



NOVAO[®]
LEARNING

SPRING-BOOT

Introduction à Spring Boot

QU'EST-CE QUE SPRING BOOT ?

Framework open-
source JAVA

2002: Spring
Framework

2014: Spring Boot
(extension
de Spring)

Création
d'applications de
production
(notamment WEB)

POURQUOI UTILISER SPRING BOOT ?

Application WEB

Application BATCH

Monolithique

WebService

Micro-services

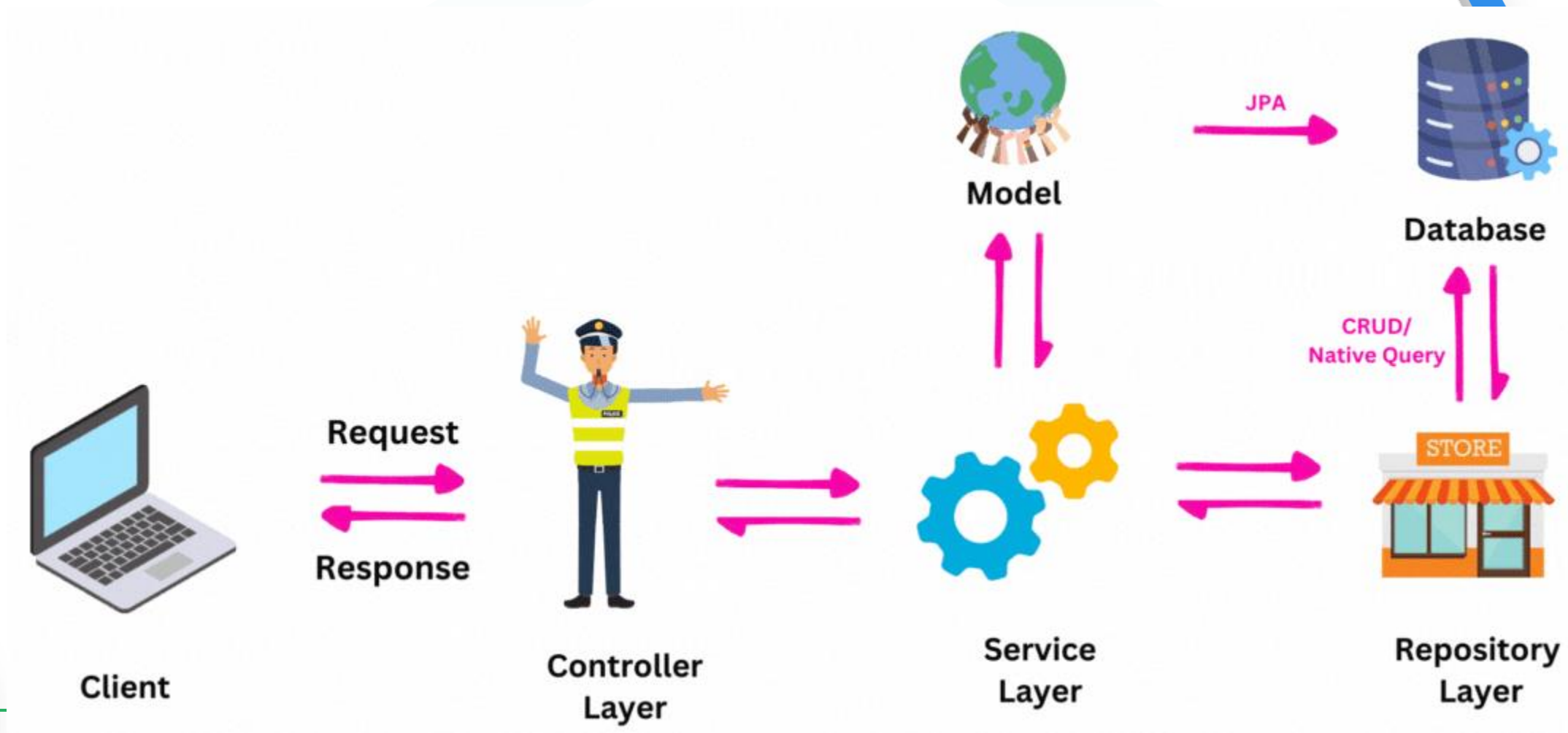
AVANTAGE ET FONCTIONNALITÉS PRINCIPALES

- Configuration automatique:
 - Convention plutôt que Configuration
 - Applications initialisée avec dépendances prédéfinies
 - Auto-configuration des packages Spring Framework et tiers
 - Reconfiguration possible
- Starters:
 - Ensembles de dépendances nécessaires pour démarrer un projet
 - Web, donnée, sécurité, etc.
- Inversion de contrôle (IoC):
 - Spring instancie les Objets
 - Injections des dépendances
- Autres:
 - Serveur intégré
 - Architecture modulaire
 - Monitoring
 - Facilité de tests

Architecture



SCHÉMA



CLIENT



Client

Interface utilisateur (UI)

Peut-être intégré dans le projet Spring:

- Html, thymeleaf, JSP
- React, Angular, Vue

Peut-être sur un autre serveur:

- Plus courant
- WebService

COUCHE CONTROLLER



**Controller
Layer**

Point d'entrée de l'application

Réception et traitement des requêtes HTTP

Déclenchement des actions appropriées

Traitement et envois de la réponse appropriée

COUCHE SERVICE



**Service
Layer**

Contient la logique métier

- Opérations
- Règles

Réutilisation

Interactions avec couche de persistances

Transactions et Validations

Appel à d'autres services

COUCHE REPOSITORY / PERSISTANCE



**Repository
Layer**

Gestion de persistance de donnée

Interaction avec Databases

- SQL (postgres, MySql, etc.)
- NoSql (Mongo)

Interface de définition d'opérations CRUD vers la BDD

Va convertir de:

- Objet Java → SQL Data
- SQL Data → Objet Java

COUCHE MODEL / ENTITÉE / DOMAIN



Model

Données métiers (ex: Produit, Utilisateur, Commande, etc.)

POJO (Plain Old Java Object):

- Attributs
- Constructeurs
- Getter
- Setter
- ToString
- Etc.

Correspond aux différentes tables de database avec leur relations

Vulgarisation:

- Model: Dictionnaires des données de JAVA ↔ DATABASE
- Repository: Traducteur usant de ce dictionnaire

DATABASE



Database

Système de stockage des données en SQL ou NoSQL

Serveur à part

Configuration automatisé par Spring

Premiers pas

ENVIRONNEMENT



Avoir une JDK
installé sur sa
machine



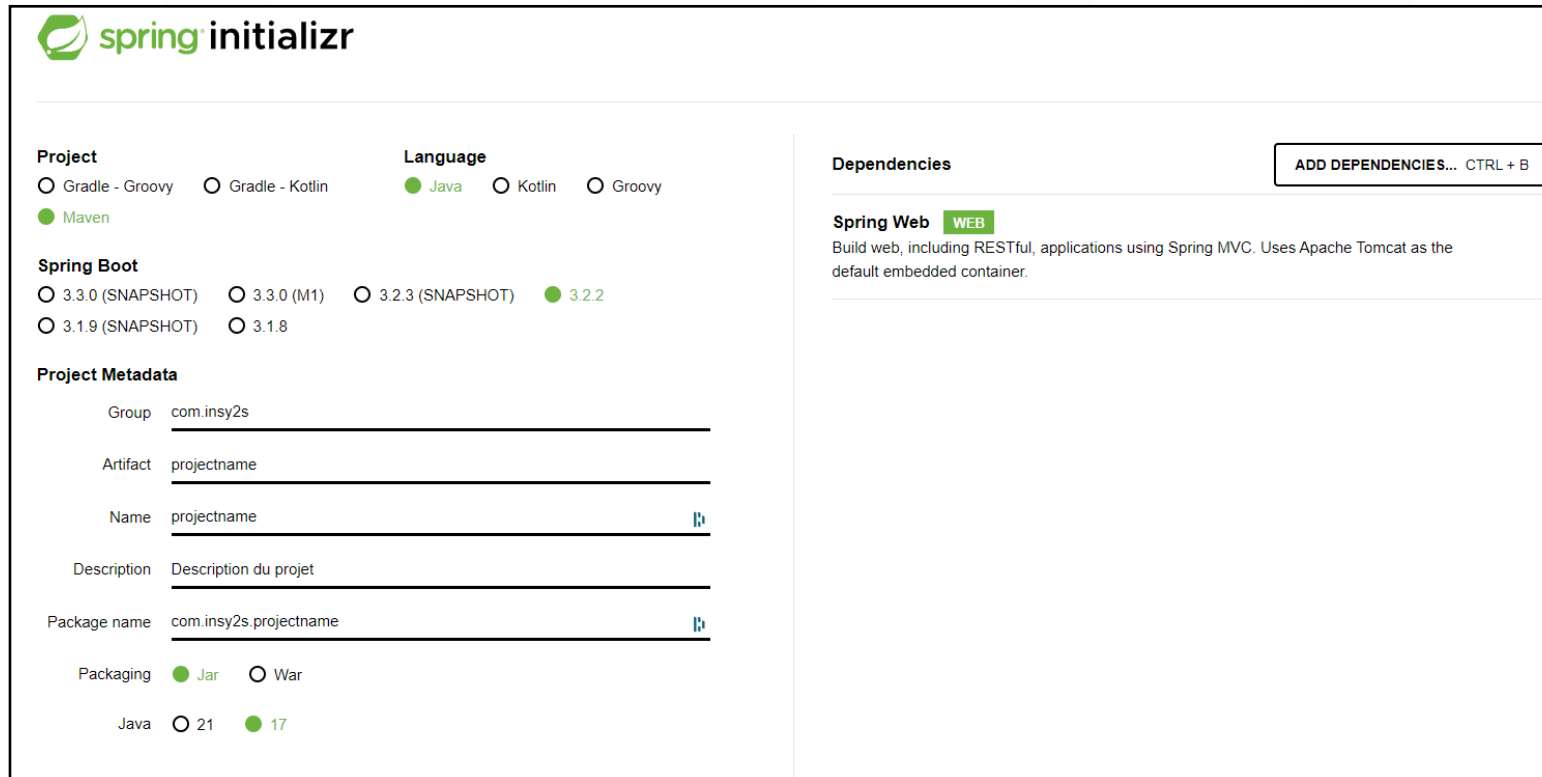
Avoir un IDE
installé sur sa
machine

```
openjdk version "17.0.8" 2023-07-18  
OpenJDK Runtime Environment GraalVM CE 22.3.3 (build 17.0.8+7-jvmci-22.3-b22)  
OpenJDK 64-Bit Server VM GraalVM CE 22.3.3 (build 17.0.8+7-jvmci-22.3-b22, mixed mode, sharing)
```

CRÉATION D'UN PROJET SIMPLE (1)

<https://start.spring.io/>

Application officielle de Spring pour créer une base de projet



The screenshot shows the Spring Initializr web application interface. The header features the "spring initializr" logo. The main content area is divided into three sections: Project, Language, and Dependencies.

Project

- ☐ Gradle - Groovy
- ☐ Gradle - Kotlin
- ☒ Maven

Language

- ☒ Java
- ☐ Kotlin
- ☐ Groovy

Spring Boot

- ☐ 3.3.0 (SNAPSHOT)
- ☐ 3.3.0 (M1)
- ☐ 3.2.3 (SNAPSHOT)
- ☒ 3.2.2
- ☐ 3.1.9 (SNAPSHOT)
- ☐ 3.1.8

Project Metadata

Group:

Artifact:

Name:

Description:

Package name:

Packaging: ☒ Jar ☐ War

Java: ☐ 21 ☒ 17

Dependencies

Spring Web ☒ WEB

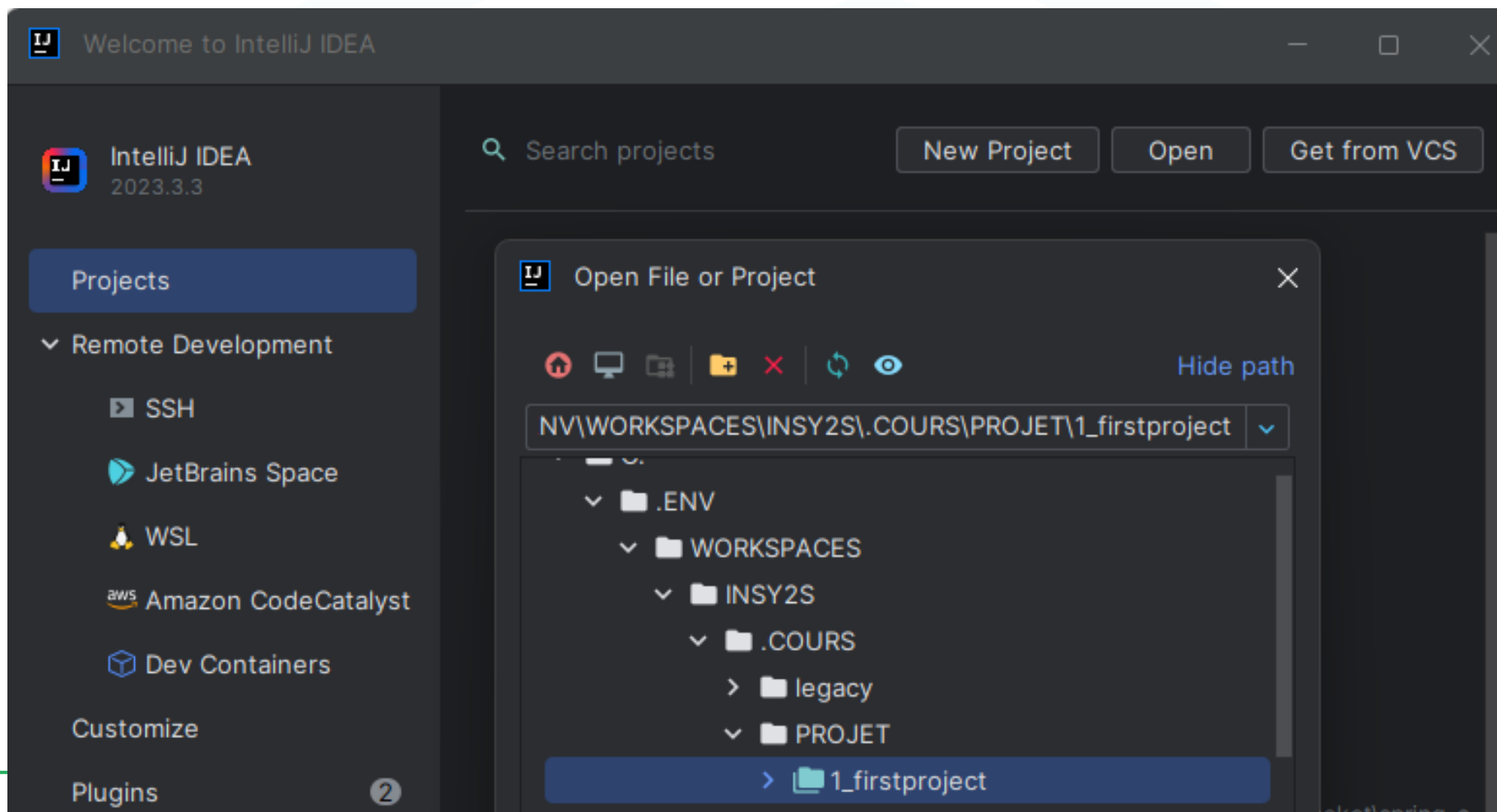
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

ADD DEPENDENCIES... CTRL + B

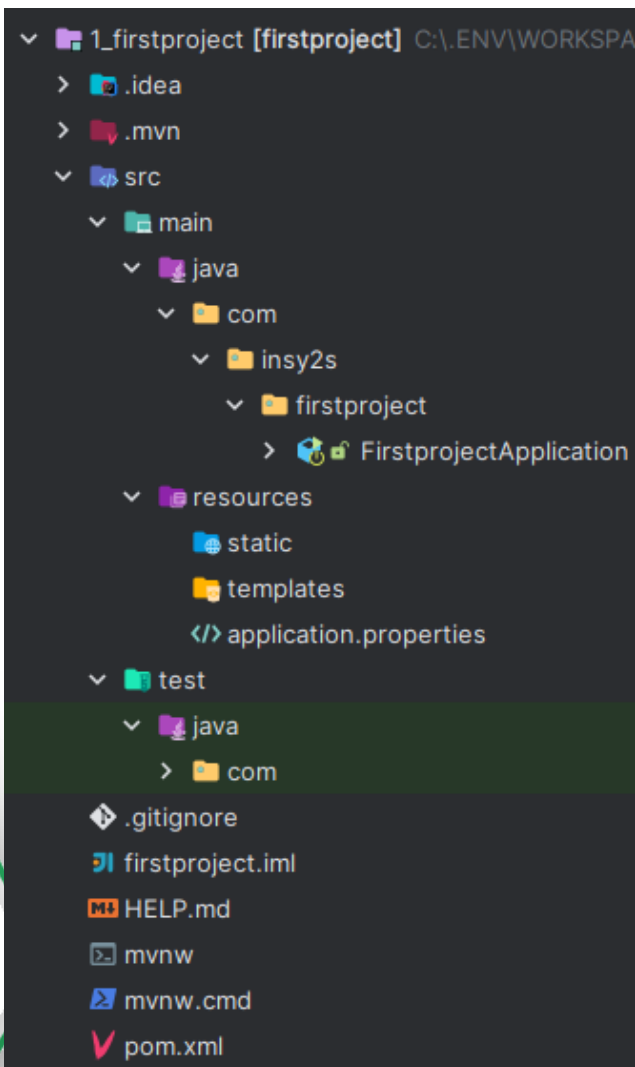
CRÉATION D'UN PROJET SIMPLE (2)

- **Project:**
 - Outils de gestion et automatisation (Maven ou Gradle)
- **Language:**
 - Langage utilisé dans l'application (Java, Kotlin ou Groovy)
- **Spring Boot:**
 - Version de spring boot (ex: 3.2.2).
- **Project Metadata:**
 - Description du projet:
 - *Group*: Identifiant unique de l'organisation (ex: com.insy2s)
 - *Artifact*: Identifiant unique du projet
 - *Name*: Nom du projet (souvent le même que l'artifact)
 - *Description*: description littéraire du projet
 - *Package Name*: Association du group et de l'artifact
 - *Packaging*: type d'emballage pour le projet
 - JAR: Java ARchive
 - WAR: Web Archive
- **Dependencies:**
 - Librairies ou framework tier à ajouter dans le projet
 - Ex: Spring WEB, Spring JPA, Spring Security, Lombok, Mapstruct, etc

CRÉATION D'UN PROJET SIMPLE (3)



STRUCTURE DE BASE



- **.idea**: dossier de configuration créer par IntelliJ IDEA (dossier local à ne pas toucher et pas envoyer sur GIT)
- **.mvn**: contient le script (maven wrapper) pour exécuter Maven
- **src/main/java**: code source java du projet, avec les différentes couches
- **src/main/ressources**: ressources de l'application, tel que:
 - configuration:
 - application.properties
 - application.yml
 - Template de vue
 - Html
 - Thymeleaf
 - Fichiers statiques:
 - CSS
 - JS
 - Images
- **src/test/java**: code source des tests unitaires de l'application
- **src/test/ressources**: ressources de l'application dans un environnement de tests
- **target**: généré par maven, contient les jar, ars, fichier de dépendances, classes des fichiers java, etc.
- **.gitignore**: contient une base de gitignore pour un projet spring-boot classique
- **pom.xml**: descripteur du projet Maven, avec metadata, dépendances à télécharger, plugins, propriétés, etc.

MAVEN ET POM.XML (1)

Maven est un outil de gestion de projet utilisé dans le développement logiciel JAVA, qui fournit un cadre complet pour la construction, le test et le déploiement d'application JAVA.

Aide à:

- **Gestion de dépendances:**
Simplifie la gestion de dépendances en permettant de déclarer celles voulu, Maven résolvant automatique celles-ci et les téléchargements sur la machine depuis un repo distant.
- **Cycle de vie:**
Série de script avec phase distinct tel que *compile, test, package, install, deploy*
- **Convention plutôt que configuration:**
Maven favorise cette approche qui suit des conventions prédéfinies pour la structure du projet et tâches de construction
- **Gestions de Plugins:**
Extensible via plugins pour des tâches spécifiques tel que compilation, testing, génération de rapport, exécution de packages, etc.
- **Gestion de profiles:**
 - Permet de créer différent type d'environnement de build d'application, tel que des profiles de développement ou de production

MAVEN ET POM.XML (2)

```
<?xml version="1.0" encoding="UTF-8"?>
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://
>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.2</version>
    <relativePath/>
  </parent>
  <groupId>com.insy2s</groupId>
  <artifactId>firstproject</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>firstproject</name>
  <description>
    Mon premier projet spring pour voir les bases
  </description>

  <properties>
    <java.version>17</java.version>
  </properties>

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

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

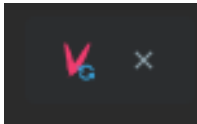
- Les metadata défini lors de la création du projet. Auquel on peut rajouter par exemple l'organisation et les contacts des responsables du projet.
- Les **properties** qui permettent de définir des versions de dépendances utilisé. Par défaut ne contient que la version de la JDK.
- La liste des **dépendencies** qui contiennent celles défini à la création et où on peut en rajouter selon les besoins du projet.
- Les **plugins** qui contiennent par défaut le plugin maven pour le bon fonctionnement de celui-ci dans un projet springboot.

MAVEN ET POM.XML (3)

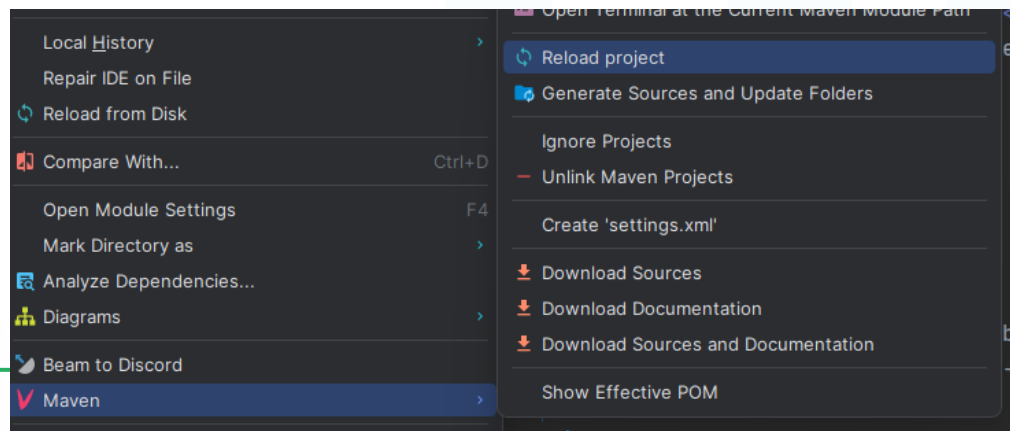
! IMPORTANT !

Quand on modifie ce fichier cela n'est pas pris en compte dans le projet tant qu'on ne **reload** pas maven.

- Soit en appuyant sur le bouton apparaissant en haut à droite d'IntelliJ lorsqu'on modifie le pom.xml et qu'il y a besoin d'un reload.



- Soit: *clic droit sur le projet → Maven → Reload Project*



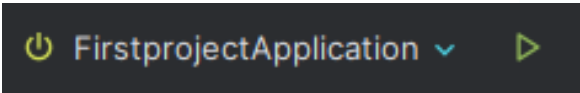
LANCER LE PROJET (1)

Plusieurs moyens pour run le projet:

- Allez dans: `src/main/java/com/insy2s/firstproject/FirstprojectApplication.java`

Puis run le projet avec l'icone: 

- Si disponible, run avec l'auto-configuration dans la navbar



- Dans un cmd à la racine du projet, faire:

```
mvnw spring-boot:run
```

```
package com.insy2s.firstproject;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class FirstprojectApplication {

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

}
```


EXERCICE : GÉNÉRER UN PROJET AVEC SPRING INITIALIZR

Allez sur <https://start.spring.io/> puis remplir les paramètres suivants :

- **Project:** Maven
- **Language:** Java
- **Spring Boot:** 3.2.3. (version choisie le 27/02/2024)
- **Project Metadata:**
 - Description du projet:
 - *Group:* com.insy2s
 - *Artifact et Name :* firstProject
 - *Package Name:* com.insy2s.firstProject
 - *Packaging:* JAR
 - Java : 17
- **Dependencies: Ajoutez les Outils suivant :**
 - Spring Web
 - Spring Data JPA
 - PostgreSQL Driver
 - Lombok

Générez le projet .zip
dézippez le puis l'ouvrir avec votre IDE

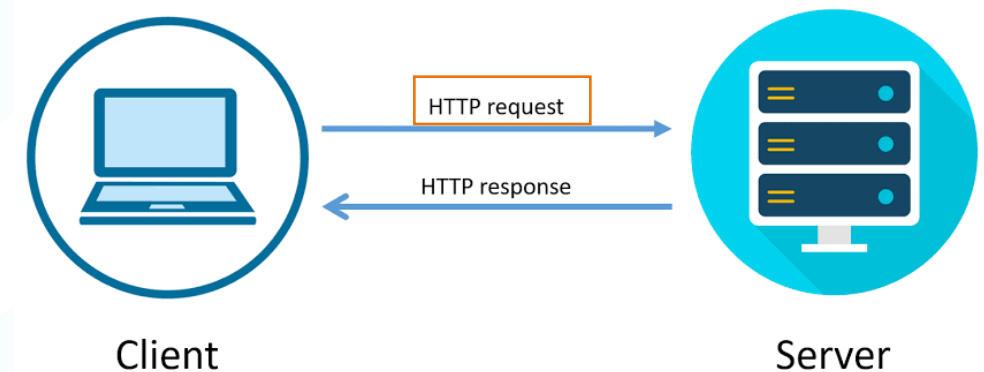
Bases de développement d'une application web

LES REQUÊTES HTTP (1)

Une requête HTTP (Hypertext Transfer Protocol) est un message envoyé par un client à un serveur pour demander une action.

Ceci est utilisé pour communiquer avec un serveur web pour soit:

- Demander des ressources:
 - Page web
 - Image
 - Fichier
 - Donnée
- Demander une action serveur:
 - Création d'une donnée en BDD
 - Modification d'une donnée de la BDD
 - Suppression d'une donnée de la BDD



LES REQUÊTES HTTP (2)

COMPOSITION D'UNE REQUÊTE HTTP

- **Ligne de requête**: contenant:
 - La méthode (GET, POST, PUT, DELETE)
 - L'URL de la ressource demandée
- **En-têtes (headers)**: Informations supplémentaires, tel que l'authentification, le type de contenu acceptables, les cookies, etc.
- **Corps (body)**: Optionnel, contient des données envoyées dans la requête, généralement au format JSON.

REQUEST

GET http://127.0.0.1:5500/styles/navigation.css **HTTP/1.1**

HTTP request line

HOST: 127.0.0.1:5500
 Accept: text/css,*/*;q=0.1
 Accept-Language: en-GB,en;q=0.5
 Accept-Encoding: gzip, deflate, br
 User-Agent: Mozilla/5.0 (Windows NT 10.0;
 Win64; x64; rv:102.0) Gecko/20100101 Firefox/102.0
 Connection: keep-alive
<CRLF>

HTTP headers



HTTP body
(empty)



Client



Server

HTTP request

HTTP response

LES REQUÊTES HTTP (3)

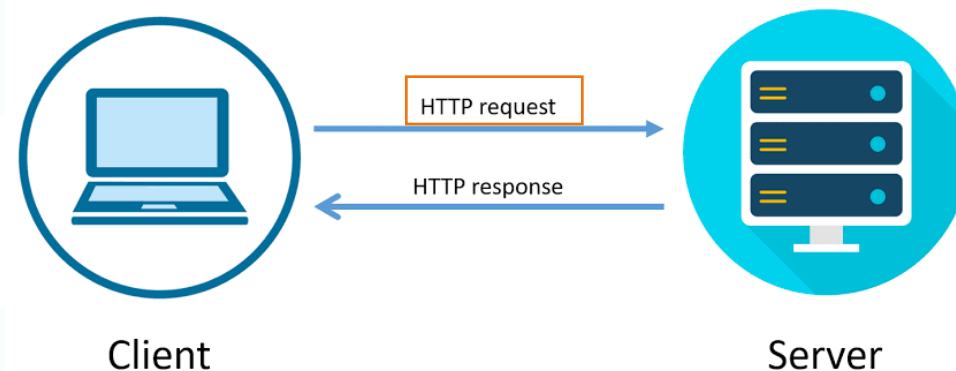
LES MÉTHODES HTTP

- **GET**: Utilisé pour récupérer une ressource
- **POST**: Ajouté une nouvelle donnée/ressource
- **PUT**: Modifié ou remplace une donnée/ressource existante
- **DELETE**: Supprime une donnée/ressource
- **PATCH**: Modifie partiellement une donnée/ressource

! Important !

Techniquement on peut tout faire avec un seul type de méthode. Rien n'empêche de faire une suppression avec un POST, ou une récupération avec un DELETE.

Il s'agit d'une convention, d'une bonne pratique à respecter.

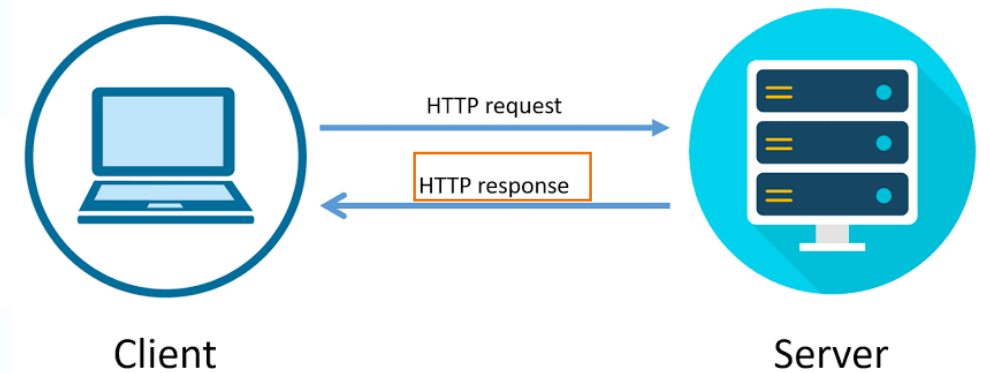


LES RÉPONSES HTTP (1)

Une réponse HTTP est un message envoyé par un serveur à un client en réponse à une requête HTTP de celui-ci.

Utilisé pour:

- Envoyer la ressource demandée par le client
- Informer le client de l'état de la requête:
 - Bon fonctionnement
 - Éventuelles erreurs



LES RÉPONSES HTTP (2)

COMPOSITION D'UNE RÉPONSE HTTP

- **Ligne de statut**: contient le code statut HTTP indiquant le résultat de la requête (200, 201, 400, 401, 404, etc.) Composé de nombres à 3 chiffres indiquent si la requête HTTP a réussi, rencontré un problème serveur ou client, ou nécessite une action supplémentaire (connexion).
- **En-têtes (headers)**: Comme dans la requête HTTP, les headers fournissent des informations supplémentaires sur la réponse, telles que le type de contenu, la date et heures de la réponse, les cookies, etc.
- **Corps (body)**: Optionnel, contient les données renvoyées par le serveur en réponse à la requête du client. Peut-être une donnée au format JSON, ou un fichier, ou tout autre type de données.

RESPONSE

```
HTTP/1.1 200 OK
```

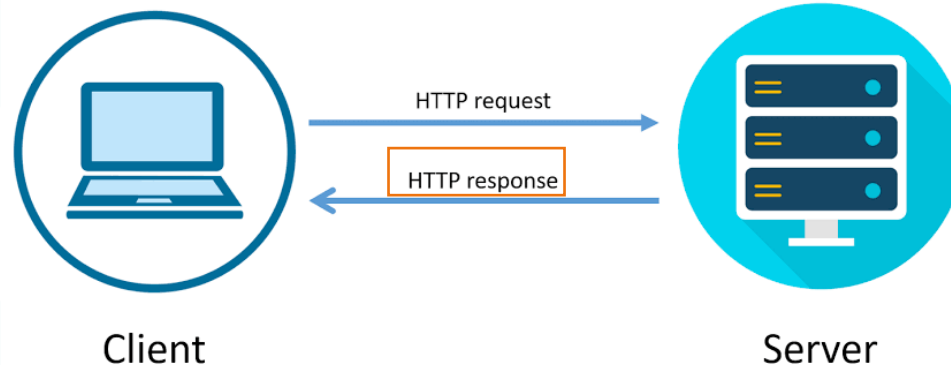
HTTP response
status line

```
Date: Wed, 06 Jul 2022 09:30:28 GMT  
Accept-Ranges: bytes  
Content-Length: 2005  
Content-Type: text/css; charset=UTF-8  
<CRLF>
```

HTTP response
headers

```
nav.navbar {  
  ...some style  
}
```

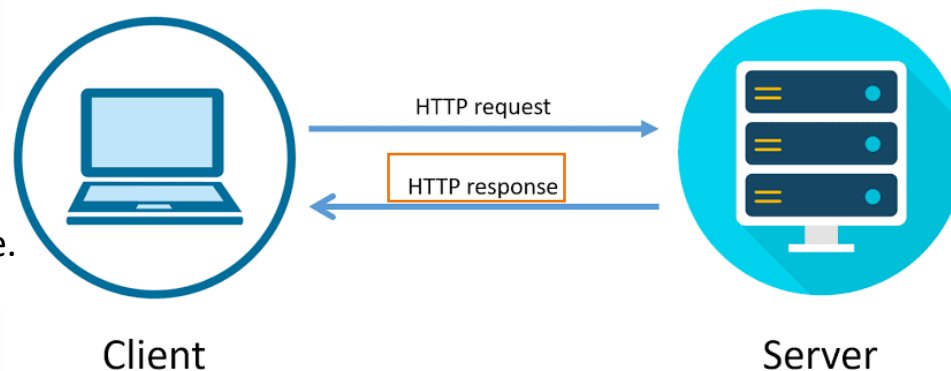
HTTP response
body



LES RÉPONSES HTTP (3)

CODES STATUTS DES RÉPONSES HTTP

- **1XX – Informations:** fourni des informations supplémentaires sur le traitement de la requête
 - 100: continue
 - 101: Switching Protocol
- **2XX – Succès:** indique que la requête a été reçue, comprise, acceptée et traitée avec succès.
 - 200: OK
 - 201: Created
- **3XX – Redirection:** indique que des actions supplémentaires doivent être prise pour compléter la requête.
 - 301: Moved Permanently
 - 304: Not Modified
- **4XX – Erreur client:** indique que la requête contient une erreur venant du client.
 - 400: Bad Request
 - 404: Not Found
- **5XX – Erreur du serveur:** indique que le serveur a rencontré un problème interne.
 - 500: Internal Server Error
 - 503: Service Unavailable



LES ANNOTATIONS

Déjà existant dans java (ex: `@Override`), les annotations dans Spring sont des marqueurs qui fournissent des métadonnées à Spring Framework sur la manière dont les composants doivent être gérés et configurés.

Elles sont largement utilisées pour configurer les différents composants de l'application, tel que:

`@RestController` : Marque une classe comme contrôleur REST (signifiant qu'elle retourne de la donnée)

`@Service` : Marque une classe comme un composant Service, qui encapsule la logique métier de l'application.

`@Repository` : Marque la classe comme un composant de repository, donc de l'accès aux données (BDD)

`@Entity` : Marque la classe comme une entité persistante, donc qui sera enregistré en BDD.

⚠ **Important** ⚠ Pratiquement chaque classe en Spring sera annotée. Donc première chose à se demander en créant une nouvelle classe est: *Quelle annotation dois-je mettre ?*

La couche Controller

INTRODUCTION



Controller
Layer

Rappel:

- Point d'entrée de l'application
- Réceptionne les requêtes clients
- Fait appel aux Services pour les traitements logiques/métiers
- Envoi les réponses au client

```
package com.insy2s.firstproject.controller;  
  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class HelloWorldController {  
  
}
```

Chaque fichier Controller devra être annoté soit:

@Controller: Classique dans une app Spring MVC traditionnelle. Qui généralement renvoi des vues (HTML)

@RestController: Accumulation de @Controller et @ResponseBody, signifiant qu'il s'agit d'un controller qui envoi de la donnée sous forme de JSON ou XML.

INTRODUCTION



Controller
Layer

En simplifié, un Controller n'est ni plus ni moins qu'une méthode JAVA avec une annotation qui spécifiera que celle-ci s'enclenchera dès qu'on appellera l'URL spécifié dessus, avec le type de méthode HTTP spécifiée.

`@GetMapping(« /hello »)` :

→ crée un contrôleur accessible via <http://localhost:8080/hello> en méthode GET.

```
package com.insy2s.firstproject.controller;

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

@RestController
public class HelloWorldController {

    @GetMapping("/hello")
    public String helloWorld() {
        return "Hello World";
    }

}
```

LES TYPES DE RETOUR



Controller
Layer

- **String**: pour envoyer une chaîne de caractère, ce qu'on évite souvent puisqu'on préfère un objet java plus descriptif.
- **Void**: Certaines fonctionnalités n'ont pas besoin de retour puisque le client demande juste un traitement de la part du serveur.
- **Object**: Permet d'envoyer un POJO qui sera traduit en JSON (ou XML, mais plus rare) que le client pour réceptionner et traiter (par exemple afficher les données d'un produit que le serveur a renvoyé).
- **ResponseEntity<?>**: Est un encapsuleur permettant de modifier certaines parties de la requête, tel que le code statut ou les headers. Le « ? » étant le type de retour comme défini au-dessus.

GET (1)



Controller
Layer

A savoir que votre navigateur fait par défaut des requêtes GET.
C'est-à-dire qu'on peut tester directement l'url: <http://localhost:8080/hello> sur un navigateur.

Ou on peut tester avec une application de test d'API comme Postman.



```
package com.insy2s.firstproject.controller;  
  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class HelloWorldController {  
  
    @GetMapping("/hello")  
    public String helloWorld() {  
        return "Hello World";  
    }  
  
}
```

GET (2): PATHVARIABLE (1)



Controller
Layer

Comme n'importe quelle méthode JAVA, les contrôleurs peuvent aussi avoir des paramètres. Ils seront automatiquement créés grâce à la requête selon d'où vient cette donnée (de l'URL, des options de l'URL ou du Body).

`@PathVariable` permet de récupérer des données directement dans le path de l'URL de la requête. Il s'agit d'une variable qui est **obligatoire** de récupérer de la requête de la part du client. Cela est notamment utilisé pour récupérer un ID pour spécifier quelle ressource spécifique on veut récupérer de la BDD.

Exemples:

/users/**1**

/products/**100**/commentaries

GET (2): PATHVARIABLE (2)



Controller
Layer

```
@GetMapping("/users/{id}")
public void getUserById(@PathVariable Long id) {
    System.out.println("request to get user with id: " + id);
}
```

- **{maVariable}** : Spécifie à votre URL qu'il y a une variable (nommé ici « maVariable ») dans son path. Elle peut être où on veut dans l'URL, au début, à la fin ou au milieu.
- **Type maVariable** : Dans les paramètres de la méthode on récupère cette même variable, avec le type dont on veut la caster. Le nom de la variable est le même que celle dans l'URL, il y a néanmoins tout de fois possibilité de renommé avec un peu de configuration.
- **@PathVariable** : Placé devant le paramètre, spécifiera à votre controller où il doit récupérer cette variable. Donc ici dans le path directement. On peut rajouter des parenthèses à cette annotation pour spécifier des options, tel que:
 - name**: Pour spécifier le nom de la variable dans l'URL, utile lorsqu'on veut un nom de variable différent en JAVA.
 - required**: Pour spécifier si on veut que la variable soit obligatoire ou non (par défaut est true). On laisse souvent required, on préférera un autre moyen pour avoir des variables non requises.
- Si dans l'URL on envoi une variable qui ne correspond pas au type défini dans le controller, Spring envoi automatiquement une erreur 400 (Bad Request) au client !

GET (3): REQUESTPARAM (1)



Controller
Layer



En plus du domain (adresse du serveur) et du path, l'URL contient des paramètres, aussi appelé Options, permettant d'envoyer également des variables aux controllers.

Généralement utilisé de manière optionnelle pour possiblement affiner la demande du client. Très utile pour avoir une requête générique avec filtre.

Exemple:

`/users` : récupère tout les utilisateurs

`/users?email=gmail` : récupère tous les utilisateurs ayant une adresse gmail

`/users?email=gmail&banned=true` : récupère tous les utilisateurs ayant une adresse gmail et étant banni

GET (3): REQUESTPARAM (2)



Controller
Layer

Etant donné qu'il s'agit de paramètres d'URL et non de variables de path, cela n'apparaît pas dans l'URL définie dans l'annotation.

Tout comme les PathVariable, on a besoin de définir dans les paramètres de la méthode le nom de la variable récupérée.

`@RequestParam`: Cette annotation permet de spécifier au controller que la variable provient des params de la requête. Elle aussi a des options en parenthèse, tel que `required` (aussi `true` par défaut, souvent changé en `false`).

```
@GetMapping("/users")
public void getAllUsers(@RequestParam(required = false) String email) {
    System.out.println("request to get all users");
    if(email != null) System.out.println("filtered by email domain: " + email);
}
```

GET (3): REQUESTPARAM (3)



Controller
Layer

L'annotation `@RequestParam` est optionnel, par défaut s'il n'y a pas d'annotations devant un paramètre de méthode du controller cela sera considéré comme un `RequestParam`. Le plus souvent on ajoute l'annotation quand on a besoin de spécifier des options à l'annotation.

Si on a plusieurs `RequestParam`, plutôt que des tous les mettre dans les paramètres de la méthode, on peut créer un objet les contenant tous pour simplifier la lecture et l'utilisation (ainsi que la réutilisabilité).

```
@GetMapping("/users")
public void getAllUsers(UserFilter userFilter) {
    System.out.println("request to get all users");
    if(userFilter.getBanned() != null) System.out.println("filtered by banned: " + userFilter.getBanned());
    if(userFilter.getEmail() != null) System.out.println("filtered by email domain: " + userFilter.getEmail());
}
```

GET (4)



Controller
Layer

Encore une fois, les `PathVariable` et `RequestParam` peuvent très bien être interchangeables et on peut les utiliser pour faire la même chose.

Il s'agit d'une **bonne pratique** d'avoir:

- `PathVariable` pour des données **requises**
- `RequestParam` pour des données **optionnelles** fait pour du **filtre**.

DELETE



Controller
Layer

Comme le nom l'indique, les requêtes du type DELETE sont utilisés pour spécifier la suppression d'une ressource. Généralement elle ne retourne rien, et auront un code statut 204 (No Content) si cela a fonctionné.

`@DeleteMapping` : Annotation à mettre au-dessus de la méthode pour spécifier au controller qu'il s'agit d'une requête de méthode DELETE.

Tout comme GET, on peut utiliser les `@PathVariable` et `@RequestParam` pour affiner la requête. Notamment pour cibler plus spécifiquement ce que le client veut supprimer.

En dehors de l'annotation, les requêtes DELETE sont en tout point identiques aux requêtes GET. Il s'agit encore une fois de bonnes pratiques d'utilisation.

```
@DeleteMapping("/users/{id}")  
public void deleteById(@PathVariable Long id) {  
    System.out.println("request to delete user with id: " + id);  
}
```

POST (1)



Controller
Layer

Les requêtes POST sont utilisées pour de la création de ressources. Par exemple un nouvel utilisateur ou un nouveau produit en BDD.

`@PostMapping` : On utilise cette annotation pour spécifier au controller qu'il s'agit d'une requête POST demandé par le client.

```
@PostMapping("/users")
public void create() {
    System.out.println("request to create user");
}
```

POST(2): REQUESTBODY



Controller
Layer

Il faut pouvoir récupérer la donnée à créer. On pourrait techniquement utiliser les PathVariable et RequestParam, mais pour une meilleure (bonne) pratique on reçoit les données via le body de la requête.

`@RequestBody` : On utilise cette annotation pour spécifier au controller que la requête contiendra une donnée dans le body.

On a juste besoin d'avoir en JAVA un objet correspondant à la donnée envoyée par le client. Avec les mêmes champs aux mêmes noms.

```
@PostMapping("/users")
public void create(@RequestBody UserData data) {
    System.out.println("request to create user : " + data);
}
```

```
{
  "firstName": "John",
  "lastName": "Doe",
  "email": "jdoe@gmail.com"
}
```

Objet Json envoyé par le client. En JAVA on doit avoir un POJO avec ces champs.

PUT

Le PUT, fait pour modifier ou remplacer une ressource, est très similaire au POST.

Par convention, on rajoute l'ID (ou autres champs uniques de la ressource) dans l'URL.



Controller
Layer

```
@PutMapping("/users/{id}")  
public void update(@PathVariable Long id, @RequestBody UserData data) {  
    System.out.println("request to update user with id: " + id + " with data: " + data);  
}
```

- On a l'ID dans le path de l'URL, récupéré par PathVariable, qui nous aidera à cibler quelle ressource est à modifier. Par exemple un produit en BDD. On peut utiliser un autre champ que l'ID du moment que ce champ permet de cibler précisément la donnée. Donc unique à celui-ci (exemple une adresse mail, un username, un UUID, un code référence, etc.)
- On récupère le body avec les données à modifier, et on modifiera toutes les données (qui peuvent être modifiée, on peut avoir des données non modifiables, comme une date de création).

PATCH



Controller
Layer

Le **PATCH** est fait pour faire de la modification **partielle** de ressource mais sans possibilité de remplacé.

***Exemple:** Je veux modifier le prix d'un article. Je ne veux pas modifier tout le reste, je veux juste dire au backend « pour l'article référence GZ-4589 met le prix à 10€. Voici la référence, voici le prix, je ne donne pas le reste ».*

Le **PATCH** est donc un **PUT** où on ne modifie que ce qu'on envoi, ce qui n'est pas envoyé (ce qui est null donc) ne devra pas être modifié. Le problème de ça est donc qu'on ne pourra pas modifier une donnée pour y mettre null.

```
@PatchMapping("/users/{id}")
public void patch(@PathVariable Long id, @RequestBody UserData data) {
    System.out.println("request to patch user with id: " + id );
    if (data.getFirstName() != null) {
        System.out.println("update first name to: " + data.getFirstName());
    }
    if (data.getLastName() != null) {
        System.out.println("update last name to: " + data.getLastName());
    }
    if (data.getEmail() != null) {
        System.out.println("update email to: " + data.getEmail());
    }
}
```

RESPONSEENTITY



Controller
Layer

Par défaut, tout *return* d'un controller enverra un statut code 200.
Et si la méthode n'est pas *void* enverra le *return* en body.

Si on veut modifier le code statut, ou modifier la réponse (par exemple ajouter des choses dans le header), on utilise le type `ResponseEntity<?>`. Le ? Étant le type du body.

⚠ **Important** ⚠ Ne laisser pas « ? » ! Mettez bien le type du return voulu !

```
public ResponseEntity<UserData> getById(@PathVariable Long id) {  
    System.out.println("request to get user with id: " + id);  
    /* get user data from database */  
    UserData userFromDB = new UserData();  
    return ResponseEntity.ok(userFromDB);  
}  
  
...  
  
return ResponseEntity.noContent().build();  
  
...  
  
return ResponseEntity.status(HttpStatus.CREATED).body(savedUser);
```

EXERCICE : CRÉER UN CONTROLLER (1)

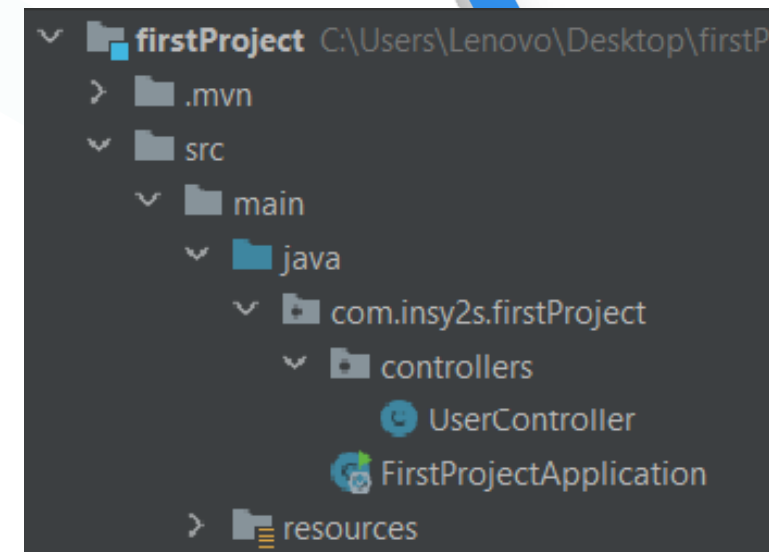
MISE EN PLACE

- Ouvrez le dossier : **firstProject/src/main/java**
- Dans le dossier **com.insy2s.firstProject** ajoutez un package **controllers**
- Dans le nouveau package **controllers**, ajoutez **UserController**
- Dans **UserController** remplissez avec le code suivant :

```
@RestController
@RequestMapping("/users")
public class UserController {

    // Endpoints REST

}
```



En utilisant le IntelliJ, les annotations peuvent être importées automatiquement

EXERCICE : CRÉER UN CONTROLLER (2)

ENDPOINTS GET

Nous allons ajouter tous les Endpoints ici

Rajouter les Endpoints suivant :

```
@GetMapping("/{id}")
public void getUserById(@PathVariable Long id){
    System.out.println("request to get user with id : " + id);
}

@GetMapping("/")
public void getAllUsers(){
    System.out.println("request to get all users");
}
```

```
@RestController
@RequestMapping("/users")
public class UserController {

    // Endpoints REST

}
```

EXERCICE : CRÉER UN CONTROLLER (3)

ENDPOINTS POST, PATCH ET DELETE

```
@PostMapping("/")
public void createUser(@RequestBody User user){
    System.out.println("request to create new user");
}

@PatchMapping("/{id}")
public void updateUser(@PathVariable Long id, @RequestBody User user){
    System.out.println("request to update user by id");
}

@DeleteMapping("/{id}")
public void deleteUserById(@PathVariable Long id){
    System.out.println("request to delete user with id : " + id);
}
```

Accès aux données

- database

CONFIGURATION D'UNE SOURCE DE DONNÉES

DÉPENDANCES POM.XML

Dans le pom.xml on doit ajouter 2 dépendances:

- **BDD** : Dépendance du driver de la BDD (ici pour une base de données postgresql)
- **ORM (Object-Relational Mapping)** : Dépendance pour le mapper qui va traduire de JAVA ↔ SQL
Ici on utilise **JPA** (Java Persistence API) framework de l'ORM **Hibernate** permettant de simplifier l'utilisation de ce dernier.

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

⚠ **Important** ⚠ Ne pas oublier de reload Maven pour que les dépendances soient téléchargées et le projet mis à jour !

CONFIGURATION D'UNE SOURCE DE DONNÉES

L'APPLICATION NE SE LANCE PAS

Après avoir mis les dépendances et rechargé le projet, si on essaye de lancer le projet l'application **ne fonctionnera pas**. Cela est dû à configuration automatique de Spring, puisqu'on a mis des dépendances pour la BDD elles sont mise dans le projet et configuré. Mais il manque certaines choses à configurer manuellement. Comme l'accès à la BDD.

APPLICATION FAILED TO START

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class

Action:

Consider the following:

If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.

If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).



CONFIGURATION D'UNE SOURCE DE DONNÉES

APPLICATION.PROPERTIES

Dans Spring on peut définir des variables d'environnement du projet pour modifier certaines configurations. Certaines de ces variables ont un défaut, d'autres doivent être obligatoirement créée (comme pour l'accès à la BDD).

Ceci ce fait dans le fichier: `src/main/resources/application.properties`

```
server.port=9090
spring.application.name=first-app-spring
spring.datasource.url=jdbc:postgresql://localhost:5432/databasetest
spring.datasource.username=postgres
spring.datasource.password=postgres
```

- `server.port`: Par défaut 8080, permet de changer le port sur lequel le projet se lance
- `spring.application.name`: Permet de donner un nom à l'application
- `spring.datasource.url`: Permet de stipulé l'adresse de la BDD
- `spring.datasource.username`: Le username pour se connecter à la BDD
- `spring.datasource.password`: Le mot de passe pour se connecter à la BDD

CONFIGURATION D'UNE SOURCE DE DONNÉES

APPLICATION.YML

Une autre façon d'écrire les propriétés est d'écrire ça au format yml.
Il s'agit des mêmes variables, mais au lieu de mettre des « . » on fait un saut à la ligne avec une tabulation.

Cela permet une meilleure lisibilité (surtout quand le fichier devient long) et de mieux ordonner les variables.

```
server:
  port: 9090
spring:
  application:
    name: first-app-spring
  datasource:
    password: postgres
    url: jdbc:postgresql://localhost:5432/databasetest
    username: postgres
```

ENTITY (1)

- Classes représentant une table de base de données relationnelle. Doivent être des POJO (Plain Old Java Object).
- Toutes tables (excepté celles d'association) auront leur équivalence en Classe JAVA
- `@Entity` : Spécifie à Spring que cette classe est une entité (obligatoire)
- `@Table` : Permet de donner des options à l'entité comme renommer la table sur la BDD (optionnel). Utile quand on utilise des mots réservés comme *User*.
- `@Id` : Spécifie à Spring quel est l'attribut de cette classe qui correspond à l'identifiant de l'entité/table
- Les noms en *camelCase* sur JAVA seront traduits en *snake_case* sur la BDD (exemple: `firstName` → `first_name`)

```
@Entity
@Table(name = "users")
public class User {

    @Id
    private Long id;

    private String firstName;

    private String lastName;

    private String email;

    private LocalDate dateOfBirth;

    private String phoneNumber;

    // getters, setters, toString, etc.
```

ENTITY(2)

GÉNÉRATION AUTOMATIQUE DE LA BDD (1)

En temps normal on doit créer la BDD nous-même, mais Spring a une option pour générer automatiquement les tables grâce aux entités créer dans l'application.

Il nous faut pour cela rajouter dans les properties:

```
spring:
  jpa:
    hibernate:
      ddl-auto: update
```

- **create** : Précise que les tables doivent être créées automatiquement, si déjà existante cela les supprimera avant de les recréer.
- **create-drop** : Similaire à create mais supprimera les tables à l'arrêt de l'application.
- **update** : Créera automatiquement les tables si non existante, ou ne fera rien si existante, ou mettra à jour celles-ci si l'entité a été modifié. Pas parfait, peut avoir besoin d'interventions manuelles.
- **none** : Précise que Spring ne doit pas générer les tables.

ENTITY(2)

GÉNÉRATION AUTOMATIQUE DE LA BDD (2)

! Important !

- ddl-auto n'est fait que pour du **développement** et du **testing** !
- Ne doit être en aucun cas utilisé en production où il devra être mis en *non* !
- Les tables devront être créer avec des scripts (SQL, yml, xml, etc.).

On peut également rajouter des options dans les propriétés pour voir le SQL généré par Spring dans la console. On peut également le formater pour une meilleure lisibilité.

```
spring:
  jpa:
    properties:
      hibernate:
        show_sql: true
        format_sql: true
```

ENTITY(3)

ID

- Dans une base de données (et donc entité), un champ unique est toujours obligatoire pour pouvoir cibler une donnée en particulier.
- On peut en avoir plusieurs.
- Peuvent être de différents types (string ou integer par exemple)
- Peut-être automatisé (par une séquence de BDD).

```
@Id
private String reference;

@Id
private UUID uuid;

@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "user_sequence")
@SequenceGenerator(name = "user_sequence", allocationSize = 1, initialValue = 1000)
private Long id;
```

ENTITY(4)

COLUMN

Sur chaque attribut on peut (et il est de bonnes pratiques de le faire) ajouter l'annotation `@Column` qui permettra de définir des options pour la column en BDD, tel que:

- **name** : pour renommé la table en BDD.
- **nullable** : spécifie que ce champ est obligatoire en BDD (false par défaut)
- **unique** : spécifie que ce champ n'est pas duplicable en BDD (false par défaut)
- **length** : spécifie la limite de caractère de champs (255 par défaut)
- **updatable** : spécifie que ce champ ne pourra être que créé mais pas modifié en BDD.

```
@Column(name = "first_name", nullable = false, length = 50)
private String firstName;

@Column(name = "last_name", nullable = false, length = 50)
private String lastName;

@Column(name = "email", nullable = false, unique = true, updatable = false)
private String email;

@Column(name = "date_of_birth")
private LocalDate dateOfBirth;

@Column(name = "phone_number", nullable = false, unique = true, length = 10)
private String phoneNumber;
```

ENTITY(5)

RELATIONSHIPS (1)

JPA aide à la gestion des relations entre entités pour des base de données relationnelles.

On a les mêmes types qu'en PostgreSQL + 1:

- OneToOne
- OneToMany
- ManyToMany
- ManyToOne

ENTITY(5)

RELATIONSHIPS (2) : ONETOONE

Utilisé quand une entité contient une autre entité de manière unique.

Exemple: Un *Utilisateur* a une et une seul *Adresse*, cette *Adresse* n'appartient qu'à un et un seul *Utilisateur*.

```
@Entity
@Table(name = "users")
public class User {

    @Id
    private Long id;

    //attributs

    @OneToOne(mappedBy = "user")
    private Address address;
}
```

La table user n'aura pas l'id de address

```
@Entity
@Table(name = "address")
public class Address {

    @Id
    private Long id;

    //attributs

    @OneToOne
    private User user;
}
```

La table address aura l'id de user

ENTITY(5)

RELATIONSHIPS (3) : MANYTOMANY

Utilisé quand une entité contient plusieurs fois une autre entité et inversement.

Exemple: Un *Utilisateur* a une ou plusieurs *Adresse*, cette *Adresse* appartient à un ou plusieurs *Utilisateur*.

```
@Entity
@Table(name = "users")
public class User {

    @Id
    private Long id;

    //attributs

    @ManyToMany
    private List<Address> addresses = new ArrayList<>();
}
```

Une table de relation user → address sera créé. Contenant user_id et address_id

```
@Entity
@Table(name = "address")
public class Address {

    @Id
    private Long id;

    //attributs

    @ManyToMany(mappedBy = "addresses")
    private List<User> users = new ArrayList<>();
}
```

ENTITY(5)

RELATIONSHIPS (4) : ONETOMANY / MANYTOONE

Utilisé quand une entité contient plusieurs fois une autre entité et que celle-ci contient une seule fois la première.

Exemple: Un *Utilisateur* a une et une seule *Adresse*, cette *Adresse* appartient à un ou plusieurs *Utilisateur*.

```
@Entity
@Table(name = "users")
public class User {

    @Id
    private Long id;

    //attributs

    @ManyToOne
    private Address address;
}
```

La table user aura l'id de l'address

```
@Entity
@Table(name = "address")
public class Address {

    @Id
    private Long id;

    //attributs

    @OneToMany(mappedBy = "address")
    private List<User> users = new ArrayList<>();
}
```

ENTITY(5)

RELATIONSHIPS (5) : OPTIONS

- Relationship options:

- `optional` : pour rendre la relation obligatoire ou non (ainsi que la colonne)

- `cascade` : propagation des opérations de persistance aux enfants:

- `PERSIST` : opérations d'insertion
 - `MERGE` : opérations de mise à jour
 - `REMOVE` : opérations de suppression
 - `ALL` : toutes les opérations

- `mappedBy` : permet de définir qu'elle est la relation inverse (donc non principale)

- `@JoinColumn` :

- `name` : redéfinit le nom de la colonne
 - `nullable` : définit la colonne comme nullable ou non

- `@JoinTable` :

- `name` : redéfinit le nom de la table de relation
 - `joinColumns = @joinColumn(name = « »)` : redéfinit le nom de l'entité actuelle dans la table de relation
 - `inverseJoinColumns` : pareil que `joinColumns` mais pour l'enfant

ENTITY(5)

RELATIONSHIPS (6) : UNIDIRECTIONAL OU BIDIRECTIONAL

Pour chaque relation (excepté OneToMany qui aura besoin de ManyToOne) peut être défini en :

- **Bidirectional** : La relation est définie dans le parent et l'enfant.
 - La relation est définie dans le parent et l'enfant.
 - Chaque entité, quand on la récupérera de la BDD, récupérera son entité relationnelle.
 - Peut entraîner une boucle infinie où entité 1 récupère entité 2 qui récupère entité 1, et ainsi de suite.
- **Unidirectional** :
 - La relation est définie uniquement dans le parent.
 - Seule le parent récupérera l'entité relationnelle. L'enfant ne récupérera rien.
 - Evite les boucles infinies.
 - Besoin de faire manuellement un appel à la BDD si on veut que l'enfant récupère le parent.

Le choix de l'un ou de l'autre dépendra des appels qui se feront en BDD dont on a besoin.

Il faut se poser les questions :

« Est-ce que je veux toujours récupérer la ou les entité(s) relationnelle(s) ?

Ou je veux la(les) récupérer dans des situations bien particulière et auquel cas autant faire un appel manuel à la BDD ? »

Le plus souvent une unidirectionnelle sera suffisante et évitera des problèmes de performances.

ENTITY(6)

RELATIONSHIPS (7) : RELATIONS CYCLIQUES (1)

Un problème récurrent lors de l'utilisation des relations, notamment des relations bidirectionnelles, sont les relations cycliques.

Imaginons une Adresse ayant un Utilisateur, et l'Utilisateur ayant plusieurs Adresse.

Si je récupère l'Adresse, je récupère l'Utilisateur avec, qui récupère sa liste d'Adresse, qui elles-mêmes retourneront leur Utilisateur, etc, etc...

On aura donc une boucle infinie qui apparaîtra.

```
@Entity
public class Address {

    @Id
    private Long id;

    @ManyToOne
    private User user;
```

```
@Entity
public class User {

    @Id
    private Long id;

    @OneToMany(mappedBy = "user")
    private List<Address> addresses = new ArrayList<>();
```

ENTITY(6)

RELATIONSHIPS (7) : RELATIONS CYCLIQUES (2)

Pour résoudre il existe plusieurs solutions :

- Eviter les relations bidirectionnelles et utiliser de l'unidirectionnelles. Si on peut se le permettre, autant avoir la relation seulement du côté qui nous intéresse, et pas de l'autre.
- `@JsonIgnore`
- `@JsonIgnoreProperties`

ENTITY(6)

RELATIONSHIPS (7) : RELATIONS CYCLIQUES (2) JSONIGNORE

L'annotation `@JsonIgnore` permet de spécifier qu'un champs ne sera pas envoyé en réponse de controller.

```
@Entity
public class Address {

    @Id
    private Long id;

    @JsonIgnore
    @ManyToOne
    private User user;
```

Dans notre cas ici, on peut trouver intéressant d'avoir la liste des adresses dans l'utilisateur, mais on a pas besoin d'avoir l'utilisateur dedans. Mais ayant un `ManyToOne`, on doit le laisser pour avoir la clé étrangère de l'utilisateur dans l'adresse.

Donc on laisse la relation bidirectionnelle, mais on stipule que dès qu'on envoie l'objet au client, que cela soit `Address` ou `User`, le champs du `ManyToOne` sera ignoré.

ENTITY(6)

RELATIONSHIPS (7) : RELATIONS CYCLIQUES (2) JSONIGNOREPROPERTIES

A d'autre moment, on voudra également la relation bidirectionnelle, mais le client veut à la fois récupérer la liste d'adresse lorsqu'il récupère l'utilisateur directement, mais veut également récupérer l'utilisateur quand il récupère l'adresse directement.

```
@Entity
public class Address {

    @JsonIgnoreProperties("addresses")
    @ManyToOne
    private User user;
```

Pour cela on utilise l'annotation `@JsonIgnoreProperties`. Celle-ci permet de spécifier à une relation, quand on récupère l'entité de celle-ci, on ignorera un champs en particulier.

```
@Entity
public class User {

    @JsonIgnoreProperties("user")
    @OneToMany(mappedBy = "user")
    private List<Address> addresses = new ArrayList<>();
```

Ici, lorsqu'on récupérera le User, on aura également la liste d'Address, mais dans celles-ci on ne récupérera pas le champs « user ». Evitant la boucle infinie.

De même, lorsqu'on récupère Address, on récupérera le User, mais la liste d'Address sera ignorée.

UTILISATION DE JPA (1)

JPA, ou JAVA Persistence API, est une spécification Java qui définit une interface de programmation pour la gestion de persistance de donnée.

Elle fournit une méthode standard pour les mapper les objets Java à des données tabulaires dans une base de données relationnelles, et vice versa.

Composé de:

- **ORM** (Object Relational Mapping) : Le mapper de conversion Java ↔ SQL
- **Gestion du cycle de vie des entités** : persistant, détaché, géré, supprimé
- **Requêtes JPQL** (Java Persistence Query Language) : Langage de requêtes orienté objet similaire à SQL
- **Relations entre Entités** : ManyToMany, ManyToOne, OneToOne, OneToMany
- **Gestion de transactions** : permettant de définir plusieurs opérations sur une entité en une.
- **Caches** : pour améliorer les performances en réduisant l'accès à la BDD.

UTILISATION DE JPA (1)

JPA, ou JAVA Persistence API, est une spécification Java qui définit une interface de programmation pour la gestion de persistance de donnée.

Elle fournit une méthode standard pour les mapper les objets Java à des données tabulaires dans une base de données relationnelles, et vice versa.

Composé de:

- **ORM** (Object Relational Mapping) : Le mapper de conversion Java ↔ SQL
- **Gestion du cycle de vie des entités** : persistant, détaché, géré, supprimé
- **Requêtes JPQL** (Java Persistence Query Language) : Langage de requêtes orienté objet similaire à SQL
- **Relations entre Entités** : ManyToMany, ManyToOne, OneToOne, OneToMany
- **Gestion de transactions** : permettant de définir plusieurs opérations sur une entité en une.
- **Caches** : pour améliorer les performances en réduisant l'accès à la BDD.

UTILISATION DE JPA (2)

MISE EN PLACE

```
@Repository  
public interface IUserRepository extends JpaRepository<User, Long> {  
}
```

- En utilisant JPA, les repositories créés sont des *interfaces*.
- `@Repository` : Spécifie à Spring que cette interface est un repository (optionnel avec `JpaRepository` qui l'injecte automatiquement).
- `JpaRepository<C, I>` :
 - `JpaRepository` : interface principale à *extends* permettant d'utiliser les fonctionnalités de JPA.
 - `C` : est la classe de l'entité que ce fichier repository doit traiter.
 - `I` : Le type de l'identifiant de cette entité.

UTILISATION DE JPA (3)

JPQL (1)

- Java Persistence Query Language (ou JPQL) est un langage de requête orienté objet, similaire à SQL.
- Contrairement à SQL qui opère sur les tables, JPQL opère sur les entités Java et leurs attributs.
- Il est fortement préconisé d'utiliser JPQL plutôt que du SQL lorsqu'on utilise JPA.
- Deux moyens de l'utiliser:
 - Query : on définit une requête à l'écrit.
 - Method : on définit une requête grâce au nom de la méthode.
- JPA vient avec des requêtes par défaut déjà défini, tel que:
 - `findById` : Récupère une donnée par son ID
 - `findAll` : Récupère toutes les données de la table
 - `save` : Insert ou Update une donnée dans la table, dépendamment si l'ID a été spécifié dans l'objet.
 - `deleteById` : Supprimer une donnée en BDD par son ID
 - Etc.

UTILISATION DE JPA (3)

JPQL (2) : QUERY (1)

JPQL est un langage très très similaire à SQL, excepté qu'on utilise les noms des entités et leurs attributs plutôt que les noms des tables et columns.

- On utilise l'entité *User* plutôt que la table *users*.
- On utilise l'attribut *firstName* plutôt que la colonne *first_name*.

Cela permet une facilité d'utilisation puisqu'on n'a pas à sans cesse passé d'un langage JAVA d'un langage SQL.

JPQL	SQL
SELECT e FROM EntityName e	SELECT * FROM table_name
SELECT e FROM EntityName e WHERE e.attributName = :valeur	SELECT * FROM table_name WHERE column_name = 'valeur'
SELECT e FROM EntityName e ORDER BY e.attributName ASC	SELECT * FROM table_name ORDER BY column_name ASC
SELECT COUNT(e) FROM EntityName e	SELECT COUNT(*) FROM table_name

UTILISATION DE JPA (3)

JPQL (2) : QUERY (2)

- `@Query`: pour pouvoir définir en *String* la requête JPQL
- `?1`: Permet de récupérer le premier paramètre de la méthode et de l'injecter dans la requête (on mettrait ?2 pour le deuxième paramètre, et ainsi de suite).

```
@Query("select u from User u where u.email = ?1")
Optional<User> findByEmail(String email);

@Query("select u from User u where u.phoneNumber = ?1")
Optional<User> findByPhoneNumber(String phoneNumber);

@Query("select u from User u where upper(u.lastName) = upper(?1)")
List<User> findByLastNameIgnoreCase(String lastName);

@Query("select count(u) from User u where u.email like '%@gmail.com'")
long countByEmailGmail();
```

UTILISATION DE JPA (3)

JPQL (3) : OPTIONAL

`Optional<User>` : Pour des récupérations singulières, on préfère utiliser *Optional* pour éviter l'utilisation d'objets potentiellement null

Classe introduite comme container immuable de valeur pour éviter de manipuler des données potentiellement *null* pouvant amener à des `NullPointerException`. Il est fortement conseillé de l'utiliser.

- Contient deux états:
 - Valeur présente
 - Valeur absente
- Contient des méthodes utilitaires:
 - `.isPresent()` et `.isEmpty()`
 - `.get()` : récupère la valeur si présente (lance exception si absente)
 - `.orElse()` : récupère la donnée ou ce qui est spécifié dans la fonction si non présente
 - `.orElseThrow()` : récupère la donnée ou throw une exception spécifiée

UTILISATION DE JPA (3)

JPQL (4) : METHOD (1)

Autrement qu'en utilisant des Queries JPQL, on peut également créer des requêtes avec simplement le nom de la méthode.

JPQL contient des mots clés pour traduire la méthode en requête:

- **findBy**: créer une méthode SELECT (le type de return spécifique définira si on reçoit une seule ou plusieurs résultat(s))
- Suffixes: permet de spécifier des options de requêtes, soit le WHERE ou ORDER BY, etc. :
 - **NomDeLAttribut**: En mettant le nom de l'attribut permet de rajouter une clause WHERE sur cette attribut (par défaut en equals)
 - **Contains**: Permet de spécifier la clause WHERE précédente comme étant une contains, donc si la donnée cherchée est quelque part dans le string
 - **IgnoreCase**: précise que la clause WHERE ne fait pas attention à la case (majuscule/minuscule)
 - **OrderByNomDeLAttribut**: Rajoute une clause ORDER BY sur le champ spécifié (ASC ou DESC peut être rajouté à la fin).

```
Optional<User> findByEmail(String email);  
  
Optional<User> findByPhoneNumber(String phoneNumber);  
  
List<User> findByLastNameIgnoreCase(String lastName);  
  
long countByEmailEndingWith(String emailSuffix);
```

UTILISATION DE JPA (3)

JPQL (4) : METHOD (2)

JPQL Method	SQL
findByEmail(String email)	SELECT * FROM table_name WHERE email = 'valeur'
findByEmailAndUsername(String email, String username)	SELECT * FROM table_name WHERE email = 'valeur' and username = 'valeur'
findByEmailEndsWithOrderByEmailDesc(String emailSuffix)	SELECT * FROM table_name WHERE email = 'valeur' ORDER BY email DESC
findByLastNameContainsIgnoreCase(String lastName)	SELECT * FROM table_name WHERE UPPER(last_name) LIKE UPPER('%valeur%')

<https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html>

UTILISATION DE JPA (3)

JPQL (5) : QUERY VS METHOD VS SQL ?

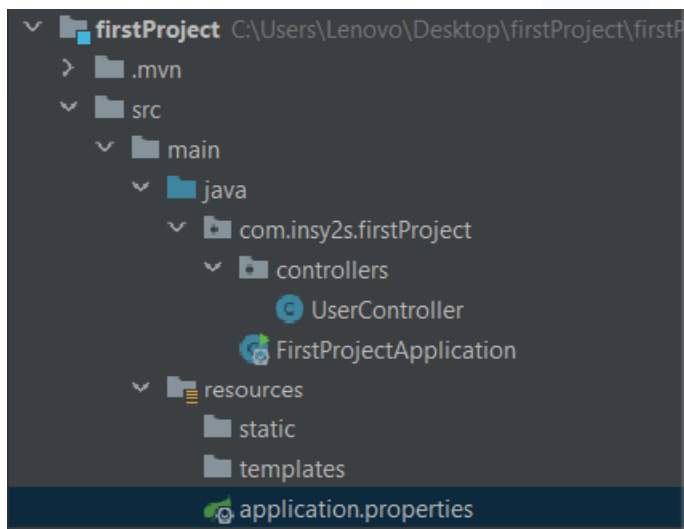
- **METHOD** : Tant que les requêtes sont simples et les noms de méthodes sont facilement lisibles et compréhensibles, on préférera utiliser les requêtes sous forme de Méthode.
- **QUERY** : Si la méthode/requête devient complexe à lire, on préférera utiliser une Query dont il sera plus compréhensible à définir l'utilité. On pourra également les écrire sur plusieurs lignes pour gagner en lisibilité.
- **SQL** : Dans certains cas précis on utilisera le SQL à la place du JPQL :
 - Principalement pour des fonctionnalités spécifiques à la BDD utilisée.
 - Également, dans de rares cas pour de l'optimisation de performance si la requête JPQL est très très complexe.

EXERCICE : MISE EN PLACE DE LA COUCHE DATA (1)

PROPERTIES

Dans **application.properties** ajouter les propriétés suivantes :

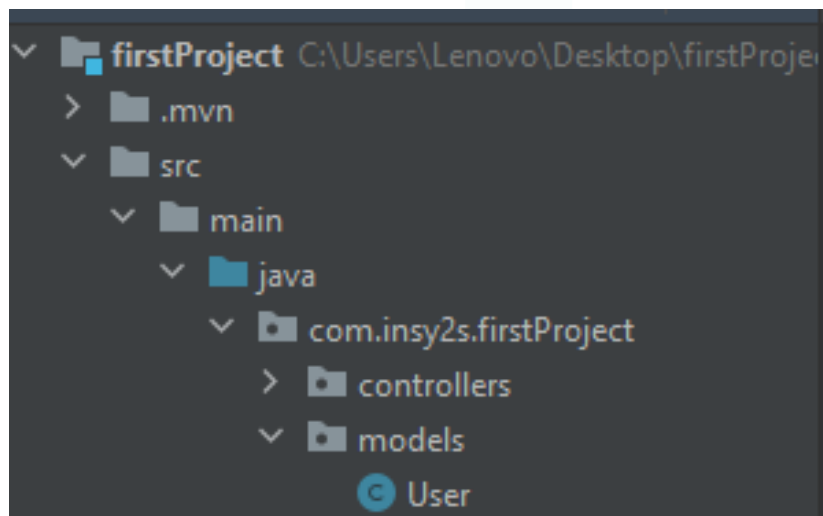
```
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres  
spring.datasource.username=postgres  
spring.datasource.password=postgres  
spring.jpa.hibernate.ddl-auto=update
```



EXERCICE : MISE EN PLACE DE LA DB (2)

ENTITY(1) : USER

Créez un package models :
ajoutez une classe User



Dans la classe User : ajoutez aussi un constructeur vide,
un constructeur avec tous les arguments, et les getters
et les setters

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "user_sequence")
    @SequenceGenerator(name = "user_sequence", allocationSize = 1, initialValue = 1000)
    private Long id;
    private String firstName;
    private String lastName;
    private String email;

    // constructeur vide

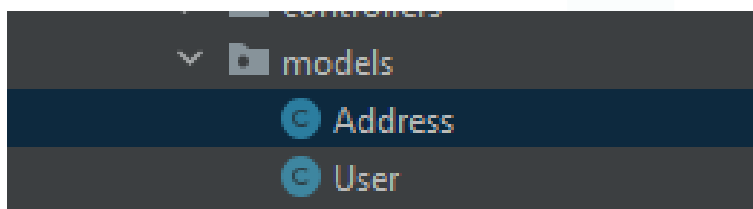
    // constructeur avec tout les arguments

    // getters et setters
}
```

EXERCICE : MISE EN PLACE DE LA DB (2)

ENTITY(2) : ADDRESS

Ajoutez une classe Address



Dans la classe Address : ajoutez aussi un constructeur vide, un constructeur avec tous les arguments, et les getters et les setters

```
@Entity
@Table(name= "address")
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "Address_GEN")
    @SequenceGenerator(name = "Address_GEN", sequenceName = "Address_SEQ", allocationSize = 1000)
    private Long id;
    @Column(name="street_number")
    private String streetNumber;
    @Column(name="street_name")
    private String streetName;
    @Column(name="zip_code")
    private String zipCode;
    private String city;

    // constructeur vide

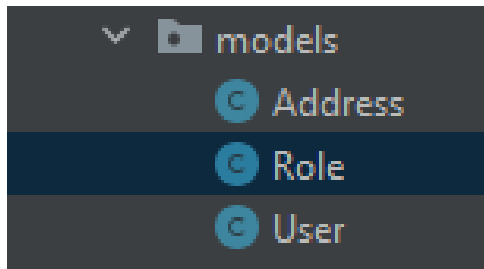
    // constructeur avec tous les arguments

    // getters et setters
}
```

EXERCICE : MISE EN PLACE DE LA DB (2)

ENTITY(3) : ROLE

Ajoutez une classe Role



Dans la classe Role : ajoutez aussi un constructeur vide, un constructeur avec tous les arguments, et les getters et les setters

```
@Entity
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "Role_GEN")
    @SequenceGenerator(name = "Role_GEN", sequenceName = "Role_SEQ", allocationSize = 1000)
    private Long id;
    private String name;

    // constructeur vide

    // constructeur avec tous les arguments

    // getters et setters
}
```

EXERCICE : MISE EN PLACE DE LA DB (2)

ENTITY(4) : RELATIONSHIPS

Un *Utilisateur* a une et une seule *Adresse*, cette *Adresse* appartient à un ou plusieurs *Utilisateur*.

Un *Utilisateur* a un ou plusieurs *Role*, cette *Role* est attribué à un ou plusieurs *Utilisateur*.

Classe *User.java*

```
@ManyToOne
@JoinColumn(name = "address_id")
private Address address;

@ManyToMany
private List<Role> roles = new ArrayList<>();
```

Classe *Address.java*

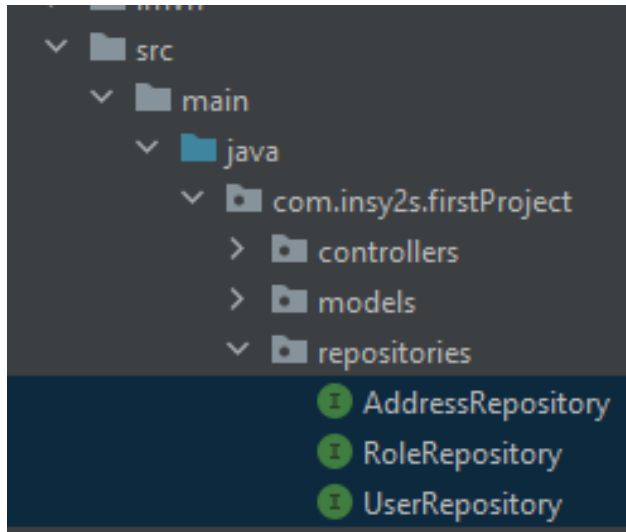
```
@OneToMany(mappedBy = "address")
@JsonIgnore
private List<User> users = new ArrayList<>();
```

Classe *Role.java*

```
@ManyToMany(mappedBy = "roles")
@JsonIgnore
private List<User> users = new ArrayList<>();
```


EXERCICE : MISE EN PLACE DE LA DB (3)

REPOSITORY



```

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}
    
```

```

@Repository
public interface RoleRepository extends JpaRepository<Role, Long> {
    Role findByName(String name);
}
    
```

```

@Repository
public interface AddressRepository extends JpaRepository<Address, Long> {

    @Query("select a from Address a where a.streetNumber = ?1 and a.streetName = ?2 and a.city = ?3")
    Optional<Address> findAddress(String streetNumber, String streetName, String city);
}
    
```

Injection de Dépendance

INTRODUCTION

Concept clé de SpringBoot, l'injection de dépendance provient de l'Inversion de Contrôle (IoC) où les dépendances sont injectées automatiquement dans le parent où elles sont utilisées.

Donc si on veut utiliser UserRepository dans UserController, on n'instancie pas le repository dans le controller, mais on va juste spécifier qu'on veut l'utiliser et Spring injectera la dépendance lors du run de l'application.

@AUTOWIRED ET CONSTRUCTEUR

- Un des moyens d'injecter une dépendance est de la définir en attributs sans l'instancier et de l'annoter avec @Autowired .

```
@Autowired  
private IUserRepository userRepository;
```

- On peut également injecter via constructeur, avec la dépendance en attributs **final**

```
private final IUserRepository userRepository;  
  
public UserController(IUserRepository userRepository) {  
    this.userRepository = userRepository;  
}
```

- Bien que les deux soient possible, l'injection par constructeur est préconisée:
 - Plus facile à tester (à mocker)
 - Immuable
 - Moins de dépendances à Spring

Couche Service



INTRODUCTION

- La couche service (*Service Layer*) se situe entre la couche Controller et la couche Repository.
- Contient la logique métier du projet.
- Contient les opérations et les traitements sur les données pour répondre aux besoins fonctionnels.
- Coordonne les opérations pour satisfaire la(les) demande(s) du(des) contrôleur(s).
- Chaque service/méthode respecte la *Separation of Concerns* puisque chaque service contiendra sa propre logique métier et que chaque méthode aura son objectif bien particulier. Permettant une meilleure réutilisabilité et une meilleure organisation du code.

INTRODUCTION

`@Service` : permet de spécifier à SpringBoot que cette classe fait partie de la couche Service.

```
@Service
public class UserService {

    private final IUserRepository userRepository;

    public UserService(IUserRepository userRepository) {
        this.userRepository = userRepository;
    }

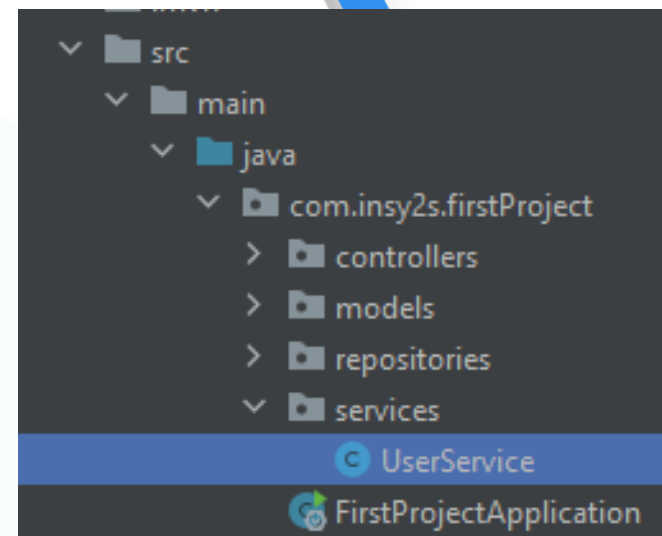
    public UserData findById(Long id) {
        Optional<User> user = userRepository.findById(id);
        if (user.isPresent()) {
            return new UserData(user.get().getFirstName(), user.get().getLastName(), user.get().getEmail());
        }
        return null;
    }

    public Long save(UserData userData) {
        User user = new User();
        user.setFirstName(userData.getFirstName());
        user.setLastName(userData.getLastName());
        user.setEmail(userData.getEmail());
        userRepository.save(user);
        return user.getId();
    }
}
```

EXERCICE : CRÉER UN SERVICE(1)

AJOUT DU PACKAGE ET DU FICHIER

On crée un package services on y ajoute le UserService où on déclare les 3 repositories avec un constructeur pour les utiliser dans le service



```
@Service
public class UserService {

    private final UserRepository userRepository;
    private final AddressRepository addressRepository;
    private final RoleRepository roleRepository;

    public UserService(UserRepository userRepository, AddressRepository addressRepository, RoleRepository roleRepository){
        this.userRepository = userRepository;
        this.addressRepository = addressRepository;
        this.roleRepository = roleRepository;
    }

    // methods
}
```


EXERCICE : CRÉER UN SERVICE(2)

MÉTHODES DE RÉCUPÉRATION

Complétez le service avec les 2 **Méthodes de récupération** :

- getAllUsers : récupérer tous les utilisateurs
- getUserById : récupérer un utilisateur avec son id

```
public List<User> getAllUsers(){  
    return userRepository.findAll();  
}  
  
public User getUserById(Long id){  
    Optional<User> user = userRepository.findById(id);  
    return user.get();  
}
```

EXERCICE : CRÉER UN SERVICE(3)

MÉTHODE DE CRÉATION

Complétez le service avec un **Méthode de Création** : createUser

```
public void createUser(User user){
    if(user.getAddress() != null){
        Address address = user.getAddress();
        Optional<Address> existingAddress = addressRepository.findAddress(
            address.getStreetNumber(), address.getStreetName(), address.getCity());

        existingAddress.ifPresentOrElse(
            foundAddress -> user.getAddress().setId(foundAddress.getId()),
            () -> {
                addressRepository.save(address);
            });
    }

    if(user.getRoles() != null){
        List<Role> roles = user.getRoles();
        roles.stream().forEach(userRole -> {
            Role role = roleRepository.findByName(userRole.getName());
            userRole.setId(role.getId());
        });
    }

    userRepository.save(user);
}
```

EXERCICE : CRÉER UN SERVICE(4)

MÉTHODE DE MISE À JOUR

Complétez le service avec un **Méthode de mise à jour** : updateUserLastName

```
public void updateUserLastName(Long id, String lastName){  
    Optional<User> existingUser = userRepository.findById(id);  
    existingUser.ifPresent(user -> {  
        if(lastName != null){  
            user.setLastName(lastName);  
        }  
        userRepository.save(user);  
    });  
}
```

EXERCICE : CRÉER UN SERVICE(5)

MÉTHODE DE SUPPRESSION

Complétez le service avec un [Méthode de suppression](#) : deleteUser

```
public void deleteUser(Long id){  
    Optional<User> existingUser = userRepository.findById(id);  
    existingUser.ifPresent(user -> {  
        userRepository.delete(user);  
    });  
}
```

EXERCICE : COMPLÉTER LE CONTROLLER

AJOUT DU SERVICE ET MODIFICATION DE LA METHODE GET

```
private final UserService userService;

public UserController(UserService userService) {
    this.userService = userService;
}

@GetMapping("/{id}")
public ResponseEntity<User> getUserById(@PathVariable Long id){
    System.out.println("request to get user with id : " + id);
    return ResponseEntity.ok(userService.getUserById(id));
}

@GetMapping("/")
public ResponseEntity<List<User>> getAllUsers(){
    System.out.println("request to get all users");
    return ResponseEntity.ok(userService.getAllUsers());
}
```

EXERCICE : COMPLÉTER LE CONTROLLER

MODIFICATION DES MÉTHODES CREATE, PATCH, DELETE

```
@DeleteMapping("/{id}")
public void deleteUserById(@PathVariable Long id){
    userService.deleteUser(id);
    System.out.println("request to delete user with id : " + id);
}

@PostMapping("/")
public ResponseEntity<String> createUser(@RequestBody User user){
    userService.createUser(user);
    return ResponseEntity.ok("created user");
}

@PatchMapping("/{id}")
public void updateUser(@PathVariable Long id, @RequestBody User user){
    System.out.println("request to update user by id");
    userService.updateUserLastName(id, user.getLastName());
}
```