

# SPRING MVC



**UP ASI**  
**Bureau E204**

# Plan du Cours

- Spring MVC ( Définition + Spring web)
  - Serveur web vs. Serveur d'application
  - Les architectures physiques et logiques
  - Spring MVC + Postman
  - Postman
  - Dépendance Web
  - Cycle de Vie d'une requête HTTP (Spring Boot + Postman)
  - RestController
- TP Spring Boot + Spring Data JPA + Spring MVC (REST) + Postman

# Introduction

- Un **Conteneur de Servlets** (Servlet container en anglais) ou **Conteneur Web** (web container en anglais) est un logiciel qui exécute des servlets.
- Un ou une **Servlet** est une classe Java qui permet de créer dynamiquement des données au sein d'un serveur HTTP.
- Il existe plusieurs conteneurs de servlets, dont **Apache Tomcat** ou encore Jetty. Le serveur d'application JBoss Application Server(Wildfly) utilise Apache Tomcat.
- Nous allons nous intéresser au développement de la couche **Web** (Web Services REST + Contrôleur + Service + Repository) dans ce cours.
- Nous allons aussi pratiquer la consommation des services par Postman.

# Spring WEB

- Plusieurs Projets Spring permettent d'implémenter des applications Web :
- Framework Spring (qui contient Spring MVC)
- Spring Web Flow (Implémenter les navigations Stateful).
- Spring mobile (Détecter le type de l'appareil connecté).
- Spring Social (Facebook, Twitter, LinkedIn).
- ...
- Nous allons nous intéresser à **Spring MVC**.

# SPRING MVC

- **Spring MVC** est un Framework Web basé sur le design pattern **MVC** (Model / View / Controller).
- Spring MVC fait partie du projet “Spring Framework”.
- Spring MVC s'intègre avec les différentes technologies de vue tel que JSF, JSP, Velocity, Thymeleaf...
- Spring MVC n'offre pas une technologie de vue mais permet en revanche de communiquer avec toutes les technologies web les plus performantes telles que Angular, React, etc...
- Spring MVC est construit en se basant sur la spécification JavaEE : **Java Servlet**.

# Architecture Physique

- **Tier** est un mot anglais qui signifie étage ou niveau.
- Une application peut être **1-Tier**, **2-Tiers**, **3-Tiers** ou **N-Tiers**.

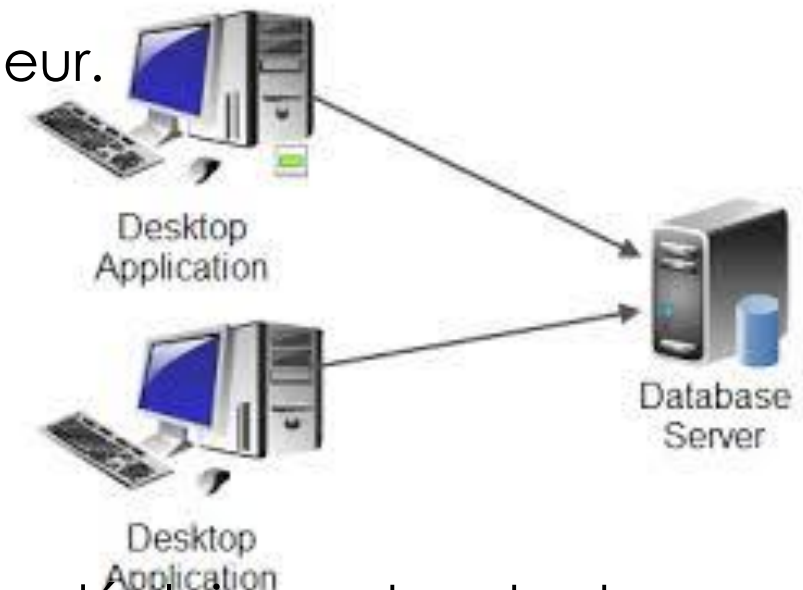
# Architecture Physique - 1-Tiers

- Une application **1-Tier** est, par exemple, la Modification d'un document Word sur un ordinateur Local.
- Tout est sur la même machine et les couches sont fortement liées.
- Inconvénients : Risque de perte des données (non sauvegardées à distance), Impossible d'accéder à une même ressource par deux utilisateurs en même temps.



# Architecture Physique - 2-Tiers

- Une application **2-Tiers** est typiquement une application **client lourd**.
- Le niveau **Présentation (IHM)** et le niveau **Traitement** sont sur la machine de l'utilisateur.
- Le niveau **Base de Données** est sur un autre serveur.
- C'est une architecture **Client / Serveur**.
- Client = demandeur de ressource
- Serveur = fournisseur de ressource



## Inconvénients

- Toute mise à jour des fonctionnalités nécessite un déploiement sur toutes les machines des utilisateurs.
- Le serveur ne fait pas appel à une autre application pour fournir le service.



# Architecture Physique - 3-Tiers

- Une application **3-Tiers** introduit un niveau intermédiaire ( middleware) entre le client et le serveur.
- Le niveau intermédiaire est chargé de fournir la ressource en faisant appel à un autre serveur.

## Avantages

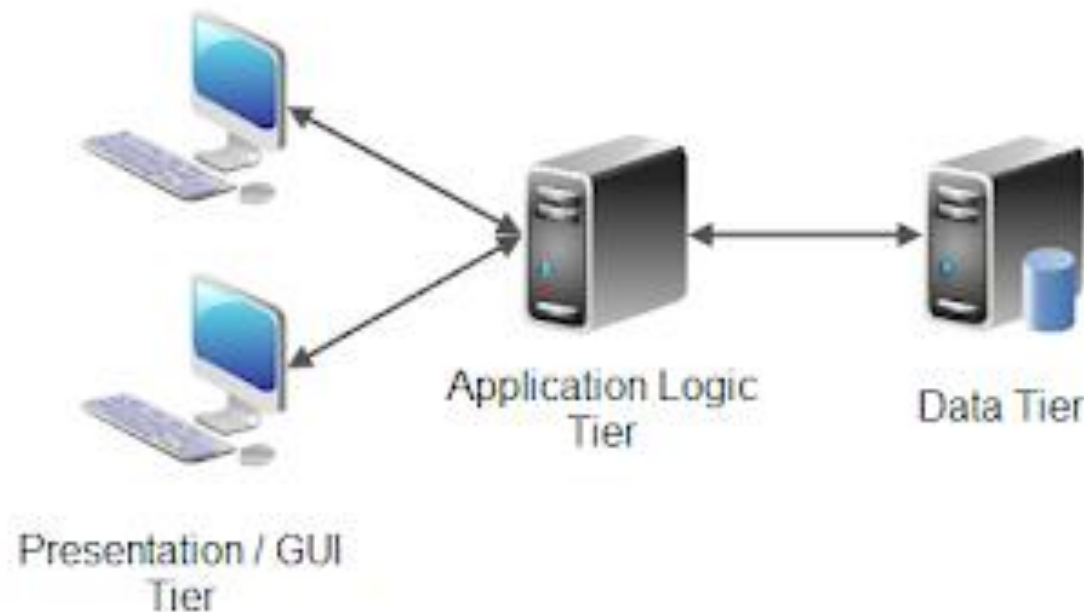
- Centraliser la logique application sur un serveur HTTP

## Inconvénients

- Le serveur HTTP ( élément principal de l'architecture )est fortement sollicité d'où une charge de demandes provenant à la fois du client et du serveur.
- Bien que cette architecture résout le problème du client lourd de l'architecture deux tiers, le soulagement du client est remplacé par un serveur fortement sollicité.

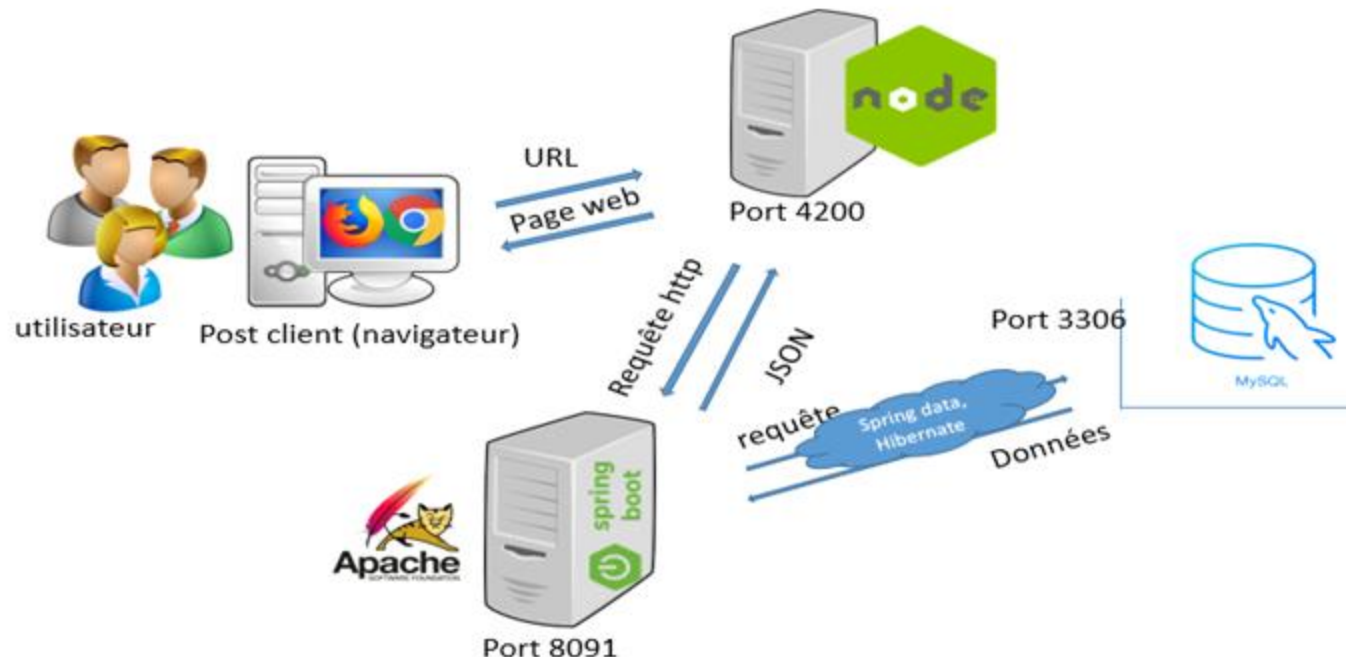
# Architecture Physique - 3-Tiers

- Une application **3-Tiers** est typiquement une application Web :
  - Niveau **Présentation** : IHM (Navigateur sur la machine de l'utilisateur)
  - Niveau **Traitement** : Un serveur web (Tomcat, ...) qui contient le WAR de notre application.
  - Niveau **Base de données** : Un serveur de BD qui stocke les données de notre application.

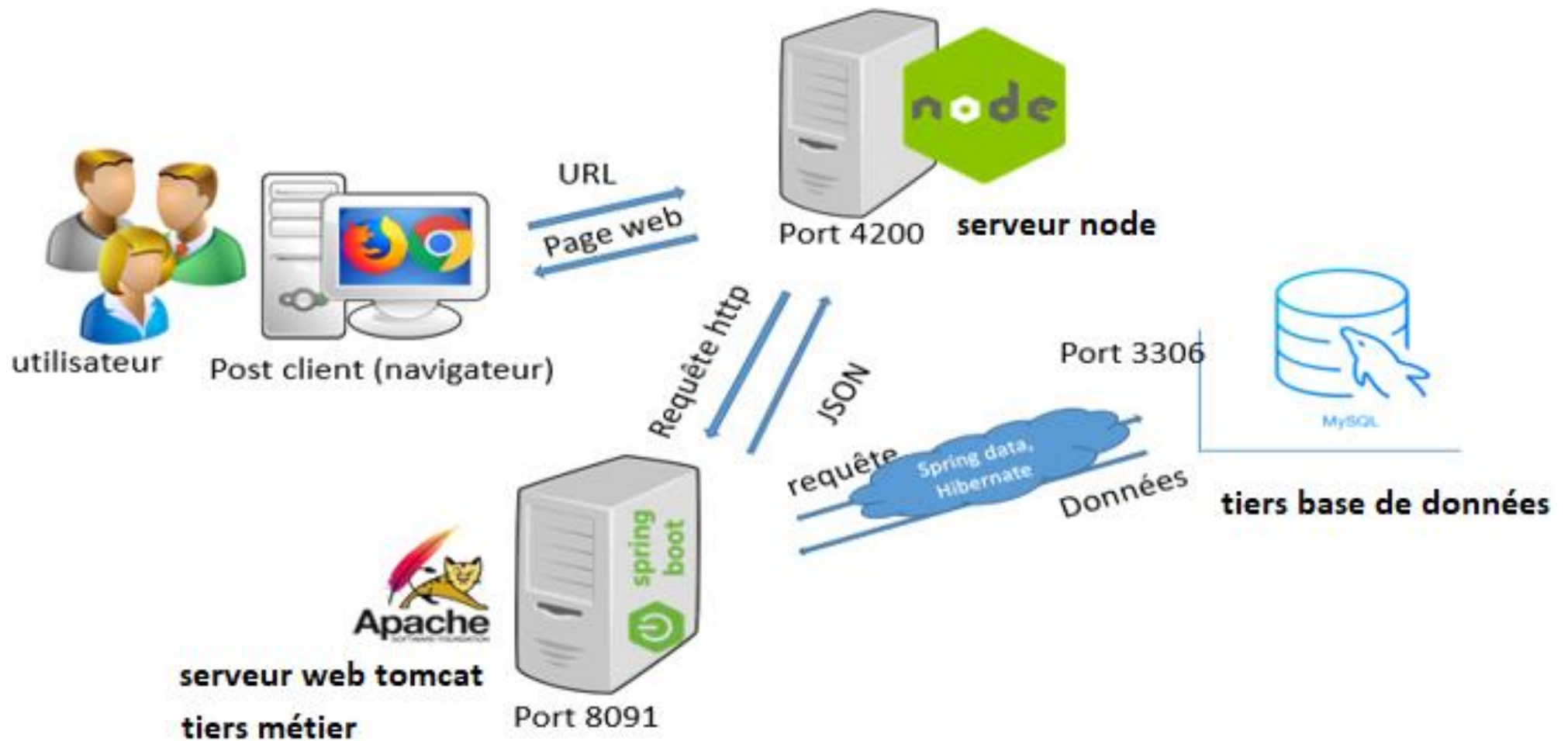


# Architecture Physique - N-Tiers

- L'architecture N tiers assure un équilibre de charge entre le client et le serveur par l'introduction de nouvelles couches.
- Voici une architecture 4-Tiers d'une application web développée par un étudiant Esprit pendant son projet de fin d'étude (GUI – Angular sur le Serveur NodeJS – Spring Boot (Serveur Web Tomcat embarqué) – Serveur de base de données MySQL) :

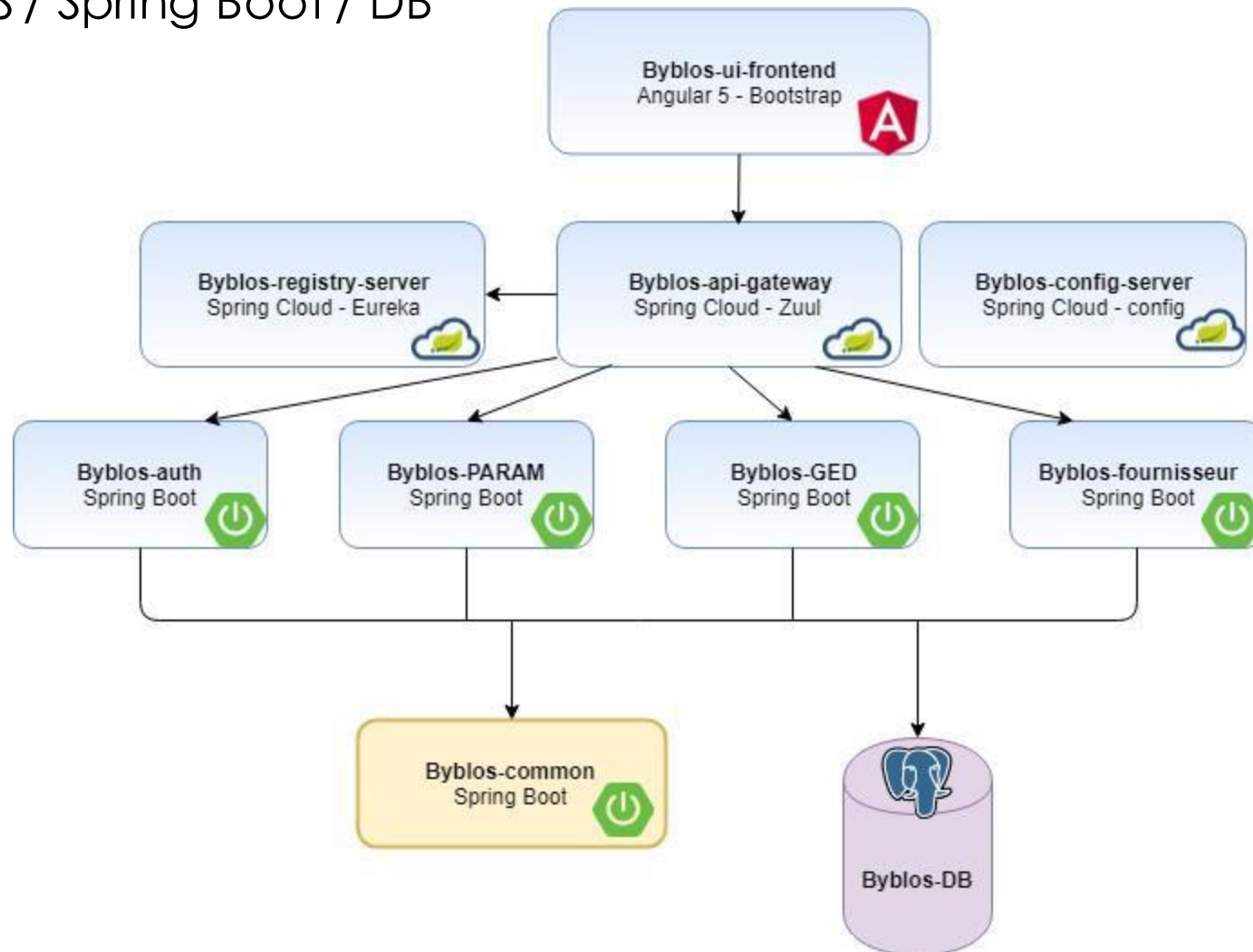


# Architecture Physique - N-Tiers



# Architecture Physique - N-Tiers

- Voici une architecture n-Tiers, **en Micro-Servcies**, d'une application web développée par un étudiant Esprit pendant son projet de fin d'étude : GUI / NodeJS / Spring Boot / DB

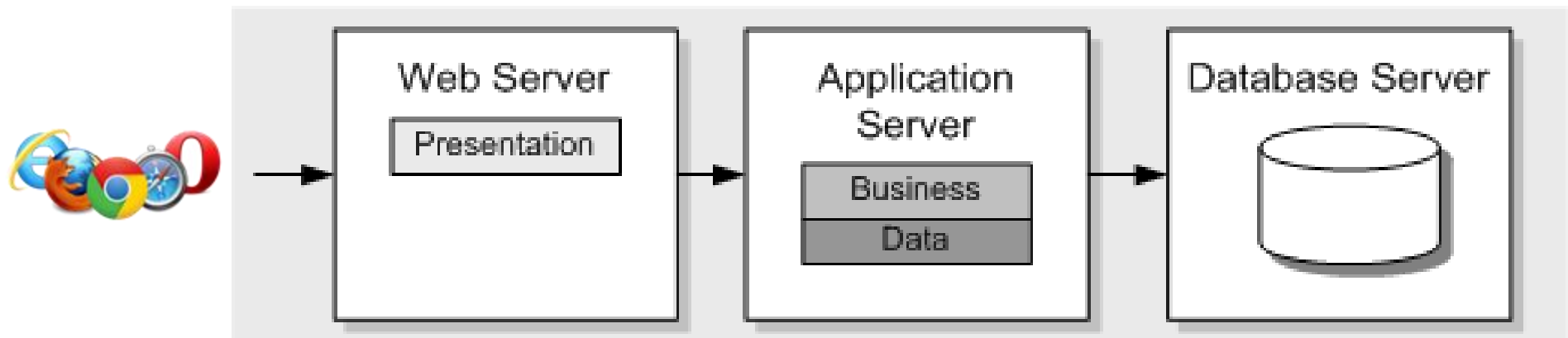


# Architecture logique

- Une application typique utilisant Spring est généralement structurée en couches (5 couches) :
  - Couche **Web** (non fournie par Spring)
  - Couche **Contrôleur**
  - ⇒ Couche **Présentation** (Web + Contrôleur)
  - Couche **Service** : interface métier avec mise en œuvre de certaines fonctionnalités.
  - Couche **Repository** (Recherche et Persistance des objets, CRUD)
  - Couche **Entités** (Modèle de données)
  - ⇒ Couche **Accès aux Données** (Repo (DAO) + Entités)
- **Spring est un Framework utilisé pour créer et injecter les objets requis pour communiquer entre les différentes couches.**

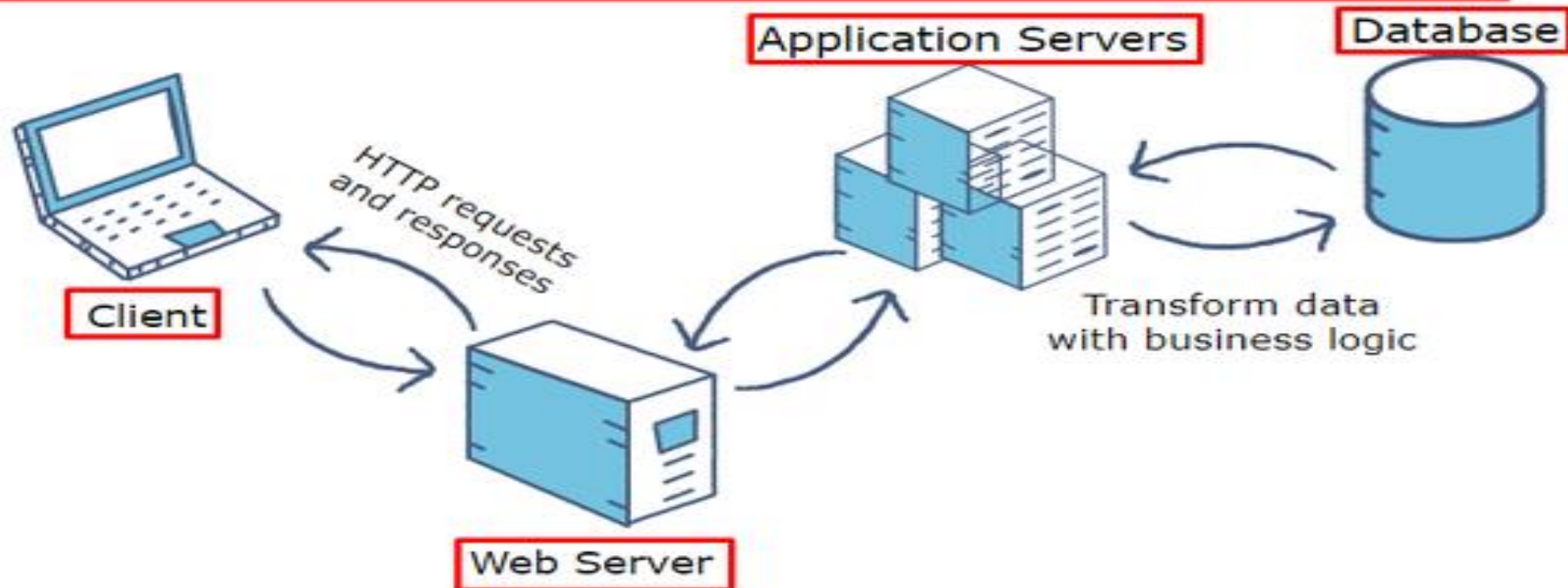
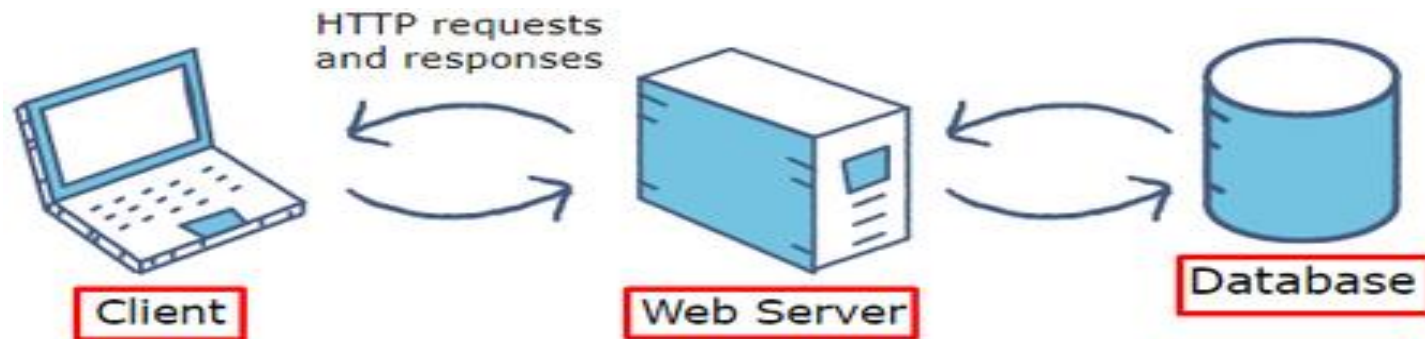
# Serveur Web vs Serveur d'Application

Serveur Web	Serveur d'application JavaEE Serveur web + container
Héberge que la couche présentation et l'expose qu'à travers le protocole HTTP(S)	Héberge la logique métier et peut aussi héberger la couche présentation (supporte différents protocoles : HTTP, JNDI, ...).
Ne peut pas inclure un EJB Container.	Doit inclure un EJB Container.
lightweight	Relativement gourmand en ressources (CPU, RAM et Disk).
Exp : Apache HTTP Server, Tomcat, Jetty ....	Exp : Wildfly, WebSphere ...





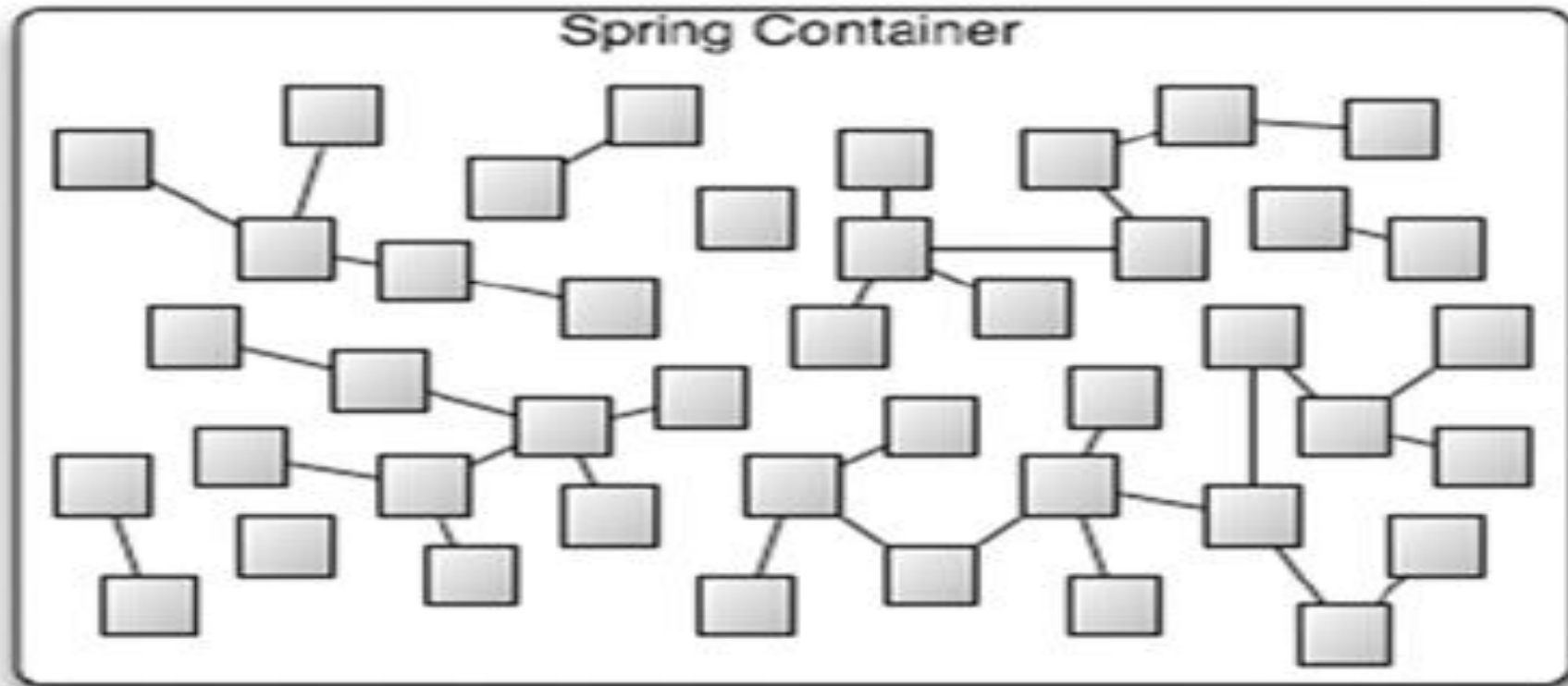
# Serveur Web vs Serveur d'Application





# Serveur Web vs Serveur d'Application

## Spring IOC Container



Dans une application Spring, les objets sont créés, sont liés ensemble et communiquent dans le Spring IOC Container.

# Spring MVC + Postman

# Postman

- Parmi les nombreuses solutions pour interroger ou tester les web services et les API, Postman propose de nombreuses fonctionnalités, une prise en main rapide et une interface graphique agréable.
- Postman permet de construire et d'exécuter des requêtes HTTP, de les stocker dans un historique afin de pouvoir les rejouer.



# Postman

The screenshot displays the Postman application interface. At the top, the breadcrumb navigation shows 'tpAchat / opérateur / modifier opérateur'. The main area is configured for a PUT request to 'http://localhost:8089/SpringMVC/opérateur/modify-operateur'. The 'Body' tab is selected, showing a JSON payload: 

```
{
  "idOperateur": 1,
  "nom": "Lahiani",
  "prenom": "ahmed",
  "password": "pwd1"
}
```

. The status bar at the bottom indicates a successful response: 'Status: 200 OK', 'Time: 318 ms', and 'Size: 232 B'. The response body is also shown in the 'Body' tab, displaying the same JSON payload in a pretty-printed format.

tpAchat / opérateur / modifier opérateur

Save

PUT http://localhost:8089/SpringMVC/opérateur/modify-operateur Send

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded **raw** binary GraphQL JSON Beautify

```
1 {
2   ... "idOperateur": 1,
3   ... "nom": "Lahiani",
4   ... "prenom": "ahmed",
5   ... "password": "pwd1"
6 }
```

Body Cookies Headers (5) Test Results Status: 200 OK Time: 318 ms Size: 232 B Save Response

Pretty Raw Preview Visualize JSON

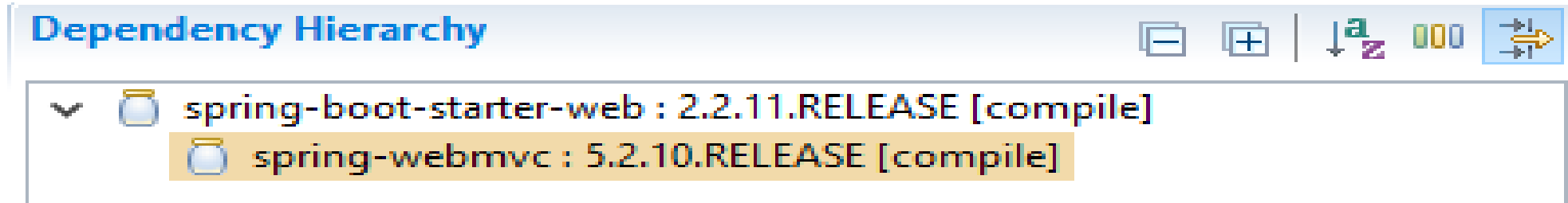
```
1 {
2   "idOperateur": 1,
3   "nom": "Lahiani",
4   "prenom": "ahmed",
5   "password": "pwd1"
6 }
```

Activer Windows



# Dépendance web

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

**Dependency Hierarchy**

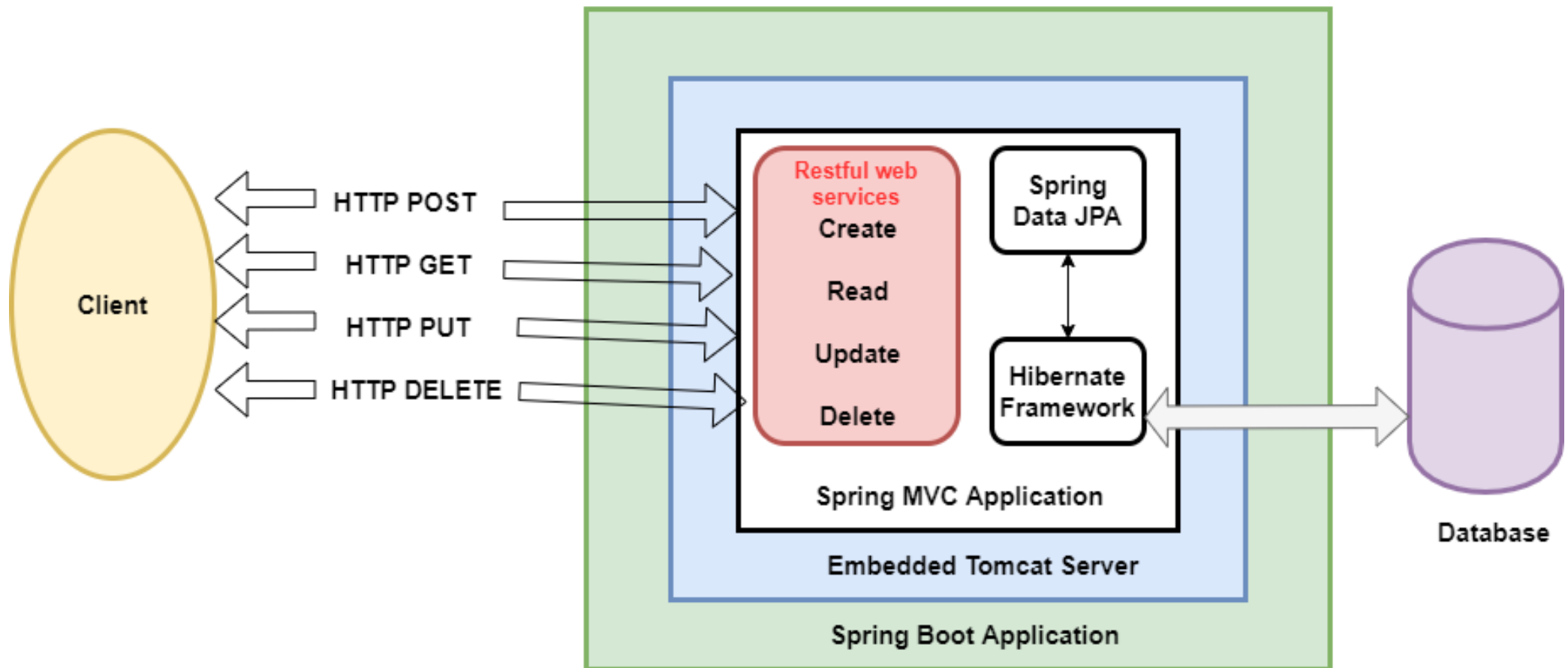


The screenshot shows the 'Dependency Hierarchy' view in an IDE. It features a toolbar with icons for zooming in/out, sorting, and refreshing. The main area shows a tree structure where 'spring-boot-starter-web : 2.2.11.RELEASE [compile]' is expanded, revealing its dependency on 'spring-webmvc : 5.2.10.RELEASE [compile]'.

- ▼  spring-boot-starter-web : 2.2.11.RELEASE [compile]
  -  spring-webmvc : 5.2.10.RELEASE [compile]

- Le starter web permet d'ajouter toutes les dépendances liées à la partie web notamment celles liées à Spring MVC et l'exposition des web services.

# Cycle de Vie d'une requête HTTP (Spring Boot+Postman)



# Url de notre Application Web

- Dans le fichier de propriétés application.properties, ajouter les lignes suivantes, pour définir l'url de notre application :

```
#Server configuration
```

```
server.port=8089
```

```
server.servlet.context-path=/SpringMVC
```

# RestController

```
@RestController
@RequestMapping("/projet-detail")
public class ProjetDetailRestController {

    @Autowired
    IProjetDetailService projetDetailService;

    // http://localhost:8089/SpringMVC/projet-detail/retrieve-all-projetdetails
    @GetMapping("/retrieve-all-projetdetails")
    public List<ProjetDetail> getProjetDetails() {
        List<ProjetDetail> listProjetDetails = projetDetailService.retrieveAllProjetDetails();
        return listProjetDetails;
    }
}
```



# TP - Spring : Boot – Core – Data JPA – MVC (REST)

- Nous allons commencer par exposer des Web Service REST : Spring : Boot – Core – Data JPA – MVC (**REST**) -Postman
- Vous avez déjà créé un projet : Spring (Boot – Core – Data JPA) avec un CRUD sur l'entité Client. Ce projet a été testé avec JUnit.
- Nous allons reprendre le même projet et exposer ces méthodes (CRUD) avec des Web Service REST.
- Ces Web Services seront testé avec **Postman**.

# TP - Spring : Boot – Core – Data JPA – MVC (REST)

- Installation de Postman :
- L'exécutable est sur le **Drive** du cours Spring (dossier **Outils**), à télécharger et à installer.



# TP - Spring : Boot – Core – Data JPA – MVC (REST)

- Vérifier que le fichier de propriétés contient les propriétés nécessaires (web, base de données, ...) :

```
#Server configuration
```

```
server.servlet.context-path=/SpringMVC
```

```
server.port=8089
```

```
### DATABASE ###
```

```
spring.datasource.url=jdbc:mysql://localhost:3306/springdb?useUnicode=true
```

```
&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC
```

```
spring.datasource.username=root
```

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

```
### JPA / HIBERNATE ###
```

```
spring.jpa.show-sql=true
```

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

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
```

# TP - Spring : Boot – Core – Data JPA – MVC (REST)

- Créer le package **tn.esprit.spring.control**
- Créer le bean Spring **ProjetDetailRestController** annoté **@RestController**
- Créer les méthodes nécessaires pour exposer le CRUD (voir pages suivantes).
- **Nous allons commencer par exposer des Web Services vides (sans appels à la couche Service) :**

# TP - Spring : Boot – Core – Data JPA – MVC (REST)

```
@RestController
@RequestMapping("/projet-detail")
public class ProjetDetailRestController {

    // http://localhost:8089/SpringMVC/projet-detail/retrieve-all-projetdetails
    @GetMapping("/retrieve-all-projetdetails")
    public List<ProjetDetail> getProjetDetails() {
        return null;
    }
}
```

# TP - Spring : Boot – Core – Data JPA – MVC (REST)

// <http://localhost:8089/SpringMVC/projet-detail/retrieve-proejt-detail/8>

**@GetMapping**("/[retrieve-proejt-detail/{projet-detail-id}](#)")

```
public ProjetDetail retrieveProjetDetail( @PathVariable("projet-detail-id") Long projetDetailId) {  
    return null;  
}
```

// <http://localhost:8089/SpringMVC/projet-detail/add-projet-detail>

**@PostMapping**("/[add-projet-detail](#)")

```
public ProjetDetail addProjetDetail( @RequestBody ProjetDetail o)  
{  
    return null;  
}
```

# TP - Spring : Boot – Core – Data JPA – MVC (REST)

```
// http://localhost:8089/SpringMVC/projet-detail/remove-proejt-detail/{projet-detail-id}
@DeleteMapping("/remove-proejt-detail/{projet-detail-id}")
public void removeProjetDetail(@PathVariable("projet-detail-id") Long projetDetailId) {

}
```

```
// http://localhost:8089/SpringMVC/projet-detail/modify-proejt-detail
@PutMapping("/modify-proejt-detail")
public ProjetDetail modifyProjetDetail(@RequestBody ProjetDetail projetDetail) {
return null;
}
```

# TP - Spring : Boot – Core – Data JPA – MVC (REST)

- Maintenant nous allons créer le Service `ProjetDetailServiceImpl` et l'injecter dans le contrôleur.
- Créer le package **`tn.esprit.spring.service`**
- Créer le bean Spring **`ProjetDetailServiceImpl`** annoté **`@Service`** et l'interface **`ProjetDetailService`**
- Créer le CRUD dans le Service, Puis Créer les méthodes nécessaires pour exposer le CRUD (voir pages suivantes) :



# TP - Spring : Boot – Core – Data JPA – MVC (REST)

@Service

```
public class ProjetDetailServiceImpl {
```

@Autowired

```
ProjetDetailRepository projetDetailRepository;
```

```
public List<ProjetDetail> retrieveAllProjetDetails()
```

```
{
```

```
    return projetDetailRepository.findAll();
```

```
}
```

```
public ProjetDetail retrieveProjetDetail(Long projetDetailId)
```

```
{
```

```
    return projetDetailRepository.findById(projetDetailId).get();
```

```
}
```

# TP - Spring : Boot – Core – Data JPA – MVC (REST)

```
public ProjetDetail addProjetDetail(ProjetDetail p)
{
    return projetDetailRepository.save(p);
}
```

```
public void removeProjetDetail(Long projetDetailId)
{
    projetDetailRepository.deleteById(projetDetailId);
}
```

```
public ProjetDetail modifyProjetDetail(ProjetDetail projetDetail)
{
    return projetDetailRepository.save(projetDetail);
}
```

# TP - Spring : Boot – Core – Data JPA – MVC (REST)

```
@RestController
@RequestMapping("/projet-detail")
public class ProjetDetailRestController {

    @Autowired
    IProjetDetailService projetDetailService;

    // http://localhost:8089/SpringMVC/projet-detail/retrieve-all-projetdetails
    @GetMapping("/retrieve-all-projetdetails")
    public List<ProjetDetail> getProjetDetails() {
        List< ProjetDetail > listProjetDetails = projetDetailService.retrieveAllProjetDetails();
        return listProjetDetails;
    }
}
```

# TP - Spring : Boot – Core – Data JPA – MVC (REST)

// <http://localhost:8089/SpringMVC/projet-detail/retrieve-proejt-detail/8>

**@GetMapping**("/retrieve-proejt-detail/{projet-detail-id}")

```
public ProjetDetail retrieveProjetDetail(@PathVariable("projet-detail-id") Long projetDetailId) {  
    ProjetDetail projetDetail = projetDetailService.retrieveProjetDetail(projetDetailId);  
    return projetDetail;  
}
```

// <http://localhost:8089/SpringMVC/projet-detail/add-proejt-detail>

**@PostMapping**("/add-proejt-detail")

```
public ProjetDetail addProjetDetail(@RequestBody ProjetDetail p)  
{  
    ProjetDetail projetDetail = projetDetailService.addProjetDetail(p);  
    return projetDetail;  
}
```

# TP - Spring : Boot – Core – Data JPA – MVC (REST)

```
// http://localhost:8089/SpringMVC/projet-detail/remove-proejt-detail/{projet-detail-id}
@DeleteMapping("/remove-proejt-detail/{projet-detail-id}")
public void removeProjetDetail(@PathVariable("projet-detail-id") Long projetDetailId) {
    projetDetailService.removeProjetDetail(projetDetailId);
}
```

```
// http://localhost:8089/SpringMVC/projet-detail/modify-proejt-detail
@PutMapping("/modify-proejt-detail")
public ProjetDetail modifyProjetDetail(@RequestBody ProjetDetail p) {
    ProjetDetail projetDetail = projetDetailService.modifyProjetDetail(p);
    return projetDetail;
}
```

# TP - Spring : Boot – Core – Data JPA – MVC (REST)

- N'oubliez pas d'ajouter les getters et les setters au niveau des entités (en attendant le cours Lombok qui permettra de ne plus les écrire).
- Tester l'ensemble des méthodes exposées avec Postman

# TP - Spring : Boot – Core – Data JPA – MVC (REST)

The screenshot displays a REST client interface with the following components:

- Method:** PUT
- URL:** http://localhost:8089/SpringMVC/client/modify-client
- Request Body:** A JSON object representing a client:

```
{  "nom": "Nasri",  "prenom": "Ahmed",  "dateNaissance": "1995-05-04",  "email": "nasri.ahmed@gmail.tn",  "password": "pwd1",  "profession": "Ingenieur",  "categorieClient": "Fidele"}
```
- Response:** A JSON object representing the result of the update:

```
{  "idClient": 1,  "nom": "Nasri",  "prenom": "Ahmed",  "dateNaissance": "1995-05-04T00:00:00.000+00:00",  "email": "nasri.ahmed@gmail.tn",  "password": "pwd1",}
```

Additional interface details include tabs for Params, Authorization, Headers (9), Body, Pre-request Script, Tests, and Settings. The response status is 200 OK, with a time of 903 ms and a size of 358 B.

# Liste des fournisseurs ( navigateur + Postman )

← → ↻ ⓘ localhost:8089/SpringMVC/fournisseur/retrieve-all-fournisseurs



```
[{"idFournisseur":12,"code":"f01","libelle":"food Services","categorieFournisseur":"ORDINAIRE","detailFournisseur":  
{"idDetailFournisseur":6,"email":"foodServices@gmail.com","dateDebutCollaboration":"2019-01-01","adresse":"mnhla tunis","matricule":"MF1523"}}]
```

```
[  
  {  
    "idFournisseur": 12,  
    "code": "f01",  
    "libelle": "food Services",  
    "categorieFournisseur": "ORDINAIRE",  
    "detailFournisseur": {  
      "idDetailFournisseur": 6,  
      "email": "foodServices@gmail.com",  
      "dateDebutCollaboration": "2019-01-01",  
      "adresse": "mnhla tunis",  
      "matricule": "MF1523"  
    }  
  }  
]
```