

Architecture Backend



Comment est structuré un projet Web?

Architecture Monolithique



Architecture Monolithique

Tout ce fait dans une seule application.

- l'affichage des pages
 - le traitement des données
 - la gestion de la persistance
-

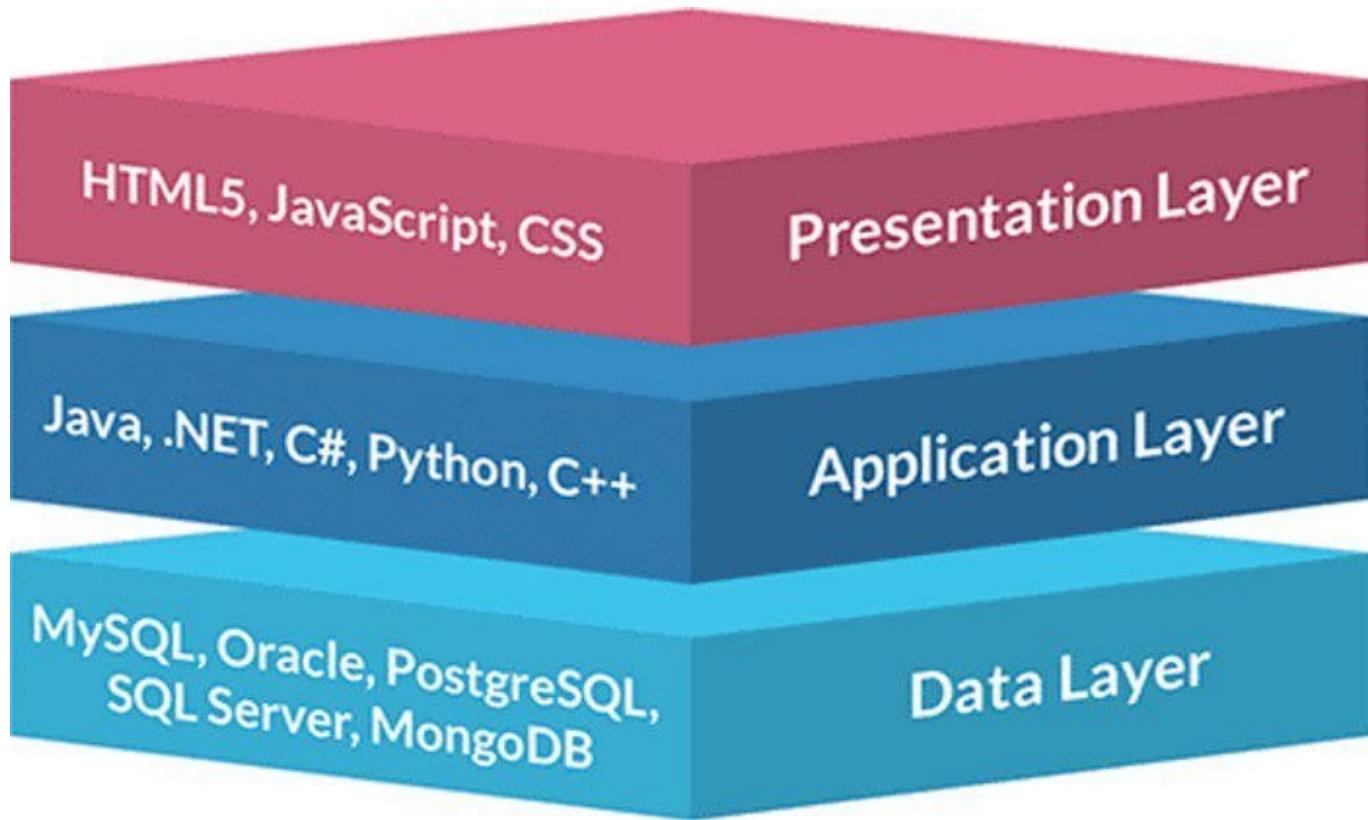
Spring boot et l'architecture Monolithique

Spring boot permet de faire des architectures Monolithique.

Mais...

Ce n'est plus ce que nous faisons

Architecture 3 tiers



Architecture 3 tiers

Nous utilisons trois applications au lieu d'une:

- **le frontend**: l'affichage des pages exécuté dans le navigateur
 - **le backend**: le traitement des données exécuté sur le serveur
 - **la base de données**: la gestion de la persistance
-

Le frontend

Java n'est pas le meilleur langage pour faire des applications web.

Il est plus adapté d'utiliser JavaScript pour faire des applications web.

La base de données

Il existe plusieurs types de bases de données:

- **SQL** : MySQL, PostgreSQL, Oracle, SQL Server, ...
 - **NoSQL** : MongoDB, Cassandra, Redis, ...
-

Le backend

C'est ici que nous allons nous concentrer.

Spring boot est un framework adapté pour la création d'application backend.

Le backend

Le backend est le cœur de l'application.



Le rôle du backend

Le backend est responsable de:

- la gestion des données
 - la gestion du métier
-

Le backend et le frontend

Le frontend envoie des requêtes au backend grâce à des **API**.

Une **API** est une interface de programmation qui permet de communiquer avec un autre programme.

Communication entre Backend et Frontend

Pour communiquer entre le Front et le Back nous allons avoir besoin d'utiliser des protocoles

Un protocole est un ensemble de règles et de codes de langage qui définissent comment se déroule la communication

L'architecture 3 couches

L'architecture 3 couches est une architecture logicielle qui sépare les données, la logique métier et l'interface utilisateur.

3 couches

Les couches

- **Contrôleur:** c'est la couche qui reçoit les requêtes HTTP et qui les transmet à la couche métier.
 - **Service:** c'est la couche qui contient la logique métier.
 - **Repository:** c'est la couche qui contient la logique d'accès aux données.
-

Les contrôleurs

Les contrôleurs sont des classes qui reçoivent les requêtes HTTP et qui les transmettent à la couche métier.

Ils se chargent de retourner la réponse HTTP.

Ils ne font pas de connexions à la base de données ou de logique métier.

Les services

Les services sont des classes qui contiennent la logique métier.

Ils ne font pas de connexions à la base.

Les repositories

Les repositories sont des classes qui contiennent la logique d'accès aux données.

Ils ne font pas de logique métier.

Analogie du restaurant

L'on peut voir l'architecture 3 couches comme un restaurant.



Le serveur

Le serveur est là pour prendre vos commandes et les transmettre à la cuisine.

Il est aussi là pour vous servir votre plat.

Mais il n'est pas là pour cuisiner ou gérer les stocks.

La cuisine

La cuisine est là pour cuisiner les plats.

Elle ne s'occupe pas de prendre les commandes ou de servir les plats.

Elle ne s'occupe pas non plus de gérer les stocks.

Le stock

Le stock est là pour gérer les ressources et les entreposer.

Le stock ne s'occupe pas de prendre les commandes ou de servir les plats.

Le stock ne s'occupe pas non plus de cuisiner.

Chaqu'un à sa place

ZONE FRAICHE 4-6°C

- Légumes cuits
- Viandes et poissons cuits
- Laitages, desserts lactés
- Fromages affinés emballés
- Fromages à pâte cuite

ZONE FROIDE 0-4°C

- Viandes crues emballées
- Charcuteries emballées
- Poisson cru emballé
- Salade en sachet
- Pâtisseries fraîches

BAC À LÉGUMES >6°C

- Légumes frais
- Fromages en cours d'affinage



PORTE 6-10°C

- Beurre
- Lait
- Jus de fruits entamés bien refermés
- Confitures
- Condiments
- Sauces en pot



Qu'est ce que Spring Spring boot.

Spring boot est une surcouche de Spring.

Il permet de démarrer rapidement un projet **Spring** en fournissant des configurations par défaut.

Spring vs Spring boot

Spring ne fournit que les composants de base.

Il faut entièrement configurer le projet ce qui peut être fastidieux.

Spring boot fournit des configurations par défaut.

Il permet de rapidement créer un projet **web** mais pas que.

Spring boot

Spring Boot propose un ensemble de modules pré-configurés.

Voici quelques modules :

- **Spring Web** : pour créer des applications web
 - **Spring Data** : pour accéder à des bases de données
 - **Spring Security** : pour sécuriser les applications
 - **Spring Cloud** : pour créer des applications distribuées
-

Spring boot, un serveur web?

Spring boot n'est pas un serveur web. **Spring boot** ne sait rien faire par défaut. Il faut lui ajouter des dépendances pour lui permettre de faire des choses.

Pour faire du **web** vous pouvez ajouter la dépendance **spring-boot-starter-web**.

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

Spring boot, connexion à une base de données?

Spring boot ne sait pas se connecter à une base de données.

Il y a plusieurs dépendances pour se connecter à une base de données:

- **spring-boot-starter-jdbc**: connecter à une base de données relationnelle avec JDBC.
- **spring-boot-starter-data-jpa**: connecter à une base de données relationnelle avec JPA.
- **spring-boot-starter-data-mongodb**: connecter à une base de données NoSQL avec MongoDB.
- **spring-boot-starter-data-redis**: connecter à une base de données NoSQL avec Redis.
- **spring-boot-starter-data-elasticsearch**: connecter à une base de données NoSQL avec Elasticsearch.

- ...
-

Qu'est ce que Spring boot web

Spring boot web est un module de Spring boot qui permet de créer des applications web.

Sans lui, spring boot ne permet pas de créer des applications web.

Dépendances

Pour utiliser Spring boot web, il faut ajouter les dépendances suivantes dans le fichier pom.xml :

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

Les controllers

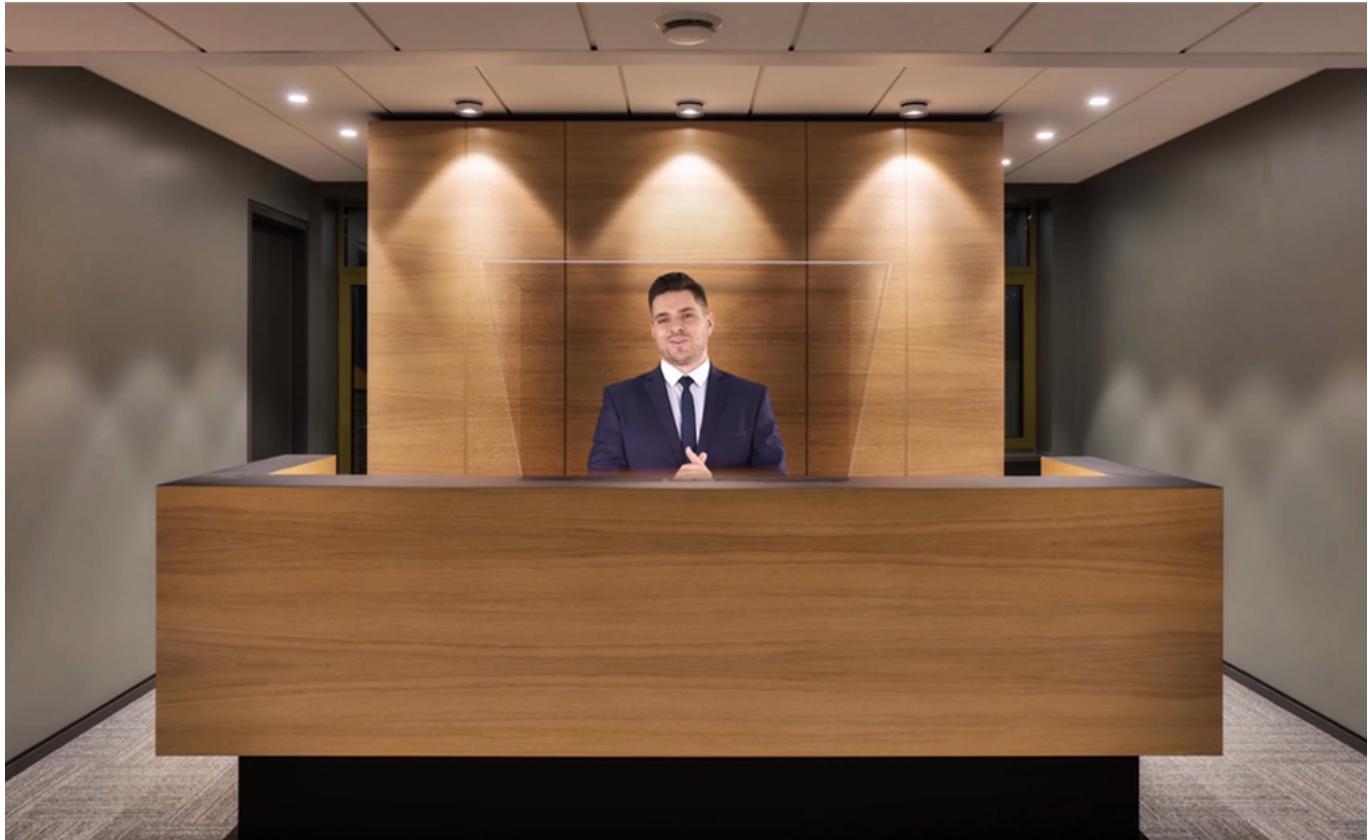
Les controllers sont des classes qui permettent de gérer les requêtes HTTP.

Elles ont les rôles suivants :

- Définir les routes
 - Récupérer les informations de la requête
 - Envoyer une réponse
-

Son rôle

Être l'accueil de l'application web.



Création d'un controller

Pour créer un controller, il faut créer une classe annoté par `org.springframework.web.bind.annotation.RestController`.

Par convention, le nom de la classe doit se terminer par `Controller`.

```
@RestController  
public class HelloController {  
    // ...  
}
```

@RestController

`@RestController` est une annotation qui permet de définir une classe comme controller.

C'est une annotation qui `hérite` de `@Component`.

La classe doit être dans le même package (ou package enfant) que la classe principale.

@RequestMapping

Dans un controller, il est possible de définir les routes.

`@RequestMapping` est une annotation qui permet de définir une route.

Dans l'exemple suivant, la route est `/hello` retourne le message `Hello World`.

```
@RestController
public class HelloController {
    @RequestMapping("/hello")
    public String hello() {
        return "Hello world";
    }
}
```

Spécifier le type de requête

Une méthode annotée avec `@RequestMapping` peut être appelée pour n'importe **verbe**(GET, POST, PUT, ...).

Pour spécifier le type de requête, il faut utiliser les annotations de type `XXXMapping` :

- `GetMapping` pour les requêtes `GET`
- `PostMapping` pour les requêtes `POST`
- `PutMapping` pour les requêtes `PUT`
- `DeleteMapping` pour les requêtes `DELETE`

Exemple

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        return "Hello Getter";
    }

    @PostMapping("/hello")
    public String helloPost() {
        return "Hello Poster";
    }
}
```

@RequestMapping sur un controller

`@RequestMapping` peut être utilisé sur un controller pour définir une route par défaut.

Dans l'exemple suivant, la route par défaut est `/parent`.

```
@RestController
@RequestMapping("/parent")
```

```

public class HelloController {
    // Appelé pour GET /parent/hello
    @GetMapping("/hello")
    public String hello() {
        return "Hello Getter";
    }

    // Appelé pour POST /parent/hello
    @PostMapping("/hello")
    public String helloPost() {
        return "Hello Poster";
    }
}

```

Les annotations de paramètres

@RequestMapping et #XXXMapping permettent d'exécuter une méthode en fonction de la route.

Pour récupérer des informations de la requête, il faut utiliser les annotations suivantes :

- @PathVariable pour récupérer une partie de la route
- @RequestParam pour récupérer un paramètre de la requête
- @RequestBody pour récupérer le corps de la requête

@PathVariable

@PathVariable permet de récupérer une partie de la route.

Dans l'exemple suivant, la route est /hello/{name}.

```

@RestController
public class HelloController {
    /*
     * Ici quand le client fait une requête GET sur /hello/John, le paramètre name
     * vaut "John"
     */
    @GetMapping("/hello/{name}")
    public String hello(@PathVariable String name) {
        return "Hello " + name;
    }
}

```

@RequestParam

@RequestParam permet de récupérer un paramètre de la requête.

Dans l'exemple suivant, la route est /hello et le paramètre name est récupéré.

```
@RestController
public class HelloController {
    /*
    Ici quand le client fait une requête GET sur /hello?name=John, le paramètre
    name vaut "John"
    */
    @GetMapping("/hello") // /hello?name=John
    public String hello(@RequestParam String name) {
        return "Hello " + name;
    }
}
```

@RequestParam avec valeur par défaut

@RequestParam demande que le paramètre soit présent dans la requête.

Pour définir une valeur par défaut, il faut utiliser l'attribut `defaultValue`.

```
@RestController
public class HelloController {
    /*
    Ici quand le client fait une requête GET sur /hello, le paramètre name vaut
    "World"
    */
    @GetMapping("/hello")
    public String hello(@RequestParam(defaultValue = "World") String name) {
        return "Hello " + name; // = Hello World
    }
}
```

@RequestParam optionnel

@RequestParam peut être utilisé sans valeur.

Dans ce cas, le paramètre est optionnel.

```
@RestController
public class HelloController {
    /*
    Ici quand le client fait une requête GET sur /hello, le paramètre name vaut
    null
    */
    @GetMapping("/hello")
    public String hello(@RequestParam(required = false) String name) {
        return "Hello " + name; // = Hello null
    }
}
```

```
    }  
}
```

@RequestBody

@RequestBody permet de récupérer le corps de la requête.

Dans l'exemple suivant, la route est `/hello` et le corps de la requête est récupéré.

```
@RestController  
public class HelloController {  
    /*  
     * Ici quand le client fait une requête POST sur /hello avec le corps "John", le  
     * paramètre name vaut "John"  
     */  
    @PostMapping("/hello")  
    public String hello(@RequestBody String name) {  
        return "Hello " + name;  
    }  
}
```

Sérialisation et deserialisation



Sérialisation

La **Sérialisation** est le processus de conversion d'un objet en une séquence de bits.

Les objets java sont représenté en mémoire de manière complexe. Pour les envoyer au client, il faut les convertir en une séquence de bits.

Exemple

```
class Personne {  
    private String name;  
    private int age;  
  
    public Personne(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    // Getters et Setters  
}
```

Une instance de cette classe est représentée en mémoire avec des références vers des objets.

Exemple

```
Personne p = new Personne("John", 42);
```

peut être représenté en Json de la manière suivante :

```
{  
    "name": "John",  
    "age": 42  
}
```

Et le format **JSON** est une séquence de bits qui peut facilement être envoyé au client.

C'est la **Sérialisation** de l'instance de la classe **Personne** en **JSON**.

Deserialisation

La **Deserialisation** est le processus inverse de la **Sérialisation**.



Deserialisation

Il est nécessaire de déserialiser les objets pour les récupérer depuis le client.

Le plus souvent, on utilise le format **JSON** pour la deserialization.

Exemple

```
{  
    "name": "John",  
    "age": 42  
}
```

peut être désérialisé en une instance de la classe **Personne**.

```
Personne personne = Deserialiseur.deserialize(json, Personne.class);  
personne.getName(); // = John  
personne.getAge(); // = 42
```

Sérialisation et deserialization avec Spring

Spring boot utilise **Jackson** pour la sérialisation et la deserialization au format **JSON**.

Pour le bon fonctionnement de la sérialisation et de la deserialization, il faut que:

- les classes soient **public**
 - les getters et setters soient **public**
 - Qu'il y ai un constructeur sans paramètre public
 - Qu'il n'y ait pas de boucle dans les objets (Personne -> Personne -> Personne -> ...)
-

Sérialisation Jackson

Jackson ne sérialise que les méthodes **getter**.

```
class Personne {  
    private String name;  
    private int age;  
  
    public Personne(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setAge(int age) {  
        return age;  
    }  
}
```

sera sérialisé en **JSON** de la manière suivante :

```
{  
    "name": "John"  
}
```

Deserialisation Jackson

Jackson ne désérialise que les méthodes **setter**.

```
{  
    "name": "John",  
    "age": 42  
}
```

Seul le champ **age** sera affecté lors de la désérialisation s'il n'y a pas de setter sur **name**.

Sérialisation avec @RequestBody

L'annotation `@RequestBody` permet de déserialiser le corps de la requête.

```
@RestController
public class HelloController {
    @PostMapping("/persons")
    public String hello(@RequestBody Person person) {
        return "Hello " + person.getName();
    }
}
```

Désérialisation de la réponse

Lors du retour d'une réponse, `Spring boot` sérialise automatiquement l'objet en `JSON`.

```
@RestController
public class HelloController {
    @GetMapping("/persons")
    public Person hello() {
        return new Person("John", 42);
    }
}
```

Gestion des status

`Spring boot web` permet de gérer les status de la réponse.

L'annotation @ResponseStatus

L'annotation `@ResponseStatus` permet de définir le status de la réponse.

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    @ResponseStatus(HttpStatus.OK)
    public String hello() {
        return "Hello World";
    }
}
```

L'annotation @ResponseStatus

L'annotation `@ResponseStatus` prend en paramètre un objet `HttpStatus` qui permet de définir le status de la réponse.

L'annotation @ResponseStatus sur les exceptions

L'annotation `@ResponseStatus` peut être utilisée sur les exceptions.

Dans ce cas, si l'annotation est retournée, spring boot web retourne le status défini.

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        throw new HelloException();
    }
}

@ResponseStatus(HttpStatus.NOT_FOUND)
class HelloException extends RuntimeException {
```

Les exceptions ResponseStatusException

Spring boot web fournit des exceptions `ResponseStatusException`.

Ces exceptions permettent de retourner un status et un message.

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello() {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Hello not
found");
    }
}
```

Exercice 1

Les services

Pour créer un service, il faut utiliser l'annotation `@Service`.

`@Service` est équivalent à `@Component`.

```
@Service
public class HelloService {
    public String hello() {
        return "Hello World";
    }
}
```

Injection de dépendances

Pour injecter un service dans un controller, utilisez le constructeur.

```
@RestController
public class HelloController {
    private final HelloService helloService;

    public HelloController(HelloService helloService) {
        this.helloService = helloService;
    }

    @GetMapping("/hello")
    public String hello() {
        return helloService.hello();
    }
}
```

Les repositories

Pour créer un repository, il faut utiliser l'annotation `@Repository`.

`@Repository` est équivalent à `@Component`.

```
@Repository
public class TrucRepository {

    private final List<Truc> Trucs = new ArrayList<>();

    public List<Truc> findAll() {
        return Trucs;
    }

    public void save(Truc Truc) {
        trucs.add(Truc);
    }
}
```

```
public void deleteAll() {
    trucs.clear();
}

public void deleteById(int id) {
    trucs.removeIf(Truc -> Truc.getId() == id);
}

public Optional<Truc> findById(int id) {
    return trucs.stream()
        .filter(Truc -> Truc.getId() == id)
        .findFirst();
}

public void update(Truc Truc) {
    deleteById(Truc.getId());
    save(Truc);
}
}
```

Injection de dépendances

Pour injecter un repository dans un service, utilisez le constructeur.

```
@Service
public class TrucService {
    private final TrucRepository TrucRepository;

    public TrucService(TrucRepository TrucRepository) {
        this.trucRepository = TrucRepository;
    }

    public List<Truc> findAll() {
        return trucRepository.findAll();
    }

    public void save(Truc Truc) {
        trucRepository.save(Truc);
    }

    public void deleteAll() {
        trucRepository.deleteAll();
    }

    public void deleteById(int id) {
        trucRepository.deleteById(id);
    }

    public Optional<Truc> findById(int id) {
        return trucRepository.findById(id);
    }
}
```

```
}

public void update(Truc Truc) {
    trucRepository.update(Truc);
}
}
```

Exercice 2