Spring 5, développer des applications d'entreprise

Dr. Ing. Ramzi FARHAT

ramzi.farhat@yahoo.fr

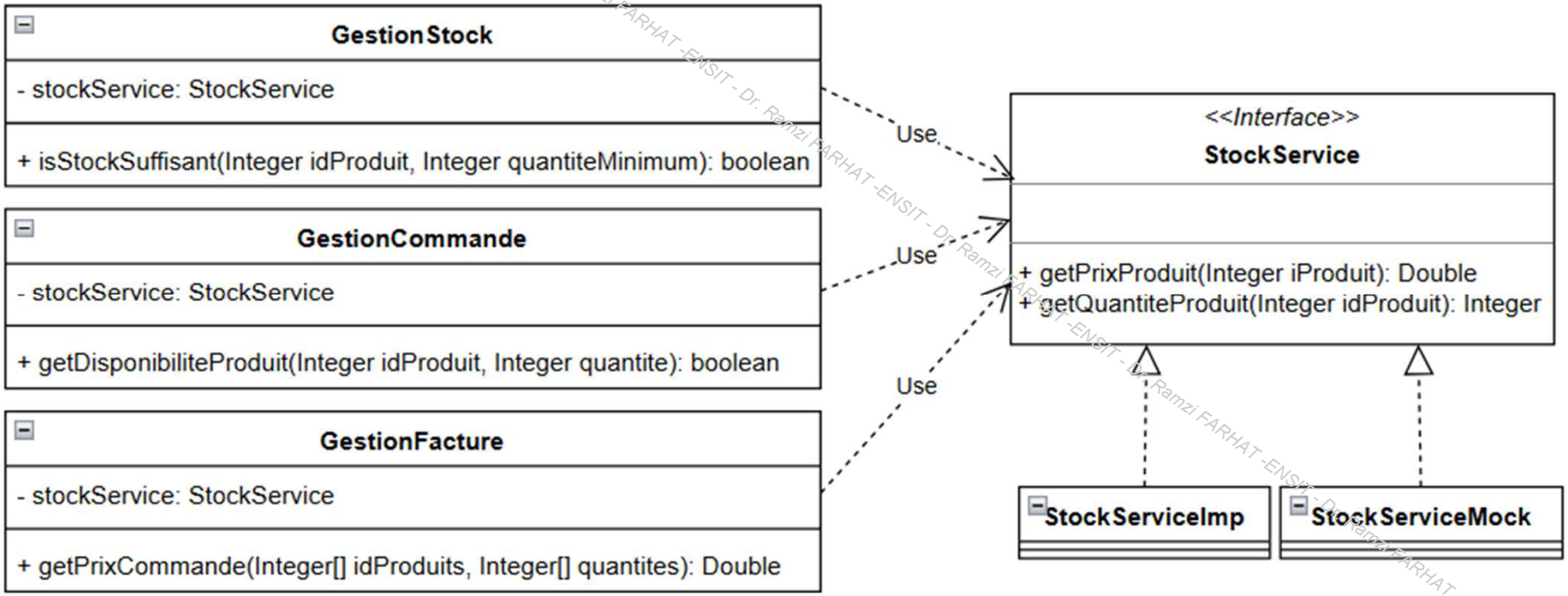
Gestion des beans avec Spring Core

Chapitre 2

- Injection de Dépendances
- IoD par Annotations
- IoD par Configuration Java
- IoD par Configuration XML

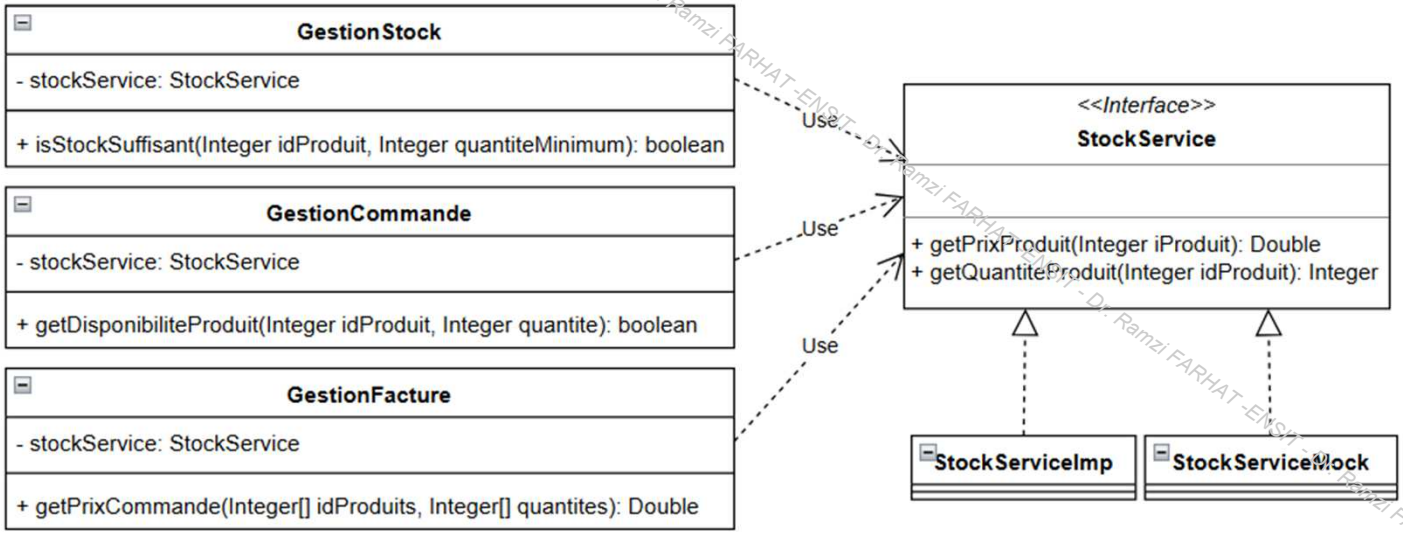
Sans Injection de Dépendances

- On va créer un projet Java boutique_v1 qui consiste correspond au diagramme de classes suivant :



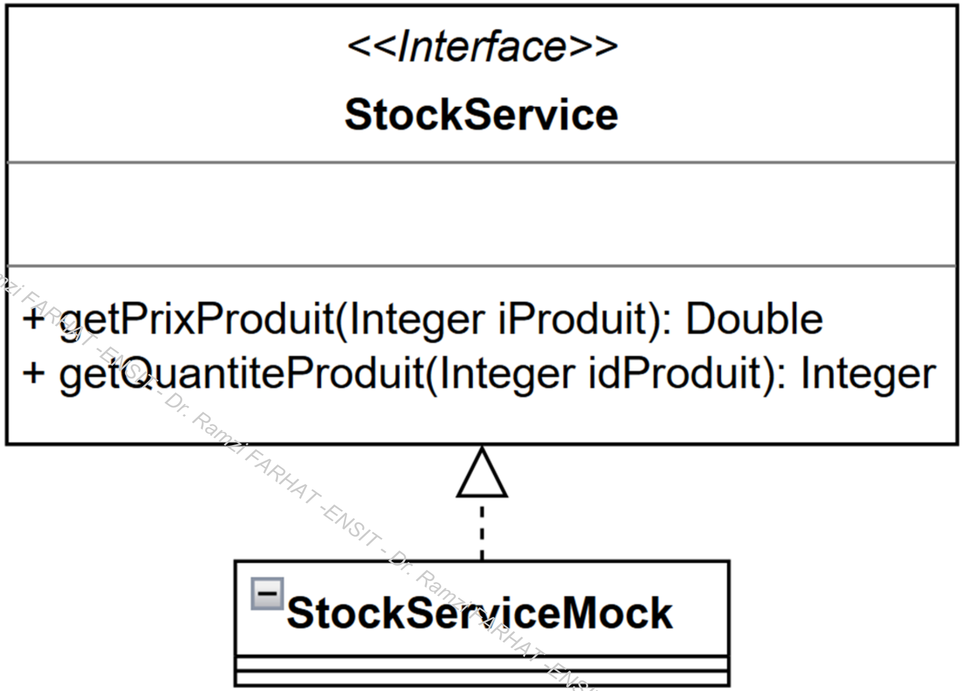
Sans Injection de Dépendances

- **Mock :**
  - Au départ on s'intéresse à la partie Gestion, donc on va utiliser un **StockServiceMock** pour tester la partie Gestion. Implémentez le code sous forme d'un projet Java :



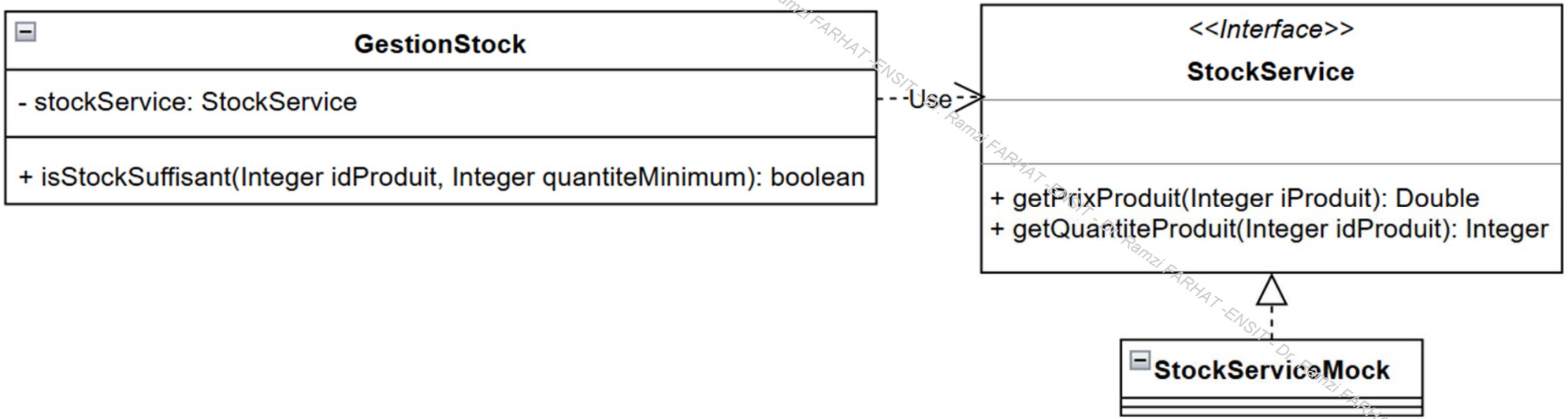
Sans Injection de Dépendances

- Implémentez l’interface **StockService**
- Implémenter la classe **StockServiceMock** :
  - **getPrixProduit()** retourne comme prix l’id du produit multiplié par 3,5
  - **getQuantiteProduit()** retourne comme quantité soit id x 2 si l’id est pair, sinon 0.



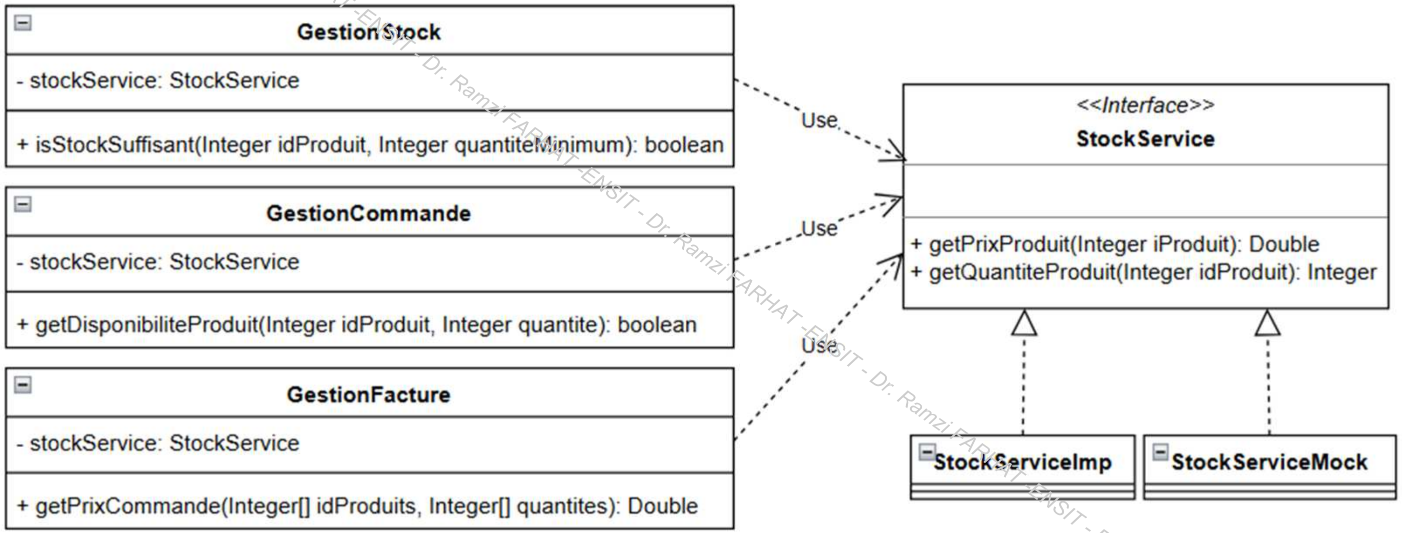
Sans Injection de Dépendances

- Implémentez la classe **GestionStock**
  - Le constructeur permet d'initialiser **stockService** avec une instance de **StockServiceMock**
  - **isStockSuffisant()** utilise le **stockService** pour retourner un booléen.



Sans Injection de Dépendances

- Implémenter les classes **GestionDeCommande** et **GestionDeFacture** de la même façon



Sans Injection de Dépendances

- Créez une classe de Test :

```
public class Main {
 public static void main(String[] args) {
   GestionStock gs = new GestionStock();
   for (int i = 1; i <= 3; i++) {
      if (gs.isStockSuffisant(i, 3)) {
         System.out.println("Stock produit "+ i + " est suffisant.");
      }else {
         System.out.println("Il faut réapprovisionner le stock du produit : "+i);
      }
  }
  GestionCommande gc = new GestionCommande();
  for (int i = 1; i <= 3; i++) {
     if (gc.getDisponibiliteProduit(i, 1)) {
        System.out.println("Produit "+ i +" disponible.");
     }else {
        System.out.println("Un exemplaire du produit "+ i + " manque ...");
     }
  }
  GestionFacture gf = new GestionFacture();
  System.out.println("Total commande : " + gf.getPrixCommande(new Integer[] {1, 2, 3}, new Integer[]
{1, 1, 1}));
 }
}
```

```
<terminated> Main [Java Application] D:\Applications\eclipse-jee-2023-12-R-win32-x86_64\eclipse
>>> GESTION DU STOCK
Il faut réapprovisionner le stock du produit : 1
Stock produit 2 est suffisant.
Il faut réapprovisionner le stock du produit : 3
>>> GESTION DE COMMANDE
Un exemplaire du produit 1 manque ...
Produit 2 disponible.
Un exemplaire du produit 3 manque ...
>>> GESTION DE FACTURE
Total commande : 21.0
```

Sans Injection de Dépendances

C'est quoi le problème ?

```
public class GestionCommande {
    private StockService stockService;
    public GestionCommande() {
        stockService = new StockServiceMock();
    }
    public boolean getDisponibiliteProduit(Integer idProduit, Integer quantite) {
}
```

```
public class GestionFacture {
    private StockService stockService;
    public GestionFacture() {
        this.stockService = new StockServiceMock();
    }
    public Double getPrixCommande(Integer[] idProduits, Integer[] quantites) {
}
```

```
public class GestionStock {
    private StockService stockService;
    public GestionStock() {
        this.stockService = new StockServiceMock();
    }
    public boolean isStockSuffisant(Integer idProduit, Integer quantiteMinimum) {
}
```

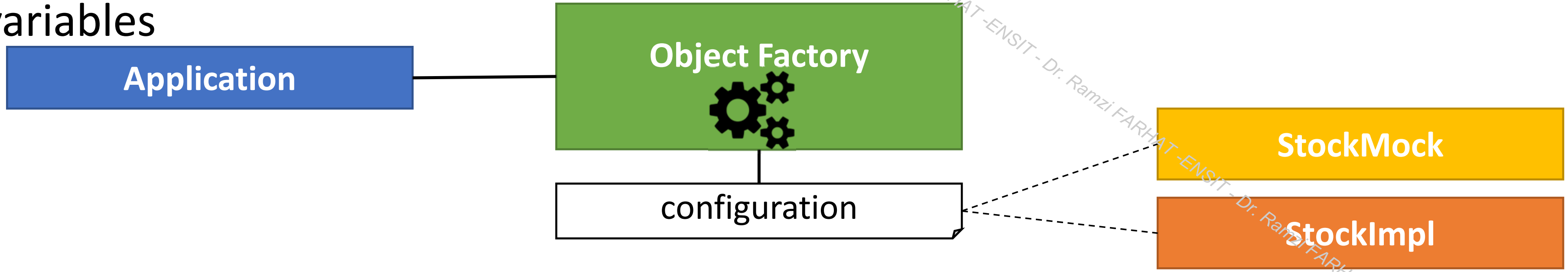
Injection de dépendances

- Principe :
  - Eviter de créer explicitement les objets :

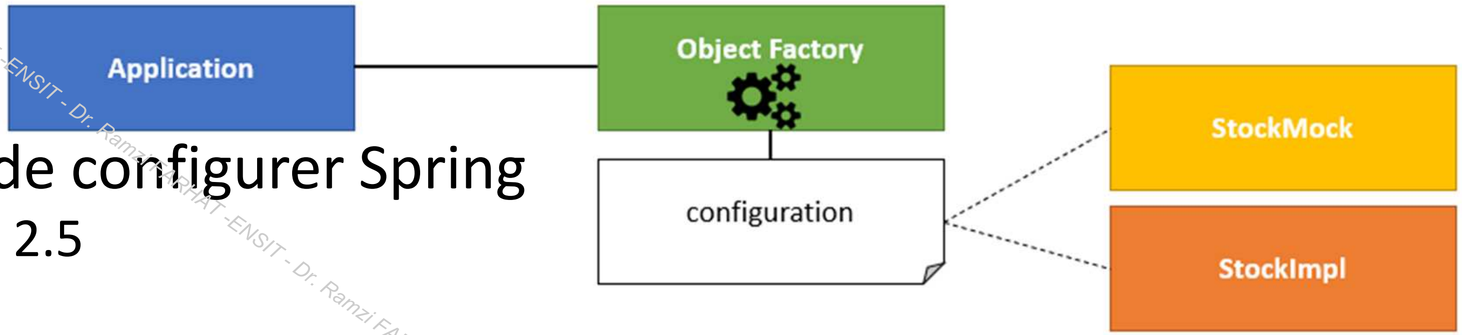
    StockService stockService = **new** StockServiceMock();

    Ou

    StockService stockService = **new** StockServiceImp();
- Démarche :
  - Le conteneur Spring va se charger de la création des objets et d'initialiser les variables



Configurer l'injection de dépendances



- Il existe en fait 3 manières de configurer Spring
  - **Annotations** : depuis Spring 2.5
    - Plus rapide à utiliser
    - Plus simple : ne convient qu'à de la configuration «métier»
  - **Java** : depuis Spring 3.0
    - Permet de coder en Java quelque chose de similaire à la configuration XML
    - Plus puissant (c'est du code, on peut faire ce qu'on veut)
    - Moins simple à modifier, en particulier pour de la configuration «d'infrastructure»
    - Moins répandu
  - **XML** : méthode «classique», très souple et très puissante
    - Essentielle à connaître
    - Convient très bien à la configuration dite «d'infrastructure»

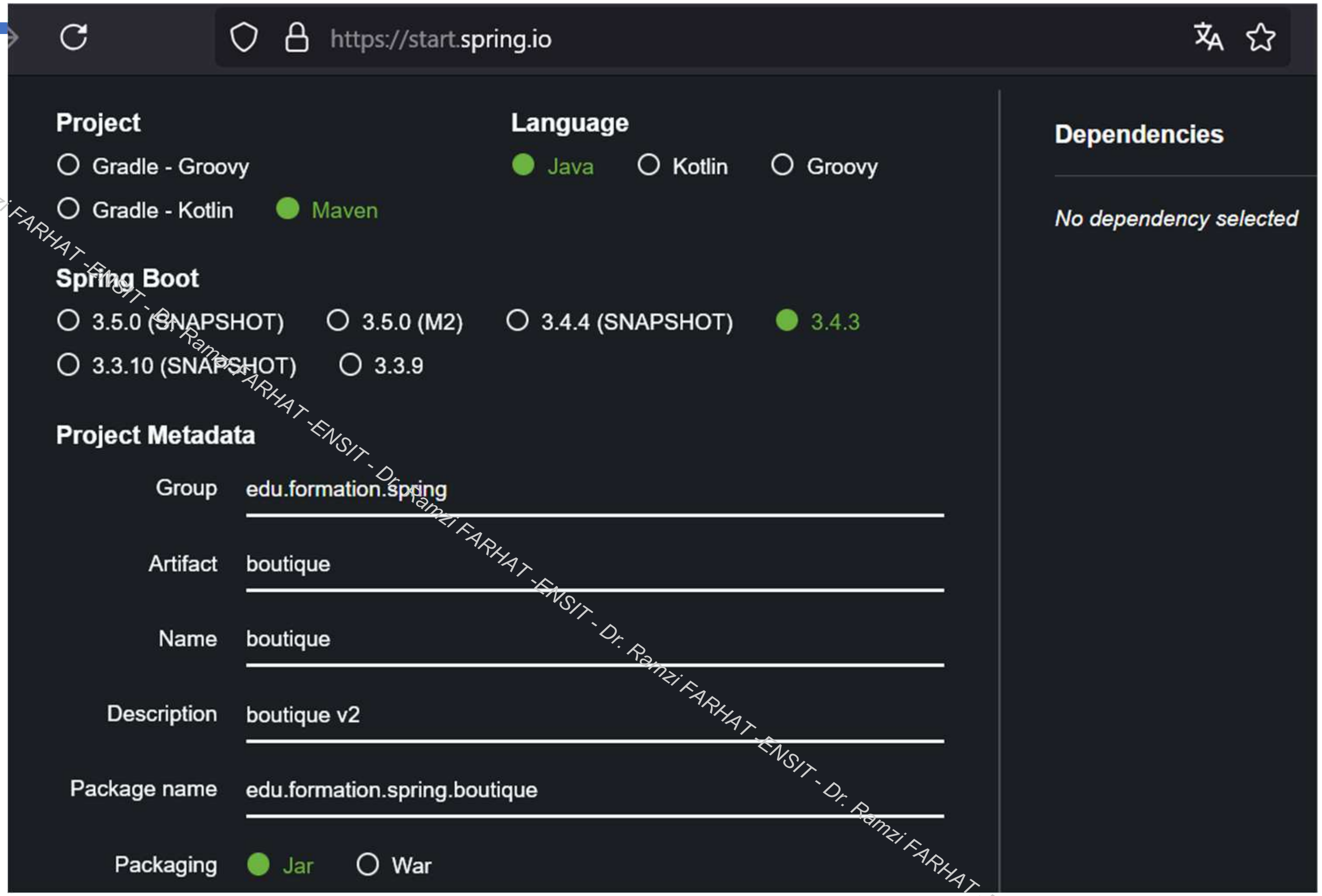
Spring Container

- Fonctions principales
  - Créer et gérer les objets (Inversion of Control)
  - Injecter les dépendances d'objets (Dependency Injection)

- Il faut créer un projet Spring pour avoir le conteneur Spring

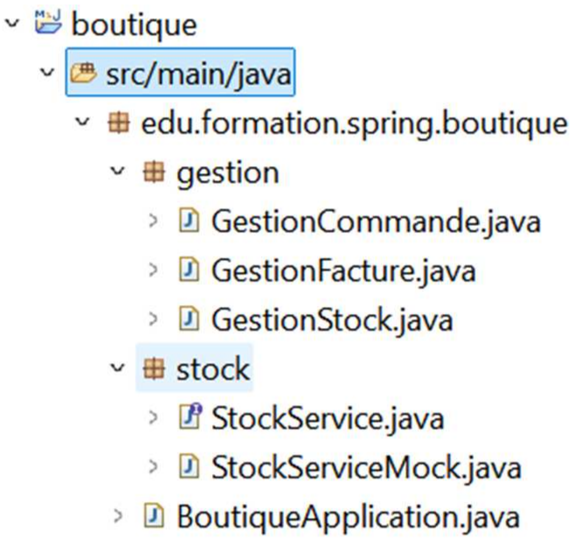
Injection de Dépendances

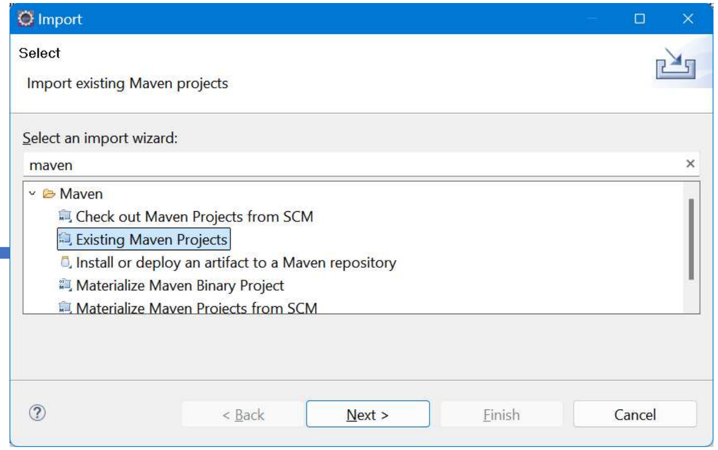
- Créer un projet avec Spring Initializr
- Générer le projet et stockez le sur votre ordinateur
- Décompresser le fichier téléchargé

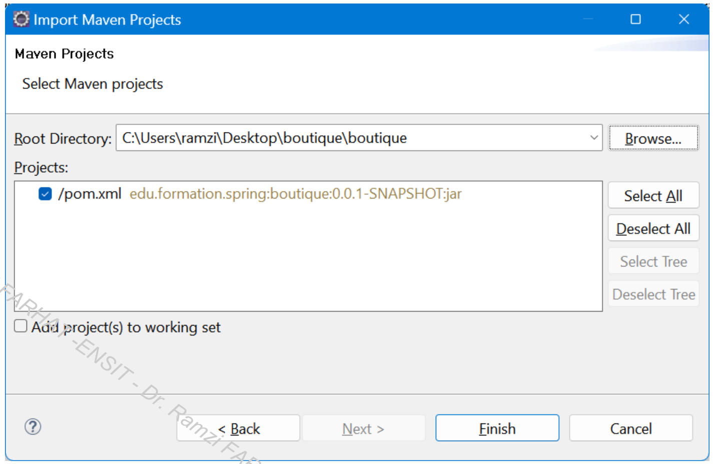


Injection de Dépendances

- Dans Eclipse
  - Allez à File > import > Existing Maven Project
  - Sélectionnez le répertoire du projet maven
  - Copier le code du projet boutique_v1







Démarche

1. Configurer le container
2. Configurer les Beans
3. Récupérer les Beans du Spring Container

Démarche 1: via les annotations

Etape 1 : Configurer le container

- Vérifiez que la classe principale est annotée avec **@SpringBootApplication** qui permet d'activer la recherche automatique des composants et leur gestion via le conteneur Spring

```java
package edu.formation.spring.boutique;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class BoutiqueApplication {

    public static void main(String[] args) {
        SpringApplication.run(BoutiqueApplication.class, args);
    }

}
```

18. Using the @SpringBootApplication Annotation

Prev | Part III. Using Spring Boot | Next

18. Using the @SpringBootApplication Annotation

Many Spring Boot developers like their apps to use auto-configuration, component scan and be able to define extra configuration on their "application class". A single `@SpringBootApplication` annotation can be used to enable those three features, that is:

- `@EnableAutoConfiguration`: enable Spring Boot's auto-configuration mechanism
- `@ComponentScan`: enable `@Component` scan on the package where the application is located (see the best practices)
- `@Configuration`: allow to register extra beans in the context or import additional configuration classes

Démarche 1: via les annotations

Etape 2 : Configurer les Beans

- On marque toutes les classes (StockServiceMock, GestionStock, GestionCommande, GestionFacture) via l'annotation **@Component**

```java
package edu.formation.spring.boutique.stock;

import org.springframework.stereotype.Component;

@Component
public class StockServiceMock implements StockService{

    public Double getPrixProduit(Integer idProduit) {

    public Integer getQuantiteProduit(Integer idProduit) {

}
```

Package org.springframework.stereotype

Annotation Interface Component

```java
@Target (TYPE )
@Retention (RUNTIME )
@Documented
@Indexed
public @interface Component
```

Indicates that an annotated class is a "component". Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.

Other class-level annotations may be considered as identifying a component as well, typically a special kind of component: e.g. the `@Repository` annotation or AspectJ's `@Aspect` annotation.

Since:

2.5

Author:

Mark Fisher

See Also:

Repository, Service, Controller, ClassPathBeanDefinitionScanner

Démarche 1: via les annotations

Etape 3 : Récupérer les Beans du Spring Container

- Utilisation de l'annotation **@Autowired** pour injecter le composant **StockService** automatiquement dans la variable **stockService**
- Supprimer l'instanciation implicite de l'objet **StockServiceMock** dans le constructeur

```java
package edu.formation.spring.boutique.gestion;

import org.springframework.beans.factory.annotation.Autowired;
@Component
public class GestionCommande {
    @Autowired
    private StockService stockService;

    public GestionCommande() {
    }
    public boolean getDisponibiliteProduit(Integer idProduit, Integer quantite) {
        return stockService.getQuantiteProduit(idProduit) >= quantite;
    }
}
```

Lancer l'application

- Modifiez la classe BoutiqueApplication :
  - Implémentez l'interface **CommandLineRunner** et ajoutez la méthode **run(String… args)**
  - Ajoutez trois arguments pour chaque classe de gestion et utilisez l'annotation **@Autowired** pour injecter les objets
  - Copiez dans la méthode le code de la méthode main() de la version 1 du projet tout en supprimant les instructions de création des objets de gestion.
- Lancez l'application

Exercice

- Créez la classe **StockServiceImp** annotée comme composant et qui contient une liste d'id d'articles et une liste de quantités d'articles et utilisez ces listes dans les méthodes d'instance :

```
Double getPrixProduit(Integer idProduit);
Integer getQuantiteProduit(Integer idProduit);
```

- Supprimez l'annotation **@Component** de la classe **StockServiceMock** et annotez à sa place la classe **StockServiceImp**
- Testez l'application

Injection de dépendance

- Injection par **champ**
  + Moins de code et simple à écrire
  - Objets mutables
  - Difficile à tester avec des tests unitaires sans charger le contexte Spring
- Injection par **setter**
  + Respecte la convention JavaBeans
  + Héritage automatique
  + Permet d'avoir des dépendances optionnelles
  - Objets mutables
- Injection par **constructeur**
  + Permet d'avoir des objets immutables
  + Oblige à avoir toutes les dépendances correctement définies
  + Respecte les principes SOLID

Injection de dépendance

- Injection par **champ** :

```
@Component
public class GestionCommande {
  @Autowired
  private StockService stockService;

  public boolean getDisponibiliteProduit(Integer idProduit, Integer quantite) {
    return stockService.getQuantiteProduit(idProduit)>= quantite;
  }
}
```

Injection de dépendance

- Injection par **setter**

```
@Component
public class GestionStock {
  private StockService stockService;
  @Autowired
  public void setStockService(StockService stockService) {
    this.stockService = stockService;
  }
  public boolean isStockSuffisant(Integer idProduit, Integer quantiteMinimum) {
    …
  }
}
```

Injection de dépendance

- Injection par **constructeur**

```
@Component
public class GestionFacture {
  final private StockService stockService;
  @Autowired
  public GestionFacture(StockService stockService) {
    this.stockService = stockService;
  }
  public Double getPrixCommande(Integer[] idProduits, Integer[] quantites) {
    …
  }
}
```

Démarche 2: via la configuration Java

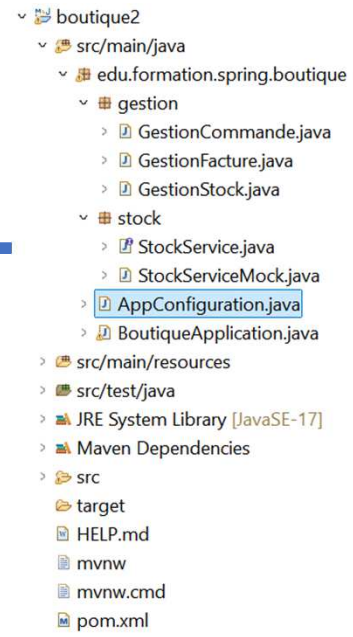
- **Principe :**
  - Plus d'utilisation de l'annotation @Component (ou de ses dérivées) pour déclarer les Beans
  - Centraliser la déclaration des Beans dans une classe de configuration annotée par @Configuration
- **Démarche :**
  - On crée une classe de configuration annotée par **@Configuration**
  - On définit une méthode par Bean annotée **@Bean** qui permet de retourner une instance de la classe souhaitée
  - On marque les endroits où on veut injecter les Beans (attribut, setter ou constructeur) par l'annotation **@Autowired**

Démarche 2: via la configuration Java

Exercice :

- Copiez le projet Spring et dans la nouvelle version du projet utilisez une classe de configuration pour gérer les Beans en complétant ce code.

```
@Configuration
public class AppConfiguration {
  @Bean
  public StockService getStockService() {
    return new StockServiceMock();
  }
  …
}
```




```
 .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::                (v3.4.3)

2025-03-16T16:16:01.042+01:00  INFO 50208 --- [boutique]
2025-03-16T16:16:01.045+01:00  INFO 50208 --- [boutique]
2025-03-16T16:16:01.582+01:00  INFO 50208 --- [boutique]
>>> GESTION DU STOCK
Il faut réapprovisionner le stock du produit : 1
Stock produit 2 est suffisant.
Il faut réapprovisionner le stock du produit : 3
>>> GESTION DE COMMANDE
Un exemplaire du produit 1 manque ...
Produit 2 disponible.
Un exemplaire du produit 3 manque ...
>>> GESTION DE FACTURE
Total commande : 21.0
```

Démarche 3: via la configuration XML

- Principe :
  - Utiliser un fichier XML pour gérer les Beans
- Démarche :
  - On crée un fichier XML dans **src/main/resources** nommé **applicationContext.xml**
  - Pour chaque bean on ajoute une balise <bean> :

```
<bean id="stockService" class="edu.formation.spring.boutique.stock.StockServiceMock"/>
```

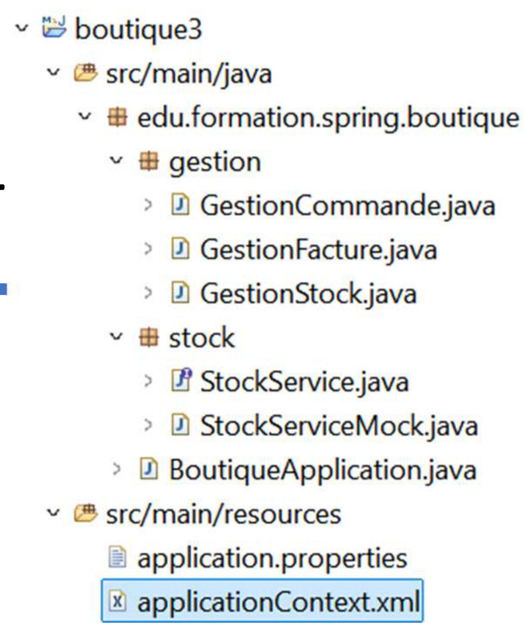
  - On peut gérer l'injection de dépendance localement, ou utiliser l'annotation @Autowired

```
<property name="stockService" ref="stockService"></property> <!--Via setter-->
<constructor-arg index="0" ref="stockService"></constructor-arg> <!--Via constructeur-->
```

  - Annoter la classe principale avec :

```
@ImportResource("classpath:applicationContext.xml")
```

Démarche 3: via la configuration XML



- Exercice :
  - Créez une nouvelle copie du projet, supprimer la classe de configuration, ajoutez le fichier XML et complétez-le et ajoutez l'annotation @ImportResource("classpath:applicationContext.xml") dans la classe principale.

```
<?xml version="1.0" encoding="UTF-8"?>
<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
https://www.springframework.org/schema/beans/spring-beans.xsd">

   <!-- Déclaration du service StockService -->
   <bean id="stockService" class="edu.formation.spring.boutique.stock.StockServiceMock"/>
   …
</beans>
```

- Injection de Dépendances
- IoD par Annotations
- IoD par Configuration Java
- IoD par Configuration XML