

 **La Plateforme**

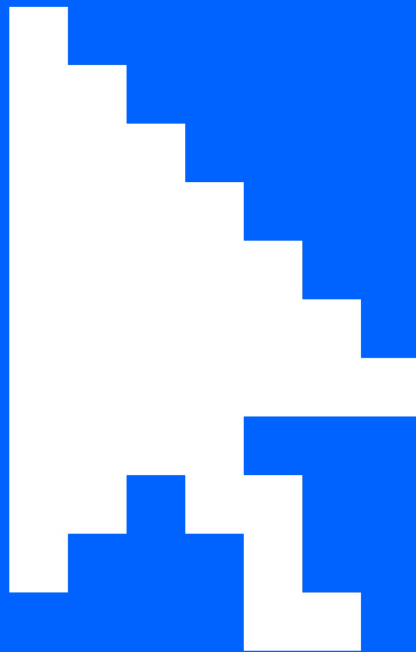
La grande école du numérique pour tous

Jour 2 – Spring Boot et l'architecture MVC & beans



Sommaire

- **présentation de l'architecture MVC**
- **La structure de Spring Boot**
- **Les beans**
- **Annotations**



Rappel

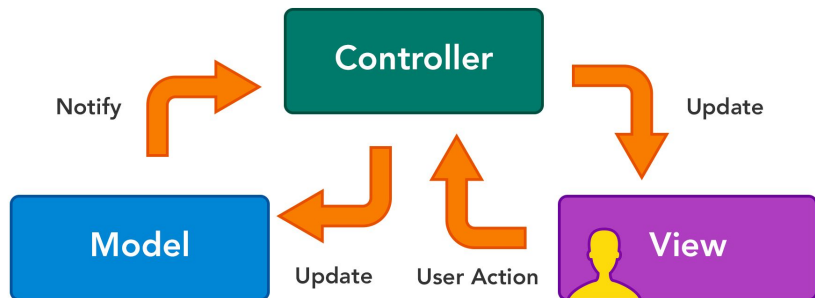
Framework : ensemble d'outils et de bibliothèques qui fournit une structure préétablie pour développer des applications. **Spring** est un framework qui simplifie le développement d'applications Java en automatisant certaines tâches comme la gestion des objets et des dépendances.

IoC : principe qui consiste à déléguer la gestion des objets à un **conteneur** (comme **Spring**) au lieu que l'application crée et gère ces objets elle-même. Spring utilise **IoC** pour créer, configurer et gérer automatiquement les objets de l'application.

Injection de dépendances : C'est un **mécanisme d'IoC** où les dépendances d'un objet lui sont fournies automatiquement par **Spring**, au lieu qu'il les crée lui-même. Cela rend le code plus modulaire, facile à tester et à maintenir.

Présentation de la l'architecture MVC

Model-View-Controller



Le pattern MVC permet de bien **organiser** son code source.

Il va vous aider à savoir quels fichiers **créer**, mais surtout à **définir leur rôle**.

- **Modèle** : gère la **logique métier**. Son objectif est de fournir une interface d'action la plus simple possible au contrôleur.
- **Vue** : se concentre sur l'**affichage**.
- **Contrôleur** : C'est l'intermédiaire entre l'utilisateur, le modèle et la vue.



En Spring Boot, le pattern MVC est logique, pas forcément lié à des dossiers fixes.

Model = les données et la logique métier. Ça peut être :

- des **entités JPA** (classes annotées **@Entity**)
- des **DTOs** (objets de transfert de données)
- des **services** qui contiennent la logique métier.

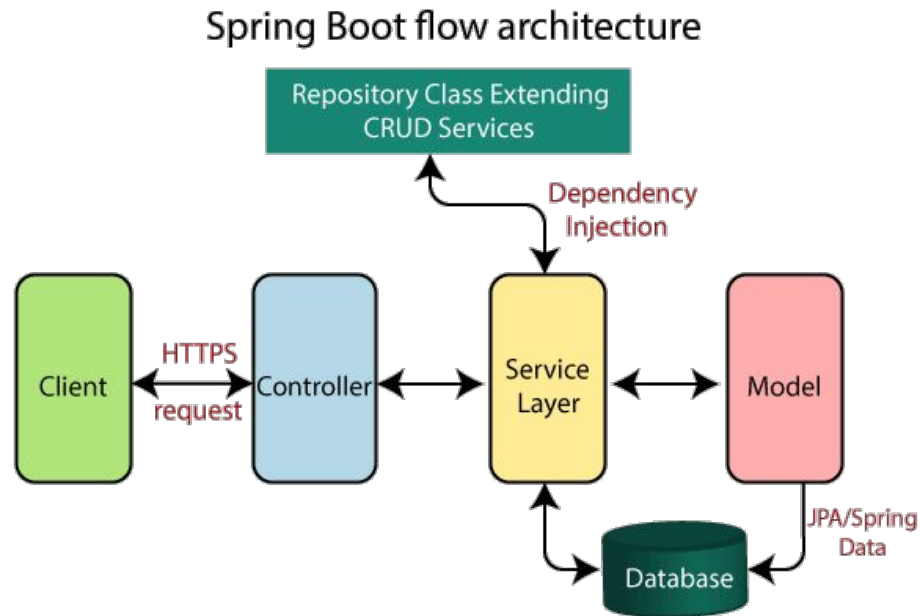
Avantages et séparation des responsabilités

- **Modularité**
- **Réutilisabilité**
- **Facilité de test**
- **Évolutivité**
- **Collaborative
Development**

MVC et Spring Boot

Pourquoi utiliser l'architecture MVC avec Spring Boot :

- **Séparation claire** des responsabilités (code organisé)
- **Facilite la maintenance et les évolutions**
- **Support natif de Spring Boot avec Spring MVC**
- **Compatible avec les applications web et API REST**





 **La Plateforme**

La grande école du numérique pour tous

Structure

Solutions entreprises

Structure classique Spring boot







Structure classique Spring boot



```
# src/main/resources/
```

```
resources
```

- └─  **static** → Contient les fichiers statiques (CSS, JS, images)
- └─  **templates** → Vues HTML (si on utilise Thymeleaf)
- └─  **application.yml** → Configuration de l'application (base de données, port...)
- └─  **data.sql** → Fichier SQL pour insérer des données initiales

Structure classique Spring boot



```
# src/test/java/com/example/monprojet/
```

```
tests
```

	folder icon	integration	→ Tests d'intégration (API, bases de données...)
	folder icon	unitaires	→ Tests unitaires (logique métier)

Examples

```
import jakarta.persistence.*;
```

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private Long id;
    private String name;

    // getters/setters
}
```



```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
```

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

```
import org.springframework.stereotype.Service;
import org.springframework.beans.factory.annotation.Autowired;
```

```
@Service
public class UserService {

    private final UserRepository repo;

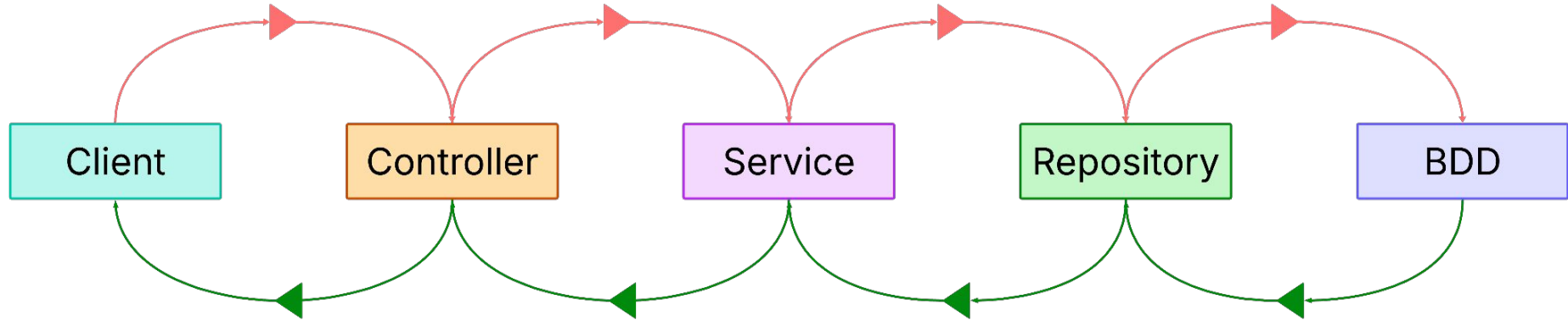
    @Autowired
    public UserService(UserRepository repo) {
        this.repo = repo;
    }

    public User createUser(String name) {
        User user = new User();
        user.setName(name);
        return repo.save(user);
    }
}
```

Les couches classiques

Couche	Rôle	Pourquoi est-ce important ?
Controller	Gère les requêtes HTTP (REST API, interface utilisateur) et appelle la couche service .	Sépare la logique métier de la gestion des requêtes (MVC).
Service	Contient la logique métier (traitement des données, règles métiers).	Évite la surcharge du contrôleur et facilite la réutilisation.
Repository (DAO)	Communique avec la base de données via JPA/Hibernate.	Permet une abstraction de la base de données et simplifie les tests.

Exemple process inscription



 **La Plateforme**

La grande école du numérique pour tous

Beans



Solutions entreprises

Beans



Un Bean est un **objet qui est instancié, assemblé et géré** par Spring IoC Container.

IoC est un **processus qui définit les dépendances d'un objet sans avoir à les créer**. C'est lors de la création des objets, que Spring va **injecter les Beans** entre eux afin d'avoir toutes **leurs dépendances**.

Dans un Bean, on déclare les dépendances nécessaires et **on n'a pas à se charger de les instancier**.

Il y a plusieurs façons de décrire un Bean mais la façon la plus courante aujourd'hui est d'utiliser des **annotations**. Les annotations **@Component**, **@Service** ou **@Repository** par exemple, vont indiquer que notre classe est un Bean.

Anecdote



Le terme *bean* vient des JavaBeans, qui représentent les composants élémentaires en Java.

Le mot est aussi une métaphore humoristique : Java tire son nom du café, et un bon café nécessite de bons grains (*beans* en anglais).

Les composants essentiels des programmes Java ont donc été baptisés *beans*.

Gestion automatique de l'injection de dépendance (variable)

```
package dev.gayerie.apptask;

import
org.springframework.beans.factory.annotation.Autowired;
public class TaskManager() {

    @Autowired
    private Runnable tache;

    public void start() {
        // ...
    }
}
```

```
public interface Runnable {
    void run();
}
```

Quand Spring doit injecter un objet de type **Runnable**, il regarde dans son conteneur IoC s'il existe un **bean qui implémente Runnable**.

Si un seul bean correspond, il l'injecte directement.

S'il y en a plusieurs, Spring cherche à désambiguïser :

il compare le **nom du bean** avec le **nom de l'attribut** où il doit injecter.

Exemple : si un attribut s'appelle **tache**, Spring privilégie le bean nommé **tache**.

Donc : en cas de conflit, le nom du champ sert de clé pour choisir le bon bean.

Exemple @Qualifier (constructor)

```
// ----- Abstraction -----  
public interface NotificationService {  
    void send(String message);  
}  
  
// ----- Implémentation 1 -----  
@Component("emailService")  
class EmailService implements NotificationService {  
    @Override  
    public void send(String message) {  
        System.out.println("Envoi d'un email : " + message);  
    }  
}  
  
// ----- Implémentation 2 -----  
@Component("smsService")  
class SmsService implements NotificationService {  
    @Override  
    public void send(String message) {  
        System.out.println("Envoi d'un SMS : " + message);  
    }  
}
```

```
// ----- Controller -----  
@RestController  
public class UserController {  
  
    private final NotificationService notificationService;  
  
    // Injection de l'implémentation choisie  
    @Autowired  
    public UserController(@Qualifier("emailService") NotificationService notificationService) {  
        this.notificationService = notificationService;  
    }  
  
    @GetMapping("/notify")  
    public String notifyUser() {  
        notificationService.send("Bienvenue sur l'application !");  
        return "Notification envoyée";  
    }  
}
```

Ici si l'on utilise pas @Qualifier alors au démarrage on obtiendra **NoUniqueBeanDefinitionException**

Scope d'un bean



Par défaut un Bean a pour scope **Singleton**.

singleton : Cela signifie qu'une seule instance de ce bean existe dans le conteneur IoC = chaque appel retourne le même objet.

On peut faire en sorte d'avoir plusieurs instances d'un Bean en spécifiant le scope **Prototype**. Pour se faire, il faut ajouter l'annotation `@Scope("prototype")` à notre bean.

prototype: À chaque fois qu'un programme appelle une méthode `getBean` pour récupérer ce bean, chaque appel retourne une nouvelle instance du bean.

Bean : cycle de vie

Un bean Spring a 3 grandes phases:

- **Initialisation:** le bean est créé par le conteneur Spring , ses dépendances sont résolues
- **Usage:** le bean est utilisable dans l'application
- **Destruction:** les beans sont envoyés au garbage collector

La méthode d'initialisation est appelée immédiatement lors de l'instanciation, et la méthode de destruction juste avant que le Bean ne soit supprimé du container.



 **La Plateforme**

La grande école du numérique pour tous

Annotations en Spring Boot

Solutions entreprises

Annotations

Les annotations permettent de **déclarer des comportements**, de **configurer des composants** et de **mapper les requêtes HTTP** sans besoin d'un lourd fichier de configuration XML.

Permettent de simplifier le code en le rendant plus expressif et moins verbeux. Elles facilitent :

1. **La configuration automatique** : il n'est plus nécessaire de définir chaque paramètre de configuration manuellement.
2. **La lisibilité** : Elles rendent le code **plus facile à lire et à maintenir**, en indiquant clairement l'intention des développeurs.
3. **La réduction du couplage** : permettent de réduire le couplage entre les composants, en favorisant une **approche déclarative**.
4. **La productivité** : le développeur peut se concentrer sur **l'implémentation des fonctionnalités** plutôt que sur la configuration de l'infrastructure.

Types d'injection de dépendances

Injection via annotation

L'**injection de dépendances** est un principe fondamental en **Spring Boot** qui permet de **découpler** les composants d'une application pour faciliter la maintenance et les tests.

En **Spring Boot**, l'injection de dépendances peut être faite grâce à plusieurs **annotations**, qui indiquent comment **Spring doit gérer et injecter les objets**.



Annotation	Description
@Component	Marque une classe comme composant Spring générique.
@Service	Indique qu'une classe contient la logique métier .
@Repository	Indique qu'une classe est un DAO qui gère l'accès aux données.
@Controller	Déclare un contrôleur MVC Spring .
@RestController	Équivalent de @Controller, mais pour une API REST.
@Autowired	Injecte automatiquement une dépendance.
@Qualifier	Spécifie quelle implémentation injecter si plusieurs existent.
@Primary	Définit un bean par défaut en cas de plusieurs implémentations.
@Value	Injecte une valeur provenant d'un fichier de configuration.

@RestController = **@Controller** + **@ResponseBody** ce qu'il retourne est **écrit directement dans la réponse HTTP** en JSON ou XML (via Jackson).

Injection automatique

Spring détecte automatiquement les dépendances à injecter si un seul constructeur est présent.

```
● ● ●  
  
@Service  
public class MonService {  
    private final MonRepository repository;  
  
    // Pas besoin de @Autowired si un seul constructeur est présent  
    public MonService(MonRepository repository) {  
        this.repository = repository;  
    }  
}
```

✓ Plus propre, réduit le couplage avec Spring.

✓ Recommandé pour les classes testables et immutables.

✗ Ne fonctionne pas si plusieurs constructeurs existent sans @Autowired.

Injection automatique

S'il y a plusieurs constructeurs, Spring ne sait pas lequel utiliser

```
@Service
public class MonService {
    private final MonRepository repository;
    private final AutreService autreService;

    @Autowired // Obligatoire pour que Spring sache quel constructeur utiliser
    public MonService(MonRepository repository) {
        this.repository = repository;
        this.autreService = null; // Pas injecté ici
    }

    public MonService(MonRepository repository, AutreService autreService) {
        this.repository = repository;
        this.autreService = autreService;
    }
}
```

✓ Contrôle précis de l'injection.

✓ Compatible avec plusieurs constructeurs ou setters.

✗ Dépendance explicite à Spring (moins portable).



 La Plateforme