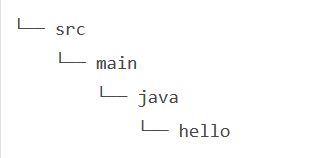
# Systèmes distribués (Manip 01)

Séance 01 :

* 1. **HelloWorld avec Maven**

1) Créer la structure des répertoires :



2) créer ces deux classes : HelloWorld.java et Greeter.java

Greeter.java :  
package hello;  
  
public class Greeter {  
 public String sayHello() {  
 return "Hello world!";  
 }  
}

HelloWorld.java :

package hello;  
  
public class HelloWorld {  
 public static void main(String[] args) {  
 Greeter greeter = new Greeter();  
 System.*out*.println(greeter.sayHello());  
 }  
}

3) Pour tester l'installation de Maven, il faut exécuter « mvn  -v» à partir de la ligne de commande.

4) créer une définition de projet Maven. Les projets Maven sont définis avec un fichier XML nommé pom.xml. Entre autres choses, ce fichier donne le nom du projet, sa version et ses dépendances vis-à-vis de bibliothèques externes.

Créez un fichier nommé pom.xml à la racine du projet (c'est-à-dire placez-le à côté du dossier src).

* <modelVersion >. Version du modèle POM (toujours 4.0.0).
* <IDgroupe>. Groupe ou organisation auquel appartient le projet. Souvent exprimé sous la forme d'un nom de domaine inversé.
* <artefactId>. Nom à donner à l'artefact de bibliothèque du projet (par exemple, le nom de son fichier JAR ou WAR).
* <version>. Version du projet en cours de construction.
* <packaging> - Comment le projet doit être emballé. La valeur par défaut est "jar" pour l'empaquetage de fichiers JAR. Utilisez "war" pour l'empaquetage des fichiers WAR.

5) Build Java code :

* **mvn compile**: il exécute Maven, lui disant d'exécuter l'objectif de compilation. Une fois terminé, vous devriez trouver les fichiers .class compilés dans le répertoire target/classes.
* **mvn package** : L'objectif package compile le code Java, exécute tous les tests et termine en empaquetant le code dans un fichier JAR dans le répertoire target.

6) Pour exécuter le fichier JAR : java -jar target/gs-maven-0.1.0.jar

7) Déclarer des dépendances : L’exemple simple Hello World est entièrement autonome et ne dépend d’aucune bibliothèque supplémentaire. Toutefois, la plupart des applications dépendent de bibliothèques externes pour gérer des fonctionnalités courantes et complexes.

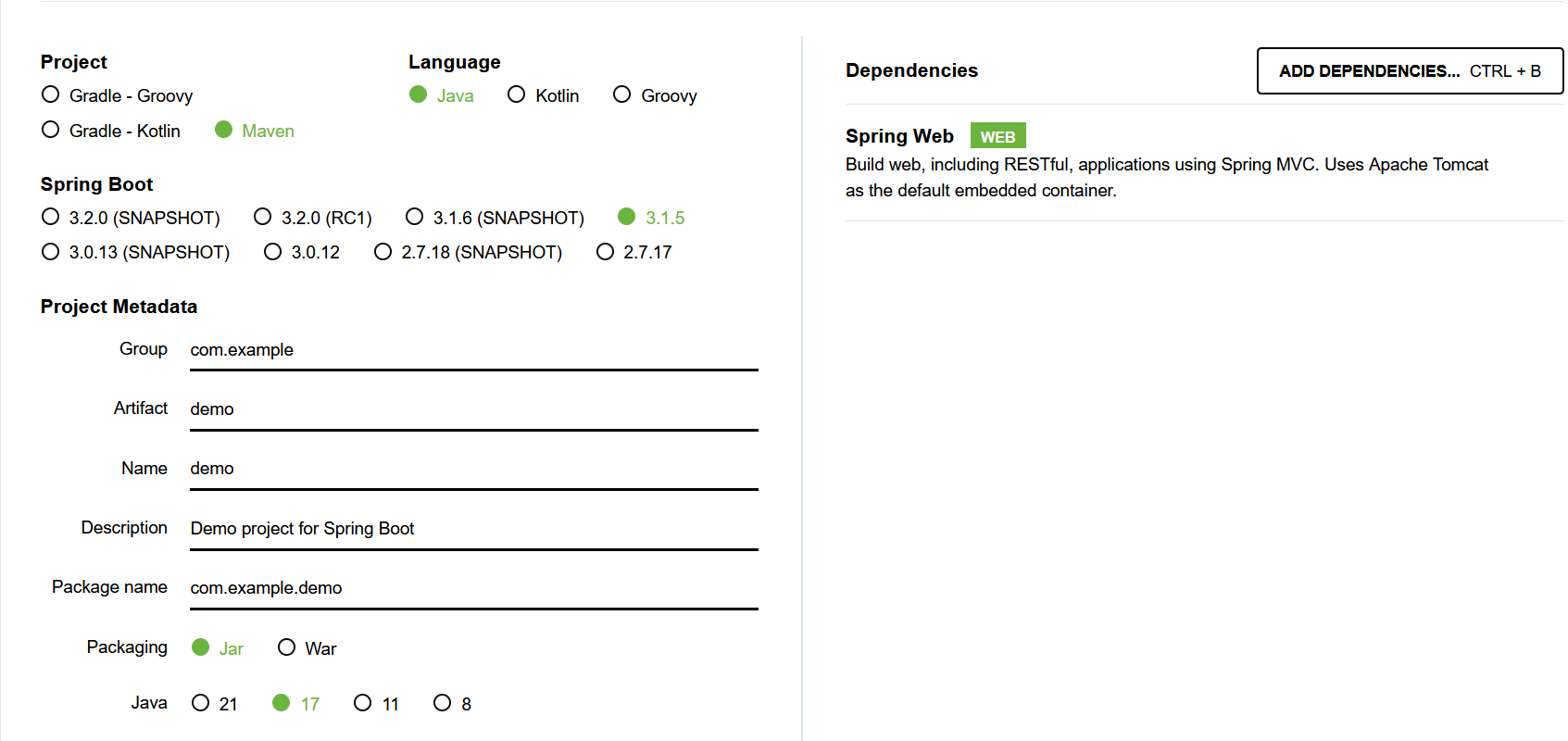
Utilisation de la librairie Joda Time :

HelloWorld.java

package hello;  
  
import org.joda.time.LocalTime;  
  
public class HelloWorld {  
 public static void main(String[] args) {  
 LocalTime currentTime = new LocalTime();  
 System.*out*.println("The current local time is: " + currentTime);  
 Greeter greeter = new Greeter();  
 System.*out*.println(greeter.sayHello());  
 }  
}

* 1. **Prise en main Spring Boot :**

1) générer le projet spring sur start.spring.io :



2) On crée une simple application web. On commence par créer le contrôleur :

src/main/java/com/example/springboot/HelloController.java

package com.example.springboot;  
  
@RestController  
public class HelloController {  
  
 @GetMapping("/")  
 public String index() {  
 return "Greetings from Spring Boot!";  
 }  
  
}

La classe est annoté comme @RestController, ce qui signifie qu'elle est prête à être utilisée par Spring MVC pour gérer les requêtes Web. @GetMapping mappe / à la méthode index(). Lorsqu'elle est invoquée depuis un navigateur ou en utilisant curl sur la ligne de commande, la méthode renvoie du texte pur. En effet, @RestController combine @Controller et @ResponseBody, deux annotations qui entraînent des requêtes Web renvoyant des données plutôt qu'une vue.

3) Création/modification de la classe Application :

src/main/java/com/example/springboot/Application.java

package com.example.springboot;  
  
@SpringBootApplication  
public class Application {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(Application.class, args);  
 }  
  
 @Bean  
 public CommandLineRunner commandLineRunner(ApplicationContext ctx) {  
 return args -> {  
  
 System.*out*.println("Let's inspect the beans provided by Spring Boot:");  
  
 String[] beanNames = ctx.getBeanDefinitionNames();  
 Arrays.*sort*(beanNames);  
 for (String beanName : beanNames) {  
 System.*out*.println(beanName);  
 }  
  
 };  
 }  
  
}

5) Pour lancer l’application : mvn spring-boot:run

* 1. **HelloWorld WebService / Construire un web service avec Spring :**

On va créer un service qui accepte les requêtes HTTP GET sur http://localhost:8080/greeting.

Il répond avec une représentation JSON de Greeting.

1) Générer un projet web (avec la dépendance web) sur start.spring.io

2) Créer un contrôleur de ressources :

Dans l'approche de Spring pour créer des services Web RESTful, les requêtes HTTP sont gérées par un contrôleur. Ces composants sont identifiés par l'annotation @RestController, et le GreetingController gère les requêtes GET pour /greeting en renvoyant une nouvelle instance de la classe Greeting :

Greeting.java :

package com.example.demo;  
  
public record Greeting(Long id,String content ) {  
}

GreetingController.java :

package com.example.restservice;  
  
@RestController  
public class GreetingController {  
  
 private static final String *template* = "Hello, %s!";  
 private final AtomicLong counter = new AtomicLong();  
  
 @GetMapping("/greeting")  
 public Greeting greeting(@RequestParam(value = "name", defaultValue = "World") String name) {  
 return new Greeting(counter.incrementAndGet(), String.*format*(*template*, name));  
 }  
}

DemoApplication.java :

package com.example.demo;  
  
@SpringBootApplication  
public class DemoApplication {  
 public static void main(String[] args) {  
 SpringApplication.*run*(DemoApplication.class, args);  
 }  
}

3) ./mvnw spring-boot:run ou mvn spring-boot:run

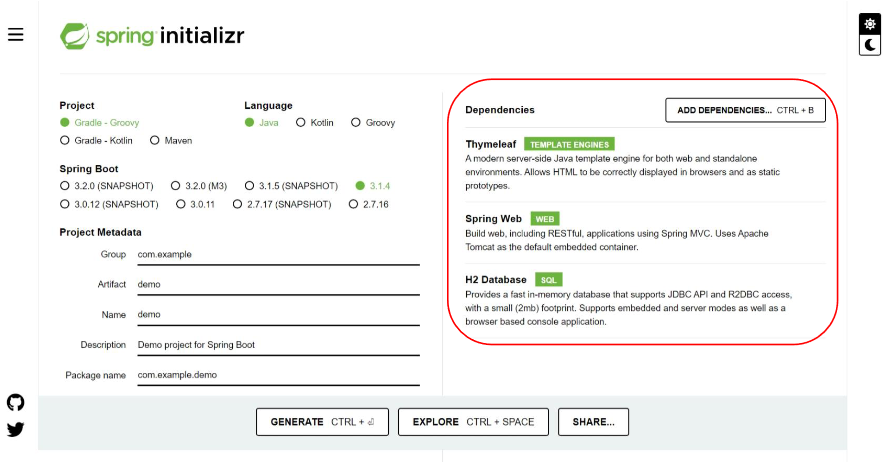
4) Maintenant que le service est opérationnel, visiter : [**http://localhost:8080/greeting**](http://localhost:8080/greeting)

5)[**http://localhost:8080/greeting?name=User**](http://localhost:8080/greeting?name=User)

Séance 02 :

**2.1 Démo Spring avec repository**

1) Start.spring.io : Dépendances Thymeleaf, Web, H2



2) Création de packages bootstrap, repositories, controllers et model

3) Création d’une entité Account : Account.java dans model  
  
@Entity  
public class Account {  
  
 @Id  
 private Long accountId;  
 private double accountAmount;  
  
 public Account() {  
 accountAmount = 0;  
 }  
  
 public Account(Long id, double amount) {  
 this.accountId = id;  
 this.accountAmount = amount;  
 }  
  
 public Long getAccountId() {  
 return accountId;  
 }  
  
 public void setAccountId(Long accountId) {  
 this.accountId = accountId;  
 }  
  
 public double getAccountAmount() {  
 return accountAmount;  
 }  
  
 public void setAccountAmount(double accountAmount) {  
 this.accountAmount = accountAmount;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
  
 Account account = (Account) o;  
  
 return accountId != null ? accountId.equals(account.accountId) : account.accountId == null;  
 }  
  
 @Override  
 public int hashCode() {  
 return accountId != null ? accountId.hashCode() : 0;  
 }  
  
 @Override  
 public String toString() {  
 return "Account{" +  
 "accountId=" + accountId +  
 ", accountAmount=" + accountAmount +  
 '}';  
 }  
}

4) Ajout d’une interface AccountRepository : dans repositories

package hepl.sysdist.springbootdemo.repositories;  
  
@Repository  
public interface AccountRepository extends CrudRepository<Account, Long> {  
}

5) Création de la classe InitDb : dans boostrap

package hepl.sysdist.springbootdemo.bootstrap;  
  
import hepl.sysdist.springbootdemo.model.Account;  
import hepl.sysdist.springbootdemo.repositories.AccountRepository;  
import org.springframework.boot.CommandLineRunner;  
import org.springframework.stereotype.Component;  
  
@Component  
public class InitDB implements CommandLineRunner {  
  
 private final AccountRepository accountRepository;  
  
 public InitDB(AccountRepository accountRepository) {  
 this.accountRepository = accountRepository;  
 }  
  
 @Override  
 public void run(String... args) throws Exception {  
 Account account1 = new Account((long) 931254956, 1000);  
 accountRepository.save(account1);  
  
 Account account2 = new Account((long) 945973355, 50000000);  
 accountRepository.save(account2);  
 }  
}

6) Création d’un contrôleur : AccountController.java

package hepl.sysdist.springbootdemo.controllers;  
  
@Controller  
public class AccountController {  
  
 private final AccountRepository accountRepository;  
  
 public AccountController(AccountRepository accountRepository) {  
 this.accountRepository = accountRepository;  
 }  
  
 @RequestMapping("/accounts")  
 public String getAccounts(Model model) {  
 model.addAttribute("accounts", accountRepository.findAll());  
 return "accounts/list";  
 }  
}

7) Création de la page list.html : dans ressources > tamplates.accounts

<!DOCTYPE html>  
<html lang="en" xmlns:th="http://www.thymeleaf.org">  
<head>  
<meta charset="UTF-8"/>  
<title>Spring MVC SYSDIST Demo</title>  
</head>  
<body>  
<h1>Account List</h1>  
<table>  
<tr>  
<th>ID</th>  
<th>Amount</th>  
</tr>  
<tr th:each="account : ${accounts}">  
<td th:text="${account.getAccountId()}"></td>  
<td th:text="${account.getAccountAmount()}"></td>  
</tr>  
</table>  
</body>  
</html>

8) Lancer l’application (créer le main : SpringbootdemoApplication ) puis **localhost:8080/accounts**

package hepl.sysdist.springbootdemo;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class SpringbootdemoApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(SpringbootdemoApplication.class, args);  
 }  
  
}

**2.2 Inversion de dépendances**

Imaginons que nous ayons une classe **ReportGenerator** qui génère des rapports, et une classe **PDFGenerator** qui génère des rapports au format PDF. Nous allons inverser la dépendance en utilisant l'injection de dépendance.

1) Créez une interface **ReportGenerator** :

public interface ReportGenerator {  
 void generateReport();  
}

2) Implémentez la classe **PDFGenerator** qui implémente cette interface :

@Service  
public class PDFGenerator implements ReportGenerator {  
 @Override  
 public void generateReport() {  
 *// Logique de génération de rapport PDF* System.*out*.println("Génération de rapport PDF...");  
 }  
}

3) Créez une classe qui dépend de **ReportGenerator** :

@Service  
public class ReportService {  
  
 private final ReportGenerator reportGenerator;  
  
 @Autowired  
 public ReportService(ReportGenerator reportGenerator) {  
 this.reportGenerator = reportGenerator;  
 }  
  
 public void generateReport() {  
 *// Utilisation de ReportGenerator pour générer le rapport* reportGenerator.generateReport();  
 }  
}

Dans cet exemple, la classe ReportService dépend de l'interface ReportGenerator, mais elle ne dépend pas directement de la classe PDFGenerator. La dépendance est inversée grâce à l'injection de dépendance.

4) Maintenant, vous pouvez créer une classe principale qui utilise **ReportService** :

@SpringBootApplication  
public class DemoApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.run(DemoApplication.class, args);  
 }  
  
 @Bean  
 public CommandLineRunner demo(ReportService reportService) {  
 return args -> {  
 *// Génération de rapport via ReportService* reportService.generateReport();  
 };  
 }  
}

Dans cette classe principale, nous injectons le **ReportService** à l'aide de **@Bean**. Lorsque l'application démarre, le rapport sera généré en utilisant la logique définie dans **PDFGenerator**.

Cela illustre comment l'inversion de dépendance permet de créer des dépendances entre des interfaces ou des contrats, et comment Spring Boot gère automatiquement l'injection de dépendances en utilisant son conteneur IoC. Cette approche rend votre code plus modulaire et flexible.

**2.3 Contexte de l’application et Récupération d’un Bean**

**(La même chose que le 1.2 !)**

**2.4 Mise en œuvre de l’injection de dépendances manuellement**

1) Création du projet avec spring.start.io

2) Création de deux répertoires : controllers et services

3) Interface GreetingService.java :

package hepl.systdist.manualdi.services;  
  
public interface GreetingService {  
 String sayGreeting();  
  
}

4) La classe GreetingServiceImpl.java :

package hepl.systdist.manualdi.services;  
  
@Service  
public class GreetingServiceImpl implements GreetingService {  
 @Override  
 public String sayGreeting() {  
 return "hello";  
 }  
}

5) La classe ConstructorInjectedController.java :

package hepl.systdist.manualdi.controllers;  
  
@Controller  
public class ConstructorInjectedController {  
  
 private final GreetingService greetingService;  
  
 public ConstructorInjectedController(GreetingService greetingService) {  
 this.greetingService = greetingService;  
 }  
  
 public String getGreeting() {  
 return greetingService.sayGreeting();  
 }  
}

6) La classe PropertyInjectedController.java :

package hepl.systdist.manualdi.controllers;  
  
import hepl.systdist.manualdi.services.GreetingService;  
  
public class PropertyInjectedController {  
  
 public GreetingService greetingService;  
  
 public GreetingService getGreetingService() {  
 return greetingService;  
 }  
  
 public String getGreeting() {  
 return getGreetingService().sayGreeting();  
 }  
  
}

7) Lancer l’application (ManualDiApplication.java):

package hepl.systdist.manualdi;  
  
@SpringBootApplication  
public class ManualDiApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(ManualDiApplication.class, args);  
 }  
  
}

**2.5 Mise en œuvre de l’injection de dépendances avec Spring**

1) Création du projet avec spring.start.io

2) Création de deux répertoires : controllers et services

3) Interface GreetingService.java :

package hepl.sysdist.springdi.services;  
  
public interface GreetingService {  
 String sayGreeting();  
  
}

4) La classe GreetingServiceImpl.java :

package hepl.sysdist.springdi.services;  
  
import org.springframework.stereotype.Service;  
  
@Service  
public class GreetingServiceImpl implements GreetingService {  
 @Override  
 public String sayGreeting() {  
 return "hello";  
 }  
}

5) La classe ConstructorInjectedController.java :

package hepl.sysdist.springdi.controllers;  
  
import hepl.sysdist.springdi.services.GreetingService;  
  
@Controller  
public class ConstructorInjectedController {  
 *// Autowired not required* private final GreetingService greetingService;  
  
 public ConstructorInjectedController(GreetingService greetingService) {  
 this.greetingService = greetingService;  
 }  
  
 public String getGreeting(){  
 return greetingService.sayGreeting();  
 }  
}

6) La classe PropertyInjectedController.java :

package hepl.sysdist.springdi.controllers;  
  
import hepl.sysdist.springdi.services.GreetingService;

@Controller  
public class PropertyInjectedController {  
  
 @Autowired  
 public GreetingService greetingService;  
  
 public GreetingService getGreetingService() {  
 return greetingService;  
 }  
  
 public String getGreeting() {  
 return getGreetingService().sayGreeting();  
 }  
}

7) Lancer l’application (ManualDiApplication.java):

package hepl.sysdist.springdi;  
  
import hepl.sysdist.springdi.controllers.ConstructorInjectedController;  
import hepl.sysdist.springdi.controllers.PropertyInjectedController;  
  
@SpringBootApplication  
public class SpringDiApplication {  
  
 public static void main(String[] args) {  
 ApplicationContext ctx = SpringApplication.*run*(SpringDiApplication.class, args);  
 PropertyInjectedController propertyInjectedController = (PropertyInjectedController) ctx.getBean("propertyInjectedController");  
 String message = propertyInjectedController.getGreeting();  
 System.*out*.println(message);  
  
 ConstructorInjectedController constructorInjectedController = (ConstructorInjectedController) ctx.getBean("constructorInjectedController");  
 message = constructorInjectedController.getGreeting();  
 System.*out*.println(message);  
 }  
  
}

**2.6 Illustration du componentScan**

1) Création du projet avec spring.start.io

2) Création de deux répertoires : controllers et services

3) Interface GreetingService.java :

package hepl.systdist.manualdi.services;  
  
public interface GreetingService {  
 String sayGreeting();  
  
}

4) La classe GreetingServiceImpl.java :

package hepl.systdist.manualdi.services;  
  
@Service  
public class GreetingServiceImpl implements GreetingService {  
 @Override  
 public String sayGreeting() {  
 return "hello";  
 }  
}

5) La classe ConstructorInjectedController.java :

package hepl.systdist.manualdi.controllers;  
  
@Controller  
public class ConstructorInjectedController {  
  
 private final GreetingService greetingService;  
  
 public ConstructorInjectedController(GreetingService greetingService) {  
 this.greetingService = greetingService;  
 }  
  
 public String getGreeting() {  
 return greetingService.sayGreeting();  
 }  
}

6) La classe PropertyInjectedController.java :

package hepl.systdist.manualdi.controllers;  
  
import hepl.systdist.manualdi.services.GreetingService;

@Controller  
public class PropertyInjectedController {

@Autowired  
 public GreetingService greetingService;  
  
 public GreetingService getGreetingService() {  
 return greetingService;  
 }  
  
 public String getGreeting() {  
 return getGreetingService().sayGreeting();  
 }  
  
}

7) Lancer l’application (ManualDiApplication.java):

package hepl.systdist.manualdi;  
  
@SpringBootApplication

@ComponentScan(basePackages = "hepl.systdist.manualdi ")  
public class ManualDiApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(ManualDiApplication.class, args);  
 }  
  
}

**2.7 Utilisation de l’annotation Qualifier**

L'annotation @Qualifier est utilisée en conjonction avec @Autowired pour résoudre l'ambiguïté lorsque plusieurs beans du même type sont disponibles pour l'injection de dépendances dans un composant Spring. Elle permet de préciser quel bean doit être injecté. Voici comment vous pouvez l'utiliser avec votre code :

Supposons que vous ayez deux implémentations de GreetingService, par exemple GreetingServiceImpl (déjà existant dans votre code) et une nouvelle classe FriendlyGreetingServiceImpl. Vous devez vous assurer que Spring sait quel bean doit être injecté dans le contrôleur.

1) Création du projet avec spring.start.io

2) Création de deux répertoires : controllers et services

3) Interface GreetingService.java :

package hepl.systdist.manualdi.services;  
  
public interface GreetingService {  
 String sayGreeting();  
  
}

4) La classe GreetingServiceImpl.java :

package hepl.systdist.manualdi.services;  
  
@Service  
public class GreetingServiceImpl implements GreetingService {  
 @Override  
 public String sayGreeting() {  
 return "hello";  
 }  
}

5) La classe FriendlyGreetingServiceImpl :

package hepl.sysdist.springdi.services;  
  
@Service  
public class FriendlyGreetingServiceImpl implements GreetingService {  
 @Override  
 public String sayGreeting() {  
 return "Hi there!";  
 }  
}

6) La classe ConstructorInjectedController.java :

package hepl.sysdist.springdi.controllers;  
  
import hepl.sysdist.springdi.services.GreetingService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Controller;  
  
@Controller  
public class ConstructorInjectedController {  
 private final GreetingService greetingService;  
  
 @Autowired  
 public ConstructorInjectedController(@Qualifier("friendlyGreetingServiceImpl") GreetingService greetingService) {  
 this.greetingService = greetingService;  
 }  
  
 public String getGreeting() {  
 return greetingService.sayGreeting();  
 }  
}

7) La classe PropertyInjectedController.java :

package hepl.systdist.manualdi.controllers;  
  
import hepl.systdist.manualdi.services.GreetingService;

@Controller  
public class PropertyInjectedController {

@Autowired  
 public GreetingService greetingService;  
  
 public GreetingService getGreetingService() {  
 return greetingService;  
 }  
  
 public String getGreeting() {  
 return getGreetingService().sayGreeting();  
 }  
  
}

8) Lancer l’application (ManualDiApplication.java):

package hepl.systdist.manualdi;  
  
@SpringBootApplication

@ComponentScan(basePackages = "hepl.systdist.manualdi ")  
public class ManualDiApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(ManualDiApplication.class, args);  
 }  
  
}

Assurez-vous d'ajouter l'annotation @Qualifier avec le nom du bean que vous souhaitez injecter ("friendlyGreetingServiceImpl") dans votre constructeur. Cela indique à Spring d'injecter le bon bean dans ConstructorInjectedController.

De cette manière, lorsque vous utilisez @Autowired avec @Qualifier, vous spécifiez explicitement quel bean Spring doit injecter. Cela résout l'ambiguïté si plusieurs beans du même type sont disponibles pour l'injection de dépendances. Vous pouvez également appliquer la même approche au PropertyInjectedController si nécessaire.

**2.8 Utilisation de profiles**

Les profils (profiles) dans Spring permettent de configurer des beans et des paramètres spécifiques pour différents environnements (développement, production, test, etc.) ou scénarios d'exécution. Cela vous permet de personnaliser votre application en fonction de l'environnement d'exécution sans avoir à changer le code source.

1) Création du projet avec spring.start.io

2) Creation de l’interface GreetingService.java :

package hepl.sysdist.springprofiledi.services;  
  
public interface GreetingService {  
 String sayGreeting();  
}

2) Création de la classe EnGreetingService.java :

package hepl.sysdist.springprofiledi.services;  
  
@Profile("EN")  
@Service("greetingservice")  
public class EnGreetingService implements GreetingService{  
 @Override  
 public String sayGreeting() {  
 return "Hello from EnGreetingService";  
 }  
}

3) Création de la classe FrGreetingService.java :

package hepl.sysdist.springprofiledi.services;  
  
@Profile("FR")  
@Service("greetingservice")  
public class FrGreetingService implements GreetingService{  
 @Override  
 public String sayGreeting() {  
 return "Salut de FrGreetingService";  
 }  
}

4) Création de la classe AController.java :

package hepl.sysdist.springprofiledi.controllers;  
  
@Controller  
public class AController {  
  
 private final GreetingService greetingService;  
  
 public AController(@Qualifier("greetingservice") GreetingService greetingService) {  
 this.greetingService = greetingService;  
 }  
  
 public String getGreeting() {  
 return greetingService.sayGreeting();  
 }  
}

5) Maintenant, lorsque vous exécutez votre application avec le profil actif approprié (défini dans spring.profiles.active), Spring utilisera automatiquement les beans et les configurations spécifiques à ce profil.

Dans resources ==> spring.profiles.active : spring.profiles.active=FR

6) Lancer l’application :

package hepl.sysdist.springprofiledi;  
  
import hepl.sysdist.springprofiledi.controllers.AController;  
  
@SpringBootApplication  
public class SpringprofileDiApplication {  
  
 public static void main(String[] args) {  
 ApplicationContext ctx = SpringApplication.*run*(SpringprofileDiApplication.class, args);  
 AController controller=(AController)ctx.getBean("AController");  
 System.*out*.println(controller.getGreeting());  
 }  
  
}

Séance 03 :

**3.1 Web Service avec requête POST**

1) Création du projet avec start.spring.io (avec dépendance web)

2) Classe principale Spring Boot (Application.java):  
@SpringBootApplication  
public class Application {  
 public static void main(String[] args) {  
 SpringApplication.*run*(Application.class, args);  
 }  
}

3) Contrôleur REST (SimpleController.java):  
@RestController  
public class SimpleController {  
  
 @PostMapping("/postendpoint")  
 public String handlePostRequest(@RequestBody String requestBody) {  
 return "Requête POST reçue avec le contenu : " + requestBody;  
 }  
}

Dans cet exemple, lorsque vous envoyez une requête POST à /postendpoint avec du contenu dans le corps de la requête, le contrôleur renvoie simplement une chaîne indiquant que la requête a été reçue avec le contenu fourni.

Pour tester votre service web, vous pouvez utiliser des outils comme curl ou Postman pour envoyer une requête POST à l'URL http://localhost:8080/postendpoint après avoir démarré l'application Spring Boot.

4) Client (SimpleClient.java):  
public class SimpleClient {  
  
 public static void main(String[] args) {  
 *// Créer une instance du client HTTP* HttpClient client = HttpClient.newHttpClient();  
  
 *// Créer une requête POST avec un corps JSON* HttpRequest request = HttpRequest.newBuilder()  
 .uri(URI.create("http://localhost:8080/postendpoint"))  
 .header("Content-Type", "application/json")  
 .POST(BodyPublishers.ofString("{\"key\":\"value\"}"))  
 .build();  
  
 *// Envoyer la requête et obtenir la réponse* client.sendAsync(request, BodyHandlers.ofString())  
 .thenApply(HttpResponse::body)  
 .thenAccept(System.*out*::println)  
 .join();  
 }  
}

Le message envoyé par le client Java que j'ai fourni est une requête HTTP POST avec un corps JSON. Le corps JSON dans l'exemple est `{"key":"value"}`. Voici ce qui se passe lorsque le client envoie ce message :

1. \*\*URI :\*\* La requête est destinée à l'URI `http://localhost:8080/postendpoint`. Cela signifie que le client tente de communiquer avec un service web qui écoute les requêtes POST sur cet endpoint.

2. \*\*Header :\*\* La requête inclut un en-tête `Content-Type` défini sur `application/json`, indiquant que le corps de la requête est au format JSON.

3. \*\*Corps de la requête :\*\* Le corps de la requête contient un objet JSON, qui est une manière courante de transmettre des données structurées via HTTP. Dans ce cas, l'objet JSON est très simple et contient une paire clé-valeur où la clé est `"key"` et la valeur est `"value"`.

4. \*\*Envoi de la requête :\*\* Le client utilise la méthode `sendAsync` pour envoyer la requête de manière asynchrone. Cela signifie que le programme n'attend pas que la réponse soit reçue avant de continuer son exécution.

5. \*\*Gestion de la réponse :\*\* Une fois la réponse reçue du service web, le corps de la réponse (supposé être une chaîne de caractères) est imprimé sur la console. Ceci est réalisé par la chaîne d'appels de méthodes `thenApply` et `thenAccept` qui sont des opérations de la programmation fonctionnelle en Java pour gérer les futures promesses retournées par les appels asynchrones.

En résumé, le message envoyé est une requête POST avec un corps JSON qui dit au service web de faire quelque chose avec les données `{ "key": "value" }`. Ce quelque chose dépend entièrement de la logique implémentée dans le service web à l'endpoint `/postendpoint`.

**3.2 Mise en place d’un appel distant (RPC) avec gRPC (« Hello World » avec BlockingStub)**

Pour créer un service gRPC en utilisant Protocol Buffers, vous devez suivre ces étapes :

1. Définir le service dans un fichier .proto : Un fichier .proto est un fichier de définition pour les services gRPC et les messages Protocol Buffers. Il spécifie la structure des messages envoyés et reçus par votre service gRPC.
2. Générer le code serveur et client : En utilisant le compilateur de Protocol Buffers.
3. Écrire le client et le serveur en utilisant l'API gRPC Java : Avec le code généré, vous écrivez la logique pour le serveur qui gère les appels RPC et pour le client qui les effectue.

Voici un exemple simple de chaque étape pour un service gRPC qui envoie un message "Hello" :

Étape 1 : Définir le service dans un fichier .proto (hello\_service.proto)

syntax = "proto3";  
  
 package hello;  
  
*// Le service Hello défini avec une méthode sayHello* service HelloService {  
 *// Envoie un message HelloRequest et reçoit une réponse HelloResponse* rpc sayHello (HelloRequest) returns (HelloResponse);  
 }  
  
*// Le message de demande contenant un nom* message HelloRequest {  
 string name = 1;  
 }  
  
*// Le message de réponse contenant un message de salutation* message HelloResponse {  
 string greeting = 1;  
 }

Étape 2 : Générer le code serveur et client

Pour générer le code, vous utiliseriez :

**mvn generate-sources**

Cela générerait des fichiers dans votre répertoire target/generated-sources que vous pouvez ensuite utiliser dans votre projet Java.

Étape 3 : Écrire le serveur en utilisant l'API gRPC Java (HelloServer.java)

import io.grpc.Server;  
import io.grpc.ServerBuilder;  
import io.grpc.stub.StreamObserver;  
import hello.HelloServiceGrpc;  
import hello.HelloRequest;  
import hello.HelloResponse;  
  
public class HelloServer {  
 public static void main(String[] args) throws Exception {  
 Server server = ServerBuilder.forPort(8080)  
 .addService(new HelloServiceImpl())  
 .build()  
 .start();  
  
 System.*out*.println("Server started on port 8080.");  
 server.awaitTermination();  
 }  
  
 static class HelloServiceImpl extends HelloServiceGrpc.HelloServiceImplBase {  
 @Override  
 public void sayHello(HelloRequest req, StreamObserver<HelloResponse> responseObserver) {  
 HelloResponse reply = HelloResponse.newBuilder()  
 .setGreeting("Hello " + req.getName())  
 .build();  
 responseObserver.onNext(reply);  
 responseObserver.onCompleted();  
 }  
 }  
}

Étape 3 : Écrire le client en utilisant l'API gRPC Java (HelloClient.java)

import io.grpc.ManagedChannel;  
import io.grpc.ManagedChannelBuilder;  
import hello.HelloServiceGrpc;  
import hello.HelloRequest;  
import hello.HelloResponse;  
  
public class HelloClient {  
 public static void main(String[] args) throws Exception {  
 ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost", 8080)  
 .usePlaintext()  
 .build();  
  
 HelloServiceGrpc.HelloServiceBlockingStub stub = HelloServiceGrpc.newBlockingStub(channel);  
  
 HelloResponse response = stub.sayHello(HelloRequest.newBuilder().setName("World").build());  
  
 System.*out*.println("Greeting: " + response.getGreeting());  
 channel.shutdown();  
 }  
}

Dans cet exemple, le service HelloService définit une seule méthode RPC sayHello, qui prend un HelloRequest et renvoie un HelloResponse. Le serveur implémente cette méthode et renvoie un message de salutation, tandis que le client envoie une demande et imprime la réponse.

Pour exécuter l'exemple, vous devrez ajouter les dépendances gRPC nécessaires à votre projet, générer le code à partir du fichier .proto, et exécuter le serveur avant le client. Assurez-vous que gRPC est inclus dans votre gestionnaire de dépendances (comme Maven ou Gradle) pour que les bibliothèques nécessaires soient importées dans votre projet.

**3.3 Mise en œuvre de JMS avec Spring**

Java Message Service (JMS) est une API Java qui permet d'envoyer des messages entre deux ou plusieurs clients. Elle est utilisée pour la communication entre composants informatiques basée sur le concept de mise en file d'attente des messages. Spring fournit un support pour JMS via le module spring-jms, qui simplifie l'utilisation de JMS en gérant les détails de bas niveau.

Pour mettre en œuvre JMS avec Spring, vous devez suivre ces étapes :

Ajouter les dépendances : Incluez spring-jms et un fournisseur JMS comme ActiveMQ ou Artemis dans votre fichier de configuration de build (par exemple, pom.xml pour Maven).

Configurer la connexion JMS : Configurez un ConnectionFactory et éventuellement un JmsTemplate pour simplifier l'envoi et la réception de messages.

Créer un message listener : Créez une classe qui écoute les messages sur une queue ou un topic spécifique.

Envoyer des messages : Utilisez le JmsTemplate pour envoyer des messages à la queue ou au topic.

Voici un exemple très simple :

1. Ajouter les dépendances dans pom.xml:

<dependencies>  
<!-- Spring Boot starter for JMS -->  
<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-activemq</artifactId>  
</dependency>  
<!-- Spring Boot starter web, for REST Controller -->  
<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-web</artifactId>  
</dependency>  
</dependencies>

2. Configurer la connexion JMS dans application.properties:

spring.activemq.broker-url=tcp://localhost:61616

spring.activemq.user=admin

spring.activemq.password=admin

3. Créer un message listener (JmsListener.java):

import org.springframework.jms.annotation.JmsListener;  
import org.springframework.stereotype.Component;  
  
@Component  
public class JmsListener {  
  
 @JmsListener(destination = "test-queue")  
 public void receiveMessage(String message) {  
 System.*out*.println("Received <" + message + ">");  
 }  
}

4. Envoyer des messages en utilisant JmsTemplate (MessageSender.java):

import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.jms.core.JmsTemplate;  
import org.springframework.stereotype.Service;  
  
@Service  
public class MessageSender {  
  
 @Autowired  
 private JmsTemplate jmsTemplate;  
  
 public void sendMessage(String destination, String message) {  
 jmsTemplate.convertAndSend(destination, message);  
 }  
}

5. Créer une application Spring Boot (Application.java):

import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class Application {  
 public static void main(String[] args) {  
 SpringApplication.*run*(Application.class, args);  
 }  
}

6. Créer un contrôleur pour déclencher l'envoi de messages (MessageController.java):

import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.RequestParam;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class MessageController {  
  
 @Autowired  
 private MessageSender messageSender;  
  
 @PostMapping("/send")  
 public String sendMessage(@RequestParam String message) {  
 messageSender.sendMessage("test-queue", message);  
 return "Message sent to the queue!";  
 }  
}

Ce code de base vous donne un système de messagerie JMS fonctionnel avec Spring Boot. Vous pouvez démarrer l'application Spring Boot et envoyer un message via le contrôleur REST. Le JmsListener écoute ensuite sur la queue "test-queue" et affiche les messages reçus.

Assurez-vous que votre serveur de messagerie JMS est en cours d'exécution et accessible à l'URL spécifiée dans application.properties. Pour Apache ActiveMQ, l'URL par défaut est tcp://localhost:61616 et les identifiants par défaut sont souvent "admin" pour le nom d'utilisateur et le mot de passe, mais cela peut varier en fonction de votre configuration.

**3.4 Mise en œuvre d’une Queue avec ActiveMQ**

**(La même chose que le 3.3 !)**

**bin\activemq.bat start**

**3.5 Mise en œuvre d’un Topic avec ActiveMQ**

La mise en œuvre d'un topic avec ActiveMQ dans une application Spring Boot est similaire à celle d'une queue, avec quelques différences clés. Un topic permet de publier des messages à plusieurs abonnés, ce qui est différent du modèle de queue où le message est consommé par un seul consommateur. Voici les étapes pour configurer un topic avec ActiveMQ et Spring Boot :

1. \*\*Ajoutez les dépendances nécessaires\*\* dans votre projet Spring Boot. Si vous utilisez Maven, ajoutez ceci à votre fichier `pom.xml` :

<dependencies><!-- Spring Boot Starter for JMS -->  
<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-activemq</artifactId>  
</dependency>  
<!-- Spring Boot Starter Web (facultatif, pour l'exposition de l'API REST) -->  
<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-web</artifactId>  
</dependency>  
</dependencies>

2. \*\*Configurez ActiveMQ\*\* dans votre fichier `application.properties` de Spring Boot :

spring.activemq.broker-url=tcp:*//localhost:61616*spring.activemq.user=admin  
spring.activemq.password=admin

3. \*\*Créez un composant Producteur\*\* pour publier des messages au topic :

import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.jms.core.JmsTemplate;  
import org.springframework.stereotype.Component;  
import javax.jms.Topic;  
  
@Component  
public class TopicPublisher {  
  
 @Autowired  
 private JmsTemplate jmsTemplate;  
  
 @Autowired  
 private Topic topic;  
  
 public void publish(String message) {  
 jmsTemplate.convertAndSend(topic, message);  
 }  
}

4. \*\*Créez un composant Consommateur\*\* pour s'abonner et recevoir des messages du topic :  
import org.springframework.jms.annotation.JmsListener;  
import org.springframework.stereotype.Component;  
  
@Component  
public class TopicSubscriber {  
  
 @JmsListener(destination = "your-topic-name", containerFactory = "jmsListenerContainerFactory")  
 public void receive(String message) {  
 System.*out*.println("Subscriber received message: " + message);  
 }  
}

5. \*\*Configurer un `JmsListenerContainerFactory`\*\* pour les topics :

import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;  
import org.springframework.jms.config.JmsListenerContainerFactory;  
import org.springframework.jms.core.JmsTemplate;  
  
import javax.jms.ConnectionFactory;  
  
@Configuration  
public class ActiveMQConfig {  
  
 @Bean  
 public JmsListenerContainerFactory<?> jmsListenerContainerFactory(ConnectionFactory connectionFactory) {  
 DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory();  
 factory.setPubSubDomain(true); *// This will configure the factory for topic support* factory.setConnectionFactory(connectionFactory);  
 return factory;  
 }  
  
 @Bean  
 public Topic topic() {  
 return new ActiveMQTopic("your-topic-name");  
 }  
}

6. \*\*Créez un point de terminaison API REST\*\* pour déclencher la publication de messages (facultatif) :

import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.RequestBody;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class MessageController {  
  
 @Autowired  
 private TopicPublisher topicPublisher;  
  
 @PostMapping("/publish")  
 public String publish(@RequestBody String message) {  
 topicPublisher.publish(message);  
 return "Message published to the topic!";  
 }  
}

7. \*\*Exécutez votre application Spring Boot\*\* :

import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class ActiveMqTopicApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(ActiveMqTopicApplication.class, args);  
 }  
}

Après avoir démarré votre application Spring Boot, vous pouvez tester la publication de messages au topic en utilisant le point de terminaison `/publish`. Tous les abonnés au topic `your-topic-name` recevront les messages publiés.

Assurez-vous qu'ActiveMQ est installé et en cours d'exécution sur votre machine et que le serveur de messages est accessible à `tcp://localhost:61616` avec les identifiants par défaut `admin/admin`.

Cette configuration vous permet de publier des messages à un topic dans ActiveMQ, et tous les abonnés inscrits au topic recevront ces messages.