Frameworks

TP1 : Inversion de contrôle et injection de dépendances avec Spring

---

Groupe : Anatole BOISSERIE, Julie GUILLOU, Mattis RIVET

3 février 2026, Polytech Paris-Saclay

Partie 1 : Sans Spring (problème initial)

Question 1 : Où est le couplage fort ?
Le couplage fort se situe à la ligne : *private EmailSender sender = new EmailSender();*. Ici, la classe NotificationService est directement liée à une implémentation concrète (EmailSender) au lieu de passer par une interface.

Question 2 : Peut-on facilement remplacer EmailSender ?
Non, pour remplacer EmailSender par SmsSender, on serait obligé de :

- Modifier le code source de la classe NotificationService.
- Recompiler l'application. Cela viole le principe de conception qui veut que le code soit ouvert à l'extension mais fermé à la modification.

Question 3 : Ce code respecte-t-il l'IoC ?
Non, dans ce code, c'est la classe NotificationService qui décide elle-même de créer son propre outil de travail avec l'opérateur new. Il n'y a pas d'Inversion de Contrôle car le "contrôle" de l'instanciation appartient toujours au service métier et non à un conteneur externe.

Partie 2 : IoC avec Spring et XML

Question 1 : Qui crée les objets ?
Ce n'est plus moi avec le mot clé *new* qui crée les objets, c'est le conteneur Spring

Question 2 : Où est l'IoC ?
L'inversion de contrôle se situe dans le fait que NotificationService ne choisit plus son implémentation de MessageSender. C'est la configuration externe (le XML) qui décide de lui "donner" un EmailSender.

Question 3 : Comment changer le canal de notification ?
Il suffit de modifier une seule ligne du fichier applicationContext.xml : class="com.tp.EmailSender" par class="com.tp.SmsSender".
Il n'y a pas besoin de recompiler ou de toucher au code Java.

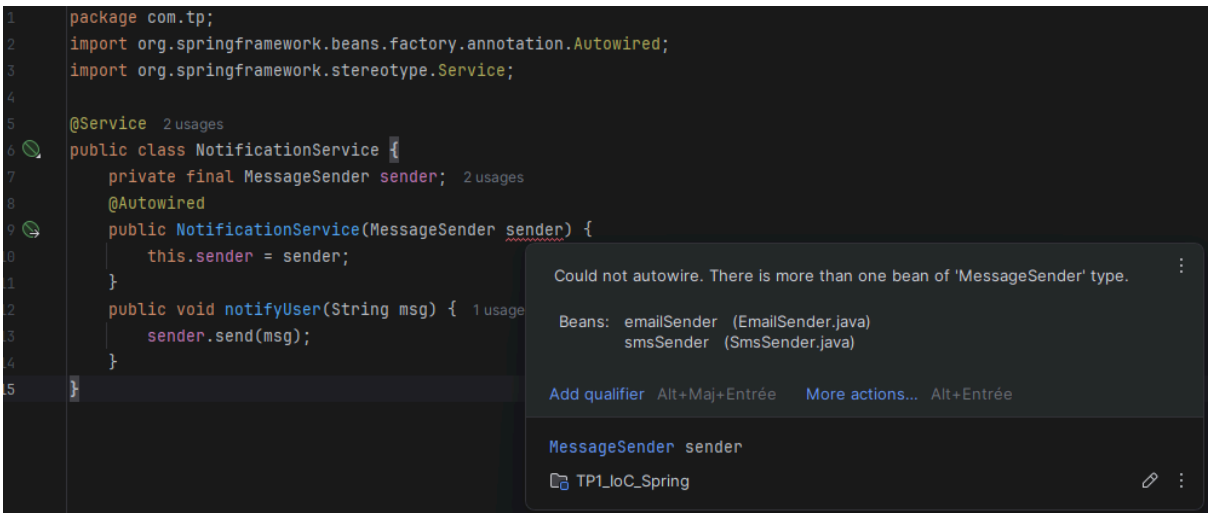
Comparaison Email/SMS : En modifiant le code comme montré, lors de l'exécution ce n'est plus un email que l'on reçoit mais un SMS.

Critère	Injection par Constructeur	Injection par Setter
Type de dépendance	Idéal pour les dépendances obligatoires.	Idéal pour les dépendances optionnelles ou modifiables.

Immuabilité	Permet de déclarer les champs en *final* (l'objet ne change plus après création).	Impossible d'utiliser *final* ; l'objet est mutable (on peut changer le sender à tout moment).
État de l'objet	Garantit que l'objet est toujours prêt à l'emploi dès son instanciation.	Risque de *NullPointerException* si on appelle une méthode avant d'avoir injecté la dépendance.
Lisibilité	Très clair : on voit immédiatement ce dont la classe a besoin pour fonctionner.	Moins explicite, surtout si la classe possède de nombreux setters.

Partie 3 : IoC avec annotations

3.3 Gestion des ambiguïtés



Après implémentation des deux components, on remarque en effet un problème d'ambiguïté sur le sender.
L'utilisation d'un @Qualifier permet de résoudre ce conflit et permettra ensuite de gérer plus facilement le choix du sender.

Partie 4 : Comparaison XML vs annotations

Critère	Configuration XML	Configuration par Annotations
Lisibilité	**Faible** : La configuration est centralisée mais peut devenir très verbeuse et complexe pour de gros projets.	**Excellente** : La configuration est placée directement sur le code concerné, ce qui permet de comprendre immédiatement le rôle d'une classe.
Couplage	**Faible** : Les classes Java restent des "POJO" purs, elles n'ont aucune dépendance vers le framework Spring.	**Élevé** : Le code source devient dépendant de Spring car il importe et utilise des annotations spécifiques (@Service, @Autowired).
Refactoring	**Plus difficile** : Si on renomme une classe, il faut penser à mettre à jour manuellement les chemins dans le fichier XML (risque d'erreurs).	**Facile** : Les IDE (comme IntelliJ) gèrent parfaitement le renommage des classes et des dépendances directement via les annotations.
Configuration dynamique	**Très flexible** : Permet de changer une implémentation sans recompiler le code (juste en modifiant le fichier .xml).	**Statique** : Toute modification de la structure des dépendances nécessite une recompilation du code Java.

Partie 6 : Questions de réflexion

Question 1 : En quoi Spring implémente-t-il l'IoC ?
Spring agit comme un chef d'orchestre : c'est lui qui gère le cycle de vie des objets (instanciation, configuration, destruction) à la place du développeur.

Question 2 : Quelle différence entre IoC et DI ?
L'IoC est le principe architectural (inverser le contrôle du flux) , tandis que la DI (Injection de Dépendances) est le mécanisme technique utilisé pour fournir les dépendances à un objet.

Question 3 : Quel pattern GoF est utilisé implicitement ?
Le pattern Stratégie, car l'implémentation de l'envoi (Email ou SMS) est interchangeable sans modifier le service.

Question 4 : Pourquoi Sprint favorise-t-il le constructor injection ?
Pour garantir que les dépendances sont obligatoires, faciliter les tests unitaires et permettre l'utilisation de champs immuables (final).