EPI-Polytec
Ecole Privée Internationale d'Ingénieurs
المدرسة الدولية العليا الخاصة للمهندسين بسوسة

Leçon 1 : Introduction à Spring Boot

Présentation de Spring Boot :

Le Framework Spring Boot est une extension du Framework Spring qui a pour but de mettre en place rapidement des applications Web MVC et/ou des API Rest. Grâce à son système modulaire de dépendances (géré par Maven) et son principe de configuration automatique, il permet de disposer d'une structure de projet immédiatement opérationnelle et extensible.

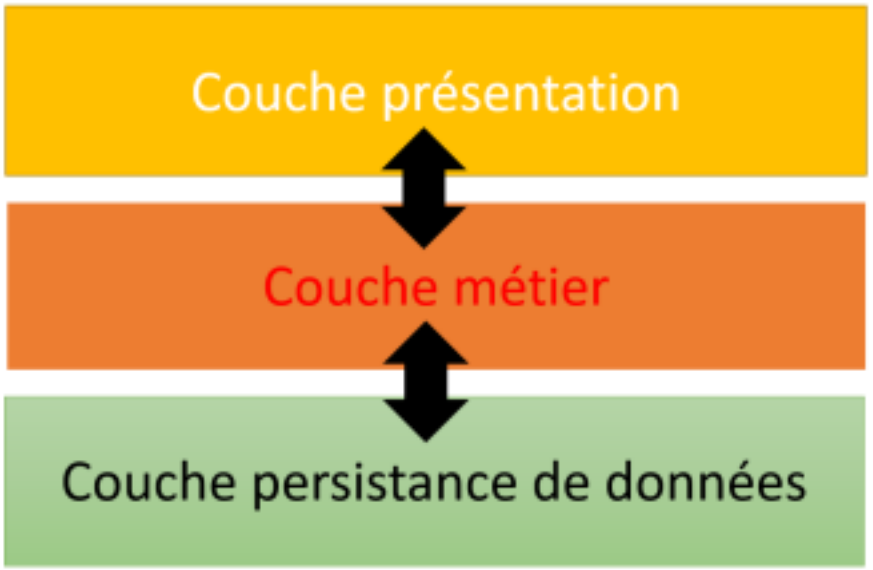
Les principaux avantages du Framework Spring Boot :

- **Un site starter** : (`https://start.spring.io/`) qui permet de générer rapidement la structure du projet en y incluant toutes les dépendances (bibliothèques) nécessaires à l'application. De ce fait, toutes les dépendances sont regroupées dans un même dépôt afin de faciliter la gestion de celles-ci et assurer la compatibilité entre eux.
- **Déploiement simple** : un conteneur intégré (Tomcat) dans le projet afin de faciliter son déploiement. Plus besoin d'un serveur externe pour le déploiement comme en Java EE. -
- **L'autoconfiguration** : Spring Boot applique une configuration par défaut au démarrage de l'application pour toutes dépendances présentes dans celle-ci.

Architecture logique d'une application Spring Boot :

Par définition, l'architecture d'un logiciel décrit la manière dont seront organisésles différents éléments d'une application et comment ils interagissent entre eux.

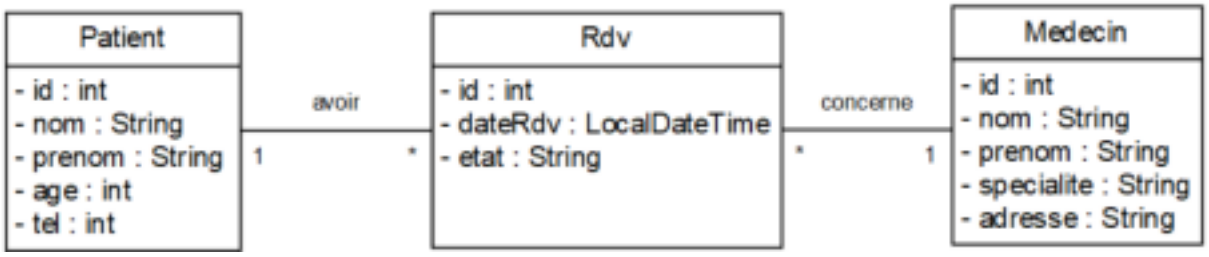
L'architecture logique d'une application Spring Boot est hiérarchique (organisée en couches) où chaque couche utilise les services de la couche en dessous et offre ses services à la couche en dessus comme le montre la figure suivante :



➢ La couche présentation a pour rôle de gérer les requêtes et les réponses HTTP avec le client et de traduire les paramètres des requêtes en objets. Les composants de cette couche différent selon le type de l'application Spring Boot à réaliser (Web ou API REST). ➢ La couche métier gère toute la logique métier de l'application : elle récupère les données de la couche persistance, effectue les traitements et validation sur ces données et les expose à la couche présentation comme des services.

➢ La couche persistance de données contient toute la logique de stockage et accès aux données (couche DAO). Elle effectue aussi la correspondance entre les objets entités depuis et vers des lignes dans la BD moyennant un ORM (Object Relational Mapping). **Enoncé du TP1 :**

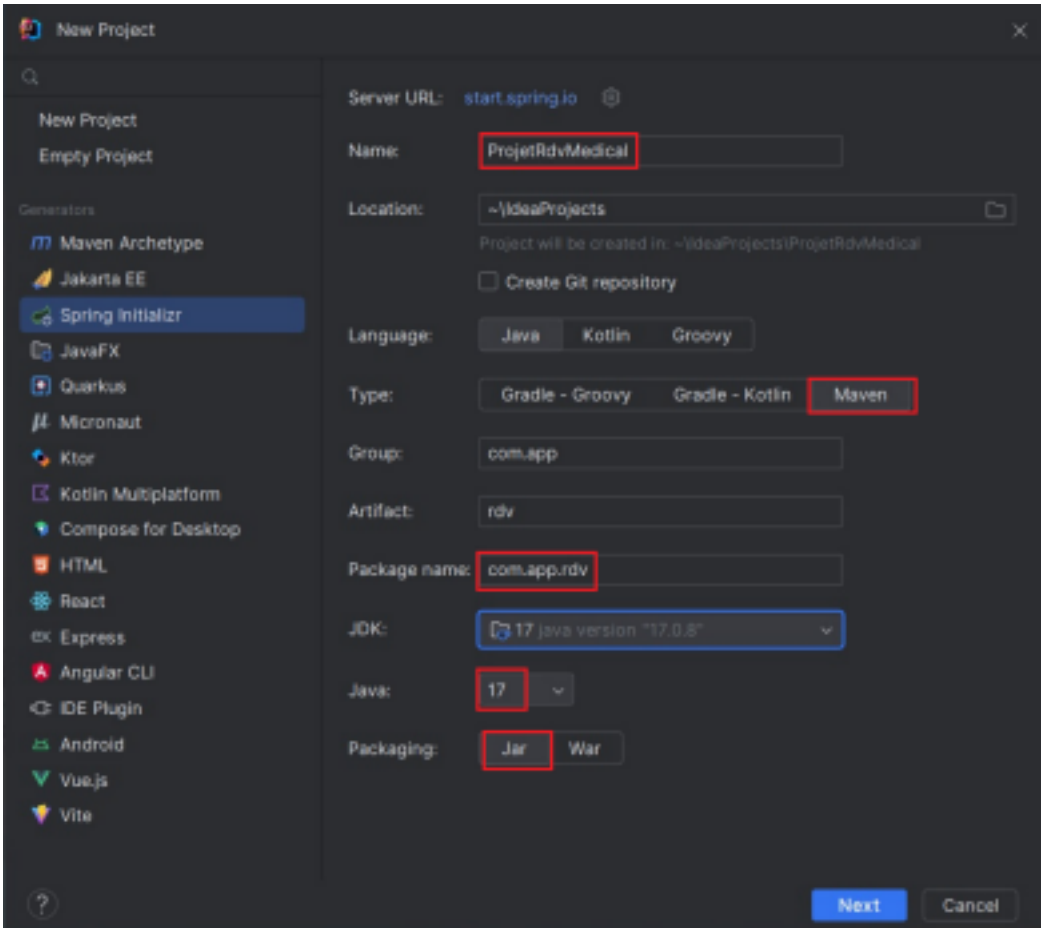
L'objectif final de ce TP est de développer une application Spring Boot Web MVC et qui expose des API Rest permettant la gestion de 3 entités reliées entre elles `Patient`, `Medecin` et `Rdv`. Le diagramme de classes suivant représente les relations entre les 3 entités :



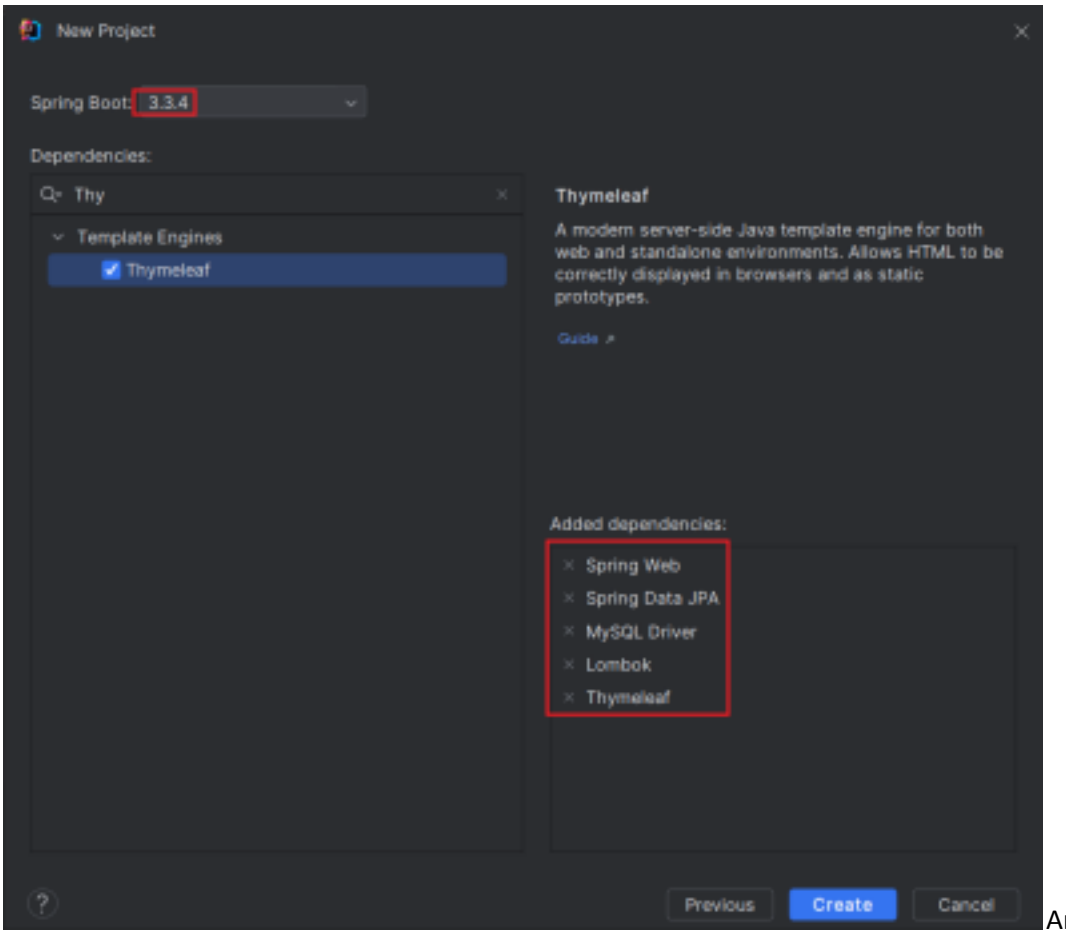
Un Patient peut avoir un ou plusieurs RDV. Un médecin peut être impliqué dans plusieurs RDV. Un RDV concerne un Patient avec un Médecin à une date et heure donnée. Nous allons commercer par créer dans l'IDE IntelliJ la structure du projet Spring Boot avec des dépendances gérées par Maven. Maven est un gestionnaire des dépendances des projets Java. Cette gestion s'articule sur un élément POM (Project Object Model) matérialisé par le fichier pom.xml qui contient la description des dépendances du projet. Les dépendances à ajouter dans le projet seront :

- `Spring Web` : définit une application web MVC RestFull avec Tomcat comme serveur intégré - `Spring Data JPA` : c'est la dépendance qui permet de persister les données dans une BD en utilisant le Framework Hibernate.
- `MySQL Driver` : le pilote pour se connecter à des BD MySQL
- `Lombok` : dépendance pour réduire le code des classes entités.
- `Thymeleaf` : dépendance du moteur de Template.

Lancer le IDE IntelliJ, choisir `New Project` avec le générateur `Spring initializr` comme le montre la fenêtre suivante :



En cliquant sur Next, on va choisir les 4 dépendances du projet citées plus haut :



En cliquant sur `Create`, Maven va télécharger les dépendances depuis le dépôt central vers le dépôt local (le dossier `.m2` situé sous `C:\users\votre_session`). Pour la première création d'un projet, le téléchargement peut prendre quelques minutes selon le débit de la connexion.

Le package principale du projet `com.app.rdv` contient la classe exécutable `ProjetRdvMedicalApplication.java`; qui représente le point de démarrage de l'application Spring Boot (qui s'exécute comme une Application Java). En respectant l'architecture d'une application Spring Boot (voir plus haut), Vous devez créer au fur et à mesure les packages suivants :

➢ `com.app.rdv.entities` contenant la définition des 3 entités `Patient` , `Medecin` et `Rdv`.

➢ `com.app.rdv.repository` contenant les 3 interfaces (`PatientRepository`, ...) héritant chacune de l'interface générique `JpaRepository<T,ID>`

➢ Les paramètres d'accès à la BD sont définis dans le fichier `application.properties` sous `src/main/resources`.

Ces 3 composants représentent **la couche persistance** dans l'architecture de l'application. ➢ `com.app.rdv.service` contenant 3 interfaces (`IServicePatient`, ...) qui contiennent chacune la signature des méthodes métier sur chacune des entités et 3 classes (`ServicePatient`, ...) annotées par `@Service` implémentant chacune les méthodes métier déclarées dans chacune des interfaces.

Le contenu de ce package représente **la couche métier** de l'application.

➢ `com.app.rdv.controller` contenant les différentes classes des contrôleurs Web et contrôleurs Rest. Chaque contrôleur contiendra les actions traitant les requêtes HTTP. Les contrôleurs Web font appel à des vues Thymeleaf pour l'affichage des interfaces. Le contenu de ce package avec les vues Thymeleaf représentent **la couche présentation** de l'application.

Etape 1 : Définition des entités

Lombok est une dépendance qui a pour but de réduire le code des classes entités en remplaçant la définition des setters, getters et constructeurs par des annotations. Les annotations `@Getter @Setter` permettent de remplacer tousles getters et setters des attributs.

L'annotation `@NoArgsConstructor` permet de remplacer le constructeur par défaut

L'annotation `@AllArgsConstructor` permet de remplacer le constructeur avec paramètres - Créer le package `com.app.rdv.entities`

- Créer successivement 3 classes chacune correspondante à une entité. Pour l'instant on va ignorer les relations entre les entités. Respecter les types des attributs mentionnés dans le diagramme de classe. Pour exemple l'entité Patient se définie comme suit :

```java
package com.app.rdv.entities;

import ...
no usages
@Getter @Setter @NoArgsConstructor @AllArgsConstructor
@Entity
public class Patient {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String nom;
    private String prenom;
    private int age;
    private int tel;
}
```

Relations entre entités

Selon le diagramme de classes, Il existe entre chaque 2 entités une relation bidirectionnelle. Par exemple un Patient peut avoir plusieurs Rdvs (relation de type `OneToMany` dans l'entité `Patient`). De plus, plusieurs Rdvs concerne un seul Patient (relation de type `ManyToOne` dans l'entité `Rdv`).

En choisissant l'entité `Patient` comme esclave dans cette relation (on ajoute la propriété `mappedBy="patient"` dans la relation de type `OneToMany`). Ce qui nous donne : Dans l'entité `Patient` :

```java
private int tel;
@OneToMany(mappedBy = "patient")
List<Rdv> rdvList;
```

Dans l'entité `Rdv` :

```java
private String etat;
@ManyToOne
Patient patient;
```

➢ Faite de même pour relier les 2 entités `Medecin` et `Rdv`.

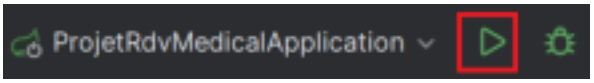
Etape 2 : Définir les paramètres d'accès à la BD

Ajouter dans le fichier `application.properties` les paramètres suivants :

```
spring.application.name=ProjetRdvMedical

spring.datasource.url=jdbc:mysql://localhost/bd_rdv?createDatabaseIfNotExist=true&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Pour exécuter votre application afin de générer les tables dans la BD `bd_rdv`, Cliquer sur :



Normalement la BD sera créé contenant les 3 tables liées `Patient`, `Medecin` et `Rdv`.

Etape 3 : Création des Repository

Dans un package repository, créer l'interface `RdvRepository` Comme suit.

```java
package com.app.rdv.repository;

import com.app.rdv.entities.Patient;
import org.springframework.data.jpa.repository.JpaRepository;

no usages
public interface PatientRepository extends JpaRepository<Patient, Integer> {
}
```

Cette interface hérite toutes les méthodes CRUD dont les signatures se trouvent dans l'interface `JpaRepository` et leurs implémentations se trouvent dans une classe de Hibernate. Définissez de la même manière les interfaces `PatientRepository` et `RdvRepository`.

Par la suite on ajoutera quelques requêtes dérivées (derived Query) à ces interfaces.

Principe de l'injection des dépendances :

Le noyau du Framework Spring et de Spring Boot Spring Core se base sur un patron de conception appelé *Inversion of Control (IoC)*.

L'injection des dépendances implémente le principe de l'IoC. Elle permet **d'instancier** des objets et de créer des dépendances nécessaires entre elles, sans avoir besoin de coder cela par les développeurs (en utilisant l'instanciation statique avec l'opérateur new). C'est au Framework de gérer ces dépendances dynamiquement au moment du déploiement de l'application Spring Boot.

Ceci a pour but de diminuer le couplage entre les objets (appliquer le principe de couplage faible entre les objets) afin d'obtenir un code lisible et facilement extensible.

Etape 4 : Création des composants de la couche service

Créer le package `com.app.rdv.service` qui contient pour chacune des 3 entités : - une interface (nommée `IServiceRdv` pour l'entité `Rdv`) contenant la signature des méthodes d'ajout et récupération de la liste sur cette entité.

- une classe (nommée `ServiceRdv` pour l'entité `Rdv`) qui implémente cette interface et qui est annotée par `@Service` (pour injecter sa dépendance dans les classes contrôleurs de la couche Présentation). Cette classe doit injecter (par constructeur) une dépendance sur l'interface Repository.

Etape 5 : Création des composants de la couche présentation

Un contrôleur Rest est une classe java annotée par `@RestController`. C'est un composant du modèle MVC. Il contient des méthodes (appelées actions) qui vont traiter les requêtes HTTP envoyés par le client.

Les annotations utilisées dans un contrôleur :

`@RequestMapping` : est l'une des annotations les plus couramment utilisées dans les applications Web Spring. Cette annotation 'mappe' les requêtes HTTP (get et post) aux actions des contrôleurs MVC. Elle possède son équivalant pour une requête spécifique : - `@GetMapping` : pour une requête get. Exemple : `@GetMapping("all")` - `@PostMapping` : pour une requête post. Exemple : `@PostMapping("save")`

`@PathVariable` : est une annotation qui indique que l'action possède un paramètre qui devrait être associé à une valeur dans l'URL traité. Exemple :

```
@GetMapping("delete/{id}")
public String delete(@PathVariable Long id) {
            ...
}
```

`@RequestParam` : utilisé pour lier un paramètre de la requête HTTP au paramètre de l'action. Exemple :

```
@PostMapping("/find")
public String find (@RequestParam String mot) {
            ...
}
```

Créer dans le package `com.app.rdv.controller` une classe `RdvController` qui sera le contrôleur Web MVC pour l'entité `Rdv`. Ce contrôleur traitera toutes les URLs qui commencent par `/rdv/` et injectera une dépendance (par constructeur) sur la classe `ServiceRdv` à travers l'interface implémentée par cette classe pour ajouter un Rdv puis afficher la liste des Rdvs selon l'ordre chronologique.

Développer les 2 vues : une pour l'ajout d'un RDv et l'autre pour l'affichage des Rdvs

Créer de même les 2 autres contrôleurs Web pour les entités `Patient` et `Medecin`. ➢ Développer les 3 contrôleurs Rest correspondants aux 3 entités.

Etape 6 : Conversion entre le type LocalDateTime et le format JSON Dans une application API Rest, il est très courant de faire appel à la conversion des objets de la classe LocalDateTime en format JSON et vice-versa.

Dans l'entité `Rdv`, utiliser l'annotation `@JsonFormat` pour préciser le format de conversion entre des objets de type `LocalDateTime` et JSON :

```
@JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss", shape = JsonFormat.Shape.STRING)
private LocalDateTime dateRdv;
```

Le format de l'attribut `dateRdv` est spécifié dans l'attribut `pattern`. ➢ Tester les API Rest des 3 contrôleurs pour gérer (en ajout et affichage) des patients, médecins et Rdv. Régler le problème de la boucle itérative entre les entités `Rdv - Patient` et `Rdv - Medecin` avec l'annotation `@JsonIgnore` sur la liste des Rdvs. ➢ Rechercher et ajouter la dépendance '`Springdoc Openapi starter`' pour la documentation Swagger. L'URL est : `http://localhost:8080/swagger-ui.html`

Etape 7 : les requêtes dérivées des méthodes (Derived Query Methods)

Définition : Les requêtes dérivées des méthodes(Derived Query Methods) sont des méthodes mentionnées dans l'interface Repository et que Spring Data JPA va les traduire en requêtes de sélection avec des critères personnalisés.

Dans une interface Repository on peut déclarer les prototypes des requêtes dérivées (ce prototype de base correspond à `findByAttribut` ou `findAllByAttribut`)

Application 1 : on veut vérifier avant d'ajouter un nouveau Rdv d'un patient avec un médecin à une Date et heure donné que d'une part ce patient et d'autre part ce médecin n'ont pas chacun un rdv à cette même Date et heure.

Pour ce besoin, vous aller ajouter dans l'interface `RdvRepository` la signature de 2 méthodes l'une qui recherche un Rdv pour un patient (par son id) à une date et heure et une autre qui recherche un Rdv pour un médecin (par son id) à la même date et heure. Appeler ces deux requêtes dérivées dans la méthode `addRdv()` pour garantir la condition qu'un patient ou médecin ne peut pas avoir deux Rdv à la même Date et heure. Tester l'ajout d'un nouveau Rdv.

Application 2 : Retourner la liste des Rdvs ordonnée par la valeur croissante de la date et l'heure.

Application 3 : Rechercher la liste des Rdv d'un médecin dans une Date donné (2 paramètres d'une requête HTTP, la date choisie sera converti en un objet `LocalDate`).

Amen Ajroud 8 5ème année GL CA (EPI Sousse)