResponseEntity.notFound().build();
}
```

L'annotation `@RequestHeader` permet d'extraire une valeur de l'entête de la requête





Mettre en place un système  
d'invalidation d'un token  
  
(suite à un vol ou une déconnexion)

*Chapitre : Spring security*

# TODO

TODO : Expiration du token : basé sur la date d'expiration + date de dernière action : si la date d'expiration du token est arrivée, alors on vérifie dans la BDD si la dernière action est inférieure à 5 min. Si oui on renouvel via l'autorization ajoutée dans l'entête (si le token est expiré lors du remplissage d'un formulaire, prévoir le cas côté client)

TODO : Expiration du token : basé sur la volonté du backend (deconnexion utilisateur, changement de droit ...): dans utilisateur date\_dernier\_token\_valide

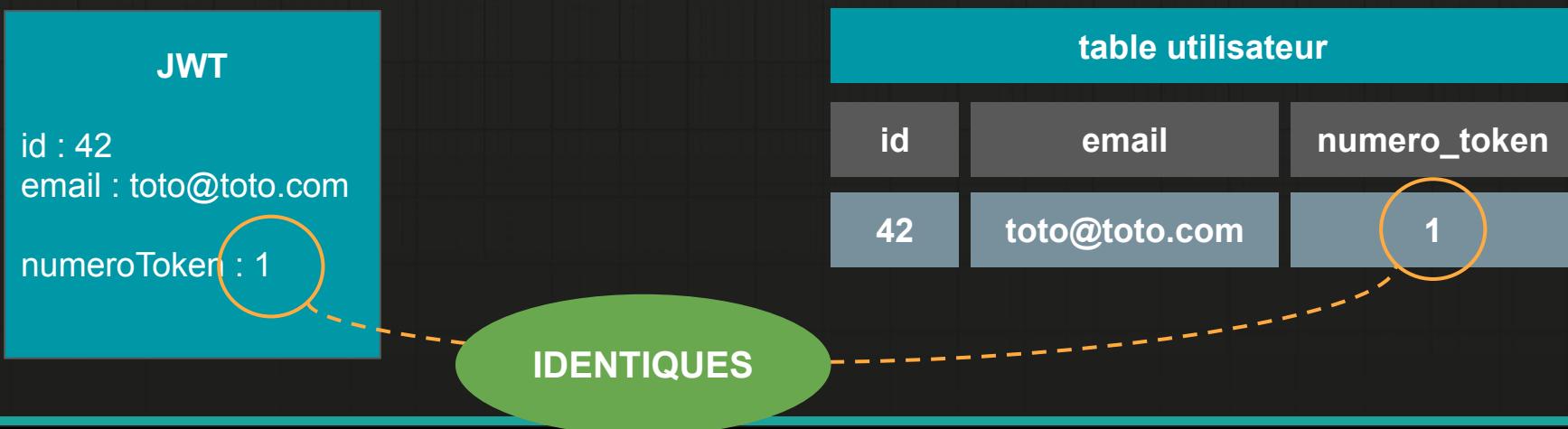


## >> Le principe d'invalidation du token

Pour rappel : à la différence des sessions, une personne connecté avec un système de token ne peut pas indiquer au serveur qu'il est "déconnecté" puisque le serveur ne stock aucune information de connexion.

Néanmoins, il est possible de rendre invalide un token malgré que sa signature soit valide.

Il suffit pour cela d'ajouter dans le corp du token une information qui devra correspondre à une information dans la table utilisateur de la base de données. Un simple nombre peut suffir :



## >> Le principe d'invalidation du token

Lors de la “déconnexion” de l’utilisateur, il suffira d’incrémenter ce numéro, ainsi le token volé contiendra une information différente que nous allons pouvoir comparer, et ainsi déclarer ce token comme invalide

Déconnexion

Incrémantation du champs  
“numero\_token” dans la table  
utilisateur

token volé		
JWT		
id : 42		
email : toto@toto.com		
numeroToken : 1		

DIFFÉRENTS

table utilisateur		
id	email	numero_token
42	toto@toto.com	2

## >> Mise en place : ajout de la propriété dans l'entité

Ajoutez une propriété “numeroToken ” dans l’entité Utilisateur :

```
private int numeroToken;
```

Modifiez le fichier data-mysql.sql en conséquence : définissez la valeur du champs “numro\_token” à 1 pour chaque utilisateur

Exemple :

```
INSERT INTO utilisateur (id, prenom, nom, email, numero token, mot de passe) VALUES  
(1, 'franky', 'bansept', 'toto@toto.com', 1 ,  
'$2a$10$AUz7WdRb8HukUyhZ4sFfHOFCY5ik2plVgyswIGLNdMicqKQOEhExO' ),  
(2, 'john', 'doe', 'tata@tata.com', 1 ,  
'$2a$10$AUz7WdRb8HukUyhZ4sFfHOFCY5ik2plVgyswIGLNdMicqKQOEhExO' ),  
(3, 'steeve', 'smith', 'titi@titi.com', 1 ,  
'$2a$10$AUz7WdRb8HukUyhZ4sFfHOFCY5ik2plVgyswIGLNdMicqKQOEhExO' );
```



## >> Mise en place : ajout de la propriété dans le JWT

Dans le fichier JwtUtils ajoutez le numéro de token dans les données du JWT :

```
public String generateToken(UserDetailsDemo userDetailsDemo) {  
  
    Map<String, Object> donnees = new HashMap<>();  
    ...  
  
    donnees.put("numeroToken", userDetailsDemo.getUtilisateur().getNumeroToken());  
  
    return Jwts.builder()  
        .setClaims(donnees)  
    ...  
}
```



>> Mise en place : Vérifier l'égalité de ce numéro entre le JWT et la table utilisateur

Toujours dans le fichier JwtUtils, modifier la méthode “tokenValide” en ajoutant un test permettant de vérifier si le numéro du token dans le JWT est identique à celui dans la table utilisateur :

```
public boolean tokenValide (String token, UserDetailsDemo userDetails) {  
    Claims claims = getTokenBody(token);  
  
    boolean utilisateurValide = claims.getSubject().equals(userDetails.getUsername());  
  
    boolean numeroTokenValide = claims  
        .get("numeroToken")  
        .equals(userDetails.getUtilisateur().getNumeroToken());  
  
    return utilisateurValide && numeroTokenValide;  
}
```



## >> Mise en place : Ajouter une méthode de déconnexion

```
GetMapping ("/deconnexion")
public ResponseEntity<String> deconnexion (
    @RequestHeader ("Authorization") String jwt){}

    String token = jwt.substring (7);
    int idUtilisateurConnecte = (int) jwtUtils.getTokenBody (token).get ("id");

    Optional<Utilisateur> utilisateurOptional =
        utilisateurDao.findById (idUtilisateurConnecte);

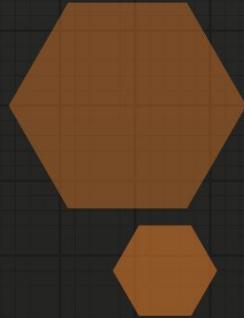
    if (utilisateurOptional.isPresent ()) {
        utilisateurOptional.get ().setNumeroToken (
            utilisateurOptional.get ().getNumeroToken () + 1);

        utilisateurDao.save (utilisateurOptional.get ());

        return ResponseEntity.ok ().build ();
    }

    return ResponseEntity.noContent ().build ();
}
```

Enfin, ajouter une méthode qui lorsqu'elle sera appelée, incrémentera le numéro de token de l'utilisateur.



## Mettre en place un système d'expiration du token

*Chapitre : Spring security*

## >> Expiration du token : Pour quel contexte ?

Il n'est pas toujours opportun de mettre en place une date d'expiration pour un token :

- Application sans transfert de donnée sensibles (*jeux mobile, application utilitaire*)
- Site e-commerce (*on préférera ne pas risquer d'avoir un client qui aurait oublié son mot de passe et rater une vente*)

Mais lorsque l'on touche à des données sensibles, il est intéressant de mettre ce système en place :

- CRM (*Customer Relationship Management*)
- DMP (*Data Management Platform*)
- Application bancaire

Dans le cas d'un CRM ou d'un DMP, on peut avoir une approche différente : aucune expiration est mise en place, mais il sera obligatoire de saisir de nouveau ses identifiants pour les opérations sensibles.



## >> Mise en place de l'expiration du token

La bibliothèque que nous avons utilisée intègre déjà un système d'expiration.  
Le JWT ainsi créé possède une propriété "exp" avec une valeur **timestamp Unix** (*Horodatage Unix*).

Un **timestamp Unix** est le nombre de secondes écoulées depuis le 1er janvier 1970.

*Ex : le 9 novembre 1989 à 19h00 correspond au timestamp 626641200*



## >> Calculer une date future

Lorsque l'on crée un objet Date en Java et que l'on ne renseigne aucun paramètre dans le constructeur, il possédera la date d'aujourd'hui.

```
Date aujourd'hui = new Date();
```

Si l'on ne le modifie pas, le fuseau horaire utilisé sera celui du système.

Nous avons déjà changé celui-ci au début du cours ([voir la slide Configurer le fuseau horaire](#))

Afin de créer une date spécifique en fonction de la date actuelle (*par exemple 5 min dans le futur*), nous allons passer en paramètre du constructeur Date, le nombre de **millisecondes** depuis le 1er janvier 1970.

Le calcul sera donc le nombre de **millisecondes** écoulées depuis aujourd'hui + le nombre de **millisecondes** que contient 5 minutes (*cad : 5 \* 60 \* 1000*). L'objet **Calendar** nous permet d'obtenir le nombre de millisecondes écoulée depuis le 1er janvier 1970 selon le fuseau horaire.

```
Calendar dateAujourd'hui = Calendar.getInstance();
long dateAujourd'huiEnMilliseconde = dateAujourd'hui.getTimeInMillis();
Date dateExpiration = new Date(dateAujourd'huiEnMilliseconde + (5 * 60 * 1000));
```

## >> Utiliser le système d'expiration de la bibliothèque

On ajoute ensuite dans le fichier **JwtUtils** une instruction pour définir cette date d'expiration lors de la génération du token.

```
return Jwts.builder()
    .setExpiration(dateExpiration)
    .setSubject(userDetailsDemo.getUsername())
    .signWith(SignatureAlgorithm.HS256, secret)
    .compact();
```

Lorsque l'on appellera la méthode **parseClaimsJws** du fichier **JwtUtils**, si la propriété **exp** possède une date plus ancienne que la date d'aujourd'hui, alors une exception sera levée, et une erreur 403 sera renvoyée.

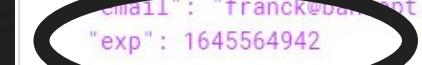
```
return Jwts.parser()
    .setSigningKey(secret)
    .parseClaimsJws(token)
    .getBody();
```

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

PAYOUT: DATA

```
{  
  "sub": "1234567890",  
  "email": "franck@bancaire.com",  
  "exp": 1645564942  
}
```



>> Comment renouveler la date d'expiration ?

TODO : cas entreprise : expiration à 12h ou minuit, ou 10h plus tard

TODO : renouvellement a chaque appel

TODO : renouvellement manuel





# Connexion OAuth2

Dans ce chapitre :

# TODO Exemple d'opérations d'Hibernate

<https://www.youtube.com/watch?v=iwr047lAc00>

<https://www.baeldung.com/rest-api-spring-oauth2-angular>



# Spring test

Dans ce chapitre :

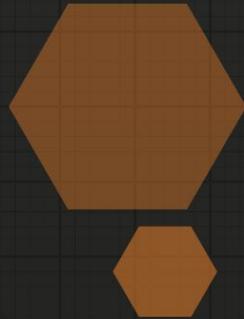
TODO : répartir sur toutes l'application

Ajouter test de performance

<https://medium.com/@johanesimarmata/performance-testing-locust-392e422e0ece>

plan de test





# Qu'est ce qu'un test unitaire ?

*Chapitre : Spring test*

## >> A quoi sert un test ?

Les tests permettent de :

- Augmenter la qualité de l'application.
- Réduire les coûts de développement du logiciel grâce à la maîtrise et à la correction en amont des défauts fonctionnels.
- Garantir l'acceptabilité du programme à la livraison
- Réduire la dette technique.



## >> Les différents familles de test Fonctionnels / non fonctionnel

Ces 2 grandes **familles** regroupent l'intégralité des tests possibles selon leur **nature**.

Un test, quelque soit son **type** (*voir slide suivante*), peut être **fonctionnel** ou **non fonctionnel**

### Les tests fonctionnels

Ils testent les fonctionnalités du logiciel vis-à-vis des demandes attendues par le client (c'est à dire ici le propriétaire du logiciel ou le Product Owner). Ils sont établis en fonction du cahier des charges et/ou users stories

### Les tests non fonctionnel

- Les tests de performance : consommation CPU, mémoire vive, montée en charge ...
- Les tests de compatibilité de plateforme
- Les tests d'ergonomie : expérience utilisateur (UX), commandes intuitives ...
- Les tests de sécurité



## >> Les différents type de test

*Le type d'un test définit la façon dont celui-ci est réalisé, ceux-ci peuvent être réalisés avec la connaissance du code à tester (boîte blanche, aussi appelé Open Box ou Glass Box ) ou sans le connaître (boîte noire)*

Les **tests unitaires** (boîte blanche) : Initiés par le développeur lui-même dans l'optique est de vérifier son code au niveau du composant qu'il doit réaliser (ex : un modèle, un contrôleur ....).

Les **tests de composants** (boîte blanche) : Ils permettent d'évaluer le bon fonctionnement d'un composant qui nécessiteraient plusieurs tests unitaires, c'est le cas par exemple d'un service d'une API qui nécessite le bon fonctionnement d'un contrôleur , d'un modèle, d'un DAO ...).

Les **tests d'intégration** (boîte blanche) : permet de s'assurer que plusieurs composantes de votre logiciel interagissent conformément aux cahiers des charges et délivrent les résultats attendus. (ex : une API et la BDD)

Les **tests systèmes** (boîte noire) : on exécute plusieurs scénarios complets qui constituent les cas d'utilisation du logiciel. L'équipe qui en a la charge est indépendante des équipes de développement.

Les **tests d'acceptation** (boîte noire, recette du logiciel) : ces tests assurent sur la conformité du logiciel aux critères d'acceptation et aux besoins des cibles. Ils sont donc généralement réalisés par le client final ou les utilisateurs.



## >> Les test unitaires en détails

Ce sont les premiers tests à mettre en place par les développeurs

Comme leur nom l'indique, ils ne doivent tester que le bon fonctionnement d'un seul composant (une classe, une méthode, un contrôleur, un DAO, un modèle ...)

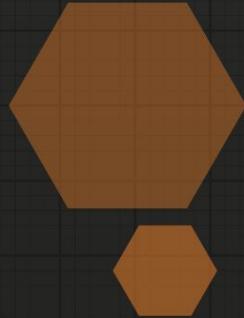
La plus grande difficulté consiste à isoler le composant à tester des autres durant le test.

*Exemple : il faut tester la méthode du contrôleur UtilisateurController permettant de récupérer un utilisateur*

*Ce test devra être effectué sans pour autant utiliser le DAO UtilisateurDao.*

*Le but est de pouvoir vérifier quel composant comporte une erreur, dans le cas où c'est le DAO qui pose problème, le test du contrôleur échouerait également.*





# Mettre en place des tests unitaire

*Chapitre : Spring test*

## >> Ecrire des tests

Dans la classe nous allons pouvoir ajouter des méthodes avec l'annotation `@Test`. Tous les tests seront exécutés les uns après les autres.

```
@Test
public void premierTest_doitEtreValide() throws Exception {
    assertThat("Hello, World").contains("Hello, World");
}

@Test
public void testQuiEchoueToujours_nePeutPasEtreValide() throws Exception {
    assertThat("Hello, World").contains("Bonjour");
}

@Test
public void testToujoursAvecException_nePeutPasEtreValide() throws Exception {
    throw new Exception("Le test a retourné une exception...");
}
```



## >> Exécuter les tests

On peut lancer la classe de test de la même manière que l'on lance une application.

Le résultat de nos 3 tests est alors affiché :

- Le premier a fonctionné
- Le deuxième a échoué car le test était faux
- Le troisième a échoué car il a retourné une Exception (Comme n'importe quel programme pourrait le faire)

Run: SpringsecurityApplicationTests

Test Results

- SpringsecurityApplicationTests
  - premierTest\_doitEtreValide() (Passed)
  - testQuiEchoueToujours\_nePeutPasEtreValide() (Failed)
  - testToujoursAvecException\_nePeutPasEtreValide() (Failed)

Tests failed: 2, passed: 1 of 3 tests – 1s 82 ms

Time	Log
10:40:01.945	[main] DEBUG org.springframework.security.core.context.SecurityContextHolder - Retrieved SecurityContext from SecurityContextHolder: SecurityContextImpl{Authentication=org.springframework.security.authentication.UsernamePasswordAuthenticationToken@1234567890: principal=, credentials=, authorities=[ ]}
10:40:01.964	[main] DEBUG org.springframework.security.core.context.SecurityContextHolder - Set SecurityContext to: SecurityContextImpl{Authentication=org.springframework.security.authentication.UsernamePasswordAuthenticationToken@1234567890: principal=, credentials=, authorities=[ ]}
10:40:02.017	[main] DEBUG org.springframework.security.core.context.SecurityContextHolder - Retrieved SecurityContext from SecurityContextHolder: SecurityContextImpl{Authentication=org.springframework.security.authentication.UsernamePasswordAuthenticationToken@1234567890: principal=, credentials=, authorities=[ ]}
10:40:02.044	[main] INFO org.springframework.security.core.context.SecurityContextHolder - Set SecurityContext to: SecurityContextImpl{Authentication=org.springframework.security.authentication.UsernamePasswordAuthenticationToken@1234567890: principal=, credentials=, authorities=[ ]}
10:40:02.052	[main] DEBUG org.springframework.security.core.context.SecurityContextHolder - Retrieved SecurityContext from SecurityContextHolder: SecurityContextImpl{Authentication=org.springframework.security.authentication.UsernamePasswordAuthenticationToken@1234567890: principal=, credentials=, authorities=[ ]}

## >> Convention d'écriture des tests

Il n'y a pas de convention universelle pour le nommage des tests. Mais le bon sens nous force à faire en sorte que le nom des test :

- Ne commence pas par une majuscule puisque ce sont des méthodes
- Comporte l'action ou l'objet qui est testé
- Comporte l'état attendu de cette action ou de cet objet
- Optionnellement : comporte le contexte si les tests sont effectués dans des environnement différents (base de donnée, requête d'api, test de méthode ...)
- Une convention de nommage doit être toujours la même pour tous les tests

Voilà quelques exemples d'une convention de nommage totalement arbitraire :

- `appelGetListeUtilisateur_devraitRenvoyer10Utilisateurs`
- `ajoutUtilisateurPuisRequeteSurUtilisateur_devraitRenvoyerNouvelUtilisateur`
- `requeteAccueil_devraitRenvoyerStatus400`



# >> Exemple test de model

## TODO

```
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.validation.ValidatorFactory;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.Set;

public class ProduitTests {

    private Validator validator;

    @BeforeEach
    public void setUp() {
        ValidatorFactory factory =
Validation.buildDefaultValidatorFactory();
        validator = factory.getValidator();
    }
}
```

```
@Test
public void testValidProduit() {
    Produit produit = new Produit();
    produit.setNom("Nom valide");
    produit.setCode("CODE123");
    produit.setPrix(10.0f);
    produit.setEtat(new Etat());

    Set<ConstraintViolation<Produit>> violations = validator.validate(produit);
    Assertions.assertTrue(violations.isEmpty());
}

@Test
public void creationProduitSansNom_shouldReturnValidationError() {

    Produit produit = new Produit();
    produit.setNom(null);
    produit.setCode("CODE123");
    produit.setPrix(10.0f);
    produit.setEtat(new Etat());

    Set<ConstraintViolation<Produit>> violations = validator.validate(produit);
    Assertions.assertFalse(violations.isEmpty());
    Assertions.assertEquals(1, violations.size());
    Assertions.assertEquals("Le nom ne peut être vide,
        null ou ne comporter que des espaces",
        violations.iterator().next().getMessage());
}
```



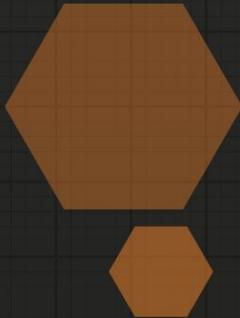
# >> Exemple test de contrôleur

## TODO

```
public class FakeProduitDao implements ProduitDao {  
    ...  
  
    @Override  
    public Optional<Produit> findById(Integer integer) {  
  
        if (integer == 1) {  
            Produit produit = new Produit();  
            produit.setId(1);  
            return Optional.of(produit);  
        } else {  
            return Optional.empty();  
        }  
    }  
    ...  
}
```

```
public class ProduitControllerTests {  
  
    @Test  
    public void testGetProduitExistant_shouldReturnProduit () {  
  
        ProduitController controller = new ProduitController( new  
FakeProduitDao());  
        ResponseEntity<Produit> reponse = controller.getProduit( 1);  
        Assertions.assertEquals(HttpStatus.OK, reponse.getStatusCode());  
    }  
  
    @Test  
    public void testGetProduitInexistant_shouldReturn404Error () {  
  
        ProduitController controller = new ProduitController( new  
FakeProduitDao());  
        ResponseEntity<Produit> reponse = controller.getProduit( 2);  
        Assertions.assertEquals(HttpStatus.NOT_FOUND, reponse.getStatusCode());  
    }  
}
```





# Mettre en place des tests de composant avec spring test

*Chapitre : Spring test*

## >> Spring security test

Si nous intégrons à la fois spring test et spring security, nous allons avoir besoin d'une dépendance qui va nous permettre d'effectuer les tests en simulant que nous sommes connectés à l'application.

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



## >> Liste des imports

Afin de ne pas confondre des classes utilisées avec des classes homonyme d'autres packages, voilà les imports qui seront effectués dans les exemples suivants.

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.security.test.context.support.WithMockUser;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
import static org.springframework.security.test.web.servlet.setup.SecurityMockMvcConfigurers.springSecurity;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
```

## >> Préparer la classe de test

La classe de test se trouve dans **src/test/.../NomApplicationTests.java**

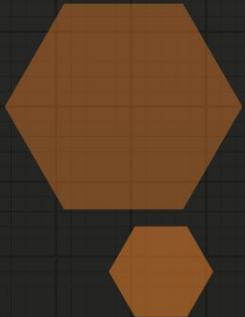
Elle possède une Annotation `@SpringBootTest` dans laquelle nous allons ajouter un paramètre permettant de démarrer notre serveur de test sur un port aléatoire afin de limiter les conflits.

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

... ICI NOUS ÉCRIRONS NOS TESTS

}





# Tester les API

*(Tests de composants)*

*Chapitre : Spring test*

## >> MockMvc et spring security

MockMvc est intégré à **Spring Test**. Il va nous permettre de simuler des requêtes vers nos API. Il nous sera alors possible de vérifier le statut des requêtes, le contenu etc ... Nous allons instancier un objet **MockMvc** en le paramétrant pour **Spring Security**.

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

    @Autowired
    private WebApplicationContext context;

    private MockMvc mvc;

    @BeforeEach
    public void setup() {
        mvc = MockMvcBuilders
            .webAppContextSetup(context)
            .apply(springSecurity())
            .build();
    }

    ...
    ICI NOUS ÉCRIRONS NOS TESTS
}
```

Avant d'effectuer nos tests nous allons créer une instance de MockMvc grâce à l'annotation `@BeforeEach`

Les tests devront être écrit ici

## >> Tester avec MockMvc

On peut simuler un utilisateur connecté via  
l'annotation @WithMockUser

```
@WithMockUser(username = "admin", roles={"ADMIN"})
@Test
public void accessAUneResourcesReserveAuxAdmin_doitRetournerUnStatut200() throws Exception {

    mvc.perform(get("/admin/hello"))
        .andExpect(status().isOk());

    mvc.perform(get("/utilisateur/${i}", 42))
        .andExpect(status().isOk());
}
```



## >> MockMvc test sur json

Chaîné à la méthode il est possible de vérifier le type de contenu qui est envoyé / reçu.

```
mvc.perform(get("/recuperation-json")
    .contentType(MediaType.APPLICATION_JSON)
)

mvc.perform(get("/envoie-json")
    .accept(MediaType.APPLICATION_JSON)
)
```



## >> MockMvc test sur json

```
@Autowired  
private ObjectMapper mapper;  
  
...  
  
Utilisateur utilisateur = new Utilisateur();  
utilisateur.setPseudo("test2");  
utilisateur.setMotDePasse("test2");  
String json = mapper.writeValueAsString(utilisateur);  
  
post("/authentication")  
    .contentType(MediaType.APPLICATION_JSON)  
    .accept(MediaType.APPLICATION_JSON)  
    .content(json))
```



## >> Exemple d'opérations d'Hibernate

### MockMvcResultMatchers

Afin de tester le JSON retourné il est possible d'utiliser MockMvcResultMatchers qui prend en paramètre une expression qui va extraire la donnée souhaitée du JSON.

Soit le retour suivant

```
{  
    "id":1,  
    "prenom":"Franck",  
    "categories":["prof","java"]  
}  
  
[  
    {"id":1,"nom":"prof"},  
    {"id":2,"nom":"java"}  
]  
  
mvc.perform(get("/utilisateur/{$i}",1))  
.andExpect(MockMvcResultMatchers.jsonPath("$.id").exists())  
.andExpect(jsonPath("$.description").value("firstName2"))  
.andExpect(jsonPath("$.categorie[1]").value("java"));  
  
mvc.perform(get("/categories"))  
.andExpect(jsonPath"][$0].nom".value("prof"));
```



# Faire évoluer l'application

Dans ce chapitre :

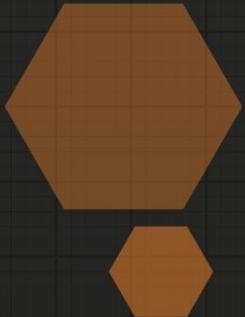
ajouter un historique de prix

droit par héritage

# Mise en production

Dans ce chapitre :

# TOMCAT

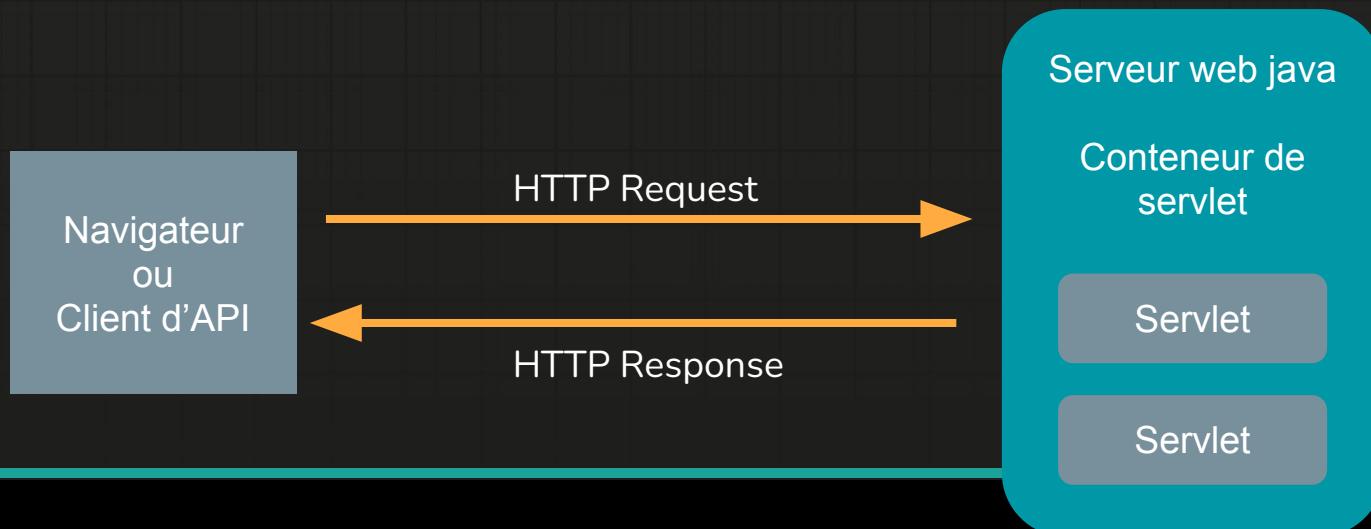


*Chapitre : Mise en production*

## >> Un conteneur de servlet

**Tomcat** est le plus populaire des serveurs pour les applications JAVA orienté web, Il est construit sur un modèle standard de serveur HTTP requête / réponse.

La **requête** atteint une **Servlet** qui générera du code HTML (ou json, xml ...) à la volé en réponse à une URL ou à un formulaire envoyé au navigateur (ou un client d'une API comme une application mobile par exemple).



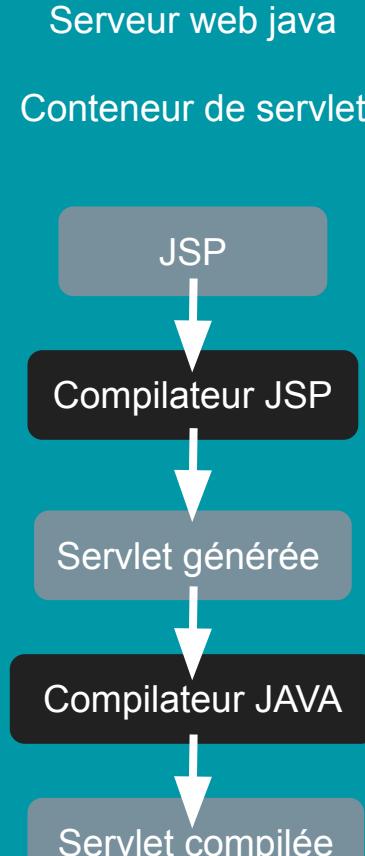
# >> Les servlets

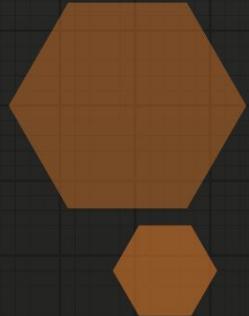
Les servlets sont des classes JAVA spécialisées, utilisées dans une application web, et pouvant s'exécuter sur des serveurs.

Elles sont capables d'intercepter les requêtes HTTP (GET, POST, DELETE ...), faisant alors office de contrôleur.

Si la ressource visée est un fichier JSP, le fichier est tout d'abord interprété par le compilateur de JSP, qui le transforme alors en Servlet.

Cette servlet est alors compilé par le compilateur de JAVA.





# Les composants de TOMCAT

*Chapitre : Mise en production*

## >> Les composants de Tomcat

Les serveurs Tomcat possèdent un certains nombre de composant, mais la quasi totalité des fonctionnalités sont partagées par 4 composants majeurs

Serveur web java

Conteneur de servlet

Catalina

Jasper

Coyote

Cluster



## >> Les composants de Tomcat

Catalina est un conteneur système basique suivant les spécifications des Servlets.

Il gère le cycle de vie des Servlets comme init(), service(), destroy() ...

Serveur web java

Conteneur de servlet

Catalina

Jasper

Coyote

Cluster



## >> Les composants de Tomcat

Jasper est un parseur de JSP.

Il doit générer le servlet correspondant

Il est également responsable de la détection des changements des fichiers JSP et de régénérer le Servlet le cas échéant.

Serveur web java

Conteneur de servlet

Catalina

Jasper

Coyote

Cluster



## >> Les composants de Tomcat

Coyote est un connecteur de composant.

Il écoute les ports TCP/IP afin de transmettre les requêtes aux conteneurs ainsi qu'au client ou au autre serveur web.

Il permet également d'utiliser le protocole SSL

Serveur web java

Conteneur de servlet

Catalina

Jasper

Coyote

Cluster



## >> Les composants de Tomcat

Comme son nom l'indique, cluster permet de gérer le comportement du serveur dans un environnement distribué.

Il permet notamment de gérer les sessions entre les différents réplicats

Serveur web java

Conteneur de servlet

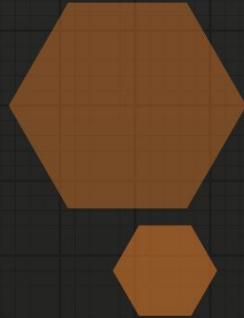
Catalina

Jasper

Coyote

Cluster





# Installer un serveur Tomcat 10

*Chapitre : Mise en production*

# >> Les composants de Tomcat

La version de **Tomcat** que nous allons sélectionner est la **version 9**.

Cette version à l'avantage de prendre en charge la **version 8 de JAVA** qui dispose d'un support poussé dans le temps (ainsi que les spécification requise pour Java EE 8).

La version 10 de Tomcat n'est pas encore assez éprouvé et les ressources concernant ses specification encore trop peu nombreuses pour constituer un choix plus intéressant. (néanmoins il utilise les spécification requise pour Jakarta ee 9)

De plus sont déploiement en entreprise reste plutôt faible à l'heure actuelle

Télécharger le dossier binaire à l'adresse <https://tomcat.apache.org/download-90.cgi> décompressez et renommez le à l'emplacement désiré (il ne nécessite pas d'installation) par exemple C:\Program Files\tomcat (ce dossier doit alors contenir les dossiers bin, conf, lib ...)

Serviet Spec	JSP Spec	EL Spec	WebSocket Spec	Authentication (JASIC) Spec	Apache Tomcat Version	Latest Released Version	Supported Java Versions
5.0	3.0	4.0	2.0	2.0	10.0.x	10.0.2	8 and later
4.0	2.3	3.0	1.1	1.1	9.0.x	9.0.43	8 and later
3.1	2.3	3.0	1.1	1.1	8.5.x	8.5.63	7 and later
3.1	2.3	3.0	1.1	N/A	8.0.x (superseded)	8.0.53 (superseded)	7 and later
3.0	2.2	2.2	1.1	N/A	7.0.x	7.0.108	
2.5	2.1	2.1	N/A	N/A	6.0.x (archived)	6.0.53 (a)	
2.4	2.0	N/A	N/A	N/A	5.5.x (archived)	5.5.36 (a)	
2.3	1.2	N/A	N/A	N/A	4.1.x (archived)	4.1.40 (archived)	1.3 and later
2.2	1.1	N/A	N/A	N/A	3.3.x (archived)	3.3.2 (archived)	1.1 and later

La version 8 de JAVA est  
en Long Term Support



## >> Installation sur Linux

Une fois les sources du JDK et de tomcat téléchargées (au format .tar.gz)

```
wget https://downloads.apache.org/tomcat/tomcat-9/v9.0.43/bin/apache-tomcat-9.0.43.tar.gz
```

```
wget https://github.com/AdoptOpenJDK/openjdk8-binaries/releases/download/jdk8u282-b08/OpenJDK8U-jdk\_x64\_linux\_hotspot\_8u282b08.tar.gz
```

Décomptez les 2 dossiers :

```
tar -zxvf apache-tomcat-9.0.43.tar.gz
```

```
tar -zxvf OpenJDK8U-jdk_x64_linux_hotspot_8u282b08.tar.gz
```

Renommez/déplacez les 3 dossiers :

```
mv apache-tomcat-9.0.43 /usr/lib/tomcat9
```

```
mv jdk8u282-b08 /usr/lib/jvm/jdk8
```



# Linux : Configuration du bash\_profile

Editez ou créez un fichier `~/.bash_profile` via un éditeur (nano, vi, vim ...) et ajoutez les ligne suivantes :

```
export JAVA_HOME=/usr/bin/java/jdk8  
export JRE_HOME=/usr/bin/java/jdk8  
export CATALINA_HOME=/usr/bin/tomcat9
```

Enregistrez le fichier et exécutez la commande :

```
source ~/.bash_profile
```



# >> Linux : Lancer le serveur Tomcat

Lancez la commande suivante :

```
$CATALINA_HOME/bin/startup.sh
```

Un résumé des différents PATH est alors affiché :

```
Using CATALINA_BASE:   /usr/bin/tomcat
Using CATALINA_HOME:  /usr/bin/tomcat
Using CATALINA_TMPDIR: /usr/bin/tomcat/temp
Using JRE_HOME:        /usr/bin/jvm/jdk8
Using CLASSPATH:       /usr/bin/tomcat/bin/bootstrap.jar:/usr/bin/tomcat/bin/tomcat-juli.jar
Using CATALINA_OPTS:
Tomcat started.
```

Si le message suivant persiste : Neither the JAVA\_HOME nor the JRE\_HOME environment variable is defined  
At least one of these environment variable is needed to run this program

Ajouter la ligne suivante au debut des fichiers **startup.sh** et **shutdown.sh**  
`export JAVA_HOME=/usr/java/jdk`

Vous pouvez alors vérifier que le serveur est disponible à l'adresse <http://localhost:8080/> via votre navigateur

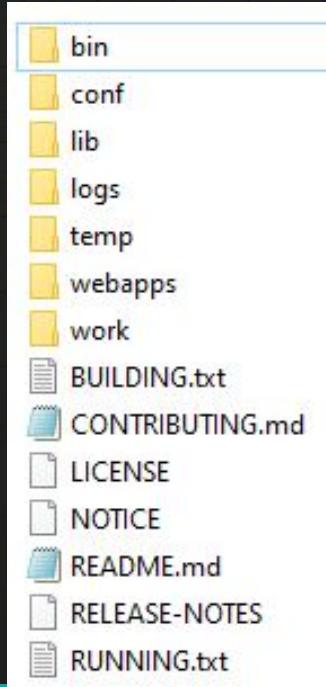


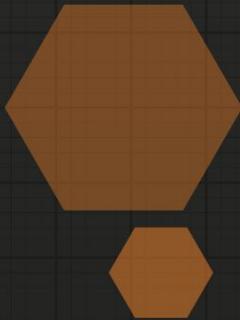
# >> Structure de Tomcat

- **bin** : contient tous les fichiers exécutable
  - dont startup.sh, shutdown.sh et leur équivalent .bat (il n'y a pas d'installation séparée)
- **conf** : contient tous les fichiers de configuration de tomcat
  - tomcat-users.xml va contenir les différents utilisateurs / rôles
  - server.xml est le comportement générale de tomcat
  - logging.properties définit le comportement des logs
- **lib** : est une bibliothèque partagée où l'on trouvera différents fichiers jar.
  - tomcat en utilise certains mais il est possible de les utiliser également en tant que dépendance pour la compilation de programmes
  - Cette bibliothèque est ajoutée aux classpaths de tous les applications déployées
- **logs** : contient les logs de chaque application
- **temp** : contient les fichiers temporaires
  - On y retrouvera les fichiers uploadés sur le serveur par exemple
- **webapps** : contient toutes les applications que nous allons déployer ainsi que les utilitaires de tomcat
- **work** : la conversion de JSP en servlet a lieu dans ce dossier

*Exemple : Dans manager app, lancez une application utilisant un JSP (/exemple puis "JSP examples" puis "Hello World Tag") une arborescence apparaît alors dans le dossier work.*

*Le fichier hello\_jsp.java est le servlet qui découle du fichier JSP original*





# Configurer les utilisateurs de TOMCAT

*Chapitre : Mise en production*

## >> Ajouter un utilisateur à Tomcat

```
vim /etc/tomcat9/tomcat-users.xml
```

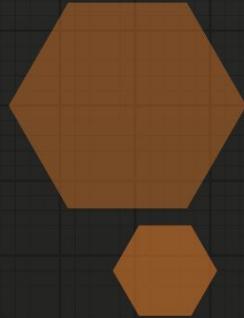
Modifier le fichier de façons à ce que dans le noeud <tomcat-users> il existe l'élément suivant :

```
<user username="          " password="mot-de-passe-a-noter"  
roles="tomcat,manager-gui,admin-gui,manager-script"/>
```

Puis redémarrez le service tomcat

```
sudo systemctl restart tomcat9
```





# Le Manager GUI

*(Graphical User Interface)*

*Chapitre : Mise en production*

## >> Accéder au Manager GUI



Gestionnaire d'applications WEB Tomcat

Message : UK

Gestionnaire

Liste des applications Aide | ITIL Gestionnaire Aide Gestionnaire Etat du serveur

Applications

Chemin	Version	Nom d'chargement	Fonctionnel	Sessions	Commandes
/	Aucun spécifié		true	0	Démarrer Arrêter Recharger Retirer Explorer les sessions Inactives depuis > 30 minutes
host-manager	Aucun spécifié	Tomcat Host Manager Application	true	0	Démarrer Arrêter Recharger Retirer Explorer les sessions Inactives depuis > 30 minutes
manager	Aucun spécifié	Tomcat Manager Application	true	1	Démarrer Arrêter Recharger Retirer Explorer les sessions Inactives depuis > 30 minutes

Le Manager GUI va nous permettre d'uploader des applications via une interface web. Elle est accessible à cette URL : <http://<IP de votre serveur>:8080/manager/html>

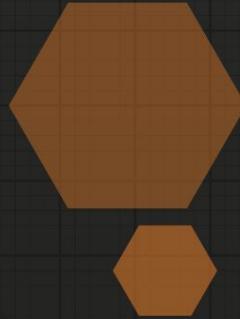
Renseignez le nom de l'utilisateur et le mot de passe que nous avons renseigné à la slide précédente

Si une page avec une erreur **403** survient, vérifier que les informations de connexion sont corrects (*celles que vous avez renseignées dans la boîte de dialogue, ainsi que le fichier tomcat-users.xml*)

Si une page avec une erreur **404** survient, installé le package nécessaire :

```
sudo apt install tomcat9-admin
```

```
sudo systemctl restart tomcat9
```



# Modification de l'application pour être déployable sur un serveur Tomcat 10

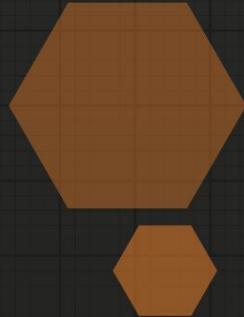
*Chapitre : Mise en production*

## >> Le fichier principal

Pour rendre compatible notre application avec tomcat 9 nous devons ajouter un héritage sur la classe de l'application, ainsi qu'une surcharge d'une méthode

```
...
public class DemoApplication extends SpringBootServletInitializer {
    ...
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(DemoApplication.class);
    }
    ...
}
```





# Ajouter des profils dans Maven

*Chapitre : Mise en production*

## >> Problématique de la mise en production

Lors de la **mise en production**, nous allons faire face à une **problématique** : Quelles informations renseigner quand celles-ci sont **différentes** entre le **PC de développement** et le **serveur de mise en production** ? Il est bien sûr possible de faire en sorte que les données soient identiques , ou bien de les modifier avant chaque mise en production. Mais ce sont des approches qui se révèlent **contre-productives** (risque d'oublis, mot de passe exposé ...)



## >> Principe des profils

Les **profils** permettent de régler ce problème : il suffit de déclarer des **propriétés** qui auront des valeurs différentes selon l' **environnement**. Dans cet exemple, si le programme est compilé avec le profil “production”, alors le mot de passe écrit dans le fichier **application.properties** sera : ivnbevnei46

Profil local

<mdp-mysql>  
root

Profil production

<mdp-mysql>  
ivnbevnei46



## >> Principe des profils

Afin que la compilation ne dure pas trop longtemps, nous allons indiquer à maven que les fichiers à modifier (*selon le profil sélectionné*) se trouvent dans le dossier “src/main/ressources”, plutôt que de lui faire parcourir l’intégralité de l’application.

Ajouter le noeud suivant dans le noeud <build> nous permet d’activer cette fonctionnalité

```
<build>
  <resources>
    <resource>
      <filtering>true</filtering>
      <directory>src/main/resources</directory>
    </resource>
  </resources>
...
</build>
```

### Profil local

<mdp-mysql>  
root

<user-mysql>  
root

### Profil production

<mdp-mysql>  
ivnbevnei46

<user-mysql>  
admin

mot de passe MySql

<mdp-mysql>

utilisateur MySql

<user-mysql>



## >> Ajouter les différents profils

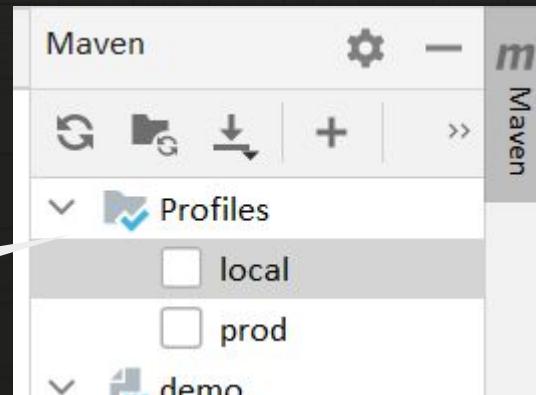
Puis dans le noeud <project> nous allons ajouter un noeud <profiles> qui contiendra tous les profils de notre application.

```
<profiles>
  <profile>
    <id>local</id>
  </profile>

  <profile>
    <id>prod</id>
  </profile>
</profiles>
```

Ici on ajoute 2 profils : prod et local.  
Le noeud id permet de les nommer

Une fois Maven rafraîchi, vous devriez constater la présence des 2 profils dans l'onglet Maven



## >> Déclarer un profil par défaut

Dans le profil local, on peut ajouter un nœud permettant de signaler ce profil comme étant sélectionné par défaut.

```
<profile>

    <activation>
        <activeByDefault>true</activeByDefault>
    </activation>

    <id>local</id>

</profile>
```



## >> Ajouter les propriétés pour le développement Local

Dans le profil local, nous allons ajouter un nœud <properties> et y ajouter toutes les propriétés susceptibles de changer entre les différents environnements. Le texte entre ces balises seront les valeurs que ces propriétés auront pour ce profil.

```
<profile>
    <activation>
        <activeByDefault>true</activeByDefault>
    </activation>
    <id>local</id>
    <properties>
        <adresse-bdd>localhost</adresse-bdd>
        <nom-bdd>spring</nom-bdd>
        <nom-utilisateur-bdd>root</nom-utilisateur-bdd>
        <mot-de-passe-utilisateur-bdd></mot-de-passe-utilisateur-bdd>
    </properties>
</profile>
```

Note : Le nom des profils, ainsi que le nom des propriétés est totalement arbitraire



## >> Ajouter les propriétés pour le déploiement en production

Ajoutez ensuite au profil de production les mêmes propriétés mais avec les valeurs propre à ce profil

```
<profile>
  <id>prod</id>
  <properties>
    <adresse-bdd>localhost</adresse-bdd>
    <nom-bdd>spring</nom-bdd>
    <nom-utilisateur-bdd>nomUtilisateur</nom-utilisateur-bdd>
    <mot-de-passe-utilisateur-bdd>motDePasse</mot-de-passe-utilisateur-bdd>
  </properties>

</profile>
```



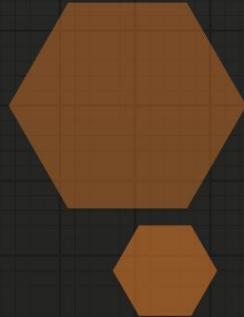
## >> Modifier le fichier application.properties

La dernière étape consiste à modifier le fichier application.properties. Nous allons placer les différentes propriétés via leur nom entouré du caractère "@"

```
spring.datasource.url =jdbc:mysql://@adresse-bdd@/@nom-bdd@?serverTimezo .. ....  
spring.datasource.username =@nom-utilisateur-bdd@  
spring.datasource.password =@mot-de-passe-utilisateur-bdd@
```

*Note : En temps normal, Maven n'utilise pas 2 caractères "@" pour placer les propriétés des profils, mais plutôt cette notation : @{nom-propriete}. Seulement Spring utilise déjà cette notation afin de récupérer les valeurs du fichier "application.properties". Spring à donc remplacer les caractères utilisés par Maven par 2 caractères "@"*





Empêcher les test unitaires lors de la compilation pour le déploiement

*Chapitre : Mise en production*

## >> Ajouter un plugin permettant de ne pas effectuer les tests unitaires

Lors de la compilation, Maven va tenter d'effectuer les tests unitaires que nous avons mis en place. Ces tests vont échouer s'ils sont exécutés en local, alors qu'ils essaient d'utiliser le profil de mise en production (ex : *pas le bon mot de passe pour la base de donnée*). Pour éviter cela, nous allons ajouter un comportement à Maven via l'ajout d'un plugin dans le noeud <build>

```
...
<build>
...
<plugin>
    <groupId>org.apache.maven.plugins</ groupId>
    <artifactId>maven-surefire-plugin</ artifactId>
    <version>3.0.0-M6</ version>
    <configuration>
        <skipTests>true</ skipTests>
    </ configuration>
</ plugin>
...
```

Si cette valeur est “true”, alors les tests unitaires ne seront pas effectués



## >> Rendre les tests unitaires optionnels via une propriété

Mais comme nous voulons tout de même effectuer les tests unitaires dans certains cas, nous allons ajouter une propriété à la place de la valeur “true”. Nous la nommerons : “sansTestsUnitaires”

```
...  
<build>  
...  
    <plugin>  
        <groupId>org.apache.maven.plugins</groupId>  
        <artifactId>maven-surefire-plugin</artifactId>  
        <version>3.0.0-M6</version>  
        <configuration>  
            <skipTests>${sansTestsUnitaires}</skipTests>  
        </configuration>  
    </plugin>  
...
```

Note : comme il ne s'agit pas du code d'un fichier de Spring, nous devons utiliser la notation normale \${nom-propriete} et non la notation @nom-propriete@



## >> Déclarer la propriété dans le projet

Par défaut nous allons déclarer cette propriété dans le fichier pom.xml avec “false” comme valeur.  
Ajoutez cette dernière dans le noeud <properties> du noeud <project>

```
<project ...>  
  
<groupId>com.example</groupId>  
...  
<description>Demo project for Spring Boot</description>  
<properties>  
    <java.version>11</java.version>  
    <sansTestsUnitaires>false</sansTestsUnitaires>  
</properties>
```



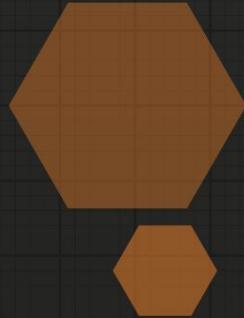
## >> Affecter une valeur particulière pour le profil de production

Puis nous allons ajouter cette propriété en plus de celles déjà présentes dans le profil de production avec comme valeur “true”, ainsi les tests ne seront ignorés que si le profil “prod” est sélectionné

```
<profile>
  <id>prod</id>
  <properties>
    <sansTestsUnitaires>true</sansTestsUnitaires>
    <secret-jwt>azerty123</secret-jwt>
    <adresse-bdd>localhost</adresse-bdd>
    <nom-bdd>spring_cda_2022</nom-bdd>
  ...

```





Définir des propriétés  
confidentielles ou spécifiques à un  
environnement local

*Chapitre : Mise en production*

## Problématique des informations sensibles

Certaines propriétés ne doivent pas être exposées dans les sources du programme. Par exemple le mot de passe de la base de données du serveur de production ou celui permettant de générer les signatures des JWT, ou encore les clés des API.

Des collaborateurs (stagiaires, développeurs sous contrat court ...) pourraient profiter de ces informations afin d'effectuer des opérations malveillantes, ou tout simplement égarer ces informations.

Les répository comme github ou gitlab sont également susceptibles d'être visibles par des personnes peu scrupuleuses au sein même de leur entreprise.

D'une manière générale, il est toujours plus prudent de ne divulguer ces informations qu'à un nombre restreint de personnes.



## >> Des informations propres à un système local

Il est également possible que certaines informations soient propres à un système local. En reprenant l'exemple de l'adresse de la base de donnée : il est possible qu'un développeur ait besoin de spécifier une adresse spécifique qui ne serait pas "localhost" mais "localhost:3307" ou un nom de domaine local comme "base\_de\_donnee\_42"

Plutôt que de définir un profil particulier pour chaque développeur ayant des besoins spécifiques, il est plus pratique de définir ces données particulières sur cet environnement local.

C'est 2 cas peuvent être réglés en utilisant un fichier en dehors des sources du projet.  
Que ce soit sur le serveur dont il faudra protéger les données, ou bien l'ordinateur ayant besoin d'informations spécifiques



## >> Le fichier settings.xml

Ce fichier se trouve dans un dossier caché :

Windows : C:\Users\nom\_utilisateur\.m2

Linux macOs : ~/.m2

Si il n'existe pas, vous pouvez le créer

Vous pouvez également définir son emplacement dans IntelliJ :

File | Settings | Build, Execution, Deployment | Build Tools | Maven | User setting file



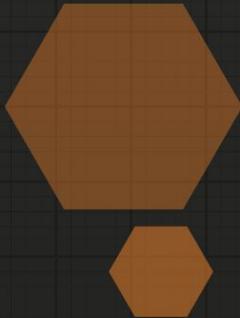
## >> Le fichier settings.xml

Le fichier permet de définir plusieurs profils à l'instar du fichier pom.xml. Le fait de sélectionner un profil dans ce fichier donnera la priorité à toutes les propriété qu'il définit

```
<profiles>
  <profile>
    <id>environnement personnel</id>
    <properties>
      <adresse-bdd>localhost:3307</adresse-bdd>
    </properties>
  </profile>
</profiles>
```

Dans cet exemple, si l'on sélectionne le profil “local” du fichier pom.xml ainsi que ce profil “environnement personnel”, la valeur de la propriété adresse-bdd sera bien “localhost:3307” et non “localhost”





Créer une archive de l'application  
afin qu'elle soit déployée sur le  
serveur

*Chapitre : Mise en production*

## >> Créer un fichier WAR

Afin de pouvoir être déployer par l'interface graphique de TOMCAT, le projet doit être archivé dans un seul fichier (comme une pièce jointe pour un email).

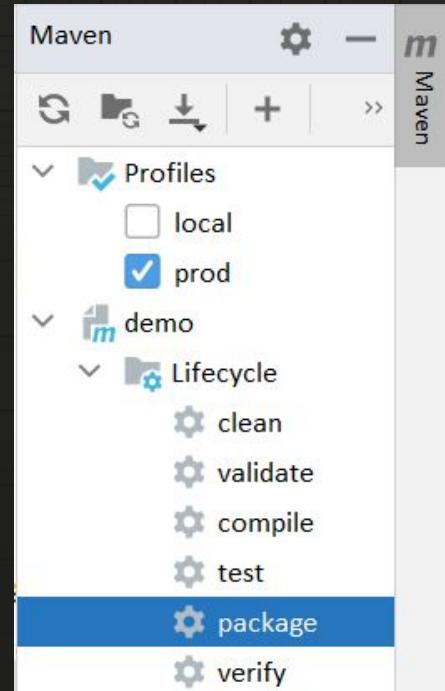
Ce fichier sera un fichier WAR (Web Archive) et peut être obtenu via le menu Maven d'IntelliJ ou bien par ligne de commande.

Afin de générer un fichier war il est nécessaire d'ajouter le noeud <pakaging> dans le noeud <projet> du fichier pom.xml

```
<packaging>war</packaging>
```

Et de double cliqué sur la commande “package” du menu Maven (sans oublier de sélectionner le profil “prod”

Une fois la commande terminé, un fichier WAR du nom de l'application suivi du numéro de la version est générée dans le dossier “target”  
ex : “demo-0.0.1-SNAPSHOT.war”



## >> Créer un fichier WAR par commande

Il est également possible de créer ce même fichier par ligne de commande.

Ce qui peut s'avérer utile dans le cas où l'on cherche à automatiser certaines tâches, comme l'automatisation de la mise en production :

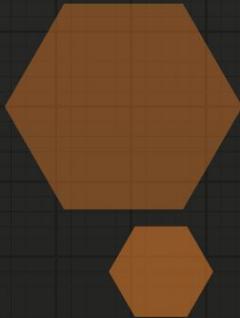
```
mvn package -Pnom-profil → mvn package -Pprod
```

Notez qu'en ayant utilisé un template de Spring via Maven, vous devriez avoir un chier “mvnw” (pour linux) et “mvnw.cmd” (pour Windows) contenant le programme Maven (Bien utile pour lancer des commandes automatique)

Naturellement la commande équivalente serait alors :

```
mvnw package -Pprod
```



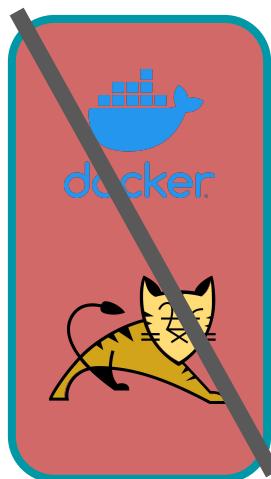


# Héberger l'application sur une architecture docker (méthode recommandée)

*Chapitre : Mise en production*

## >> Procédure à mettre en place

Suppression du conteneur de l'ancienne version



Code du Repository pull

Dockerfile

Maven™

.WAR



docker

Image basée sur Tomcat et contenant le nouveau WAR



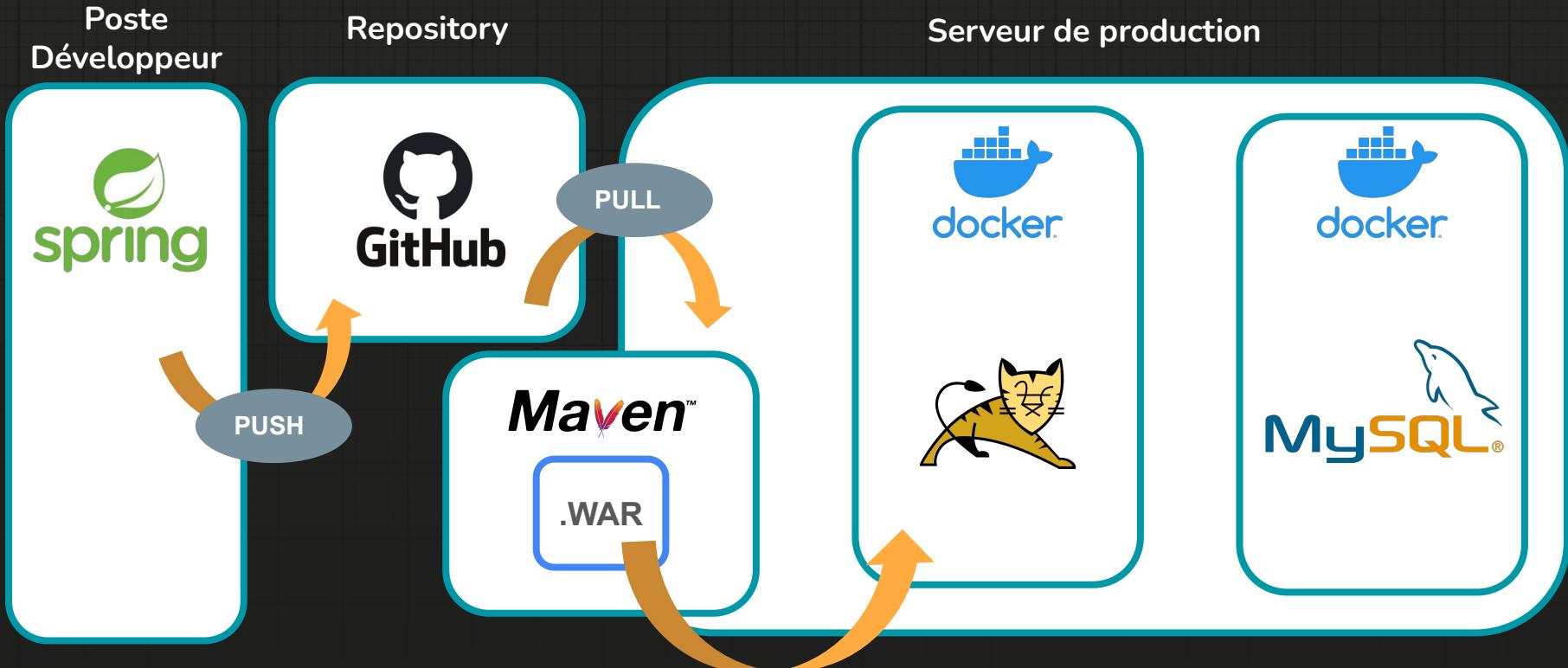
docker

Nouveau Conteneur



Application Déployée

## >> Procédure à mettre en place



# >> Installation de git sur le serveur

Pour Debian / Ubuntu :

```
sudo apt update
```

```
sudo apt install git
```

Pour Centos :

```
sudo yum update
```

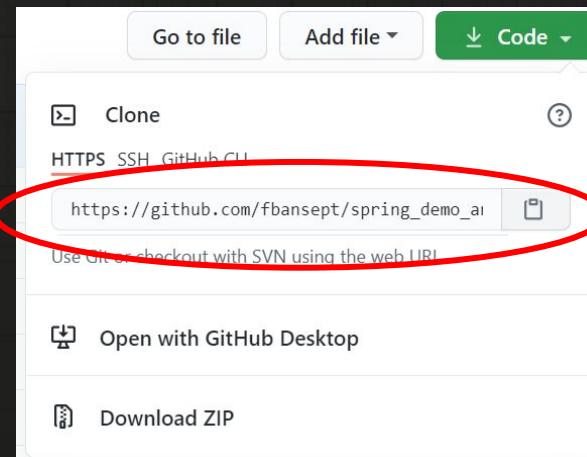
```
sudo yum install git
```

Pour Fedora :

```
sudo dnf install yum
```

Vous pouvez alors récupérer le code de l'application via la commande git clone, et l'url https de votre repository sur github

```
sudo git clone https://chemin/vers/le/repository
```



## >> Installation de docker

Pour Debian / Ubuntu

```
sudo apt update  
sudo apt install docker.io  
sudo systemctl start docker
```

Pour Centos

```
sudo yum update  
sudo yum install docker-ce docker-ce-cli containerd.io  
sudo systemctl start docker
```

Pour Fedora

```
sudo dnf update  
sudo dnf install docker-ce docker-ce-cli containerd.io  
sudo systemctl start docker
```

Dans tous les cas, vous pouvez tester le bon fonctionnement en lançant la commande suivante

```
sudo docker run hello-world
```



## >> Préparation du Dockerfile

Un fichier **Dockerfile** permet de créer une **image Docker**.

Cette **image** pourra par la suite créer un ou plusieurs **conteneurs Docker**.

Le fichier **Dockerfile** est situé généralement à la racine du repository où l'on versionne le code.

Il permet de définir l'environnement nécessaire à l'exécution de l'application.

Il permet également à tous les développeurs qui travaillent sur ce projet de bénéficier du même environnement que l'environnement de production.

Notre fichier **Dockerfile** utilise une **image officiel** de **tomcat**, elle copiera le **fichier war** de l'application, et lancera **Tomcat**. Celui-ci étant par défaut en mode d'**auto déploiement** des **fichiers war**, notre application sera lancée également .

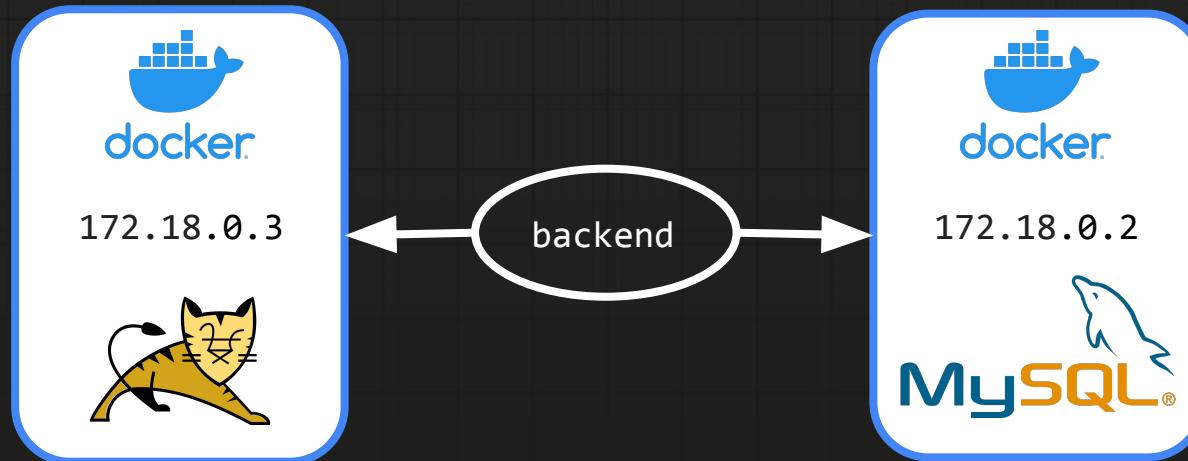
```
FROM tomcat:8.5
COPY target/demo-0.0.1-SNAPSHOT.war /usr/local/tomcat/webapps/ROOT.war
ENTRYPOINT ["/bin/bash", "/usr/local/tomcat/bin/catalina.sh", "run"]
```



## >> Préparation des réseau virtuel permettant la communication entre les conteneurs

La première étape consiste à créer un réseau docker. Les 2 conteneurs que nous allons créer pourront alors communiquer entre eux. On crée un réseau qui se nommera backend et utilisera des IP commençant par 172.18.

```
sudo docker network create --subnet=172.18.0.0/16 backend
```



## >> Préparation du volume pour stocker la base de données

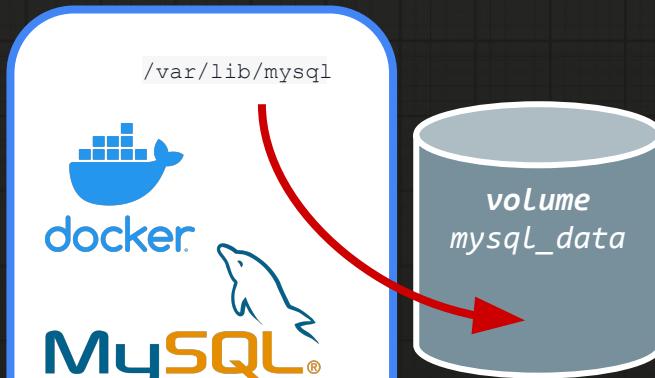
Nous allons ensuite créer un volume afin de conserver les données de la base de données, même si le conteneur est détruit.

```
sudo docker volume create mysql_data
```

Notes : Les volumes sont gérés par Docker et permettent de séparer les données des conteneurs pour les conserver indépendamment du cycle de vie des conteneurs.  
Cette fonctionnalité permet également :

- **Le Partage des données entre conteneurs** : Les volumes peuvent être montés sur plusieurs conteneurs en même temps, ce qui permet de partager des données entre différents conteneurs.
- **La migration des données** : Les volumes facilitent la migration des données entre les hôtes ou les conteneurs en fournissant une couche d'abstraction pour les données stockées.
- **La sauvegarde et restauration** : Les volumes facilitent également les sauvegardes et les restaurations des données, car ils peuvent être facilement sauvegardés, archivés et restaurés indépendamment des conteneurs.

serveur physique



Un volume est un dossier persistant entre le conteneur et le serveur (le volume n'est pas détruit lors de la suppression du conteneur)

## >> Préparation du conteneur MySQL

Nous allons ensuite créer un conteneur nommé `mysql`, basé sur une image de `Mysql 5`, ayant le mot de passe `root`, relié au réseau `backend` précédemment créé, ayant l'ip `172.18.0.2`, et dont le dossier `/var/lib/mysql` sera lié au volume `mysql_data`

```
sudo docker run --name mysql --net backend -v mysql_data:/var/lib/mysql --ip 172.18.0.2 -e MYSQL_ROOT_PASSWORD=root -d mysql:5
```

Puis nous allons copier le fichier `fichier_dump.sql` dans le conteneur `mysql`

```
sudo docker cp ~/dossier_projet/fichier_dump.sql mysql:fichier_dump.sql
```

Enfin, on lance un processus bash dans le conteneur, afin de lancer le script `fichier_dump.sql` qui créera la base de donnée

```
sudo docker exec -it mysql bash  
mysql -u root -p  
>> source fichier_dump.sql;  
>> exit
```

Si votre dump ne contient pas les instructions de création de base de donnée exécutez les instruction suivantes au préalable:

```
create database nom_base_de_donnee;  
use nom_base_de_donnee;
```

## >> Préparation du conteneur Phpmyadmin

Nous allons ensuite créer un conteneur nommé `phpmyadmin`, basé sur l'image `phpmyadmin/phpmyadmin` , relié au réseau `backend` précédemment créé, ayant l'ip `172.18.0.3`, permettant de gérer la base de donnée situé sur le conteneur ayant l'ip `172.18.0.2`, et disponible à l'adresse `ip_du_serveur:8888`

```
sudo docker run --name phpmyadmin --net backend --ip 172.18.0.3 -e PMA_HOST=172.18.0.2 -p 8888:80 -d phpmyadmin/phpmyadmin
```



## >> Préparation du volume pour stocker les documents uploadés

(uniquement si l'application Spring prévoit cette fonctionnalité)

Nous allons ensuite créer un volume afin de conserver les fichiers uploadés par un utilisateur, même si le conteneur est détruit.

```
sudo docker volume create uploaded_files
```

FichierService.java

```
@Service  
public class FichierService {  
  
    @Value("${dossier.upload}")  
    private String dossierUpload;
```

pom.xml

```
<profile>  
    <id>prod</id>  
    <properties>  
        <dossier_upload> /uploads</dossier_upload>  
    </properties>  
</profile>
```

FichierService.java

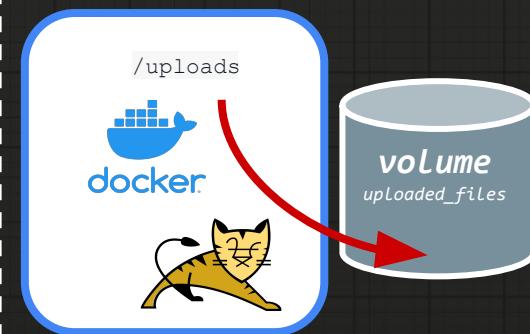
```
Path destination = Paths.get(dossierUpload + "/" + nomFichier);  
  
Files.copy(fichier.getInputStream(), destination,  
StandardCopyOption.REPLACE_EXISTING);
```

application.properties

```
dossier.upload=@dossier_upload@
```

Le chemin vers lequel on liera notre volume sera le chemin dans lequel nous enregistrons les fichiers dans l'application et dépendra de la façon dont vous le renseignez (*hard coded, dans un profil, dans le fichier pom.xml, via une variable d'environnement ...*)

serveur physique



# >> Mise en production

Récupérez la dernière version du code

```
git pull
```

Générez le .war via la commande mvnw (maven) incluse de base par **Spring initializer**

```
bash mvnw package -P prod -e
```

Créez une nouvelle image basée sur le Dockerfile du repository

```
docker build --no-cache -t image-spring-demo .
```

Supprimez l'ancien conteneur

```
docker rm -f conteneur-spring-demo
```

Lancez un nouveau conteneur sur le port 8080, basé sur la nouvelle image de votre application

```
docker run -d --net backend --ip 172.18.0.4 --name=conteneur-spring-demo -p 8080:8080 -v uploaded_files:/uploads image-spring-demo
```

Optionnellement vous pouvez consulter les logs de l'application (pour débogage)

```
docker logs conteneur-spring-demo
```

l'application est disponible via l'adresse : ip\_du\_serveur:8080 (ex : <http://123.456.789.123:8080>)

Attention si vous avez installé le certificat SSL alors le protocole est "https"

*En cas d'erreur 404 : le nom du war est peut être différent ou vous n'avez peut être pas de route GetMapping("") (dans ce cas testez une route existante)*

*En cas d'erreur 401/403 : n'oubliez pas que certaines de vos routes sont peut être réservées aux utilisateurs connectés / administrateurs*

*En cas d'erreur 500 : vérifiez les logs via la commande : sudo docker Logs mon-application*

*En cas d'erreur 400 : vérifiez que vous envoyez le bon type de donnée (du JSON par exemple)*

*Vérifier aussi que ce ne sont tout simplement pas des erreurs que vous avez géré dans vos contrôleurs*

Uniquement si votre application prévoit l'upload de fichier



# >> Créer un script de déploiement

Afin de ne pas avoir à exécuter chacune des instructions, il est possible de toutes les exécuter dans un même script, créez un script *deploy.sh*

```
sudo nano deploy.sh
```

```
#!/bin/bash

# Mettre à jour le code source
git pull

# Construire le projet avec Maven
bash mvnw package -P prod -e

# Construire l'image Docker
docker build --no-cache -t image-spring-demo .

# Arreter le conteneur existant
docker stop conteneur-spring-demo

# Supprimer le conteneur existant
docker rm -f conteneur-spring-demo

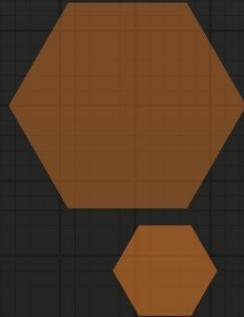
# Lancer un nouveau conteneur
docker run -d --net backend --ip 172.18.0.4 --name=conteneur-spring-demo -p 8080:8080 -v uploaded_files:/uploads image-spring-demo
```

Puis donner les droit d'exécution à ce script :

```
sudo chmod +x deploy.sh
```

Il peut être désormais appelé via la commande : **sh deploy.sh**





# Déployer l'application par l'interface graphique de TOMCAT *(méthode alternative à docker)*

*Chapitre : Mise en production*

## &gt;&gt; Utiliser l'interface pour déployer le WAR

Retournez sur l'interface installée précédemment afin de l'utiliser pour uploader puis déployer l'application

**Deployer**

Emplacement du répertoire ou fichier WAR de déploiement sur le serveur

Chemin de context (requis) :

Version (pour les déploiements en parallèle) :

URL du fichier XML de configuration :

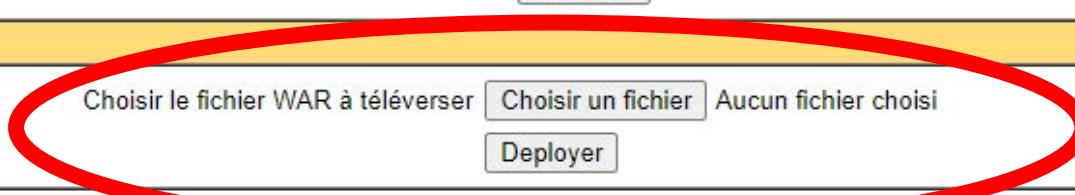
URL vers WAR ou répertoire :

**Deployer**

**Fichier WAR à déployer**

Choisir le fichier WAR à téléverser  Aucun fichier choisi

**Deployer**



## >> Utiliser l'interface pour déployer le WAR

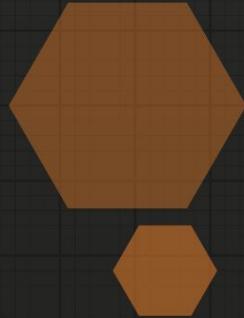
Cliquez sur le lien de l'application nouvellement installée afin de vérifier si l'installation s'est bien passée.

Dans le cas, vous devriez voir apparaître une page 404 (attention également à ce que la route vide existe bien sur votre API sinon il est logique d'avoir une page 404, dans ce cas testez une route existante de votre API disponible par une méthode GET par le navigateur, ou une autre méthode par POSTMAN)

Attention également aux erreurs 403 : elles sont également normales si vous n'êtes pas authentifiés.

Si vous ne comprenez pas votre erreur, il va falloir vérifier les logs de votre serveur (slide suivante)

Applications						
Ch... <a href="#">/demo-0.0.1-SNAPSHOT</a>	Version	Nom d'affichage	Fonctionnelle	Sessions	Commandes	
/demo-0.0.1-SNAPSHOT	Aucun spécifié		true	0	Démarrer Arrêter Recharger Retirer Expirer les sessions inactives depuis ≥ 30 minutes	
/host-manager	Aucun spécifié	Tomcat Host Manager Application	true	0	Démarrer Arrêter Recharger Retirer Expirer les sessions inactives depuis ≥ 30 minutes	
/manager	Aucun spécifié	Tomcat Manager Application	true	1	Démarrer Arrêter Recharger Retirer Expirer les sessions inactives depuis ≥ 30 minutes	

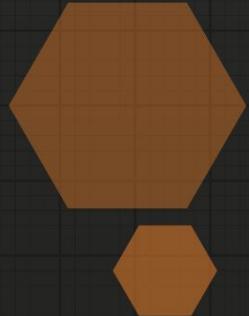


# Créer une architecture complète avec **docker compose**

*Chapitre : Mise en production*

>> TODO





# Les logs

*Chapitre : Mise en production*

## >> Inspecter les logs

Si votre serveur répond avec une erreur 404 ou 500, il se peut qu'il se soit mal installé. Afin de pouvoir comprendre le problème, il est nécessaire de pouvoir vérifier les messages qui apparaissent en tant normal dans la console. A l'exception que ces messages sont stockés dans des fichiers.

Sur un système linux, ceux-ci sont disponibles dans le dossier : var/lib/tomcat9/logs

Il seront tous archivés à l'exception des logs des dernières 24h, le fichier qui vous intéresse doit commencer par catalina, puis contenir la date d'aujourd'hui.

Note : Il y a de forte chance pour que l'erreur qu'il contienne concerne un mauvais identifiant / mot de passe afin d'accéder à la base de données en production.



# >> Créer ses propres logs

Dans le fichier application.properties ajouter les lignes suivantes :

```
logging.file.path=logs/  
logging.file.name=logs/application.log
```

Avec cette configuration, un fichier **/var/lib/tomcat9/logs/application.log** sera créé et contiendra les logs que nous allons générés.

Afin d'y ajouter un message on utilisera la propriété logger

```
logger.warn("Un message");
```

La propriété logger est disponible par héritage sur certains des composants Spring.

Dans le cas contraire, cette propriété peut être récupérée de la manière suivante :

```
Logger logger = LogManager.getLogger(NomDeLaClasseActuelle.class.getName());
```



# >> Afficher les erreurs 500

Par défaut les exceptions ne génèrent aucun log et ne retourne qu'une erreur 500.

Ces erreurs sont particulièrement compliquées à débugger du à l'absence de message.

On peut modifier notre intercepteur d'exception que nous avions déjà utilisé pour les erreurs de validation et les contraintes SQL.

*src/main/java/..../IntercepteurExceptionGlobal.java*

```
@ControllerAdvice
public class IntercepteurExceptionGlobal {

    //ici les methode précédentes

    @ExceptionHandler(value = { Exception.class })
    protected ResponseEntity<Object> handleConflict(
        RuntimeException ex, WebRequest request) {

        StringWriter sw = new StringWriter();
        PrintWriter pw = new PrintWriter(sw);
        ex.printStackTrace(pw);

        String bodyOfResponse = sw.toString();

        logger.warn("Unexpected Exception logger", ex);

        return handleExceptionInternal(ex, bodyOfResponse,
            new HttpHeaders(),
            HttpStatus.INTERNAL_SERVER_ERROR, request);
    }
}
```



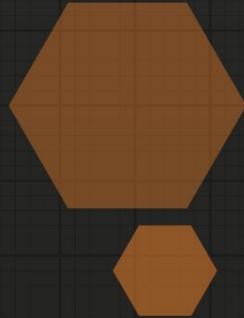
## >> Désactiver l'affichage des erreurs en production

TODO

spring.profiles.active=dev

```
@Profile("dev") // Active uniquement pour le profil "dev" @ControllerAdvice public class  
IntercepteurExceptionGlobal {
```





# Créer une stack ELK

*(ElasticSearch Logstash Kibana)*

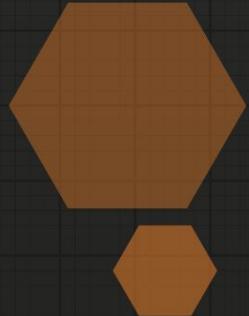
*Chapitre : Mise en production*

## >> STACK ELK

TODO

[https://www.youtube.com/watch?v=U4fD3zOtRKY&ab\\_channel=SimplifyingTech](https://www.youtube.com/watch?v=U4fD3zOtRKY&ab_channel=SimplifyingTech)





## HTTPS (SSL)

*En Local (tomcat embeded)*

*Chapitre : Mise en production*

## >> SSL Local

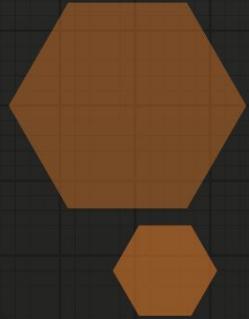
Génération certificat

```
sudo keytool -genkeypair -alias spring -keyalg RSA -keysize 2048 -storetype PKCS12 -keystore spring.p12  
-validity 3650
```

Fichier spring.p12 a placer dans le dossier src/ressources de

```
server.ssl.key-store-type =PKCS12  
server.ssl.key-store =classpath:spring.p12  
server.ssl.key-store-password =spring  
server.ssl.key-alias =spring  
server.ssl.enabled=true
```





# HTTPS (SSL)

*En production (tomcat externe)*

*Chapitre : Mise en production*

## >> SSL production

```
sudo mkdir /var/lib/tomcat9/conf/keystore  
cd /var/lib/tomcat9/conf/keystore  
sudo keytool -genkeypair -alias tomcat -keyalg RSA -keysize 2048 -storetype PKCS12 -validity 3650  
-keystore tomcat.p12 -storepass tomcat  
sudo keytool -keystore tomcat.p12 -storetype pkcs12 -exportcert -file tomcat.crt -rfc -alias tomcat  
sudo keytool -import -alias broker -keystore tomcat.ts -file tomcat.crt
```



# >> SSL production

Dans le fichier /var/lib/tomcat9/conf/server.xml , ajouter le noeud suivant dans le noeud <Service>

```
<Connector port="8443"
    protocol="org.apache.coyote.http11.Http11NioProtocol"
    maxThreads="150"
    SSLEnabled="true">
    <SSLHostConfig>
        <Certificate
            truststoreFile="conf/keystore/tomcat.ts"
            truststorePassword="tomcat"
            truststoreType="JKS"
            certificateKeyAlias="tomcat"
            certificateKeystoreFile="conf/keystore/tomcat.p12"
            certificateKeystorePassword="tomcat"
            certificateKeystoreType="PKCS12" />
    </SSLHostConfig>
</Connector>
```



## >> SSL production

Redémarrez le serveur tomcat

```
sudo systemctl restart tomcat9.service
```

Les applications déployées sont accessibles via **https://votre\_adresse:8443/nom\_application**

Un message d'avertissement sera alors visible puisque le certificat est auto signé, mais vous pouvez ignorer l'avertissement

(en cliquant sur “paramètre avancés” en bas de la page sur chrome)



# Ajouter un reverse proxy

Dans ce chapitre :

- Créer un load balancer Nginx
- Limiter les requêtes sur l'API (anti brute force)

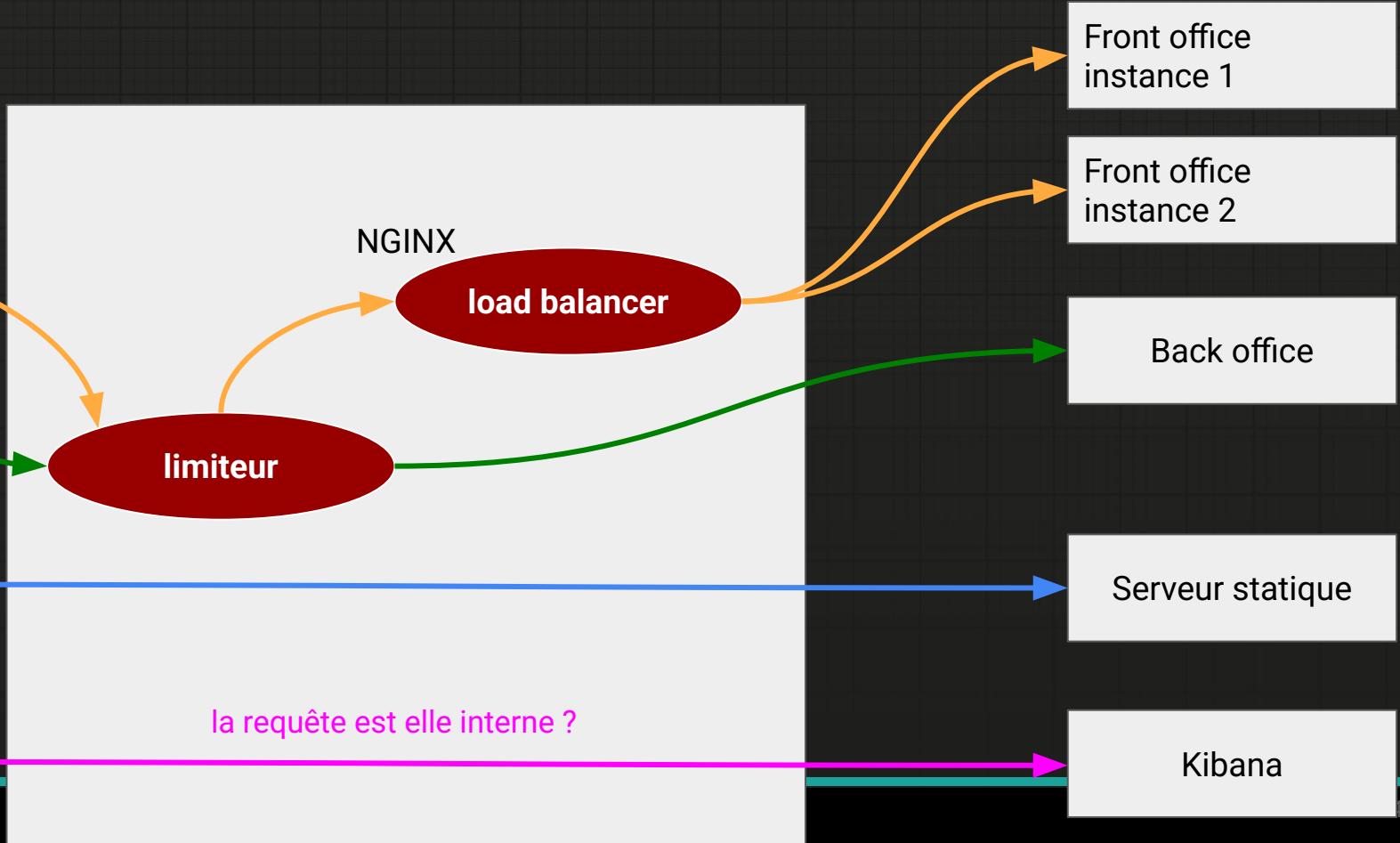
# >> SSL production

api front

api back

ressource statique

UI kibana



# Base de données NoSQL

Dans ce chapitre :

- MongoDB
- Firebase

>> TODO

<https://www.mongodb.com/resources/products/compatibilities/spring-boot>



# Créer un repository maven d'entreprise : Nexus

Dans ce chapitre :

>> TODO

# Améliorer la sécurité

Dans ce chapitre :

## >> TODO

Brut force

problematique de ReCecaptcha et des fermes à Bot (alternative : Envoie email de confirmation, popup App, problématique SMS, )

problème confirmation d'email tant que la personne n'a pas de compte ou si le compte est anonyme ou l'application account less

Double authentification (2FA)



# Techniques de déploiement avancées

Dans ce chapitre :

>> TODO

[https://www.youtube.com/watch?v=AWVTKBUnolg&t=98s&ab\\_channel\[ByteByteGo\]](https://www.youtube.com/watch?v=AWVTKBUnolg&t=98s&ab_channel[ByteByteGo])



# Spring web : architecture template

Dans ce chapitre :

- Créer des page HTML via Spring
- Thymleaf

## >> SPRING Web et le système de template

Spring permet de gérer une multitude de système de template. Même si il est possible d'utiliser plusieurs systèmes de template, il est recommandé d'en utiliser qu'un seul (l'intérêt étant limité, et la configuration du projet beaucoup plus lourde).

JSP : le système historique de template de JAVA issu de JAKARTA EE (JAVA EE)

Thymeleaf : un système de template plus complet que JSP

Les ressources statiques : ici pas de système de template. Cette option est le plus souvent opter dans le cas d'une application front end (angular, polymer, node, react, view ...)



## >> Configuration de SPRING

Spring se configure via le fichier application.properties situé dans le dossier resources

Les propriétés existent déjà est sont disponible selon les dépendances installées  
(Spring security, Spring web ...)

Ici on indique que les vues renvoyées par le contrôleur se finissent pas .html et sont à la racine du dossier static

```
spring.mvc.view.prefix = /  
spring.mvc.view.suffix = .html
```

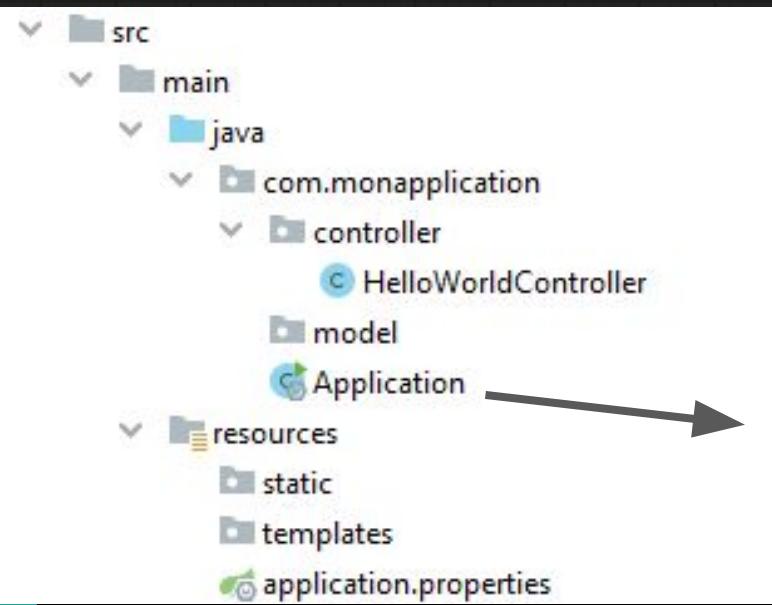


## >> Principe de SPRING

Spring permet grâce à des **annotations** de faciliter la configuration d'un projet.

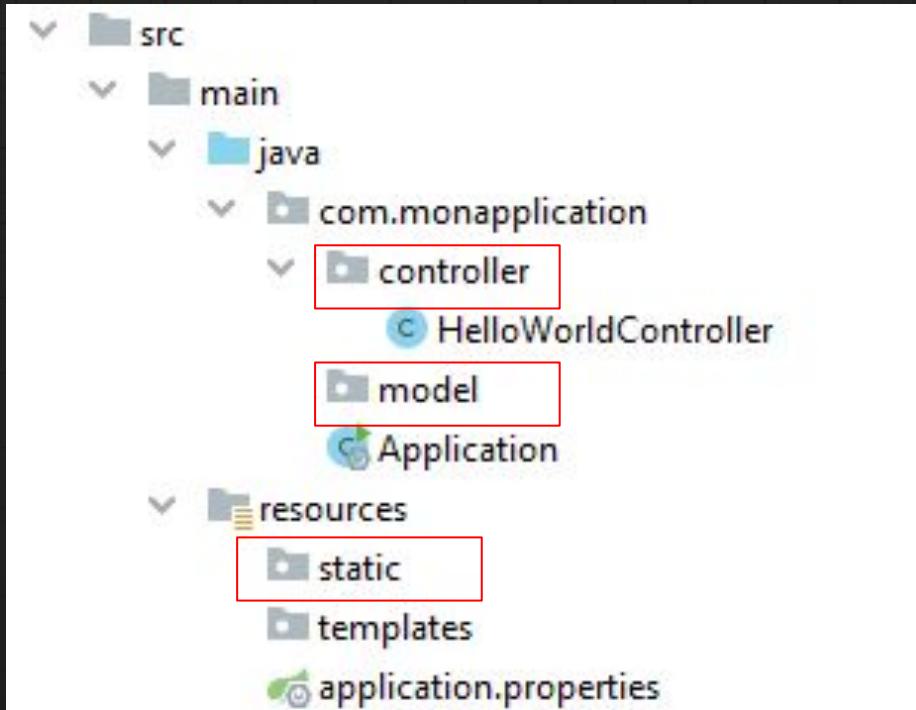
L'annotation principal de Spring boot est **@SpringBootApplication**

La classe qui reçoit cette annotation tentera d'évaluer toutes les annotations contenu dans le package dans lequel elle se trouve, ainsi que tous les packages situé en dessous de ce package



```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

## >> MVC (Model View Controller) Modèle Vue Contrôleur



Afin de respecter le modèle MVC, on créera un package controller, et un package model.

Le dossier static sera notre dossier contenant les vues

## >> Le controller de vue

Grâce à l'annotation `@Controller` on déclare notre classe en tant que contrôleur.

Grâce à l'annotation `@GetMapping` on déclare la méthode `helloPage()` comme devant se déclencher dès qu'une requête GET est effectuée avec pour url "/hello" (ex: localhost:8080/hello)

```
@CrossOrigin  
@Controller  
public class HelloWorldController {  
  
    @GetMapping({ "/hello" })  
    public String helloPage()  
        return "hello";  
}
```

Lorsque l'on a configuré le fichier application.properties, on a indiquer à SPRING que les vues se trouvait à la racine du dossier static, et se finissait par .html

Le contrôleur renverra donc le fichier /hello.html situé dans le dossier static

