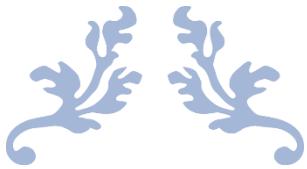

JEE – PARTIE 2

M. ABDEL-KALIF BEN HAMADOU

Activité 3: Introduction à Spring Framework

Objectifs de l'activité :

- Comprendre les concepts fondamentaux du **Spring Framework** : Inversion of Control (IoC) et injection de dépendances.
- Apprendre à configurer une application Spring à l'aide de XML et de la configuration Java.
- Créer un premier projet Spring avec un bean et une injection de dépendances.

1. Concepts clés de Spring : Inversion of Control (IoC) et injection de dépendances

1.1. Qu'est-ce que l'Inversion of Control (IoC) ?

L'**Inversion of Control (IoC)** est un concept central du framework Spring. Traditionnellement, dans une application Java, un objet est responsable de la création et de la gestion de ses dépendances (autres objets dont il a besoin pour fonctionner). Avec l'IoC, c'est le framework qui prend en charge la création et la gestion des dépendances des objets, inversant ainsi le contrôle par rapport à la méthode traditionnelle.

L'IoC est implémenté dans Spring via le **conteneur IoC**, qui gère les objets, leur cycle de vie et leurs relations de dépendances.

1.2. Injection de dépendances

L'**injection de dépendances** (Dependency Injection, DI) est une implémentation de l'IoC dans Spring. Elle consiste à fournir les dépendances (objets) d'une classe de l'extérieur plutôt que de laisser la classe les créer elle-même. Il existe trois types d'injection de dépendances dans Spring :

- **Injection par constructeur** : Les dépendances sont injectées dans un objet via son constructeur.
- **Injection par setter** : Les dépendances sont injectées via des méthodes setter.
- **Injection via les champs** : Moins courante, cette méthode utilise directement les annotations sur les propriétés de la classe.

Exemple avec injection par constructeur :

```
public class Service {
    private Repository repository;

    public Service(Repository repository) {
        this.repository = repository;
    }

    public void doSomething() {
        repository.saveData();
    }
}
```

Dans cet exemple, la classe `Service` dépend d'une instance de `Repository`. Plutôt que de la créer elle-même, `Repository` lui est injecté via le constructeur.

2. Configuration avec XML vs Java-based configuration

2.1. Configuration basée sur XML

Historiquement, la configuration des dépendances dans Spring se faisait principalement avec des fichiers XML. Ce type de configuration consiste à définir les beans et leurs dépendances dans un fichier XML que le conteneur Spring utilise pour les injecter.

Exemple d'un fichier de configuration XML (`applicationContext.xml`) :

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="repository" class="com.example.Repository" />

    <bean id="service" class="com.example.Service">
        <constructor-arg ref="repository" />
    </bean>
</beans>
```

Dans cet exemple, un bean `Repository` est défini, ainsi qu'un bean `Service` qui reçoit `Repository` via son constructeur.

2.2. Configuration basée sur Java (Java-based configuration)

Avec les versions récentes de Spring, la configuration Java (ou **Java-based configuration**) est devenue plus populaire, car elle est plus intuitive et permet de tirer parti des annotations et des classes Java sans nécessiter de fichiers XML.

Exemple d'une configuration Java à l'aide de l'annotation `@Configuration` :

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean
    public Repository repository() {
        return new Repository();
    }

    @Bean
    public Service service() {
        return new Service(repository());
    }
}
```

Ici, la classe `AppConfig` contient des méthodes annotées avec `@Bean` qui retournent des instances des objets à injecter. Ces méthodes définissent les beans et leur cycle de vie.

2.3. Avantages de la configuration Java

- **Lisibilité** : La configuration est plus lisible et intuitive, car elle utilise directement le langage Java.
- **Type-safe** : Avec la configuration Java, vous bénéficiez des avantages de la vérification des types à la compilation.
- **Réduction de la dépendance XML** : Moins de fichiers XML à gérer et plus de contrôle dans le code.

3. Premier projet Spring : création d'un bean et injection de dépendances

Pour mettre en pratique ces concepts, nous allons créer un petit projet Spring qui définit et injecte un bean dans une classe.

3.1. Structure du projet

Voici les étapes pour créer une petite application Spring :

1. **Créer les classes de base** :
    - `Repository` : une classe simple qui simule la gestion des données.
    - `Service` : une classe qui dépend de `Repository` pour effectuer des opérations métier.
2. **Configurer Spring pour gérer les beans** : Utiliser soit un fichier XML, soit une configuration Java pour déclarer les beans.

3.2. Exemple de classes pour l'application

Voici une classe `Repository` simple :

```
public class Repository {
    public void saveData() {
        System.out.println("Données enregistrées !");
    }
}
```

Et la classe `Service` qui dépend de `Repository` :

```
public class Service {
    private Repository repository;

    public Service(Repository repository) {
        this.repository = repository;
    }

    public void process() {
        System.out.println("Service en cours...");
        repository.saveData();
    }
}
```

3.3. Configuration du conteneur Spring

Nous allons utiliser une configuration Java pour injecter `Repository` dans `Service`.

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

        Service service = context.getBean(Service.class);
        service.process();
    }
}
```

3.4. Fichier de configuration Java (`AppConfig.java`)

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean
    public Repository repository() {
        return new Repository();
    }

    @Bean
    public Service service() {
        return new Service(repository());
    }
}
```

3.5. Résultat attendu

En exécutant l’application, vous devriez obtenir la sortie suivante :

```
Service en cours...
Données enregistrées !
```

Spring a géré l'injection de dépendances et a exécuté les méthodes de manière fluide.

4. Conclusion

Le **Spring Framework** introduit des concepts puissants comme l'**Inversion of Control** (IoC) et l'**injection de dépendances**, qui simplifient la gestion des objets et de leurs relations dans une application. La configuration via XML ou Java offre des approches flexibles pour déclarer les beans, et grâce à l'injection de dépendances, les applications deviennent plus modulaires, testables et maintenables.

Résumé des points clés :

- **IoC** et **DI** sont des concepts centraux dans Spring, offrant un contrôle flexible sur la création et la gestion des dépendances.
- Spring permet deux types principaux de configuration : **XML** (plus ancien) et **Java-based configuration** (plus moderne et intuitive).
- La création d’un projet Spring simple avec l’injection de dépendances permet de bien comprendre ces concepts.

Activité 4: Spring MVC pour le Développement Web

Objectifs de l'activité :

- Découvrir le framework **Spring MVC** pour le développement d'applications web.
- Apprendre à créer et configurer un **contrôleur Spring**.
- Comprendre la gestion des requêtes HTTP (GET et POST) dans Spring MVC.
- Apprendre à utiliser des vues et des modèles avec des moteurs de template comme **JSP** et **Thymeleaf**.

1. Introduction à Spring MVC

1.1. Qu'est-ce que Spring MVC ?

Spring MVC (Model-View-Controller) est un module du Spring Framework qui facilite le développement d'applications web en utilisant une architecture basée sur le modèle MVC. Ce modèle permet de séparer la logique métier (modèle), l'affichage (vue), et le contrôle des interactions utilisateur (contrôleur), améliorant ainsi la maintenabilité et l'organisation du code.

1.2. Architecture MVC

- **Modèle (Model)** : Représente les données de l'application. Il contient la logique métier et récupère les données nécessaires à partir de la base de données.
- **Vue (View)** : Représente la présentation des données. Dans Spring MVC, cela peut être géré par des moteurs de templates comme **JSP** ou **Thymeleaf**.
- **Contrôleur (Controller)** : Gère les interactions avec l'utilisateur, reçoit les requêtes HTTP, traite la logique métier via le modèle, et renvoie une vue appropriée à l'utilisateur.

L'architecture Spring MVC repose principalement sur l'utilisation de **contrôleurs** pour gérer les requêtes HTTP et retourner des vues correspondantes.

2. Création d’un contrôleur Spring

2.1. Définition d’un contrôleur

Dans Spring MVC, un **contrôleur** est une classe Java annotée avec `@Controller` qui gère les requêtes HTTP. Chaque méthode du contrôleur peut être associée à une requête spécifique (GET, POST, etc.) en utilisant des annotations comme `@GetMapping` ou `@PostMapping`.

2.2. Exemple de contrôleur Spring

Voici un exemple de contrôleur Spring basique qui gère une requête HTTP GET et renvoie une vue :

```java
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
@RequestMapping("/home")
public class HomeController {

    @GetMapping
    public ModelAndView home() {
        ModelAndView mav = new ModelAndView("home");
        mav.addObject("message", "Bienvenue dans Spring MVC !");
        return mav;
    }
}
```

- **@Controller** : Indique que cette classe est un contrôleur Spring.
- **@RequestMapping("/home")** : Mappe la classe à l'URL "/home".
- **@GetMapping** : Gère les requêtes HTTP GET pour "/home".
- **ModelAndView** : Retourne une vue nommée `home` et y associe un modèle contenant un message à afficher.

3. Gestion des requêtes HTTP : GET, POST

3.1. Gestion des requêtes GET

Les requêtes GET sont principalement utilisées pour **récupérer** des données. Spring MVC utilise l'annotation `@GetMapping` pour définir des méthodes qui traitent ces requêtes.

Exemple d'une méthode qui gère une requête GET pour afficher un formulaire de contact :

```
@GetMapping("/contact")
public String showContactForm(Model model) {
    model.addAttribute("contact", new Contact());
    return "contact-form";
}
```

Ici, le contrôleur renvoie la vue `contact-form` et ajoute un objet vide `Contact` au modèle.

3.2. Gestion des requêtes POST

Les requêtes POST sont utilisées pour **envoyer** des données, comme des informations de formulaire. Spring MVC utilise `@PostMapping` pour gérer les requêtes POST.

Exemple de méthode qui gère une soumission de formulaire de contact :

```
@PostMapping("/contact")
public String submitContactForm(@ModelAttribute("contact") Contact contact, Model
model) {
    // Logique de traitement du contact (sauvegarde, validation, etc.)
    model.addAttribute("message", "Formulaire soumis avec succès !");
    return "contact-result";
}
```

- **@ModelAttribute** : Lie les données envoyées via le formulaire à l'objet `Contact`.
- La méthode traite les données et renvoie une vue pour afficher le résultat.

3.3. Modèle et Vue (ModelAndView)

Spring MVC permet de combiner des objets de modèle avec des vues pour rendre des réponses dynamiques. L'objet `ModelAndView` contient à la fois le modèle (les données) et la vue (le modèle d'affichage).

4. Vue et gestion des modèles (JSP, Thymeleaf)

4.1. Utilisation de JSP comme moteur de vue

JSP (Java Server Pages) est un moteur de template couramment utilisé avec Spring MVC. Il permet de créer des pages web dynamiques en combinant HTML et des balises JSP pour intégrer des données Java dans la vue.

Exemple d'une page JSP (`home.jsp`) :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
    <head>
        <title>Page d'accueil</title>
    </head>
    <body>
        <h1>${message}</h1>
    </body>
</html>
```

Dans cet exemple, la variable `message` ajoutée dans le modèle par le contrôleur est affichée dans la vue.

4.2. Utilisation de Thymeleaf comme moteur de vue

Thymeleaf est un moteur de template moderne et plus flexible que JSP. Il s'intègre facilement à Spring MVC et permet de manipuler des modèles HTML de manière plus lisible.

Exemple d'une vue Thymeleaf (`home.html`) :

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Page d'accueil</title>
    </head>
    <body>
        <h1 th:text="${message}"></h1>
    </body>
</html>
```

Ici, l'attribut `th:text` remplace le contenu de l'élément avec la variable `message` provenant du modèle.

4.3. Configuration de Thymeleaf dans Spring

Pour utiliser Thymeleaf dans Spring, il est nécessaire de l'ajouter en tant que dépendance dans le projet et de configurer un résolveur de vues.

Dépendance Maven pour Thymeleaf :

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

5. Exemple complet : formulaire de contact avec Spring MVC

Voici un exemple simple d'application Spring MVC qui gère un formulaire de contact. L'utilisateur peut soumettre ses informations, et le contrôleur les traite.

5.1. Classe Contact

```
public class Contact {
    private String nom;
    private String email;

    // Getters et setters
}
```

5.2. Contrôleur de formulaire de contact

```
@Controller
@RequestMapping("/contact")
public class ContactController {

    @GetMapping
    public String showContactForm(Model model) {
        model.addAttribute("contact", new Contact());
        return "contact-form";
    }

    @PostMapping
    public String submitContactForm(@ModelAttribute("contact") Contact contact,
Model model) {
        // Traitement du formulaire (sauvegarde, validation, etc.)
        model.addAttribute("message", "Merci " + contact.getNom() + ", votre
formulaire a été soumis.");
        return "contact-result";
    }
}
```

5.3. Formulaire pour le contact (`contact-form.jsp`)

```
<form action="/contact" method="post">
    <label for="nom">Nom :</label>
    <input type="text" id="nom" />

    <label for="email">Email :</label>
    <input type="email" id="email" />

    <button type="submit">Envoyer</button>
</form>
```

5.4. Vue de confirmation (`contact-result.jsp`)

```
<h2>${message}</h2>
```

6. Conclusion

Le framework **Spring MVC** fournit une architecture robuste et flexible pour le développement d'applications web. En gérant les requêtes HTTP et en intégrant des moteurs de template comme **JSP** ou **Thymeleaf**, Spring permet de créer des applications web dynamiques et maintenables. L'utilisation de **contrôleurs** et de la **gestion des modèles et vues** simplifie considérablement la structuration des applications web complexes.

Résumé des points clés :

- **Spring MVC** implémente l'architecture MVC pour séparer la logique métier, l'affichage, et le contrôle des interactions utilisateur.
- Les **contrôleurs Spring** gèrent les requêtes HTTP et permettent de lier des modèles à des vues.
- **JSP** et **Thymeleaf** sont des moteurs de template couramment utilisés pour la création de vues dynamiques.