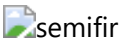


Premier projet avec Spring

Pour créer un projet avec `spring-core`, il faut utiliser les dépendances de spring core et spring contexte.

- `Spring Core` est la base de spring, il contient les annotations et les interfaces de base.
- `Spring Context` est le conteneur d'objets de spring, il permet de créer des beans et de les injecter dans d'autres beans. C'est lui qui permet de faire de l'injection de dépendances.

Création d'un projet Spring avec maven

1. Créer un projet maven sans archétype particulier
2. Ajouter les dépendances de spring-core et spring-context dans le fichier pom.xml:

```xml
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>6.0.3</version>
</dependency>
```

Création d'un projet Spring avec gradle

1. Créer un projet gradle sans archétype particulier
2. Ajouter les dépendances de spring-core et spring-context dans le fichier build.gradle:

```
dependencies {
    implementation 'org.springframework:spring-core:6.0.3'
    implementation 'org.springframework:spring-context:6.0.3'
}
```

Le contexte Spring

Création d'un contexte Spring

Pour créer un contexte Spring, il faut utiliser la classe `AnnotationConfigApplicationContext`:

```
class Main {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
        context.refresh();
    }
}
```

Qu'est ce qu'un contexte Spring?

Un contexte Spring est un conteneur d'objets. Il permet de créer des objets et de les injecter dans d'autres objets.

Ajouter un objet au contexte Spring

Pour ajouter un objet au contexte Spring, il faut utiliser la méthode `register`:

```
class Main {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
        context.refresh();
        context.register(Voiture.class);
    }
}
```

---

récupérer un objet du contexte Spring

Pour récupérer un objet du contexte Spring, il faut utiliser la méthode `getBean`:

```
class Main {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
        context.refresh();
        context.register(Voiture.class);
        Voiture voiture = context.getBean(Voiture.class);
    }
}
```

`Spring` va créer une instance de la classe `Voiture` automatiquement.

Injection de dépendances

Imaginons que la classe `Voiture` dépend de la classe `Moteur`:

```
public class Voiture {
    private Moteur moteur;

    public Voiture(Moteur moteur) {
        this.moteur = moteur;
    }

    public void rouler() {
        moteur.demarrer();
    }
}
```

Injection de dépendances

```
class Moteur {
    public void demarrer() {
        System.out.println("Le moteur démarre");
    }
}
```

Injection de dépendances

Si vous ajoutez la classe `Moteur` au contexte Spring, `Spring` va automatiquement injecter la classe `Moteur` dans la classe `Voiture`:

```
class Main {
    public static void main(String[] args) {
```

```
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
        context.refresh();
        context.register(Voiture.class);
        context.register(Moteur.class);
        Voiture voiture = context.getBean(Voiture.class);
        voiture.rouler();
    }
}
```

Terminologie: Bean

Un `bean` en anglais veut dire `chose`.

En `Spring`, un `bean` est un objet qui est géré par le contexte Spring.

Ici les classes `Voiture` et `Moteur` sont des `beans`.

Les beans

Les beans sont caractérisés par:

- `Un Type` (ici `Voiture` et `Moteur`)
- `Un identifiant` (ici `voiture` et `moteur`) qui doit être unique
- `Une portée` (ici `singleton`)

Ajouter un bean au contexte Spring

Il y a 3 façons d'ajouter un bean au contexte Spring:

- En utilisant la méthode `register`
- En utilisant l'annotation `@Bean`
- En utilisant l'annotation `@Component`

Ajouter un bean au contexte Spring avec register

La méthode `register` permet d'ajouter un bean au contexte Spring:

```
context.register(Voiture.class);
```

---

register

Il est possible d'ajouter plusieurs beans en une seule fois:

```
context.register(Voiture.class, Moteur.class);
```

---

register

Vous pouvez aussi ajouter un bean en lui définissant un identifiant:

```
context.registerBean(Voiture.class, "maVoiture");
Voiture voiture = context.getBean("maVoiture", Voiture.class);
```

Ajouter un bean au contexte Spring avec @Component

Une autre façon d'ajouter un bean au contexte Spring est d'utiliser l'annotation `@Component` sur la classe:

```
@Component
public class Voiture {
    private Moteur moteur;

    public Voiture(Moteur moteur) {
        this.moteur = moteur;
    }

    public void rouler() {
        moteur.demarrer();
    }
}
```

@Component

L'annotation `@Component` permet de définir un bean dans le contexte Spring.

Pour que `Spring` puisse détecter l'annotation `@Component`, il faut ajouter l'annotation `@ComponentScan` sur la classe `Main`:

```
@ComponentScan
class Main {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext( Main.class );
        Voiture voiture = context.getBean(Voiture.class);
        voiture.rouler();
    }
}
```

@ComponentScan

L'annotation `@ComponentScan` permet de scanner le package de la classe `Main` et de détecter les annotations `@Component`.

Ajouter un bean au contexte Spring avec @Configuration

La dernière façon d'ajouter un bean au contexte Spring est d'utiliser l'annotation `@Configuration` sur la classe:

```
@Configuration
public class Config {
    @Bean
    public Voiture voiture() {
        return new Voiture(moteur());
    }

    @Bean
    public Moteur moteur() {
        return new Moteur();
    }
}
```

@Configuration

Cette méthode demande l'utilisation de `@Configuration` sur la classe `Main`:

L'annotation `@Configuration` permet de définir que cette classe est une classe de configuration.

@Configuration

C'est l'équivalent XML de:

```xml
<beans>
    <bean id="voiture" class="com.example.Voiture">
        <constructor-arg ref="moteur"/>
    </bean>
    <bean id="moteur" class="com.example.Moteur"/>
</beans>
```

@Bean

L'annotation `@Bean` permet de définir un bean dans le contexte Spring.

```java
@Bean
public Voiture voiture() {
    return new Voiture(moteur());
}
```

Ici `Spring` va créer un bean de type `Voiture` et d'identifiant `voiture` et l'ajouter au contexte Spring.

C'est la méthode `voiture()` qui est appelée pour créer l'instance de la classe `Voiture`.

Quand utiliser @Configuration

Il est préférable d'utiliser l'annotation `@Configuration` pour définir les beans.

Cela permet de séparer la configuration du code métier.

Cas où il y a plusieurs beans du même type

Imaginons que vous avez plusieurs classes `Voiture`:

```java
@Component
public class Voiture {
    private Moteur moteur;

    public Voiture(Moteur moteur) {
        this.moteur = moteur;
    }

    public void rouler() {
        moteur.demarrer();
    }
```

```
}

@Component
public class VoitureDeCourse {
    private Moteur moteur;

    public VoitureDeCourse(Moteur moteur) {
        this.moteur = moteur;
    }

    public void rouler() {
        moteur.demarrer();
    }
}
```

Cas où il y a plusieurs beans du même type

Si je veux récupérer une instance de la classe `VoitureDeCourse`, je peux utiliser la méthode `getBean`:

```
VoitureDeCourse voitureDeCourse = context.getBean(VoitureDeCourse.class);
```

Mais si je veux récupérer une instance de la classe `Voiture`, je ne peux pas utiliser la méthode `getBean`:

```
Voiture voiture = context.getBean(Voiture.class); // Retourne une erreur
```

Cas où il y a plusieurs beans du même type

Quand `Spring` ne sait pas comment choisir le bean à retourner, il retourne une erreur.

Pour résoudre ce problème, il faut ajouter l'annotation `@Primary` sur la classe `Voiture`:

```
@Component
@Primary
public class Voiture {
    private Moteur moteur;

    public Voiture(Moteur moteur) {
        this.moteur = moteur;
    }

    public void rouler() {
        moteur.demarrer();
    }
}
```

@Primary

L'annotation `@Primary` permet de dire à `Spring` que cette classe est la classe par défaut.

Les scopes

Créer deux fois le même bean

```
@Component
public class MonBean {
    private static int compteur = 0;

    public MonBean() {
        compteur++;
        System.out.println("Création du bean n°" + compteur);
    }

    public void direBonjour() {
        System.out.println("Bonjour "+ compteur);
    }
}
```

Créer deux fois le même bean

```
MonBean monBean1 = context.getBean(MonBean.class);
MonBean monBean2 = context.getBean(MonBean.class);
monBean1.direBonjour();
monBean2.direBonjour();
System.out.println(monBean1 == monBean2); // True
```

Pourquoi?

Par défaut, `Spring` crée une seule instance pour chaque type de bean.

On dit que le scope est `singleton`.

@Scope

L'annotation `@Scope` permet de changer le scope d'un bean.

Il y a plusieurs scopes:

- `singleton` (par défaut)
- `prototype` : Créer une nouvelle instance à chaque fois

@Scope

```
@Component
@Scope("prototype")
public class MonBean {
    private static int compteur = 0;

    public MonBean() {
        compteur++;
        System.out.println("Création du bean n°" + compteur);
    }

    public void direBonjour() {
        System.out.println("Bonjour "+ compteur);
    }
}
```

@Autowired

@Autowired

L'annotation `@Autowired` permet de dire à `Spring` où injecter les dépendances. Et de lui demander de faire une injection de dépendance.

Elle peut être utilisée sur:

- Un constructeur quand il y à plusieurs constructeurs
- Un champ (même privé)
- Une méthode

@Autowired sur un constructeur

```java
@Component
public class Voiture {
    private Moteur moteur;

    // Constructeur appelé par Spring
    @Autowired
    public Voiture(Moteur moteur) {
        this.moteur = moteur;
    }

    // Constructeur appelé par l'utilisateur mais pas par Spring
    public Voiture() {
    }
}
```

@Autowired sur un champ

```
@Component
public class Voiture {
    @Autowired
    private Moteur moteur;

    public void rouler() {
        moteur.demarrer();
    }
}
```

Spring va injecter le bean `moteur` dans le champ `moteur`.

@Autowired sur une méthode

```
@Component
public class Voiture {
    private Moteur moteur;

    @Autowired
    public void setMoteur(Moteur moteur) {
        this.moteur = moteur;
    }

    public void rouler() {
        moteur.demarrer();
    }
}
```

@Autowired bonnes pratiques

- Ne pas utiliser `@Autowired` sur les champs privés
- Utiliser `@Autowired` sur les constructeurs dans certains cas (quand il y a plusieurs constructeurs et qu'il faut en choisir un)

@Qualifier

L'annotation `@Qualifier` permet de préciser le nom du bean à injecter.

```
@Component
public class Voiture {

    private Moteur moteur;

    public Voiture(
        @Qualifier("moteurEssence") Moteur moteur
```

```
    ){
        this.moteur = moteur;
    }

    public void rouler() {
        moteur.demarrer();
    }
}
```

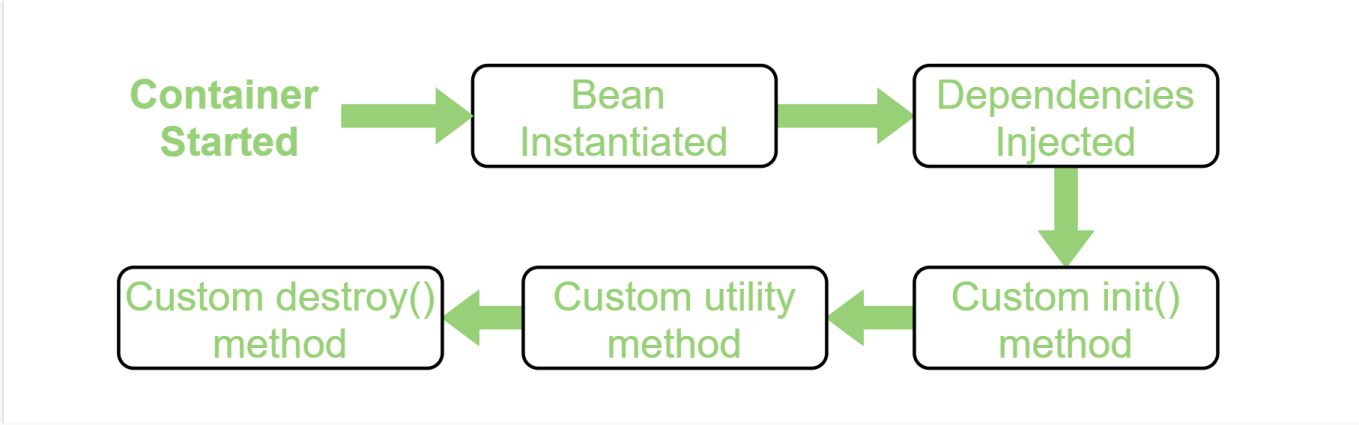
Le code précédant est équivalent à:

```
@Component
public class Voiture {

    private Moteur moteur;

    public Voiture(
         Moteur moteurEssence
    ){
        this.moteurEssence = moteurEssence;
    }

    public void rouler() {
        moteur.demarrer();
    }
}
```

Cycle de vie d'un bean

Cycle de vie d'un bean

Un bean a un cycle de vie:



Cycle de vie d'un bean

1. `Instantiation` : Création de l'objet
2. `Injection` : Injection des dépendances
3. `Initialisation` : Appel des méthodes d'initialisation
4. `Utilisation` : Utilisation de l'objet
5. `Destruction` : Destruction de l'objet

Cycle de vie d'un bean

Ce cycle de vie est géré par le `Spring container` (La BeanFactory).

Qui est responsable de la création, de l'injection, de l'initialisation et de la destruction des beans.

Cycle de vie d'un bean

Il est possible de définir des méthodes qui vont venir j'ajouter au cycle de vie d'un bean.

Il existe 2 groupes de méthodes:

- `Méthodes d'initialisation`
- `Méthodes de destruction`

Cycle de vie d'un bean

`Attention !`

Les méthodes d'initialisation sont appelées peut importe le scope du bean.

Les méthodes de destruction ne sont appelées que pour les beans de scope `singleton`.

Cycle de vie d'un bean

`Méthodes d'initialisation` : `@PostConstruct`

- Méthode exécutée après l'injection des dépendances et avant l'utilisation du bean.
- Permet de garder les constructeurs propres.

Cycle de vie d'un bean - Exemple

```
public class DemoBean {

  @PostConstruct
  public void customInit()
  {
```

```
    System.out.println("Method customInit() invoked...");
  }
}
```

Cycle de vie d'un bean

`Méthodes de destruction` : `@PreDestroy`

- Méthode exécutée avant la destruction du bean.
- Permet de fermer des connexions, des fichiers, etc.

Cycle de vie d'un bean - Exemple

```
public class DemoBean {

  @PreDestroy
  public void customDestroy()
  {
    System.out.println("Method customDestroy() invoked...");
  }
}
```

Exercice 1:

Vous voulez créer un `burger` avec un `pain`, un `steak`, une `salade` et un `fromage`.

1. Créer les classes `Pain`, `Steak`, `Salade` et `Fromage`
2. Créer une classe `Burger` qui contient les 4 ingrédients
3. Ajouter toutes les classes dans le contexte de `Spring` avec `context.register`
4. Récupérer le `burger` et afficher les ingrédients

Exercice 1.2:

1. Garder les même classes que l'exercice 1.
2. Ajouter les classes avec l'annotation `@Component`.
3. Récupérer le `burger` et afficher les ingrédients

Exercice 1.3:

1. Garder les même classes que l'exercice 2
2. Supprimer l'annotation `@Component` des classes et basculer sur l'annotation `@Configuration` et `@Bean` pour l'ajout des beans.
3. Comment faire pour avoir plusieurs `burger` qui ont des ingrédients différents?