

# Introduction à Spring

## Objectifs du cours

- ¥ Comprendre l'histoire et la philosophie de Spring
- ¥ Saisir les concepts d'Inversion de Contrôle (IoC) et d'Injection de Dépendances (DI)
- ¥ Identifier les avantages de Spring pour le développement d'applications complexes
- ¥ Comparer Spring / Spring Core avec Jakarta EE (anciennement Java EE)



L'objectif ici est d'expliquer pourquoi Spring a été développé et comment il répond aux limites de Jakarta EE, notamment en matière de configuration, de flexibilité et de testabilité.

## 1. Contexte et Historique de Spring

### 1.1 Pourquoi Spring ?

Spring est né au début des années 2000, en réaction à la complexité croissante de Java EE (aujourd'hui Jakarta EE). À cette époque, de nombreux développeurs se plaignaient de la lourdeur des technologies comme EJB (Enterprise JavaBeans), qui étaient difficilement testables, longues à configurer, et peu souples.

### 1.2 Limites de Jakarta EE

#### ! Limite 1 : Configuration lourde (XML, EJB, etc.)

Dans les premières versions de Java EE, il fallait beaucoup de configuration XML pour déclarer des composants.

Exemple :

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>CompteService</ejb-name>
      <ejb-class>com.banque.CompteServiceBean</ejb-class>
      <session-type>Stateless</session-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Soulignez que cette configuration est non seulement verbeuse, mais aussi source d'erreurs. Toute modification de nom ou de classe nécessitait une mise à jour dans plusieurs fichiers. Cela ralentissait le cycle de développement.

## ! Limite 2 : Complexité et verbosité du code

Même pour une logique simple, Java EE demandait plusieurs classes et annotations.

Exemple :

```
@Stateless
public class CompteServiceBean implements CompteService {
    public double getSolde(String numeroCompte) {
        // ...
    }
}
```

Insistez sur le contraste avec des frameworks modernes comme Spring, où une annotation `@Service` suffit dans la plupart des cas. Les développeurs perdaient du temps à respecter des conventions strictes imposées par la spécification.

## ! Limite 3 : Couplage fort au serveur d'application

Les EJB (et autres composants Java EE) ne pouvaient fonctionner qu'au sein d'un conteneur comme JBoss ou GlassFish.

Expliquez que cela réduisait la portabilité et complexifiait l'architecture : impossible d'exécuter un service métier indépendamment. Cela allait à l'encontre des principes modernes comme la containerisation ou les microservices légers.

## ! Limite 4 : Test difficile des composants

Impossible de tester un EJB sans lancer tout le serveur d'application.

Soulignez l'impact direct sur la qualité logicielle : les tests unitaires étaient souvent évités ou remplacés par des tests d'intégration lourds et lents. Ce frein à l'agilité a été l'un des déclencheurs majeurs de l'adoption de Spring.

## 2 Pourquoi Spring ? Objectifs et Philosophie

### 2.1 Alléger le développement Java

¥ Spring repose sur des POJOs : pas besoin d'hériter d'une classe spécifique.

¥ Moins de configuration XML, plus d'annotations simples.

```
@Component
public class CompteService {
    public double getSolde(String numero) {
        return 100.0;
    }
}
```



Insistez sur le fait qu'un composant métier n'a pas besoin d'hériter d'une classe ou d'implémenter une interface imposée par un conteneur. Cela facilite le développement et la maintenance.

## 2.2 Favoriser le développement modulaire et découplé

¥ Spring utilise l'injection de dépendances (DI).

¥ Réduction du couplage entre les composants de l'application.

```
@Service
public class CompteService {
    private final CompteRepository repository;

    public CompteService(CompteRepository repository) {
        this.repository = repository;
    }
}
```



Soulignez que Spring injecte automatiquement les dépendances (ici, `CompteRepository`), ce qui facilite la substitution ou l'évolution des composants.

## 2.3. Simplifier les tests

¥ Grâce aux POJOs et à la DI, les composants peuvent être testés sans serveur.

¥ Plus besoin de déployer un conteneur Java EE.

```
@Test
void testSolde() {
    CompteRepository repo = mock(CompteRepository.class);
    when(repo.findSolde("123")).thenReturn(200.0);
    CompteService service = new CompteService(repo);
    assertEquals(200.0, service.getSolde("123"));
}
```

```
}
```

!

Expliquez que cela rend les tests beaucoup plus rapides, car ils peuvent être exécutés comme de simples tests unitaires.

## 2.4. Réduire le couplage à l'infrastructure

¥ Spring fonctionne sans conteneur Java EE complet.

¥ Peut tourner dans un conteneur léger comme Tomcat, ou même en mode console.

!

Expliquez que cela facilite le déploiement, même en local, et réduit les contraintes liées à l'environnement d'exécution.

## 2.5. Encourager les bonnes pratiques d'architecture

¥ Respect du principe de séparation des responsabilités.

¥ Utilisation d'annotations : `@Controller`, `@Service`, `@Repository`.

¥ Spring propose aussi l'AOP (Aspect-Oriented Programming).

!

Donnez un exemple d'architecture en couches (web/service/persistence) que Spring encourage naturellement.

## 3. Les Fondamentaux de Spring Core

### 3.1 Inversion de Contrôle (IoC)

¥ Définition : Le contrôle de la création et de l'assemblage des objets est délégué à un conteneur (Spring IoC container)

¥ Avantages : Découplage du code : les composants n'ont plus à gérer la création de leurs dépendances Facilite la maintenance et l'évolution de l'application

¥ Mécanisme : Les classes sont déclarées comme beans à l'aide d'annotations (`@Component`, `@Service`, etc.) ou via des fichiers de configuration

### 3.1 Inversion de Contrôle (IoC)

```
@Component
public class MessageService {
    É public String getMessage() {
```

```

    return "Hello from Spring!";
}
}

```



Expliquez qu'avec IoC, c'est Spring qui organise l'assemblage des objets, permettant ainsi une évolution plus aisée et des tests plus simples.

## 3.2 Injection de Dépendances (DI)

¥ Concept : Les dépendances sont injectées par le conteneur et non créées directement par le code

¥ Types d'injection :

- ! Par constructeur (préférée pour l'immuabilité)
- ! Par setter (pour les dépendances optionnelles)

## 3.2 Injection de Dépendances (DI)

¥ Avantages :

- ! Testabilité : il devient possible de substituer facilement les dépendances par des mocks
- ! Flexibilité et réutilisabilité des composants

## 3.2 Injection de Dépendances (DI)

```

@Component
public class UserService {
    private final MessageService messageService;

    @Autowired
    public UserService(MessageService messageService) {
        this.messageService = messageService;
    }

    public void printMessage() {
        System.out.println(messageService.getMessage());
    }
}

```



Vous pouvez insister sur la notion de testabilité en soulignant qu'il est plus simple de tester `UserService` en injectant un faux `MessageService`.

## 3.3 Configuration de Spring Core

¥ Deux approches principales :

- ! Configuration XML : Historique, mais aujourd'hui moins utilisée
- ! Configuration par annotations : Utilisation de `@Configuration` et `@ComponentScan`

## 3.3 Configuration de Spring Core

```
@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
}
```

!

Précisez que la configuration par annotations est aujourd'hui la méthode privilégiée.

## 4. Comparaison Spring / Spring Core et Jakarta EE

### 4.1 Jakarta EE (ex Java EE)

¥ Caractéristiques :

- ! Standard robuste pour les applications d'entreprise
- ! Exige des conteneurs lourds (ex: GlassFish, Wildfly)
- ! Nombreuses spécifications (Servlets, JSP, EJB, JPA, etc.)

### 4.1 Jakarta EE (ex Java EE)

¥ Limites :

- ! Couplage fort au runtime
- ! Configuration verbeuse (souvent XML)
- ! Tests unitaires difficiles

!

Détaillez que l'utilisation d'EJB impose un modèle complexe et peu testable.

## 4.2 Les Avantages de Spring

¥ Souplesse et légèreté :

- ! Mise en place rapide
- ! Moins de configuration

## 4.2 Les Avantages de Spring

¥ Découplage et testabilité :

- ! DI + IoC = composants testables

## 4.2 Les Avantages de Spring

¥ Modularité et intégration :

- ! Choix des modules
- ! Intégration facile avec d'autres frameworks

!

Soulignez que Spring a transformé le développement Java.

## 5. Conclusion et perspectives

¥ Récapitulatif :

- ! Spring résout les limites de Jakarta EE
- ! Basé sur IoC + DI

¥ Prochaines étapes :

- ! Introduction à Spring Boot

!

Encouragez les questions et ouvrez sur Spring Boot.

## Références

¥ JMDoudoux @ <https://www.jmdoudoux.fr/java/dej/chap-spring.htm>

¥ JMDoudoux @ [https://www.jmdoudoux.fr/java/dej/chap-spring\\_core.htm](https://www.jmdoudoux.fr/java/dej/chap-spring_core.htm)