Table de Matière

Introduction Générale

Dans l'écosystème dynamique du développement d'entreprise Java, la complexité croissante des applications a rendu indispensable la transition de l'approche traditionnelle JEE vers des frameworks plus agiles et productifs. Dans ce contexte, Spring Boot s'est imposé comme une solution incontournable pour simplifier et accélérer le cycle de développement, devenant ainsi l'un des frameworks les plus influents et adoptés du marché.

Ce rapport a pour vocation de servir de guide pratique et complet à la maîtrise de Spring Boot.
L'objectif principal est de démystifier ce framework en proposant un parcours d'apprentissage progressif, partant des fondations de l'écosystème Spring et de ses concepts clés, pour aboutir à la mise en œuvre d'une application robuste et conforme aux architectures d'entreprise modernes. Ainsi, ce document vise à démontrer par la pratique comment Spring Boot facilite la création d'applications performantes en s'appuyant sur les standards JEE tout en réduisant drastiquement la configuration requise.

Il s'adresse à tout développeur ou étudiant possédant des connaissances fondamentales en programmation orientée objet avec Java, ainsi qu'une familiarité avec les principes de base de l'architecture JEE, et désireux de moderniser leurs compétences.

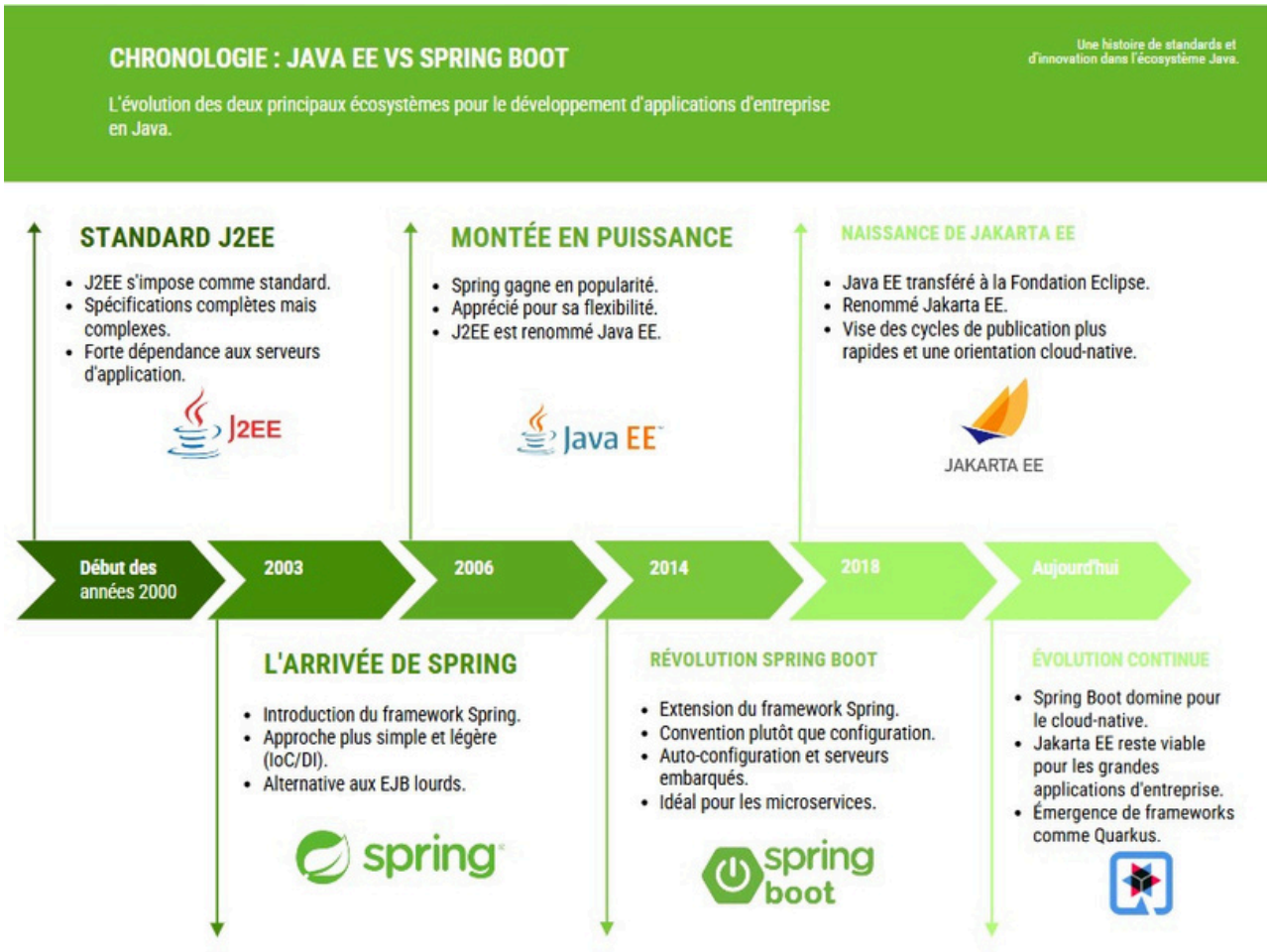
1. Évolution des applications d'entreprise : du JEE traditionnel aux frameworks modernes

Au début des années 2000, le paysage du développement Java d'entreprise était structuré autour de la plateforme J2EE (plus tard Java EE). Bien qu'elle ait établi un standard robuste, sa mise en œuvre était souvent synonyme de complexité, de configurations XML lourdes et d'une forte dépendance à des serveurs d'applications monolithiques. Cette rigidité constituait un frein notable à la productivité et à l'agilité.

En réponse directe à ces défis, le framework Spring a été introduit en 2003. Il a provoqué une véritable rupture en proposant une alternative plus légère, fondée sur les principes d'Inversion de Contrôle (IoC) et d'Injection de Dépendances (DI). En offrant une alternative plus simple aux EJB (Enterprise JavaBeans), Spring a rapidement gagné en popularité, redonnant de la flexibilité aux développeurs.

Cette quête de simplification a franchi une nouvelle étape décisive avec l'arrivée de Spring Boot.
En adoptant une philosophie radicale de "convention plutôt que configuration", Spring Boot élimine la majorité du paramétrage initial grâce à des mécanismes d'auto-configuration et l'intégration de serveurs embarqués. Cette approche a rendu possible le développement rapide d'applications autonomes, notamment pour les architectures microservices. Parallèlement, Java EE a poursuivi sa propre modernisation en devenant Jakarta EE sous l'égide de la Fondation Eclipse, avec une orientation plus marquée vers le cloud, illustrant une tendance de fond de l'écosystème vers plus de légèreté et de rapidité.

1-1 Frise chronologique. *JEE et Spring*



2. Présentation de Spring et Spring Boot

2.1. Le framework Spring : une fondation pour la productivité

Lancé en 2003, le framework Spring a émergé comme une réponse innovante à la complexité de J2EE. Ses principes fondateurs, l'Inversion de Contrôle (IoC) et l'Injection de Dépendances (DI), ont révolutionné le développement Java en permettant de créer des applications faiblement couplées, modulaires et aisément testables. Au-delà de son conteneur IoC (Spring Core), Spring s'est développé en un écosystème vaste et modulaire, offrant une multitude de projets spécialisés pour adresser les diverses facettes des applications d'entreprise :

- **Spring Core :** Le cœur du framework, fournissant le conteneur IoC/DI, la gestion des beans et l'accès aux ressources.

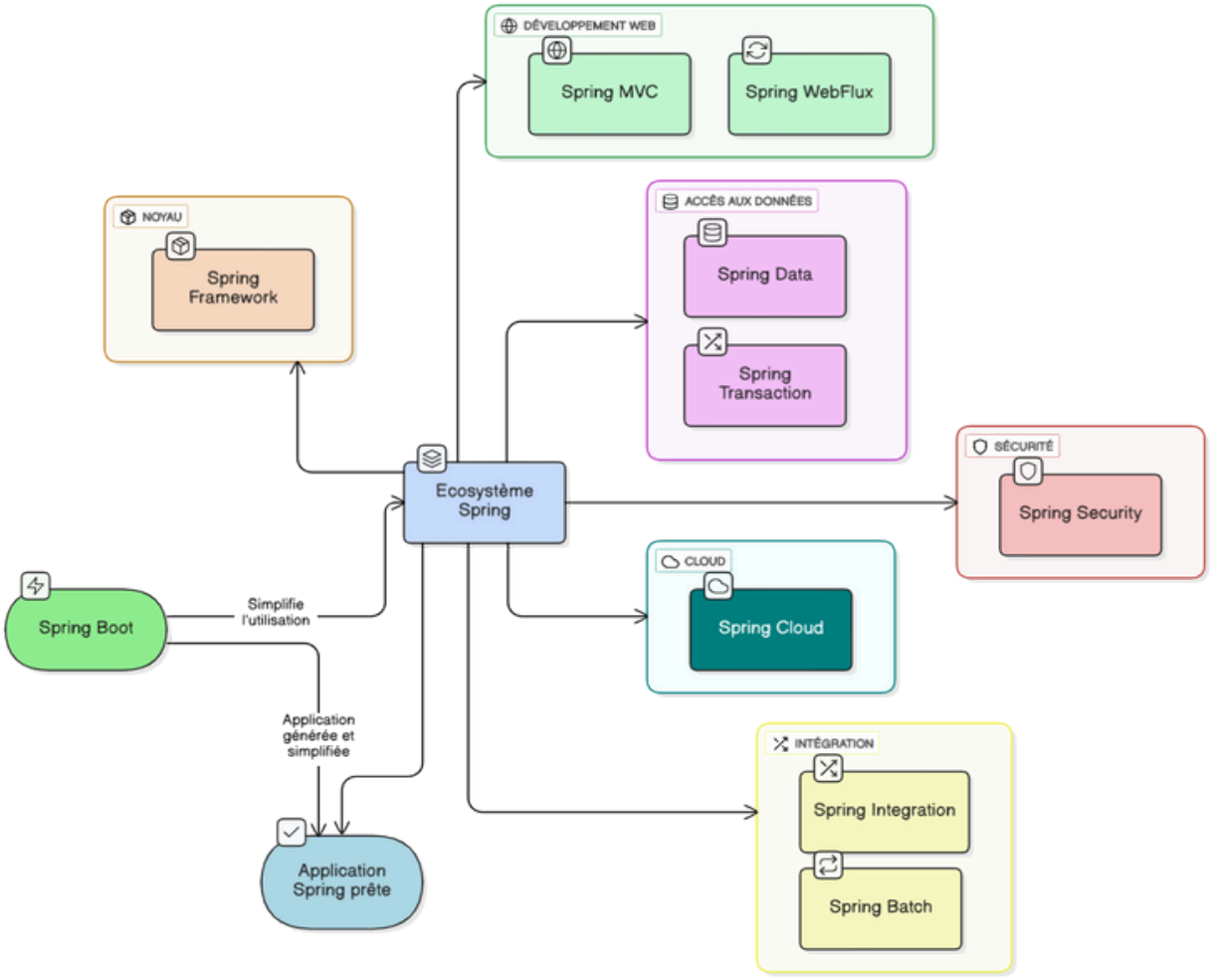
❖ **Spring MVC / Spring WebFlux :** Pour le développement d'applications web et d'APIs RESTful, avec une approche basée sur les contrôleurs pour le premier, et une approche réactive non-bloquante pour le second.

❖ **Spring Data :** Simplifie l'accès aux données avec une prise en charge uniforme de diverses technologies de persistance, qu'il s'agisse de bases de données relationnelles (JPA, JDBC) ou NoSQL (MongoDB, Redis, etc.), en générant les implémentations des dépôts de données *(repositories)*.

❖ **Spring Security :** Un framework puissant et flexible pour l'authentification, l'autorisation et la protection des applications.

D'autres projets comme Spring Batch (traitement par lots), Spring Cloud (développement de microservices cloud-native) ou Spring Integration (intégration d'applications) complètent cet écosystème riche.

1-2 Diagramme. *Vue d'ensemble de l'écosystème Spring et du rôle de Spring Boot*



2.2. Spring Boot : l'accélérateur de développement

Cependant, à mesure que l'écosystème Spring s'est enrichi, la configuration nécessaire pour orchestrer ses différents modules pouvait elle-même devenir complexe. C'est pour répondre à ce défi que Spring Boot a été créé. Il ne s'agit pas d'une réécriture de Spring, mais d'une surcouche intelligente qui adopte une approche opiniâtre (*opinionated*) basée sur la **"convention plutôt que configuration"**. Ses innovations majeures incluent :

- ❖ L'auto-configuration : Spring Boot inspecte le classpath et configure automatiquement l'application avec des paramètres par défaut pertinents.
- ❖ Les dépendances "starters" : Des descripteurs de dépendances simplifiés qui regroupent tout le nécessaire pour une fonctionnalité donnée (ex: spring-boot-starter-web).
- ❖ Les serveurs applicatifs embarqués : La possibilité de créer des applications autonomes (fichiers .jar exécutables) avec un serveur comme Tomcat ou Netty intégré, éliminant le besoin de déploiements externes complexes.

2.3. Positionnement dans l'écosystème JEE/Jakarta EE

Il est essentiel de comprendre que Spring et Spring Boot ne sont pas des adversaires de l'écosystème JEE, mais plutôt des facilitateurs qui en exploitent les standards. Loin de réinventer la roue, Spring Boot s'appuie sur des spécifications standard comme JPA (via son implémentation Hibernate) pour la persistance ou l'API Servlet pour les applications web. Il agit comme une surcouche pragmatique et productive qui utilise la puissance des standards Java tout en masquant leur complexité inhérente, permettant ainsi aux développeurs de se concentrer quasi exclusivement sur la logique métier.

3. Structure du document

Afin de proposer un parcours d'apprentissage structuré et progressif, ce rapport est organisé en une suite de chapitres logiques, allant de la configuration initiale à la réalisation d'un projet complet, puis à l'exploration de concepts avancés.

- ❖ ***Chapitre 2 :*** Préparation de l'Environnement de Développement. Ce chapitre initial se veut un guide purement pratique pour installer et configurer tous les outils indispensables (JDK, IDE, Maven/Gradle) et pour prendre en main l'outil Spring Initializr afin de créer la structure d'un premier projet.
- ❖ ***Chapitres 3 et 4 :*** Des Fondamentaux de Spring à la Simplicité de Spring Boot. Ces deux chapitres posent les fondations théoriques. Nous y aborderons d'abord les principes fondateurs du framework Spring – l'Inversion de Contrôle (IoC) et l'Injection de Dépendances (DI) – avant de découvrir comment Spring Boot vient simplifier radicalement cet écosystème grâce à ses mécanismes d'auto-configuration et ses dépendances *"starters"*.
- ❖ ***Chapitres 5, 6 et 7 :*** Construction des couches applicatives. Le cœur du rapport se concentre sur le développement des différentes couches d'une application moderne. Nous verrons comment créer une API REST avec Spring MVC, gérer la persistance des données avec Spring Data JPA, et enfin, comment sécuriser l'application avec Spring Security.

- ***Chapitre 8 :*** Tests et Qualité du Code. La qualité étant un pilier du développement logiciel, ce chapitre est entièrement consacré aux stratégies de test. Il couvre à la fois les tests unitaires avec des outils comme JUnit et Mockito, et les tests d'intégration facilités par l'écosystème Spring.
- ***Chapitre 9 :*** Étude de Cas Pratique: Application Complète. Ce chapitre majeur constitue la synthèse de toutes les connaissances acquises. Il guidera le lecteur dans la réalisation, de A à Z, d'une application fonctionnelle, depuis le cahier des charges et la conception jusqu'à l'implémentation et au déploiement.
- ***Chapitres 10 et 11 :*** Concepts Avancés et Conclusion. Pour ceux qui souhaitent approfondir le sujet, un chapitre optionnel aborde des concepts avancés comme la gestion des exceptions ou le monitoring. Enfin, la conclusion récapitulera les compétences clés acquises et offrira des ressources pour continuer à progresser.

Chapitre 1. Préparation de l'Environnement de Développement

Ce chapitre est dédié à la mise en place de l'environnement de travail technique qui servira de socle au développement de la solution logicielle en utilisant l'architecture JEE (Jakarta EE) et le framework Spring Boot. Il détaille les outils utilisés, les versions requises et la configuration initiale de l'environnement.

1.1 Configuration Système Requise et Choix Technologiques

Voici un tableau qui récapitule les exigences minimales pour la version **Spring Boot 4.0.0** et les choix technologiques adoptés.

Composant	Version Minimale Requise	Choix Adopté	Rôle
Java Development Kit (JDK)	Java 17	**Java 17 (LTS)**	Fournit l'environnement de compilation et d'exécution du code.
Spring Framework	7.0.1	7.x	Cœur du développement d'applications.
Outil de Construction (Build)	Maven 3.6.3 / Gradle 8.14	Au choix (Maven ou Gradle)	Gestion des dépendances et automatisation des tâches de construction.
IDE	N/A	**Eclipse IDE for Enterprise Java**	Environnement de codage, de débogage et de déploiement.

Figure 1-1. Tableau des spécifications techniques

1.2 Installation des Outils Nécessaires

L'installation des outils fondamentaux :

JDK (Java Development Kit)

Le **JDK** est essentiel car il contient le compilateur Java, l'environnement d'exécution (JRE) et les outils nécessaires pour développer des applications Java.

- **Version Recommandée : Java 17 (LTS)** est la version minimale et privilégiée pour sa stabilité et son support optimal par Spring Boot 4.0.0.
- **Téléchargement et Installation :** L'installation est réalisée à partir d'un fournisseur comme **Adoptium(Eclipse Temurin)**, via le téléchargement de l'installeur correspondant au système d'exploitation de l'utilisateur.

IDE (Environnement de Développement Intégré)

L'**IDE** est l'outil central qui facilite l'écriture, la compilation, le débogage et le déploiement du code.

- **ChoixAdopté:EclipseIDEforEnterpriseJavaandWebDevelopers** .
- **Justification:** Eclipse est sélectionné pour son intégration approfondie dans l'écosystème JEE et pour la disponibilité de ses extensions dédiées à Spring (Spring Tools).
- **Installation:** La version adéquate est téléchargée depuis le site officiel d'Eclipse.

Maven/Gradle : Outils de Construction (Build)

Ces outils de gestion de dépendances et de construction sont indispensables pour gérer les librairies requises par Spring Boot et automatiser les tâches de *build* (compilation, tests, packaging).

- **VersionsSupportées :**
  - **Maven :** 3.6.3 ou version ultérieure.
  - **Gradle :** 8.x (8.14+) et 9.x.
- **Installation:** L'intégration de ces outils est assurée par l'IDE Eclipse et l'outil de génération de projet (Spring Initializr), ce qui élimine le besoin d'une installation manuelle complexe pour la plupart des environnements.

1.3 Configuration et Validation de l'Environnement

La configuration de l'environnement de développement repose sur l'établissement des variables système nécessaires pour garantir que l'outil de construction (Maven/Gradle) et l'IDE (Eclipse) utilisent la version requise du JDK (Java 17).

Configuration des Variables Clés

- **JAVA_HOME :** La variable a été définie pour pointer précisément vers le répertoire d'installation de la version **Java 17 (LTS)**. Cette étape assure la compatibilité avec Spring Boot 4.0.0.
- **PATH :** Le chemin vers le sous-répertoire bin du JDK a été ajouté au PATH pour permettre l'accès direct aux exécutables Java depuis la console.

Vérification de l'Intégrité de l'Installation

L'intégrité de l'installation est validée par des commandes de vérification de version dans le terminal, qui confirment que la version de Java et l'outil de *build* sont correctement référencés.

- Pour Java :

```
C:\Users\PC-HP> java -version
```

- Pour l'outil de build:

```
C:\Users\PC-HP> mvn -v
```
ou
```
C:\Users\PC-HP> gradle -v
```

1.4 Création et Importation du Projet Fondateur

La création du projet est réalisée à l'aide de l'outil standard de l'écosystème Spring, suivi de son importation dans l'IDE.

Utilisation de Spring Initializr

Spring Initializr est le générateur de projets officiel, garantissant une structure de base propre et conforme aux normes.

- **Processus de Génération :**
  1. Accès à **start.spring.io**.

Figure 1-2.Spring initializr

2. Sélection du projet Maven, java et version de spring boot.

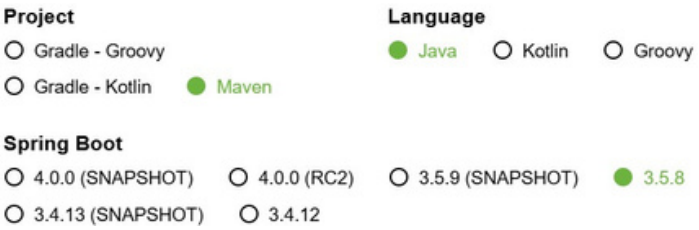


Figure 1-3. Choix des versions, projet.

3. Définition des métadonnées (Group, Artifact, Version de Spring Boot/Java 17).

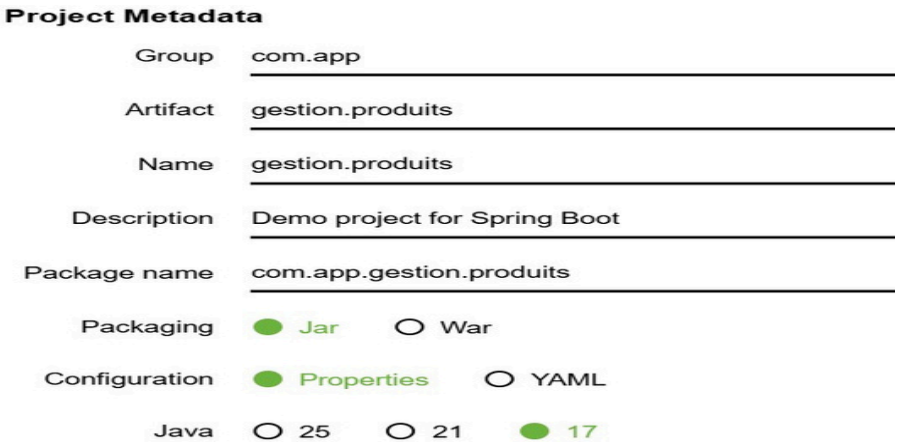


Figure 1-4. Choix des métadonnées.

4. Sélection des dépendances initiales (comme Spring Web pour la création d'API, Spring Data JPA pour la persistance, etc).

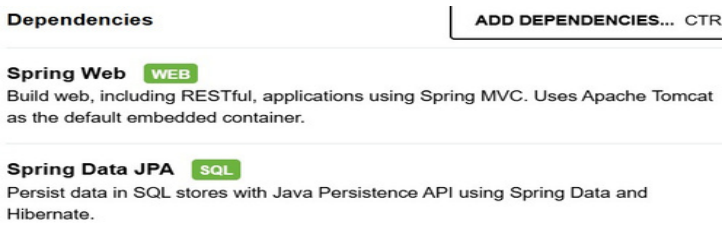


Figure 1-5. Ajout des dépendances.

5. Téléchargement du projet compressé au format ZIP.

Structure de Projet et Importation dans Eclipse

Figure 1-6.Eclipse

Leprojetgénéréestimportédansl'IDEpourlancer le développement.

- **Structure Clé :** La structure inclut le fichier de configuration de *build* (pom.xml ou build.gradle), les répertoires src/main/java pour le code source et src/main/resources pour les fichiers de configuration(commeapplication.properties).
- **Importation dans Eclipse :** Leprojet est importé via **"File -> Import... -> Existing Maven/Gradle Projects"**.
  - ➔ CetteméthodepermetàEclipse d'indexer automatiquement le projet et de télécharger touteslesdépendancesrequises, finalisant la préparation de l'environnement de développement.
  - ➔ Exemple après l'importation :

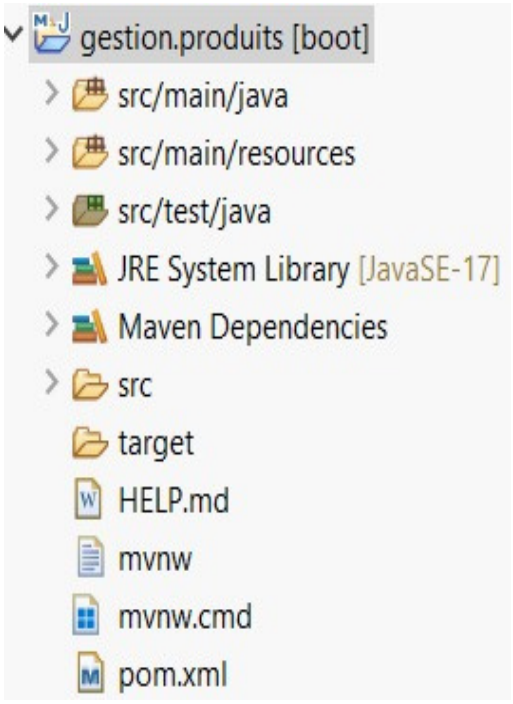


Figure 1-7. *Structure Initiale du Projet sur Eclipse*

Chapitre 2. Fondamentaux de Spring Framework

2.1 Philosophie et principes de conception de Spring

Les problématiques résolues par Spring

Spring est né à la fin des années 2000 pour répondre aux limites et douleurs du développement Java «classique ». Voici les problèmes principaux et la façon dont Spring les adresse.

1. Couplage fort entre classes

Problème : Dans les applications traditionnelles, les classes créent et gèrent elles-mêmes leurs dépendances (new ServiceImpl() partout). Résultat : difficile d'isoler une classe pour les tests, changement d'implémentation délicat.

Solution Spring :

- **Inversion deContrôle(IoC)** et **Injection de Dépendances (DI)** : Spring crée et assemble les objets (beans) pour toi, et injecte les dépendances dans les composants.
- **Effet :** on obtient des classes plus petites, axées sur la logique métier, et facilement testables via des mocks.

Exemple simple (avant / après)

Avant (couplage) :

```
public class OrderController {
    private PaymentService paymentService = new StripePaymentService();
}
```

Après (DI via constructeur) :

```
@Component
public class OrderController {
    private final PaymentService paymentService;
    public OrderController(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

 }
```

2. Boilerplate et gestion

Problème : Beaucoup de code répétitif pour créer des connexions JDBC, configurer transactions, gérer ressources, etc.

Solution Spring :

- **Abstractions pour l'accès aux données (JdbcTemplate, Spring Data)** qui réduisent le code répétitif.
- **Gestion déclarative des transactions** (annotations ou configuration) pour ne pas écrire de code transactionnel partout.

3. Cross-cutting concerns (préoccupations transverses)

Problème : Sécurité, logging, transactions, cache, monitoring s'imbriquent dans le code métier et polluent la logique.

Solution Spring AOP (programmation orientée aspect) :

- Permet d'extraire ces préoccupations transverses en **aspects** (advice, pointcuts), appliqués sans modifier la logique métier.

4. Architecture monolithique et rigidité

Problème : Applications monolithiques difficiles à modulariser et à faire évoluer.

Solution Spring :

- **Architecture modulaire** (choisissez les modules nécessaires).
- **Spring Boot** :conventions et auto-configuration pour démarrer vite, tout en restant modulaire si besoin.

5. Difficulté d'intégration (technologies variées)

Problème : Intégrer JMS, AMQP, Web Services, batch, jobs planifiés, etc., impliquait souvent du glue code complexe.

Solution Spring :

Modules d'intégration et adaptateurs (Spring Integration, Spring Cloud, Spring Batch, Spring AMQP) qui normalisent l'intégration et offrent des patterns prêts à l'emploi.

6. Configurations multiples et gestion des environnements

Problème : Comment gérer configurations pour dev/test/prod sans dupliquer ?

Solution Spring Boot + Profiles + Properties/ConfigServer :

Profiles (dev/prod/test) et sources de configuration (fichiers properties/YAML, variables d'environnement, serveur de config) centralisent la configuration.

7. Evolutivité et modernité

Problème : Besoin de réactivité,non-bloquant, microservices, déploiement cloud.

Solution Spring :

Spring WebFlux pour Web non bloquant (reactive).
Spring Cloud pour patterns cloud (service discovery, configuration, circuit breaker, gateway).

Architecture modulaire de Spring

Spring est conçu en **modules distincts** que tu peux combiner selon les besoins. Voici les modules principaux, leur rôle et quand les utiliser.

Spring Boot n'est pas un module isolé mais un *éco-système* qui assemble et configure automatiquement les modules Spring pour démarrer rapidement.

1. Spring Core

Spring Core constitue le cœur du framework Spring et fournit les fondations essentielles pour toute application Spring. Il implémente le principe d'Inversion of Control (IoC) et gère les beans via des conteneurs comme BeanFactory ou ApplicationContext. Spring Core s'occupe également du cycle de vie des beans, de leur création à leur destruction, et permet l'injection de dépendances, facilitant ainsi l'architecture modulaire et la maintenabilité des applications.

2. Spring AOP

SpringAOP (Aspect-Oriented Programming) permet d'introduire la programmation orientée aspects afin de traiter des préoccupations transversales comme la gestion des transactions, le logging, la sécurité ou la collecte de métriques. L'utilisation de Spring AOP favorise une séparation nette des responsabilités, en permettant de gérer ces aspects de manière centralisée sans polluer la logique métier.

3. Spring Data

Spring Data fournit un ensemble d'abstractions pour simplifier l'accès aux données et réduire le code des couches DAO ou repository. Grâce à des modules comme Spring Data JPA, MongoDB ou Redis, il est possible de bénéficier d'implémentations automatiques de repository et d'exploiter des méthodes de recherche prédéfinies. Cela accélère le développement et garantit des pratiques standardisées pour la gestion de la persistance.

4. Spring Transaction

Le module Spring Transaction assure la gestion déclarative des transactions dans les applications, principalement via l'annotation @Transactional. Cela permet de garantir la cohérence des opérations sur la base de données et d'assurer le rollback automatique en cas d'exception, simplifiant ainsi la gestion de la fiabilité et de l'intégrité des données.

5. Spring MVC / Web

Spring MVC est un framework de type Model-View-Controller destiné à la création d'applications web classiques. Il fournit des mécanismes pour définir des contrôleurs, gérer les vues et résoudre les requêtes HTTP. Ce module est particulièrement adapté pour construire des applications RESTful ou des pages serveur côté backend utilisant des technologies comme Thymeleaf ou JSP.

6. Spring WebFlux

Spring WebFlux introduit une approche réactive et non bloquante pour la création d'applications web. Basé sur Project Reactor, il est conçu pour gérer des charges élevées et des opérations d'entrée/sortie intensives. Contrairement à Spring MVC, WebFlux n'est pas un simple "upgrade", mais une architecture différente destinée à répondre à des besoins spécifiques de réactivité et de scalabilité.

7. Spring Security

SpringSecurity fournit un cadre complet pour sécuriser les applications. Il gère l'authentification, l'autorisation et la protection des endpoints, et prend en charge des protocoles modernes comme OAuth2 et OpenID Connect. Ce module est essentiel pour sécuriser les APIs et les applications web, en garantissant la protection des données et la conformité aux bonnes pratiques de sécurité.

8. Spring Boot

SpringBoot simplifie considérablement la configuration et le packaging des applications Spring. Il permet d'exécuter les applications de manière autonome grâce à un serveur embarqué tel que Tomcat ou Jetty. Avec ses fonctionnalités d'auto-configuration et ses starters (par exemple
spring-boot-starter-web ou spring-boot-starter-data-jpa), Spring Boot accélère le développement en fournissant des valeurs par défaut et une structure standardisée, tout en restant flexible.

9. Spring Cloud

Spring Cloud fournit un ensemble d'outils pour les architectures distribuées et les environnements cloud. Il facilite la découverte de services, la centralisation des configurations via Config Server, la gestion des gateways et des circuits breaker, et bien d'autres fonctionnalités. Ce module est particulièrement pertinent dans le contexte des microservices et du déploiement sur des plateformes cloud comme Kubernetes ou AWS.

10. Spring Integration / Spring Batch / Spring AMQP / Spring Kafka

Ces modules répondent àdesbesoins spécifiques. Spring Integration permet d'implémenter des patterns d'intégration et de communication entre systèmes via des canaux de messages et des adaptateurs. Spring Batch est conçu pour les traitements par lots, avec gestion des jobs, steps et restartabilité. Spring AMQP et Spring Kafka facilitent la messagerie asynchrone et événementielle, idéale pour les architectures basées sur les événements, les ETL ou les traitements batch.

11. Spring Web Services

Spring WebServices permet de créer et de consommer des services web SOAP. Il fournit des outils pour exposer des endpoints SOAP, manipuler des messages XML et gérer les contrats WSDL, offrant ainsi une approche robuste pour l'intégration via web services standards.

12. Spring Test

Spring Testfournit des utilitaires pour les tests unitaires et d'intégration dans le contexte Spring. Avec des annotations comme @SpringBootTest et les slices de test, il facilite la création de tests rapides, isolés et réalistes, permettant de vérifier le comportement des composants Spring tout en conservant un contexte applicatif cohérent.

Comment les modules s'articulent

Les modules Spring s'organisent de manière cohérente pour construire des applications modulaires et maintenables. À la base, on retrouve **Spring Core**, le **Context** et la gestion des **Beans**, qui sont toujours présents et constituent le socle de toute application Spring.

Au-dessus de cette base se situe la **couche métier**, composée des services annotés @Service ou @Component, qui contiennent la logique métier et orchestrent les opérations de l'application. L'**accès aux données** est pris en charge par Spring Data ou JdbcTemplate, avec les repositories qui simplifient la gestion de la persistance et des opérations sur les bases de données.

Pour la partie **Web ou API**, Spring MVC ou Spring WebFlux expose les endpoints nécessaires, permettant la communication avec les clients ou d'autres services. Les aspects transverses tels que la **sécurité** et la **programmation orientée aspects (AOP)** sont gérés par Spring Security ou des aspects personnalisés, apportant des fonctionnalités globales comme l'authentification, l'autorisation, le logging et la gestion des transactions.

Enfin, pour l'infrastructure et le déploiement, **Spring Boot** assure le démarrage rapide de l'application et l'auto-configuration, tandis que Spring Cloud fournit des fonctionnalités avancées pour les architectures microservices, comme la découverte des services ou la configuration centralisée. La **couche test** complète cette architecture grâce à Spring Test, qui permet de valider les composants et de garantir la qualité et la fiabilité du code.

Bénéfices concrets & Cas d'usage

L'adoption de Spring apporte de nombreux bénéfices concrets pour le développement d'applications. Tout d'abord, le **découplage** des composants facilite la maintenance et l'évolutivité du code, car chaque classe peut évoluer indépendamment des autres. La **testabilité** est également améliorée : grâce à l'injection de dépendances, il est possible de réaliser des tests unitaires et d'intégration plus simples et isolés. Spring favorise également la **productivité**, en réduisant le code répétitif grâce aux starters et à l'auto-configuration, permettant aux développeurs de se concentrer sur la logique métier.

La **réutilisabilité** des composants est renforcée, chaque module pouvant être facilement remplacé ou utilisé dans différents projets. Enfin, Spring contribue à la **robustesse** des applications, notamment par la gestion intégrée des transactions, des retries ou d'autres mécanismes critiques pour la fiabilité des systèmes.

Ces bénéfices se traduisent concrètement dans de nombreux cas d'usage. Pour la création d'**API REST d'entreprise**, on combine généralement Spring Boot avec Spring MVC, Spring Data JPA et Spring Security. Pour les **systèmes événementiels**, on peut s'appuyer sur Spring Boot associé à Kafka ou AMQP et Spring Integration. Les **traitements batch** utilisent quant à eux Spring Batch, tandis que les architectures de **microservices distribués** tirent parti de Spring Cloud pour la découverte des services, la gestion centralisée des configurations et l'implémentation de passerelles et de circuits de résilience.

Quelques bonnes pratiques architecturales liées à la philosophie Spring

Il est recommandé de privilégier l'injection par constructeur dans les classes Spring. Cette approche favorise l'immutabilité des objets et améliore considérablement la testabilité, car toutes les dépendances nécessaires sont explicitement déclarées dès la création de l'objet.

Une bonne architecture Spring nécessite également de séparer clairement les différentes couches de l'application. Les controllers doivent uniquement orchestrer les requêtes et déléguer la logique métier aux services, tandis que les repositories se chargent de l'accès aux données. Cette séparation favorise la lisibilité, la maintenabilité et la réutilisabilité du code.

Pour renforcer la flexibilité et permettre un découplage plus fort, il est conseillé d'utiliser des interfaces pour les services. Cela facilite le remplacement d'implémentations et les tests unitaires, tout en respectant le principe de programmation orientée interface.

Dans les controllers, il faut limiter la logique métier. Les controllers doivent agir comme des coordinateurs, orchestrant les appels aux services plutôt que de contenir des calculs ou des règles complexes. Cela rend le code plus clair et simplifie sa maintenance.

L'utilisation de Spring Boot est fortement recommandée pour accélérer le démarrage des projets et bénéficier de l'auto-configuration. Cependant, il reste important de comprendre quelles configurations sont appliquées automatiquement pour éviter des comportements inattendus ou des conflits. La gestion du scope des beans est également un point important : par défaut, les beans sont singleton, ce qui signifie qu'une seule instance est partagée dans l'application. Dans certains cas spécifiques, un scope prototype peut être utilisé pour créer une nouvelle instance à chaque demande.

Enfin, il est conseillé d'utiliser les profiles Spring (dev, test, prod)pour gérer les configurations spécifiques à chaque environnement. Cette approche centralise les paramètres et évite la duplication ou les erreurs liées à la modification manuelle des configurations lors du passage d'un environnement à un autre.

2.2. Inversion de Contrôle (IoC)

L'Inversion de Contrôle constitue le principe fondamental sur lequel repose Spring Framework. Ce mécanisme permet au framework de prendre en charge la création, l'assemblage et la gestion des composants applicatifs, ce qui favorise une architecture modulaire, maintenable et facilement testable.

Concept théorique : Qu'est-ce que l'IoC ?

1. Définition

L'Inversion de Contrôle (IoC) est un principe d'architecture logicielle selon lequel la responsabilité de créer et de gérer les objets n'est plus assurée par le code métier mais par un conteneur externe. Dans le contexte de Spring, ce rôle est rempli par le conteneur IoC.

Au lieu de créer explicitement leurs dépendances au moyen de l'instruction new, les composants déclarent uniquement ce dont ils ont besoin. C'est le conteneur Spring qui se charge ensuite de fournir les instances nécessaires.

2. Origine du terme "Inversion"

Habituellement, les objets contrôlent leur propre flux d'exécution et instancient directement leurs dépendances. L'IoC inverse ce mécanisme : le framework prend le contrôle du cycle d'exécution et de la gestion des objets. Les composants deviennent ainsi passifs dans leur création et actifs seulement dans l'exécution de leur logique métier.

3. Avantages de l'IoC

L'utilisation de Spring et de l'injection de dépendances permet de réduire fortement le couplage entre les classes, chaque composant pouvant se concentrer sur sa propre responsabilité sans dépendre directement des implémentations des autres. Cette approche favorise une meilleure maintenabilité et modularité du code, rendant les applications plus faciles à faire évoluer et à organiser.

Elle simplifie également les tests unitaires, car il devient possible d'injecter des dépendances simulées (mocks) pour isoler chaque composant et tester son comportement indépendamment du reste de l'application. En outre, Spring permet une centralisation et une uniformisation de la configuration, évitant la duplication de paramètres et facilitant la gestion des différents environnements. Enfin, cette architecture favorise la réutilisabilité des composants, qui peuvent être utilisés dans différents modules ou projets sans modification, grâce à leur faible dépendance aux autres classes et à la clarté de leur interface.

Le Conteneur Spring : ApplicationContext

Le conteneur Spring IoC est l'élément chargé de créer les objets, de résoudre leurs dépendances et de gérer leur durée de vie.

1. Rôle du conteneur

Le conteneur Spring constitue le cœur de l'architecture Spring et assure la gestion complète des composants de l'application. Il est responsable de l'instanciation des objets déclarés comme beans et de

l'injection automatique des dépendances nécessaires pour leur fonctionnement. Le conteneur gère également le cycle de vie des beans, depuis leur création jusqu'à leur destruction, tout en appliquant les mécanismes transverses tels que l'AOP pour la gestion des aspects comme le logging, la sécurité ou les transactions.

De plus, le conteneur Spring est capable de lire et d'interpréter les configurations, qu'elles soient définies en XML, via des annotations ou avec des classes de configuration Java. Il offre également une série de services supplémentaires facilitant le développement d'applications robustes, tels que l'internationalisation, la gestion des événements applicatifs ou le chargement centralisé des ressources. Grâce à ces fonctionnalités, le conteneur Spring permet de simplifier la construction, la configuration et la maintenance des applications tout en garantissant une architecture cohérente et modulable.

2. ApplicationContext

ApplicationContext est l'interface principale du conteneur IoC.

Ses implémentations les plus courantes sont :

- ClassPathXmlApplicationContext : configuration basée sur XML ;
- AnnotationConfigApplicationContext : configuration basée sur les annotations ou les classes Java annotées ;
- implémentations spécialisées utilisées par Spring Boot pour les applications Web embarquées.

```
ApplicationContext context =
        new AnnotationConfigApplicationContext(AppConfig.class);

MyService service = context.getBean(MyService.class);
service.process();
```

Cycle de vie des beans

Un bean est un objet géré par Spring. Son cycle de vie complet comprend les étapes suivantes :

1. Chargement de la configuration

Spring analyse les fichiers XML, les annotations ou les classes Java annotées pour identifier les beans à gérer.

2. Instanciation

Le conteneur crée l'objet en appelant son constructeur.

3. Injection des dépendances

Les dépendances nécessaires au bean sont injectées selon le mode défini (constructeur, méthodes setters ou champs).

4. Post-traitements

Les BeanPostProcessor permettent d'intercepter ou de modifier les beans après leur création mais avant leur initialisation. C'est à ce stade que les proxys AOP sont générés.

5. Initialisation

Le conteneur appelle les mécanismes d'initialisation :

- méthode annotée @PostConstruct ;
- méthode d'initialisation spécifiée dans la configuration ;
- implémentation éventuelle de l'interface InitializingBean.

6. Phase d'utilisation

Le bean est pleinement opérationnel et peut être utilisé dans l'application.

7. Destruction

Lors de l'arrêt du contexte, Spring exécute :

- la méthode annotée @PreDestroy ;
- la méthode de destruction déclarée dans la configuration ;
- l'interface DisposableBean si elle est implémentée.

Configuration Spring : XML, Annotations, Java Config

Spring propose trois grands modèles de configuration. Chacun présente un niveau de flexibilité distinct.

1. Configuration XML

C'est la méthode historique où les définitions de beans sont déclarées dans des fichiers XML.

Exemple :

```
<bean id="myService" class="com.example.MyService"/>
```

Caractéristiques :

- Séparation stricte entre code et configuration ;
- Procédure généralement plus verbeuse ;
- Utilisée principalement dans les projets existants ou pour des besoins très spécifiques.

2. Configuration basée sur les annotations

Introduite avec Spring 2.5, elle permet d'annoter directement les classes pour que Spring les détecte automatiquement. Par exemple :

```
@Component
public class MyService {}
```

```
@Autowired
private MyService myService;
```

3. Java Config (Configuration par classes Java)

Méthode recommandée, introduite avec Spring 3, qui permet de configurer Spring via des classes Java annotées.

Exemple :

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyService();
    }
}
```

Caractéristiques :

- Contrôle total par la programmation ;
- Vérification des erreurs à la compilation ;
- Intégration naturelle avec Spring Boot ;
- Gestion centralisée et lisible de la configuration.

Comparaison synthétique

Critère	XML	Annotations	Java Config
Niveaudemodernité	Faible	Élevé	Très élevé
Lisibilité	Moyenne	Bonne	Excellente
Flexibilité	Élevée	Moyenne	Très élevée
Usage dansSpring Boot	Rare	Fréquent	Privilégié
Utilisationrecommandée	Projets existants	Projets courants	Projets modernes

2.3 Injection de Dépendances (DI)

Principe de la DI et ses avantages

L'injection de dépendances (Dependency Injection, DI) est un mécanisme fondamental de Spring permettant de déléguer la création et la gestion des objets au conteneur IoC. Au lieu qu'une classe instancie elle-même ses dépendances, celles-ci lui sont fournies de l'extérieur. Cette approche améliore la modularité et réduit fortement le couplage entre les composants.

Les principaux avantages de la DI sont :

- **Réduction du couplage** : les classes ne dépendent plus directement de l'implémentation mais d'interfaces ou d'abstractions.
- **Facilité de test** : les dépendances peuvent être remplacées par des mocks ou des doublures.
- **Réutilisabilité accrue** : les composants deviennent plus génériques et indépendants.
- **Configuration centralisée** : toutes les dépendances sont gérées par Spring, ce qui évite les instanciations répétitives.
- **Évolution facilitée** : changer une implémentation ne nécessite aucune modification dans les classes consommatrices.

Types d'injection : constructeur, setter et field

1. Injection par constructeur

L'injection par constructeur consiste à passer les dépendances d'une classe directement via son constructeur. Cette approche présente plusieurs avantages importants : elle **assure que toutes les dépendances nécessaires sont fournies**, garantissant ainsi que l'objet est correctement initialisé dès sa création. Elle favorise également **l'immuabilité des objets**, car les dépendances peuvent être déclarées final et ne peuvent pas être modifiées après l'instanciation. Pour ces raisons, l'injection par

constructeur est particulièrement **recommandée pour les composants critiques**, où la fiabilité et la cohérence des dépendances sont essentielles.

Exemple (Spring Boot) :

```
@Service
public class OrderService {
    private final PaymentService paymentService;

    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
```

2. Injection par setter

L'injection par setter consiste à fournir les dépendances d'une classe via des méthodes publiques de type setter. Cette approche est particulièrement adaptée pour **les dépendances optionnelles**, qui ne sont pas indispensables lors de la création de l'objet. Elle est également utile lorsque la dépendance peut **changer après l'instanciation**, offrant ainsi plus de flexibilité. Cependant, cette méthode est **moins stricte que l'injection par constructeur**, car il n'est pas garanti que toutes les dépendances soient présentes au moment de la création de l'objet.

```
@Service
public class NotificationService {
    private EmailService emailService;

    @Autowired
    public void setEmailService(EmailService emailService) {
        this.emailService = emailService;
    }
}
```

3. Injection par champ (field injection)

L'injection par champ consiste à laisser Spring injecter directement les dépendances dans les attributs annotés de la classe, généralement avec @Autowired. Cette approche présente l'avantage d'une **syntaxe simple et rapide**, car il n'est pas nécessaire d'écrire des constructeurs ou des setters. Cependant, elle **est déconseillée dans les projets professionnels**, car elle complique les tests unitaires et empêche l'immuabilité des objets. Pour cette raison, l'injection par champ ne devrait être utilisée que **de manière exceptionnelle**, lorsque la simplicité prime sur la testabilité ou la flexibilité.

```java
@Service
public class ReportService {
    @Autowired
    private DataService dataService;
}
```

Bonnes pratiques d'injection

Spring recommande plusieurs règles pour garantir un code maintenable :

1. **Privilégier l'injection par constructeur** pour les dépendances principales.
2. **Limiter l'injection par champ**, principalement utilisée dans les projets simples ou dans les tests.
3. **Utiliser des interfaces** lorsque c'est pertinent pour réduire le couplage.
4. **Éviter la logique métier dans les constructeurs**, qui doivent uniquement recevoir les dépendances.
5. **Ne pas multiplier les dépendances** : si une classe nécessite trop de services externes, cela peut indiquer un problème de conception.

3.4 Autowiring

Mécanismes d'Autowiring (@Autowired, @Qualifier)

L'autowiring est une fonctionnalité permettant à Spring de résoudre automatiquement les dépendances entre les composants. Le développeur n'a pas besoin d'instancier ou de configurer explicitement les objets : Spring le fait en fonction du type des dépendances.

1. @Autowired

Indique à Spring d'injecter automatiquement la dépendance requise.

- Peut être utilisé sur :
  - un constructeur,
  - un setter,
  - un champ,
  - une méthode.

Spring recherche un bean compatible par son type.

2. @Qualifier
Permet de résoudre les ambiguïtés lorsqu'il existe plusieurs beans du même type.

```
@Service
publicclass BillingService {

    @Autowired
    @Qualifier("paypalService")
    private PaymentService paymentService;
}
```

Ici, Spring injectera spécifiquement le bean nommé *paypalService*.

Résolution des ambiguïtés

Lorsque plusieurs implémentations du même type existent, Spring ne peut pas déterminer lequel injecter. Dans ce cas :

1. **@Qualifier** identifie explicitement le bean désiré.
2. **@Primary** définit une implémentation par défaut.
3. Le nom du bean peut être utilisé implicitement si aucune annotation n'est fournie.

```
Exemple avec @Primary :
 @Service
@Primary
public class StripePaymentService implements PaymentService { }
```

@Component, @Service, @Repository, @Controller

Ces annotations indiquent à Spring que les classes doivent être détectées automatiquement via le mécanisme de scan de composants.

1. @Component
Annotation générique permettant de déclarer un bean géré par Spring.

2. @Service

Spécialisation de @Component pour les services métiers.Elle indique que la classe contient la logique applicative.

3. @Repository

Annotation indiquant une classe d'accès aux données. Elle gère automatiquement certains aspects tels que la traduction des exceptions JDBC.

4. @Controller

Utilisée dans Spring MVC pour indiquer qu'une classe gère les requêtes HTTP. Elle expose des points d'entrée via des méthodes annotées comme @GetMapping ou @PostMapping.

Chapitre 3. Spring Boot : Simplification du Développement

Alors que le framework Spring esttrès puissant,il peut devenir complexe à configurer lorsque l'application commence à grandir. Spring Boot apporte une approche plus simple et plus rapide grâce à des mécanismes comme **l'auto-configuration**, les **starters**, un système de **configuration centralisé** et différents outils destinés à améliorer la productivité du développeur. Dans ce chapitre, nous allons découvrir comment Spring Boot simplifie la création d'applications Java, en particulier pour les projets JEE modernes.

3.1 Qu'est-ce que Spring Boot ?

Spring Boot est un framework *opinionated*, c'est-à-dire qu'il propose des choix par défaut afin de faciliter le travail du développeur. Son objectif principal est de permettre de **créer rapidement des applications basées sur Spring**, sans avoir à répéter les mêmes configurations techniques à chaque nouveau projet. Au lieu d'écrire du code de configuration répétitif et souvent complexe, Spring Boot fournit un ensemble de mécanismes qui automatisent cette configuration et permettent au développeur de se concentrer uniquement sur la logique métier. Les fonctionnalités clés de Spring Boot incluent :

- **Les Spring Boot Starters**
- **L'auto-configuration**
- **Une gestion de configuration élégante et centralisée**
- **Spring Boot Actuator**
- **Le support des serveurs embarqués**

Spring Boot Starters

Spring Boot propose de nombreux modules appelés starters, qui permettent de démarrer rapidement avec les technologies les plus courantes telles que Spring MVC, JPA, MongoDB, Spring Batch, Spring Security etc.

Ces starters sont préconfigurés avec les dépendances de bibliothèques les plus utilisées, ce qui évite au développeur de devoir chercher manuellement les versions compatibles et de les configurer une par une.

Par exemple, le module spring-boot-starter-data-jpa regroupe toutes les dépendances nécessaires pour utiliser Spring Data JPA, ainsi que les bibliothèques d'Hibernate, car Hibernate est l'implémentation JPA la plus couramment utilisée.

☐ ***Note*** *Vous pouvez trouver la liste complète de tous les starters Spring Boot disponibles par défaut dans la documentation officielle à l'adresse suivante:*

`http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/ #using-boot-starter-poms.`

Spring Boot Autoconfiguration

Spring Boot résout le problème de la configuration complexe des applications Spring en éliminant la nécessité de définir manuellement toutes les configurations répétitives (*boilerplate configuration*).

Il adopte une approche dite **opinionated**, ce qui signifie qu'il fait des choix par défaut pour faciliter la mise en place de l'application. Spring Boot configure automatiquement différents composants en enregistrant des beans selon plusieurs critères, tels que :

- La présence d'une classe spécifique dans le *classpath*
- La présence ou l'absence d'un bean Spring particulier
- L'existence d'une propriété système
- L'absence d'un fichier de configuration

Par exemple, Si vous avez la dépendance spring-webmvc dans votre *classpath*, Spring Boot suppose que vous souhaitez créer une application web basée sur Spring MVC et enregistre automatiquement le DispatcherServlet, sauf s'il a déjà été défini.

Si un driver de base de données embarquée comme **H2** ou **HSQL** est présent dans le classpath, et que vous n'avez pas configuré explicitement de bean DataSource, Spring Boot créera automatiquement un DataSource en utilisant une base de données en mémoire.

Gestion de Configuration Élégante

Spring permet déjà d'externaliser les propriétés configurables grâce à l'annotation @PropertySource. Spring Boot va encore plus loin en proposant des valeurs par défaut pertinentes ainsi qu'un système puissant de **liaison de propriétés typées**, permettant d'associer directement les valeurs de configuration aux propriétés d'un bean.

Il offre également la possibilité de gérer facilement plusieurs fichiers de configuration selon les profils (dev, test, prod), sans avoir besoin d'une configuration complexe.

Spring Boot Actuator

Avoir une visibilité claire sur une application en production est essentiel pour garantir sa stabilité et détecter les problèmes rapidement.

Le module **Spring Boot Actuator** fournit un ensemble complet de fonctionnalités prêtes à l'emploi, sans nécessiter beaucoup de code ou de configuration.Parmi les fonctionnalités principales de l'Actuator, on peut citer :

- La consultation des détails de configuration des beans
- L'affichage des mappings d'URL, des informations d'environnement et des valeurs des paramètres de configuration
- L'accès aux métriques de santé et aux indicateurs de performance de l'application

Support des Serveurs Embarqués

Traditionnellement, le développement d'applications web en Java nécessitait la création de modules de type **WAR**, puis leur déploiement sur des serveurs externes tels que Tomcat, WildFly, etc.Avec Spring Boot, ce n'est plus nécessaire.Vous pouvez créer un module **JAR** et embarquer directement le conteneur servlet dans l'application. L'application devient alors une unité de déploiement autonome, facile à exécuter et à transporter.Pendant le développement, il est également possible de lancer facilement l'application Spring Boot (au format JAR) directement depuis l'IDE ou depuis la ligne de commande, à l'aide d'outils comme Maven ou Gradle.

3.2 Explorez Votre Première Application Spring Boot

Il existe plusieurs façons de créer une application Spring Boot. La méthode la plus simple consiste à utiliser Spring Initializr via l'adresse start.spring.io, qui est un générateur en ligne d'applications Spring Boot.

Dans cette section, nous verrons comment créer une application web Spring Boot simple, capable de servir une page HTML, et nous explorerons les différents éléments qui composent une application Spring Boot typique.

Créer Le Projet Avec Spring Initializr

Vous pouvez accéder à l'adresse `start.spring.io` depuis votre navigateur pour consulter les détails du projet, comme illustré dans la Figure 3-1.

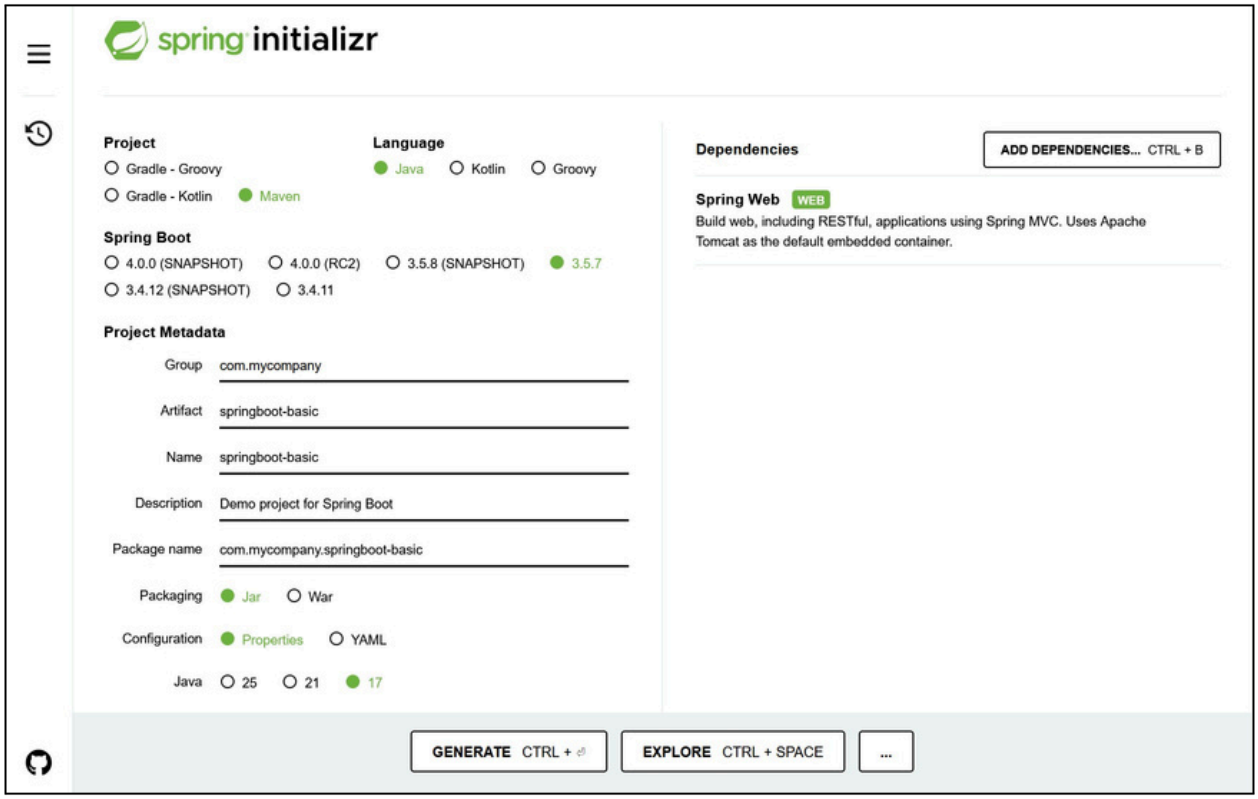


Figure 3-1. *Spring Initializr*

1. Sélectionnez **Maven Project** et la version de Spring Boot
2. Saisissez les informations du projet Maven comme suit :

    2.1. **Group** : `com.mycompany`
    2.2. **Langage** : `Java`
    2.3. **Artifact** : `springboot-basic`
    2.4. **Name** : `springboot-basic`
    2.5. **Package Name** : `com.mycompany.springboot-basic`
    2.6. **Packaging** : `JAR`
    2.7. **Version Java** : `17`

3. Vous pouvez rechercher directement les starters si vous connaissez leur nom. Vous verrez de nombreux modules regroupés par catégorie, comme **Core**, **Web**, **Data**, etc. Sélectionner **Spring Web** dans la catégorie Web. Cliquez sur le bouton **Generate**.
4. Cliquez sur le bouton **Generate**.

Vous pouvez maintenant extraire le fichier ZIP téléchargé et l'importer dans votre IDE préféré.

Exploration du Projet

Maintenant que vous avez créé un projet Spring Boot basé sur Maven avec le starter Web, vous êtes prêt à explorer le contenu de l'application générée.

1. Tout d'abord, jetezun œil au fichier pom.xml .

Listing 3-1. *Fichier* `pom.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.5.7</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <groupId>com.mycompany</groupId>
    <artifactId>springboot-basic</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springboot-basic</name>
    <description>Demo project for Spring Boot</description>
    <url/>

    <licenses>
        <license/>
    </licenses>

    <developers>
        <developer/>
    </developers>

    <scm>
        <connection/>
        <developerConnection/>
        <tag/> <url/>

    </scm>

    <properties>
        <java.version>17</java.version>
```

```
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
             </plugin>
        </plugins>
    </build>

</project>
```

La première chose à noter est que le module Maven **springboot-basic** hérite du module **spring-boot-starter-parent**. En héritant de ce module parent, le nouveau module bénéficie automatiquement des avantages suivants :

- **Version Spring Boot centralisée** : vous n'avez besoin de spécifier la version de Spring Boot qu'une seule fois dans la configuration du module parent. Il n'est pas nécessaire d'indiquer la version pour tous les starters et autres bibliothèques de support. Pour consulter la liste de ces bibliothèques, vous pouvez consulter le fichier pom.xml du module Maven `org.springframework.boot:spring-boot-dependencies:{version}`.

- **Plugins Maven préconfigurés** : le module parent spring-boot-starter-parent inclut déjà les plugins Maven les plus couramment utilisés, tels que maven-jar-plugin, maven-surefire-plugin, maven-war-plugin, exec-maven-plugin et maven-resources-plugin, avec des valeurs par défaut pertinentes.

- **Construction de JAR "fat"** : en plus des plugins mentionnés, le module parent configure également le **spring-boot-maven-plugin**, utilisé pour créer des JAR auto-exécutables (*fat JAR*).

Dans cet exemple, seul le starter Webaété sélectionné, mais le starter de test est inclus par défaut.

La version Java choisie est `17`, ce quiexplique la présence de la propriété suivante dans le `pom.xml` :

```
<java.version>17</java.version>
```

Cette valeur est utilisée pour configurer la version du JDK pour le compilateur Maven dans le module `spring-boot-starter-parent:`

```
<maven.compiler.source>${java.version}</maven.compiler.source>
<maven.compiler.target>${java.version}</maven.compiler.target>
```

2. Le module Spring Boot généré de type JAR contient une classe Java servant de point d'entrée à l'application, appelée SpringbootBasicApplication.java, avec la méthode public static void main(String[] args).C'est cette méthode que vous pouvez exécuter pour démarrer l'application.

Listing 3-2. `com.mycompany.springboot_basic`

```
package com.mycompany.springboot_basic;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringbootBasicApplication {

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

}
```

Ici, la classe SpringbootBasicApplication est annotée avec @SpringBootApplication, qui est une annotation composée.

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
@Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
  @Filter(type= FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class)
```

```
})
public @interface SpringBootApplication {
    // ...
}
```

L'annotation @SpringBootConfiguration est elle-même une annotation composée, utilisant l'annotation @Configuration de Spring .

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {
}
```

Voici la signification de ces annotations :

- @Configuration : indique que cette classe est une classe de configuration Spring.
- @ComponentScan : active le scan des composants pour détecter automatiquement les beans Spring dans le package où la classe actuelle est définie.
- @EnableAutoConfiguration : déclenche les mécanismes d'auto-configuration de Spring Boot.

L'application est initialisée en appelant la méthode SpringApplication.run(SpringbootBasicApplication.class, args)dans la méthode main(). Il est possible de passer une ou plusieurs classes de configuration Spring à la méthode SpringApplication.run().Cependant, si la classe de point d'entrée de votre application se trouve dans le package racine, il suffit de passer uniquement cette classe. Spring Boot se charge alors de scanner automatiquement toutes les autres classes de configuration Spring présentes dans les sous-packages.

3. Créez maintenant un contrôleur Spring MVC simple, appelé HomeController.java.

Listing 3-3. `HomeController.java`

```
packagecom.mycompany.springboot_basic;
importorg.springframework.stereotype.Controller;
importorg.springframework.ui.Model;
importorg.springframework.web.bind.annotation.RequestMapping;
@Controller
publicclass HomeController{
   @RequestMapping("/")
   publicString home(Model model) {
      return "index.html";
}
}
```

Il s'agit d'un contrôleur Spring MVC simple comportant une méthode de gestion des requêtes pour l'URL /, qui retourne la vue nommée index.html.

4. Créez une vue HTML appelée index.html.

Par défaut, Spring Boot sert le contenu statique depuis les répertoires src/main/public/ et src/main/static/ .Créez donc le fichier index.html dans src/main/public/.

Listing 3-4. index.html

```
<!DOCTYPE html>
<html>
<head>
<meta  charset="utf-8"/>
<title>Home</title>
</head>
<body>
<h2>Hello   World!!</h2>
</body>
</html>
```

Maintenant, depuis votre IDE, exécutez la méthode SpringbootBasicApplication.main() en tant que classe Java autonome.
Cela démarrera le **serveur Tomcat embarqué** sur le port 8080. Ensuite, ouvrez votre navigateur et rendez-vous à l'adresse :http://localhost:8080/.Vous devriez voir la réponse : Hello World!!

Il est également possible de lancer l'application Spring Boot en utilisant le plugin Maven spring-boot-maven-plugin, avec la commande suivante :

```
mvn spring-boot:run
```

La Classe Point d'Entrée de l'Application

Les applications Spring Boot doivent disposer d'une classe point d'entrée contenant la méthode : public static void main(String[] args)
Cette classe est généralement annotée avec @SpringBootApplication et sert à initialiser l'application (bootstrap).
Il est fortement recommandé de placer la classe point d'entrée dans le package racine, par exemple com.mycompany.myproject, afin que les annotations @EnableAutoConfiguration et scannent automatiquement les beans Spring, les entités JPA, etc., dans le package racine et tous ses sous-packages.
Si la classe point d'entrée se trouve dans un package imbriqué, il est nécessaire de spécifier explicitement les packages à scanner pour les composants Spring.

Listing 3.5. *Classe principale* `com.mycompany.myproject.config.Application.java` *dans un package non racine*

```
package com.mycompany.myproject.config;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.Configuration;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.context.annotation.ComponentScan;

@Configuration
@EnableAutoConfiguration
@ComponentScan(basePackages = "com.mycompany.myproject")
@EntityScan(basePackageClasses = Person.class)
publicclass Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Dans cet exemple, la classe Application.java se trouve dans le package com.mycompany.myproject.config, qui n'est pas le package racine.

Il est donc nécessaire de spécifier @ComponentScan(basePackages = "com.mycompany.myproject") pour que Spring Boot scanne le package racine et tous ses sous-packages à la recherche des composants Spring et utiliser @EntityScan(basePackageClasses = Person.class) pour que Spring Boot détecte les entités JPA dans le package où se trouve la classe Person.class.

Création d'un Fat JAR avec le Spring Boot Maven Plugin

Pendant le développement, vous pouvez exécuter votre application directement depuis l'IDE ou utiliser la commande Maven : mvn spring-boot:run. Cependant, pour la production, il est nécessaire de créer une **unité de déploiement autonome** pouvant être exécutée sans IDE.Le **spring-boot-maven-plugin** permet de créer une **unité de déploiement unique** (*fat JAR*) en exécutant les commandes Maven suivantes :

```
mvn clean package
```

Après compilation, deux fichiers intéressants apparaissent dans le répertoire target :

- springboot-basic-1.0-SNAPSHOT.jar.original : contient uniquement les classes compilées et les ressources du classpath.
- springboot-basic-1.0-SNAPSHOT.jar : contient tout ce qui est nécessaire pour exécuter l'application Spring Boot :

- Les **classes compilées** de votre code source (src/main/java) et les ressources statiques (src/main/resources) se trouvent dans le répertoire BOOT-INF/classes. .
- Tous les **JAR dépendants** sont dans BOOT-INF/lib .
- Les classes du package org.springframework.boot.loader, qui assurent la magie Spring Boot permettant d'exécuter l'application.

Il est possible de créer des JAR autonomes en utilisant des plugins comme maven-shade-plugin, qui empaquettent toutes les classes dépendantes dans un seul JAR.

Spring Boot adopte une approche différente : il permet d' imbriquer les JAR directement dans le JAR Spring Boot.

Pour plus de détails, vous pouvez consulter : Documentation Spring Boot – Fat JAR exécutable.

Pour exécuter l'application, utilisez la commande suivante :

```
java -jar springboot-basic-1.0-SNAPSHOT.jar
```

3.3 Auto-Configuration

Maintenant que vous savez comment créer une application Spring Boot simple et l'exécuter, il est intéressant de comprendre **l**e fonctionnement de l'auto-configuration de Spring Boot qui simplifie considérablement la configuration des applications Spring. Cette approche réduit le temps de développement et limite les erreurs liées à une configuration manuelle complexe.

Avant cela, il est important de connaître la fonctionnalité @Conditional de Spring, sur laquelle repose entièrement le mécanisme d'auto-configuration.

Principe du "Convention Over Configuration"

Le principe fondamental de l'auto-configuration est celui du **"convention over configuration"** (ou **convention plutôt que configuration**).Autrement dit, Spring Boot applique des paramètres par défaut pertinents pour la majorité des scénarios d'usage, de sorte que le développeur n'ait pas à configurer chaque détail manuellement.Par exemple, si vous ajoutez la dépendance spring-boot-starter-web dans votre projet, Spring Boot suppose que vous voulez créer une application web Spring MVC et configure automatiquement un DispatcherServlet, un serveur embarqué Tomcat et d'autres
composants nécessaires.

Explorer La Puissance de `@Conditional`

Lors du développement d'applications basées sur Spring, il peut être nécessaire d'**enregistrer des beans de manière conditionnelle**.Par exemple, vous pourriez vouloir enregistrer un bean DataSource pointant vers la base de données DEV lorsque vous exécutez l'application localement, et vers une base de données PRODUCTION lorsque l'application est déployée en production.

Il est possible d'externaliser les paramètres de connexion à la base de données dans des fichiers de propriétés et de sélectionner le fichier correspondant à l'environnement. Cependant, cela oblige à modifier la configuration et à redéployer l'application chaque fois que vous souhaitez changer d'environnement.

Pour résoudre ce problème, Spring 3.1 a introduit le concept de *profils*.Vous pouvez enregistrer plusieurs beans du même type et les associer à un ou plusieurs profils. Lors de l'exécution de l'application, vous activez le(s) profil(s) souhaité(s), et seuls les beans associés aux profils activés seront enregistrés.

Listing 3-6. *Configuration des DataSources par profil avec Spring*

```
@Configuration
public class AppConfig {

    @Bean
    @Profile("DEV")
    public DataSource devDataSource() {
        // configuration DEV
    }

    @Bean
    @Profile("PROD")
    public DataSource prodDataSource() {
        // configuration PROD
    }
}
```

Avec cette configuration, il est possible de spécifier le profil actif via la propriété système :

```
-Dspring.profiles.active=DEV
```

Cette approche fonctionne bien pour des cas simples, comme activer ou désactiver l'enregistrement de beans selon le profil activé.Mais si vous souhaitez enregistrer des beans selon une logique conditionnelle plus complexe, l'utilisation des profils seuls n'est pas suffisante.

Pour offrir une flexibilité beaucoup plus grande, Spring 4 a introduit le concept de @Conditional.
Avec cette approche, vous pouvez enregistrer un bean de manière conditionnelle en fonction de n'importe quelle condition arbitraire.

Par exemple, vous pouvez enregistrer un bean lorsque :

- Une classe spécifique est présente dans le classpath
- Un bean Spring d'un certain type n'est pas déjà enregistré dans le ApplicationContext
- Un fichier spécifique existe à un emplacement donné
- Une propriété spécifique est configurée dans un fichier de configuration
- Une propriété système spécifique est présente ou absente.

Ce ne sont que quelques exemples : il est possible de définir toute condition souhaitée.

Conditionnement Basé Sur Les Propriétés Système

Dans une application, il est souvent nécessaire de choisir dynamiquement entre plusieurs implémentations selon l'environnement d'exécution. Par exemple, utiliser une base de données MySQL en production et MongoDB en développement.
Considérons une interface UserDAO avec deux implémentations : JdbcUserDAO pour MySQL et MongoUserDAO pour MongoDB.

Listing 3-1. *Interface* `UserDAO` *, et implementations* `JdbcUserDAO MongoUserDAO`

```
// Interface commune
public interface UserDAO {
    List<String> getAllUserNames();
}

// Implémentation MySQL
public class JdbcUserDAO implements UserDAO {
    @Override
    public List<String> getAllUserNames() {
        System.out.println("**** Getting usernames from RDBMS *****");
        return Arrays.asList("Jim", "John", "Rob");
    }
}

//Implémentation MongoDB
publicclass MongoUserDAO implements UserDAO {
    @Override
    public List<String> getAllUserNames() {
        System.out.println("**** Getting usernames from MongoDB *****");
        return Arrays.asList("Bond", "James", "Bond");
    }
}
```

Listing 3-2. `MySQLDatabaseTypeCondition.java MongoDBDatabaseTypeCondition.java`

```
// Condition pour MySQL
public class MySQLDatabaseTypeCondition implements Condition {
    @Override
    public boolean matches(ConditionContext conditionContext,
                           AnnotatedTypeMetadata metadata) {
        String enabledDBType = System.getProperty("dbType");
        return (enabledDBType != null &&
                enabledDBType.equalsIgnoreCase("MYSQL"));
    }
}

// Condition pour MongoDB
```

```
publicclassMongoDBDatabaseTypeCondition implements Condition {
    @Override
    publicbooleanmatches(ConditionContext conditionContext,
                         AnnotatedTypeMetadata metadata) {
        StringenabledDBType = System.getProperty("dbType");
        return(enabledDBType != null &&
                enabledDBType.equalsIgnoreCase("MONGODB"));
    }
}
```

Vous pouvez maintenant configurer les beans JdbcUserDAO et MongoUserDAO de manière conditionnelle en utilisant l'annotation @Conditional.

Listing 3-3. `AppConfig.java`

```
@Configuration
public class AppConfig {
    @Bean
    @Conditional(MySQLDatabaseTypeCondition.class)
    public UserDAO jdbcUserDAO() {
        return new JdbcUserDAO();
    }

    @Bean
    @Conditional(MongoDBDatabaseTypeCondition.class)
    public UserDAO mongoUserDAO() {
         return new MongoUserDAO();
    }
}
```

Si vous exécutez l'application avec la commande :java -jar myapp.jar -DdbType=MYSQL seul le bean JdbcUserDAO sera enregistré.En revanche, si vous définissez la propriété système comme suit : -DdbType=MONGODB.le bean MongoUserDAO sera enregistré.

Les Annotations @Conditional intégrées de Spring Boot

Spring Boot propose de nombreuses annotations @Conditional personnalisées afin de répondre aux besoins d'**a**uto-configuration des développeurs selon différents critères.

Le **Tableau 3-1** répertorie les annotations @Conditional fournies par Spring Boot par défaut.

Annotation	Description
`@ConditionalOnBean`	S'applique lorsque les classes et/ou les noms de beans spécifiés sont déjà enregistrés.
`@ConditionalOnMissingBean`	Il s'applique lorsque les classes et/ou les noms de beans spécifiés ne sont pas encore enregistrés.
`@ConditionalOnClass`	S'applique lorsque les classes spécifiées sont présentes dans le classpath.
`@ConditionalOnMissingClass`	S'applique lorsque les classes spécifiées ne sont pas présentes dans le classpath.
`@ConditionalOnProperty`	S'applique lorsque les propriétés spécifiées ont une valeur particulière.
`@ConditionalOnResource`	S'applique lorsque les ressources spécifiées sont présentes dans le classpath.
`@ConditionalOnWebApplication`	S'applique lorsque le contexte de l'application est un contexte web.

Tableau 3-1. *Spring Boot* @Conditional *Annotations*

Maintenant que vous savez comment Spring Boot utilise l'annotation @Conditional pour décider conditionnellement d'enregistrer ou non un bean, vous pourriez vous demander ce qui déclenche exactement le mécanisme d'autoconfiguration.

Comment L'Autoconfiguration Fonctionne dans Spring Boot

La clé de l'autoconfiguration de Spring Boot est l'annotation @EnableAutoConfiguration. En général, on l'active en annotant la classe principale de l'application avec @SpringBootApplication, ou bien, si l'on souhaite personnaliser certains comportements, avec les trois annotations suivantes :

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application {
}
```

L'annotation @EnableAutoConfiguration permet à Spring d'activer le mécanisme d'autoconfiguration en analysant le classpath et en enregistrant automatiquement les beans correspondant à certaines conditions.

Spring Boot fournit de nombreuses classes d'autoconfiguration dans le module spring-boot-autoconfigure-{version}.jar. Chaque classe joue un rôle dans la création et la configuration automatique de composants spécifiques.

Les classes d'autoconfiguration sont généralement annotées avec @Configuration (elles représentent des configurations Spring), annotées avec @EnableConfigurationProperties pour activer la liaison automatique des propriétés de configuration et composées de méthodes qui enregistrent des beans, souvent protégées par des annotations conditionnelles.Prenons l'exemple de la classe suivante dans Listing 3-4

Listing3-4. org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration

```
@Configuration @ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class
}) @EnableConfigurationProperties(DataSourceProperties.class) @Import({
Registrar.class, DataSourcePoolMetadataProvidersConfiguration.class }) public
class DataSourceAutoConfiguration {

    ...
    @Bean
    @ConditionalOnMissingBean
    public DataSourceInitializer dataSourceInitializer(
            DataSourceProperties properties,
            ApplicationContext applicationContext) {
        return new DataSourceInitializer(properties, applicationContext);
    }
    ...
    @Conditional(EmbeddedDatabaseCondition.class)
    @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
    @Import(EmbeddedDataSourceConfiguration.class)
    protected static class EmbeddedDatabaseConfiguration { }
    ...
    @Configuration
    @Conditional(PooledDataSourceCondition.class)
    @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
    @Import({
        DataSourceConfiguration.Tomcat.class,
        DataSourceConfiguration.Hikari.class,
        DataSourceConfiguration.Dbcp2.class,
        DataSourceConfiguration.Generic.class
    })
    protected static class PooledDataSourceConfiguration { }
    ...
}
```

L'annotation @ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })indique que l'autoconfiguration ne sera appliquée que si ces classes sont présentes sur le classpath.

De plus, @EnableConfigurationProperties(DataSourceProperties.class) active la liaison automatique des propriétés externes vers une classe Java :

```
@ConfigurationProperties(prefix = DataSourceProperties.PREFIX)
public class DataSourceProperties {
    public static final String PREFIX = "spring.datasource";
    private String driverClassName;
    private String url;
    private String username;
    private String password; // getters & setters

}
```

Ainsi, les propriétés suivantes seront automatiquement injectées dans l'objet `DataSourceProperties:`

```
spring.datasource.url=jdbc:mysql://localhost:3306/test
spring.datasource.username=root
spring.datasource.password=secret
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

La classe DataSourceAutoConfiguration contient également de nombreuses méthodes ou classes internes annotées avec :

- `@ConditionalOnMissingBean`
- `@ConditionalOnClass`
- `@ConditionalOnProperty`

Ces conditions indiquent à Spring Boot quand un bean doit être créé ou non.
Par exemple : un bean ne sera enregistré que si aucun autre bean du même type n'existe déjà, ou seulement si une propriété est définie.
Dans le module d'autoconfiguration, vous pouvez aussi retrouver :

- `DispatcherServletAutoConfiguration`
- `HibernateJpaAutoConfiguration`
- `JpaRepositoriesAutoConfiguration`
- `JacksonAutoConfiguration`

Chacune applique automatiquement la configuration nécessaire selon ce qui est détecté dans le projet.

3.4 Les bases de Spring Boot

Spring Bootfournit plusieurs fonctionnalités permettant d'implémenter des fonctionnalités couramment utilisées, comme la journalisation (logging), l'externalisation des propriétés de configuration et Spring Boot Dev Tools pour redémarrer automatiquement le serveur lors de modifications du code, ce qui permet d'améliorer la productivité des développeurs.

Logging

Le logging ou la journalisation est une partie très importante de toute application et il aide à déboguer les problèmes. Par défaut, Spring Boot inclut spring-boot-starter-logging comme dépendance transitive pour le module spring-boot-starter. Par défaut, Spring Boot utilise SLF4J avec les implémentations Logback. Spring Boot possède une abstraction LoggingSystem qui configure automatiquement le logging en fonction des fichiers de configuration disponibles dans le classpath.

Si Logback est disponible, Spring Boot l'utilisera comme gestionnaire de logs. Vous pouvez facilement configurer les niveaux de logging dans le fichier application.properties, sans avoir à créer des fichiers spécifiques au fournisseur de logging comme logback.xml ou log4j.properties.

```
logging.level.org.springframework.web=INFO
logging.level.org.hibernate=ERROR
logging.level.com.mycompany=DEBUG
```

Si vous souhaitez **e**nregistrer les logs dans un fichier en plus de la console, vous pouvez spécifier le nom du fichier comme suit : `logging.path=/var/logs/app.log` ou `logging.file=myapp.log`.

Pour avoir un contrôle plus fin sur la configuration du logging, vous pouvez créer les fichiers spécifiques au fournisseur de logging dans leurs emplacements par défaut, que Spring Boot utilisera automatiquement.

Si vous souhaitez utiliser d'autres bibliothèques de logging, comme Log4J ou Log4j2, au lieu de Logback, vous pouvez exclure spring-boot-starter-logging et inclure le starter correspondant, comme suit :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j</artifactId>
</dependency>
```

Externalisation des Propriétés de Configuration

En général, vous voudrez externaliser les paramètres de configuration dans des fichiers de propriétés ou XML séparés, plutôt que de les intégrer directement dans le code, afin de pouvoir les modifier facilement selon l'environnement de l'application. Spring fournit l'annotation @PropertySource pour spécifier la liste des fichiers de configuration. Spring Boot va plus loin en enregistrant automatiquement un bean PropertyPlaceholderConfigurer en utilisant le fichier application.properties situé par défaut à la racine du classpath. Vous pouvez également créer des fichiers de configuration spécifiques à un profil en utilisant le nom de fichier application-{profil}.properties. Par exemple, vous pouvez avoir application.properties pour les valeurs par défaut, application-dev.properties pour le profil dev et application-prod.properties pour le profil production. Si vous souhaitez configurer des propriétés communes à tous les profils, vous pouvez les placer dans application-default.properties.

☐ ***Note*** *: vous pouvez également utiliser des fichiers* ***YAML (.yml)*** *comme alternative aux fichiers .properties. Voir la section « Using YAML instead of properties » dans la documentation officielle de Spring Boot* : Spring Boot Reference

---

Spring fournit l'annotation @Value pour lier une valeur de propriété à une propriété d'un bean. Cependant, lier chaque propriété individuellement avec @Value peut être fastidieux. C'est pourquoi Spring Boot a introduit un mécanisme permettant de lier automatiquement un ensemble de propriétés aux propriétés d'un bean de manière **type-safe**.

Supposons que vous ayez le fichier application.properties suivant et une classe DataSourceConfig comme suit :

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/test
jdbc.username=root
jdbc.password=secret

public class DataSourceConfig {
    private  String  driver;
    private    String    url;
    private String username;
    private String password;
```

```
    // setters et getters
}
```

Vous pouvez maintenant simplement annoter DataSourceConfig avec @ConfigurationProperties(prefix="jdbc") pour lier automatiquement toutes les propriétés commençant par jdbc.*.

```
@Component
@ConfigurationProperties(prefix="jdbc")
public class DataSourceConfig {
    ...
}
```

Vous pouvez ensuite injecter le bean DataSourceConfig dans d'autres beans Spring et accéder aux propriétés via les getters.

Les noms des propriétés du bean n'ont pas besoin d'être exactement identiques aux clés des propriétés. Spring Boot prend en charge le **relaxed binding**, ce qui signifie que la propriété du bean driverClassName peut être mappée à n'importe laquelle des clés suivantes : driverClassName, driver-class-name, ou DRIVER_CLASS_NAME.

Developer Tools

Pendant le développement, il est souvent nécessaire de modifier le code et de redémarrer le serveur pour que ces changements soient pris en compte. Spring Boot facilite ce processus grâce au module spring-boot-devtools, qui offre notamment un redémarrage rapide de l'application dès qu'une modification du classpath est détectée. Lorsque ce module est inclus, la mise en cache des templates de vues (Thymeleaf, Velocity, Freemarker, etc.) est automatiquement désactivée, permettant de visualiser immédiatement les changements effectués. Les propriétés configurées par défaut peuvent être consultées dans la classe org.springframework.boot.devtools.env.DevToolsPropertyDefaultsPostProcessor.
L'intégration du module se fait simplement via la dépendance suivante :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
</dependency>
```

Spring Boot developer tools déclenchent ainsi automatiquement le redémarrage de l'application à chaque modification du contenu du classpath.

Lorsque vous modifiez des classes ou des fichiers de configuration situés dans le classpath, Spring Boot redémarre automatiquement le serveur. En revanche, les ressources statiques comme les fichiers CSS, JS

ou HTML ne déclenchent pas de redémarrage. C'est pourquoi elles sont exclues par défaut dans la propriété suivante définie dans DevToolsProperties :

```
@ConfigurationProperties(prefix = "spring.devtools")
public class DevToolsProperties {
    ...
    public static class Restart {
        private static final String DEFAULT_RESTART_EXCLUDES =
            "META-INF/maven/**,"
            + "META-INF/resources/**,resources/**,"
            + "static/**,public/**,templates/**,"
                                +   "**/*Test.class,**/*Tests.class,git.properties,
META-INF/build-info.properties ";

        private String exclude = DEFAULT_RESTART_EXCLUDES;
        ...
    }
}
```

Vous pouvez remplacer cette liste d'exclusions via: spring.devtools.restart.exclude=assets/**,resources/**

Ou ajouter des exclusions ou chemins supplémentaires : spring.devtools.restart.additional-exclude=assets/**,setup-instructions/** : spring.devtools.restart.additional-paths=D:/global-overrides/

Lorsque vous devez effectuer plusieurs modifications avant de tester une fonctionnalité, le redémarrage automatique à chaque changement peut devenir gênant. Il est alors possible d'utiliser un fichier déclencheur grâce à spring.devtools.restart.trigger-file=restart.txt.

Le mécanisme de redémarrage s'appuie sur deux classloaders : un classloader de base pour les classes qui ne changent pas (issues des dépendances externes), et un classloader de redémarrage pour les classes de votre application. Lors d'un redémarrage, seul ce dernier est recréé, ce qui accélère significativement le processus. Vous pouvez désactiver le redémarrage automatique avec :spring.devtools.restart.enabled=false. Ou le désactiver complètement en le passant comme propriété système java -jar -Dspring.devtools.restart.enabled=false app.jar

Enfin, pour utiliser les mêmes réglages devtools sur plusieurs projets, il est possible de créer un fichier global sprint-boot-devtools.properties dans le répertoire utilisateur (C:\Users\<username>\ sous Windows ou /home/<username>/ sous Linux/MacOS).

Chapitre 4. Développement d'Applications Web avec Spring MVC

Dans ce chapitre, nous allons explorer la manière dont Spring Boot implémente le modèle architectural Modèle-Vue-Contrôleur (MVC) à travers son module Spring Web MVC, dédié au développement d'applications Web et d'APIs Web. Il couvre les concepts fondamentaux de la gestion des requêtes et des réponses, ainsi que les mécanismes de validation de données.

4.1 Architecture MVC et Contexte Spring Web

Le module Spring Web MVC est l'outil de Spring pour la construction de l'interface utilisateur et de l'interface d'API. Bien que le nom réfère au modèle MVC classique, son application est adaptée pour les APIs Web, où la Vue est remplacée par la sérialisation de données.

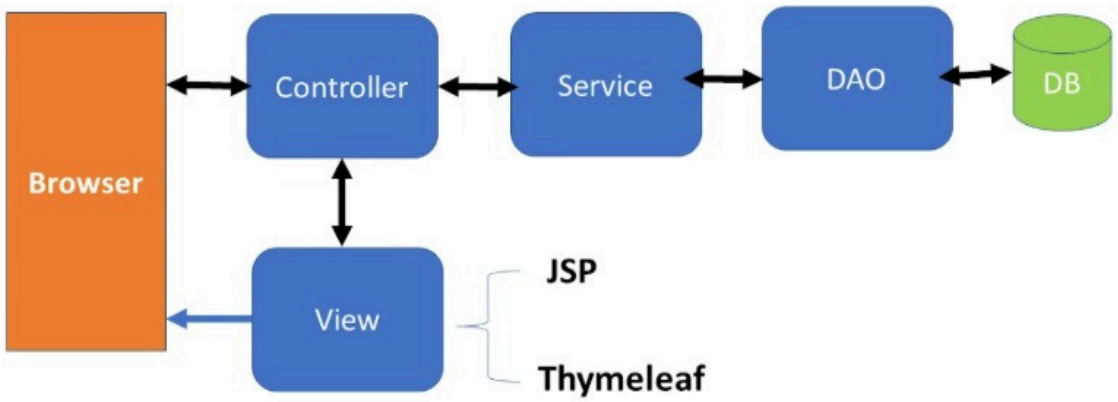


Figure 4-1. *Structure d'une Application Web JEE/Spring Boot*

Le Modèle MVC : Principes et Responsabilités

Le modèle **MVC** (Modèle Vue Contrôleur) est un patron d'architecture pour guider la conception d'applications nécessitant une interaction de l'utilisateur avec le système. Il définit trois grandes catégories de responsabilité :

- **Modèle (Model) :** Les classes appartenant à cette catégorie définissent les données applicatives (objets Java) échangées entre l'utilisateur et le système ou les données à afficher.
- **Vue (View) :** Les classes appartenant à cette catégorie gèrent la représentation graphique des données et l'interface utilisateur. Spring Web MVC permet l'utilisation de différentes technologies de vues comme Thymeleaf (le moteur recommandé) ou JSP.
- **Contrôleur (Controller) :** Les classes appartenant à cette catégorie gèrent les interactions de l'utilisateur (requêtes HTTP), valident les paramètres et assurent la cohérence entre le modèle et la vue après traitement par la couche de service.

Intégration de Spring Web MVC et Rôle du Serveur

Spring Web MVC nécessite un conteneur Web léger pour traiter les requêtes HTTP, car le Spring Framework lui-même ne fournit pas de serveur.

- **Approche Spring Boot :** Grâce à Spring Boot, l'application embarque son propre conteneur Web (comme Tomcat ou Jetty). Cela simplifie l'intégration et le déploiement par rapport aux serveurs d'applications Java EE complets.
- **Le Contrôleur dans le Flux :** Dans la logique MVC, l'utilisateur interagit avec le Contrôleur (via une requête HTTP). Ce contrôleur est chargé de valider les paramètres de la requête, de les transmettre à la couche de service pour traitement, puis d'alimenter le modèle pour le transmettre à la vue.

4.2 Controllers

Les contrôleurs sont des composants centraux qui gèrent les interactions entre le client et la logique métier. Spring utilise des annotations spécifiques pour définir et configurer ces classes.

@RestController vs @Controller

- **@Controller** est l'annotation standard utilisée pour les contrôleurs dans les applications traditionnelles basées sur les vues
  ➔ Le but est de générer et de retourner le nom logique d'une Vue (par exemple, un fichier HTML via Thymeleaf ou JSP) pour l'affichage par le navigateur.
- **@RestController** est l'annotation privilégiée pour la création d'APIs RESTful.
  ➔ C'est une annotation qui combine les fonctionnalités de @Controller et de @ResponseBody, cette combinaison permet d'indiquer à Spring que la valeur de retour de la méthode ne doit pas être interprétée comme le nom d'une vue, mais doit être sérialisée directement dans le corps de la réponse HTTP ,généralement au format JSON.

Mapping des Requêtes

Les annotations de mapping sont utilisées pour associer une méthode de contrôleur à une URL et à une méthode HTTP spécifiques :

Annotation	Méthode HTTP	Rôle
@GetMapping	GET	Récupération de données.
@PostMapping	POST	Création de nouvelles ressources.
@PutMapping	PUT	Modification complète d'une ressource.
@DeleteMapping	DELETE	Suppression d'une ressource.

Figure 4- 2.*Tableau sur les variantes de @RequestMapping*

- **Exemple de Contrôleur (Utilisation pour les APIs RESTful)**

```
@RestController
@RequestMapping("/api/produits") // Mappe toutes les requêtes de base
publicclass ProduitController {

    //@GetMapping gère la requête GET /api/produits
    @GetMapping
    public List<Produit> getAllProducts() {
        // l'appel au service pour récupérer tous les produits
        return service.findAll();
    }
    //@PostMapping gère la requête POST /api/produits
    @PostMapping
    public Produit createProduct(@RequestBody Produit nouveauProduit) {
        // l'appel au service pour sauvegarder le nouveau produit
        return service.save(nouveauProduit);
    }
}
```

Code 4-1. *Exemple de contrôleur*

Gestion des Paramètres et du Path

Spring permet de lier des parties de la requête HTTP aux arguments des méthodes du contrôleur :

- **@PathVariable :** permet d'extrairedesvaleurs dynamiques directement d'une partie de l'URL.
- **@RequestParam :** permet d'extrairedesparamètresderequête qui suivent le point d'interrogationdansl'URL (parexemple,/produits?page=1).
    - ➢ **Exemple de @PathVariable (Extraction de l'ID)**

```
    //L'ID du produit est extrait de l'URL /{id}
    @GetMapping("/{id}")
    public Produit getProductById(@PathVariable Long id) {
        // l'appel au service pour trouver le produit par ID
        return service.findById(id);
    }
```

Code 4-2. *Exemple utilisation de PathVariable*

4.3 Création d'APIs RESTful

Le développement d'API suit les principes de l'architecture REST (Representational State Transfer) pour une communication standardisée et stateless.

Principes REST

Une API RESTful modélise les données sous forme de ressources accessibles via des URI. Les interactions sont sans état (*stateless*), et l'interface est uniforme, s'appuyant sur les méthodes HTTP standard.

HTTP Methods et Codes de Statut

Les Codes de Statut HTTP sont fondamentaux pour le développement d'APIs RESTful, car ils normalisent la communication du résultat d'une requête au client.

Status Code	Action	Description
200	OK	Successfully retrieved resource
201	Created	A new resource was created
204	No Content	Request has nothing to return
301 / 302	Moved	Moved to another location (redirect)
400	Bad Request	Invalid request / syntax error
401 / 403	Unauthorized	Authentication failed / Access denied
404	Not Found	Invalid resource was requested
409	Conflict	Conflict was detected, e.g. duplicated email
500 / 503	Server Error	Internal server error / Service unavailable

Figure 4-3.*Codes de statut HTTP*

Ces codes sont divisés en classes principales :

- ❖ 2xx (Succès) : Indiquent que la requête a été traitée avec succès (ex: 200 OK, 201 Created).
- ❖ 4xx (Erreur Client) : Signalent une erreur de la part du client (ex: 400 Bad Request pour une validation échouée, 404 Not Found).
- ❖ 5xx (Erreur Serveur) : Indiquent une défaillance du serveur (ex: 500 Internal Server Error).

Pour définir et retourner explicitement ces statuts et assurer la conformité du protocole REST, on utilise la classeResponseEntity dansSpring.

Annotations de Gestion des Données

- @RequestBody : Mappe le corps de la requête HTTP (généralement JSON/XML) vers un objet Java.
- @ResponseBody : Indique que la valeur de retour doit être sérialisée dans le corps de la réponse. (Implicitement inclus avec @RestController).

4.4 Gestion des Réponses

Pour construire une API professionnelle, il faut contrôler précisément le contenu et les métadonnées (statut) des réponses.

ResponseEntity et Personnalisation des Réponses

La classe ResponseEntity permet d'envelopper l'objet de réponse pour contrôler explicitement les en-têtes HTTP et le code de statut, assurant une communication REST conforme.

➢ **Exemple d'utilisation de ResponseEntity**

```
@GetMapping("/secure/{id}")
 publicResponseEntity<Produit> getSecuredProduct(@PathVariable Long id) {
     Produitp=service.findById(id);
     if(p==null) {
         //Retourne un statut 404 Not Found si la ressource est introuvable
         returnnew ResponseEntity<>(HttpStatus.NOT_FOUND);
     }
     //Retournela ressource avec un statut 200 OK : la requête a réussi
     returnnewResponseEntity<>(p, HttpStatus.OK);
 }
```

Code 4-3. Exemple utilisation de ResponseEntity

Content Negotiation (JSON, XML)

Boot gère la négociation de contenu, qui détermine le format de la réponse (JSON, XML, etc.) en fonction des en-têtes de la requête client (notamment l'en-tête Accept). La sérialisation en JSON via la librairie Jackson est Spring le comportement par défaut pour les APIs REST.

4.5 Validation des Données

La validation des données est nécessaire pour garantir l'intégrité des informations avant leur traitement ou leur persistance.

Bean Validation (JSR-380)

Spring utilise les spécifications de Bean Validation (JSR-380), permettant de définir des contraintes déclaratives directement sur les champs des objets Java.

Annotations de Validation

Les contraintes sont définies à l'aide d'annotations :

- @NotNull / @NotEmpty / @NotBlank : Vérifie l'absence de valeur nulle, de chaîne vide,..
- @Size(min=x, max=y) : Limite la taille d'une chaîne ou d'une collection.
- @Min(value=x) / @Max(value=x) : Limite la valeur numérique.
- @Email : Valide le format de l'adresse e-mail.

➢ **Exemple de Bean (DTO) avec Annotations de Validation**

```
public class ProduitDTO {

    @NotNull(message = "Le nom ne peut pas être nul.")
    @Size(min = 3, max = 50, message = "Le nom doit contenir entre 3 et 50
caractères.")
    private String nom;
```

```
    @Min(value = 10, message = "Le prix doit être au moins de 10.")
    private double prix;

  }
```

Code 4-4. Exemple de Bean avec Annotations de Validation

Gestion des Erreurs de Validation

L'annotation @Valid ou @Validated placée devant le DTO dans le contrôleur déclenche le processus de validation. En cas d'échec, Spring lève une exception.

- Implémentation : Il est recommandé d'utiliser une classe annotée avec @ControllerAdvice pour intercepter l'exception (MethodArgumentNotValidException) et formater une réponse 400 Bad Request contenant une liste détaillée deserreurs pour le client.

➢ **Exemple d'utilisation de @Valid**

```
   @PostMapping("/validate")
    //@Valid déclenche la vérification des contraintes définies dans ProduitDTO
    public ResponseEntity<Produit> createValidProduct(@Valid @RequestBody
ProduitDTO produitDTO) {
        // Le code n'est exécuté que si la validation est réussie
        // l'appel au service de création
        return new ResponseEntity<>(produitDTO.toProduit(), HttpStatus.CREATED);
    }
```

Code 4-5. Exemple d'utilisation de @Valid

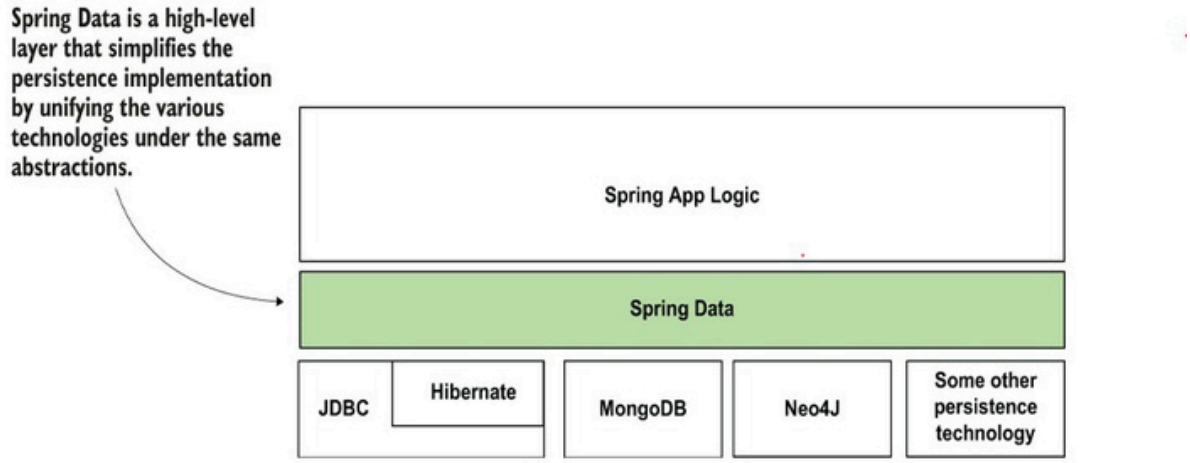
Chapitre 5. Persistance des Données avec Spring Data JPA

5.1 C’est quoi Spring Data ?

Dans l'écosystème Spring, la gestion de la persistance des données est grandement simplifiée par le projet Spring Data. Il s'agit d'un framework de haut niveau dont l'objectif principal est d'unifier et de faciliter l'accès à diverses sources de données, qu'il s'agisse de bases de données relationnelles, NoSQL, ou de systèmes de recherche. Sa philosophie est de réduire au maximum le code répétitif (boilerplate) traditionnellement nécessaire pour écrire la couche d'accès aux données, permettant ainsi aux développeurs de se concentrer sur la logique métier.

Spring Data adopte une approche modulaire selon la technologie de persistance utilisée. Ainsi, nous trouvons des modules spécialisés tels que **Spring Data JDBC** pour les accès directs via JDBC, **Spring Data MongoDB** pour les bases NoSQL document, **Spring Data Redis** pour les bases clé-valeur, Spring Data Elasticsearch pour les moteurs de recherche, ou encore **Spring Data Neo4j** pour les bases de données orientées graphes. Chaque module respecte un ensemble de contrats communs, offrant une expérience de développement cohérente quelle que soit la technologie sous-jacente.

Pour interagir avec les bases de données relationnelles, qui sont au cœur de nombreuses applications, Spring Data propose le module **Spring Data JPA**. Ce module s'appuie sur la norme établie dans le monde Java pour la persistance : la JPA (Java Persistence API). JPA est une spécification qui définit un cadre pour le mapping objet-relationnel (ORM), c'est-à-dire la technique qui permet de faire correspondre les objets de notre application aux tables d'une base de données. Des outils comme *Hibernate* sont des implémentations concrètes de cette spécification.



5-1 Illustration. *Modules de Spring Data*

La véritable puissance de **Spring Data JPA** réside dans son abstraction des Repositories (référentiels). Plutôt que d'écrire manuellement des classes d'accès aux données (DAO), le développeur se contente de définir une interface. À partir de cette simple interface, Spring Data est capable de générer dynamiquement une implémentation complète fournissant les opérations de base *(CRUD - Create, Read, Update, Delete)* ainsi que des requêtes plus complexes dérivées simplement du nom des méthodes. Cette approche déclarative sera au centre de notre étude dans ce chapitre

5.2 Introduction à JPA et Hibernate

ORM : conceptsetavantages

Le mapping objet-relationnel (ORM) résout l'une des problématiques fondamentales du développement d'applications : le décalage d'impédance entre le modèle objet utilisé dans les langages de programmation et le modèle relationnel des bases de données. Cette technique permet de manipuler les données sous forme d'objets Java tout en bénéficiant de la robustesse et des performances des bases de données relationnelles. Les principaux avantages de l'ORM incluent :

- **Abstraction de la base de données :** Le développeur travaille avec des objets Java plutôt qu'avec du SQL brut
- **Portabilité :** Le même code peut fonctionner avec différents SGBD (MySQL, PostgreSQL, Oracle, etc.)
- **Productivité accrue :** Réduction significative du code de persistance manuel
- **Gestion automatique des relations :** Mapping transparent des associations entre entités
- **Optimisations intégrées :** *Lazy loading*, cache de premier et second niveau, regroupement de requêtes

JPA comme spécification, Hibernate comme implémentation

JPA (Java Persistence API) est une spécification Java EE qui standardise les concepts ORM. Elle définit un ensemble d'annotations *(@Entity, @Table, @Column)*, d'APIs *(EntityManager, Query)* et de comportements que toute implémentation doit respecter. JPA ne fournit pas de code exécutable, mais plutôt un contrat que les fournisseurs d'ORM doivent implémenter.

Hibernate est l'implémentation de référence de JPA, développée par Red Hat. Il s'agit d'un framework ORM mature qui existait bien avant la spécification JPA et qui a largement inspiré cette dernière.
Hibernate fournit :

- Une implémentation complète et performante de JPA
- Des fonctionnalités étendues au-delà de la spécification (Criteria API native, types personnalisés, etc.)
- Un moteur de requêtes HQL (Hibernate Query Language) particulièrement puissant
- Des mécanismes avancés de cache et d'optimisation

Dans Spring Boot, Hibernate est automatiquement configuré comme implémentation JPA par défaut via le starter spring-boot-starter-data-jpa. Cette configuration transparente permet aux développeurs de

bénéficier immédiatement de toute la puissance d'Hibernate tout en respectant les standards JPA, garantissant ainsi la portabilité de leur code.

Configuration de la base de données

L'une des forces de Spring Boot réside dans sa capacité à simplifier drastiquement la configuration de la persistance. Grâce à son principe d'auto-configuration, Spring Boot peut détecter automatiquement les dépendances présentes dans le classpath et configurer la base de données en conséquence, pour travailler avec spring data jpa il faut ajouter la dépendance suivante:

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

➢ Base de données H2

La base de données H2 est un système de gestion de bases de données relationnelles (SGBDR) open source, léger, entièrement écrit en Java. Elle est principalement utilisée en mode "embarqué" (intégrée directement dans l'application) ou "en mémoire" in-memory pour offrir des performances très rapides. Grâce à son faible encombrement et sa conformité à l'API JDBC standard, H2 est particulièrement appréciée pour les phases de développement, le prototypage rapide et, surtout, les tests automatisés dans les applications Java, notamment celles basées sur le framework Spring Boot. Elle propose également une console web intégrée accessible via /h2-console, permettant d'exécuter des requêtes SQL directement depuis le navigateur.. Pour l'utiliser, il suffit d'ajouter la dépendance :

```
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <scope>runtime</scope>
</dependency>
```

Configuration H2 dans application.properties :

Par défaut, Spring Boot configure automatiquement une base de données H2 en mémoire si la dépendance est présente et qu'aucune autre source de données n'est configurée. Vous pouvez personnaliser ce comportement dans le fichier src/main/resources/application.properties :

```
# H2 In-Memory Configuration
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# Activation de la console web H2
```

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

`spring.datasource.driver-class-name` spécifie le pilote Java correct, tandis que `spring.datasource.url` définitl'adresseetle mode de fonctionnement (en mémoire) de la base de données.

Configuration JPA/Hibernate (optionnel)

Pour gérer la création et les mises à jour du schéma de base de données par Hibernate pendant le développement, vous pouvez configurer spring.jpa.hibernate.ddl-auto contrôle la manière dont le schéma de la base de données est automatiquement généré ou mis à jour au démarrage de l'application, Pour les bases de données embarquées (comme H2, HSQLDB, ou Derby), la valeur par défaut est create-drop. L'affichage des requêtes SQL est activé avec spring.jpa.show-sql=**true** et spring.jpa.properties.hibernate.format_sql formate les requêtes SQL pour une meilleure lisibilité :

```
# Configuration JPA/Hibernate
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
```

➢ Base de données MySQL

MySQL est l'un des SGBD relationnels les plus populaires en entreprise. Pour l'intégrer à Spring Boot :

```
<dependency>
	<groupId>com.mysql</groupId>
	<artifactId>mysql-connector-j</artifactId>
	<scope>runtime</scope>
</dependency>
```

Configuration MySQL dans application.properties :

```
# Configuration de la source de données MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/votre_nom_de_base_de_donnees
spring.datasource.username=votre_utilisateur
spring.datasource.password=votre_mot_de_passe

# Spécifie explicitement la classe du pilote MySQL
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

# Configuration JPA/Hibernate (facultatif mais recommandé)
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

- spring.datasource.url : L'URL inclut l'hôte (localhost si c'est sur votre machine), le port par défaut de MySQL (3306), et le nom de la base de données (votre_nom_de_base_de_donnees) que vous devez avoir préalablement créée.
- spring.datasource.username et spring.datasource.password : Les identifiants de
- connexion à votre base de données. spring.datasource.driver-class-name : Le nom de la classe du pilote MySQL moderne.
- spring.jpa.hibernate.ddl-auto : Pour MySQL, la valeur par défaut de Spring Boot est none en production.

➢ Base de données PostegreSQL

La configuration de PostgreSQL suit le même principe que MySQL. Son intégration nécessite l'ajout du connecteur JDBC pour PostgreSQL dans pom.xml :

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
```

ConfigurationPostgreSQLdansapplication.properties :

Remplacez les propriétés MySQL ou H2 par celles de PostgreSQL dans src/main/resources/ application.properties :

```
# Configuration de la source de données PostgreSQL
spring.datasource.url=jdbc:postgresql://localhost:5432/votre_nom de base_de_donnees
spring.datasource.username=votre_utilisateur
spring.datasource.password=votre_mot_de_passe

# Spécifie explicitement la classe du pilote (souvent facultatif avec Spring Boot)
spring.datasource.driver-class-name=org.postgresql.Driver

# Configuration JPA/Hibernate (facultatif mais recommandé)
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

- spring.datasource.url : Utilise le format jdbc:postgresql://[hôte]:[port]/[base_de_données]. Le port par défaut est 5432.

- spring.datasource.driver-**class-name** : La classe du pilote standard pour PostgreSQL est org.postgresql.Driver.
- `spring.jpa.properties.hibernate.dialect` : Bien que Spring Boot puisse souvent le déduire automatiquement, spécifier le dialecte org.hibernate.dialect.PostgreSQLDialect permet à Hibernate de générer le SQL le plus optimisé pour PostgreSQL**.**

Entités JPA

Une entité est une simple classe Java *(POJO - Plain Old Java Object)* dont les instances correspondent à des lignes dans une table de base de données. Le mapping entre la classe et la table est réalisé à l'aide d'annotations.

Mapping objet-relationnel (@Entity, @Table, @Column)

Les annotations de base permettent de déclarer une classe comme une entité et de personnaliser son mapping avec la structure de la base de données.

- **@Entity :** C'est l'annotation fondamentale qui marque une classe comme étant une entité JPA. Elle signale au provider de persistance (Hibernate) que cette classe doit être gérée et que ses objets peuvent être stockés en base de données.
- **@Id:** Cette annotation est placée sur le champ qui sert de clé primaire.
- **@GeneratedValue :** Combinée avec **@Id**, cette annotation spécifie la stratégie de génération de la clé primaire. Les stratégies les plus courantes (GenerationType) sont :
  - IDENTITY : S'appuie sur une colonne auto-incrémentée de la base de données .
  - SEQUENCE : Utilise une séquence de base de données pour générer la valeur .
  - AUTO : (Défaut) Laisse le provider de persistance (Hibernate) choisir la stratégie la plus appropriée en fonction du dialecte de la base de données.

Exemple de classe marquée comme une entité JPA

```
import  javax.persistence.Entity;  import
javax.persistence.GeneratedValue;  import
javax.persistence.GenerationType;  import
javax.persistence.Id;

@Entity public class Employee {

    @Id          @GeneratedValue(strategy          =
    GenerationType.AUTO)  private  long  id;  private
    String name; private String city;



    public Employee() {
    }
```

```
    public Employee(String name, String city) {
        this.name  =  name;
        this.city = city;
    }
//getters et setters
}
```

Comme aucune annotation **@Table** n'existe, il suppose que cette entité est mappée à une table nommée *Employee*.

❖ **@Table :** Optionnelle, cette annotation permet de spécifier les détails de la table à laquelle l'entité est mappée. Par défaut, le nom de la table est le nom de la classe. On l'utilise principalement pour définir un nom de table différent (name), un schéma (schema), ou des contraintes d'unicité.
❖ **@Column :** Appliquée sur un champ de l'entité, cette annotation permet de personnaliser le mapping avec la colonne correspondante. On peut spécifier son nom (name), sa longueur (length), si elle peut être nulle (nullable), ou si sa valeur doit être unique (unique).

Exemple avec @Table et @Column

```
import javax.persistence.*;

@Entity // Marque cette classe comme une entité JPA
@Table(name = "produits") // Mappe cette entité à la table "produits"
public class Produit {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @Column(name = "nom_produit", nullable = false, length = 100)
    private String nom;

    @Column(length = 500)
    private String description;

    private double prix;

    // Constructeurs, Getters et Setters...
}
```

Relations entre entités

L'un des plus grands atouts de JPA est sa capacité à modéliser les relations entre les tables directement dans le code objet.

@ManyToOne (Plusieurs-à-Un) : C'est la relation la plus commune. Par exemple, plusieurs produits (many) appartiennent à une seule catégorie (one). C'est généralement le côté "propriétaire" de la relation, celui qui porte la colonne de clé étrangère.

@OneToMany (Un-à-Plusieurs) : C'est le côté inverse de la relation @ManyToOne. Une catégorie (one) peut avoir une collection de produits (many). On utilise l'attribut mappedBy pour indiquer que la relation est gérée par l'autre entité, évitant ainsi la redondance.

@ManyToMany (Plusieurs-à-Plusieurs) : Modélise une relation où une instance d'une entité peut être associée à plusieurs instances d'une autre, et vice-versa (par exemple, des produits et des commandes). JPA gère cette relation en utilisant une table de jonction.

Exemple de relation OneToMany / ManyToOne : une catégorie et plusieurs produits

```
// Dans la classe Produit.java @Entity @Table(name = "produits") public class
Produit { //clé primaire @ManyToOne(fetch = FetchType.LAZY) // Plusieurs
produits pour une catégorie @JoinColumn(name = "categorie_id") // Nom de la
colonne de la clé étrangère private Categorie categorie;




    // Colonnes, Constructeurs, Getters et Setters...
}
// Dans une nouvelle classe Categorie.java
@Entity
@Table(name = "categories")
public class Categorie {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nom;
 // mappedBy="categorie" fait référence au champ "categorie" dans l'entité Produit
    @OneToMany(mappedBy = "categorie", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<Produit> produits = new ArrayList<>();

    // Constructeurs, Getters et Setters...
}
```

Repositories Spring Data

Spring Data s'organise autour de la notion de *repository*. Il fournit une interface marqueur générique Repository<T, ID>. Le type T correspond au type de l'objet géré par le *repository*. Le type ID correspond au type de la clé d'un objet.

L'interface CrudRepository<T, ID> hérite de Repository<T, ID> et fournit un ensemble d'opérations élémentaires pour la manipulation des objets.

Spring Data JPA fournit l'interface JpaRepository<T, ID> qui hérite de CrudRepository<T, ID> et qui fournit un ensemble de méthodes plus spécifiquement adaptées pour interagir avec une base de données relationnelle.

Pour définir un *repository*, il suffit de créer une interface qui hérite d'une des interfaces ci-dessus.

Exemple de Repository pour l'entité Produit

```
import org.springframework.data.jpa.repository.JpaRepository;
@Repository //Annotation (optionnelle mais recommandée) qui identifie ce bean comme
un repository
public interface ProduitRepository extends JpaRepository<Product, Long> {
}
```

En déclarant simplement cette interface, Spring Boot va automatiquement détecter ProduitRepository, comprendre qu'il doit la gérer, et créer à l'exécution un bean qui implémente toutes les méthodes de JpaRepository.

Méthodes CRUD automatiques

En héritant de JpaRepository (qui hérite elle-même CrudRepository), ProduitRepository dispose instantanément d'un ensemble complet de méthodes pour les opérations CRUD de base, sans écrire une seule ligne d'implémentation :

- save(Produit produit) : Sauvegarde un nouveau produit ou met à jour un produit existant.
- findById(Long id) : Récupère un produit par sa clé primaire. Renvoie un Optional<Produit>.
- findAll() : Renvoie la liste de tous les produits.
- deleteById(Long id) : Supprime un produit par sa clé primaire.
- count() : Compte le nombre total de produits.
- existsById(Long id) : Vérifie si un produit avec cet ID existe.

Ajout de méthodes dans une interface de repository

L'interface JpaRepository<T, ID> déclare beaucoup de méthodes mais elles suffisent rarement pour implémenter les fonctionnalités attendues d'une application. Spring Data JPA utilise une convention de nommage pour générer automatiquement le code sous-jacent et exécuter la requête. La requête est déduite de la signature de la méthode (on parle de *query methods*).

La convention est la suivante : Spring Data JPA supprime du début de la méthode les préfixes find, findAll, read, query, count et get et recherche la présence du mot By pour marquer le début des critères de filtre. Le terme après By fait référence à un attribut de l'entité JPA pour lequel on veut appliquer un filtre. Chaque critère doit correspondre à un paramètre de la méthode en respectant l'ordre.

Exemple de méthodes à ajouter dans ProduitRepository

```
public interface ProduitRepository extends JpaRepository<Produit, Long> {

    // SELECT p FROM Produit p WHERE p.nom = ?1
    Optional<Produit> findByNom(String nom);

    // SELECT p FROM Produit p WHERE p.prix < ?1
    List<Produit> findByPrixLessThan(double prixMax);

    // SELECT p FROM Produit p WHERE p.nom LIKE %?1%
    List<Produit> findByNomContainingIgnoreCase(String keyword);

    // SELECT p FROM Produit p WHERE p.categorie.nom = ?1
    List<Produit> findByCategorieNom(String nomCategorie);
}
```

Spring Data JPA générera une implémentation pour chaque méthode de ce *repository*.

Requêtes personnalisées

Lorsque les conventions de nommage ne suffisent pas pour des requêtes plus complexes (impliquant des jointures spécifiques, des agrégations ou des sous-requêtes), Spring Data permet d'écrire des requêtes personnalisées à l'aide de l'annotation **@Query.**

On peut écrire la requête en JPQL (Java Persistence Query Language), qui est similaire au SQL mais opère sur les entités et leurs propriétés.

Exemple avec JPQL et des paramètres nommés :

```
import org.springframework.data.repository.query.Param;

public interface ProduitRepository extends JpaRepository<Produit, Long> {
    @Query("SELECT p FROM Produit p WHERE p.categorie.id = :catId AND p.prix>
:prixMin")
    List<Produit> findProduitsChersDansCategorie(
        @Param("catId") Long categorieId,
        @Param("prixMin") double prixMinimum
    );}
```

Il est également possible d'exécuter des requêtes SQL natives en ajoutant l'attribut nativeQuery = true. C'est utile pour exploiter des fonctionnalités spécifiques à une base de données.

```
@Query(
  value = "SELECT * FROM produits p JOIN categories c ON p.categorie_id = c.id
```

```
WHERE c.nom = ?1",
  nativeQuery = true)
List<Produit> findByCategorieNomNative(String nomCategorie);
```

Grâce aux repositories, la couche d'accès aux données devient à la fois simple, puissante et extrêmement productive.

La notion de transaction est fondamentale dans les systèmes d'information. Une transaction respecte quatre propriétés désignées par l'acronyme ACID (Atomicité, Cohérence, Isolation, Durabilité). Elle est définie par un début et une fin : soit une validation des modifications (commit), soit une annulation (rollback).

La démarcation transactionnelle dans la couche Service

On parle de démarcation transactionnelle pour désigner la portion de code qui doit s'exécuter comme un bloc unique. Dans une architecture multi-couches, la couche de service (ou couche métier) est l'endroit idéal pour cette démarcation. En effet, une méthode de service représente souvent une fonctionnalité complète qui peut nécessiter plusieurs opérations sur la base de données. Ces opérations doivent réussir ou échouer en bloc.

Par défaut, Spring Data JPA active les transactions sur chaque méthode des repositories. Cela signifie qu'un appel à repository.save() est une transaction à lui seul. Cette configuration peut entraîner des incohérences : si une méthode de service appelle deux méthodes save() et qu'une erreur survient après le premier appel, ce dernier ne sera pas annulé.

Pour des applications robustes, il est donc recommandé de désactiver ce comportement et de gérer les transactions exclusivement au niveau de la couche de service.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@SpringBootApplication
@EnableJpaRepositories(enableDefaultTransactions = false) // Désactivation des
transactions par défaut
public class MyApplication {
	public static void main(String[] args) {
	  SpringApplication.run(MyApplication.class, args);
	}
}
```

Une fois cette option désactivée, tout appel à une méthode de repository modifiant des données devra obligatoirement être exécuté depuis un contexte transactionnel (comme une méthode de service annotée), sous peine d'échouer.

Gestion transactionnelle avec @Transactional

L'annotation **@Transactional**deSpring, placée sur une méthode de service, demande à Spring d'envelopper son exécution dansunetransaction :

- ❖ Si la méthode se terminesanserreur, Spring valide la transaction (commit).
- ❖ Si une RuntimeExceptionestlevée, Spring annule la transaction (rollback).

Exemple dans une couche Service:

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import javax.persistence.EntityNotFoundException;

@Service
public class ProduitService {

	private final ProduitRepository produitRepository;
	public ProduitService(ProduitRepository produitRepository) {
	this.produitRepository = produitRepository;
	}

	@Transactional
	public void updateProduitPrice(Long produitId, double nouveauPrix) {
	Produit produit = produitRepository.findById(produitId)
		.orElseThrow(() -> new EntityNotFoundException("Produit non trouvé
avec l'id : " + produitId));

	if (nouveauPrix <= 0) {
		thrownewIllegalArgumentException("Leprixdoitêtrestrictementpositif.");
	}

	produit.setPrix(nouveauPrix);

	// Pas besoin d'appeler produitRepository.save(produit).
	// Dans une transaction, Hibernate surveille les changements sur les entités (dirty
checking)
	// et propage automatiquement la mise à jour en base de données au moment du commit.
	}
}
```

Avec **@Transactional**, l'opération de mise à jour du prix est atomique : soit elle réussit complètement, soit elle est entièrement annulée, garantissant ainsi l'intégrité des données.

Propagation et Isolation

L'annotation @Transactional peut être affinée avec des attributs :

Propagation (propagation)

Définit le comportement si une méthode transactionnelle en appelle une autre.

- ❖ REQUIRED (défaut) : La méthode rejoint la transaction existante ou en crée une nouvelle.
- ❖ REQUIRES_NEW : Crée toujours une nouvelle transaction indépendante.
- ❖ Autres : SUPPORTS, NOT_SUPPORTED, MANDATORY, NEVER

Isolation (isolation)

Définit le degré d'isolation d'une transaction par rapport aux autres.

- ❖ READ_COMMITTED : Empêche la lecture de données non validées (défaut sur PostgreSQL, Oracle).
- ❖ REPEATABLE_READ : Empêche qu'une même lecture donne des résultats différents dans la même transaction (défaut sur MySQL).
- ❖ Autres : READ_UNCOMMITTED, SERIALIZABLE

Exemple d'une transaction configurée pour un audit indépendant

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void auditAction(String message) {
      // ... Logique d'audit qui sera validée même si l'appelant échoue}
```

Conclusion

Ce chapitre a démontré l'efficacité de l'écosystème Spring pour la persistance des données. En nous appuyant sur la norme JPA et son implémentation Hibernate, nous avons mis en place une couche de mapping objet-relationnel robuste, dont la configuration est grandement simplifiée par Spring Boot.

L'atout majeur réside dans Spring Data JPA et son concept de Repositories. Grâce à de simples interfaces, nous avons obtenu une couche d'accès aux données complète, incluant les opérations CRUD et des requêtes personnalisées, réduisant ainsi drastiquement le code à écrire.

Finalement, l'utilisation de @Transactional au niveau de la couche de service a permis de garantir l'intégrité et la cohérence des données via une gestion transactionnelle simple et déclarative.

En somme, Spring Data JPA offre une abstraction puissante qui accélère le développement de la couche de persistance tout en assurant la fiabilité des opérations, permettant aux développeurs de se concentrer sur la logique métier.

Chapitre 6. Sécurisation des Applications avec Spring Security

La sécurité web constitue aujourd'hui un pilier fondamental du développement d'applications en ligne. Avec l'évolution rapide des technologies et la montée en sophistication des cyberattaques, la protection des données, la prévention des accès non autorisés et la sécurisation des échanges deviennent des exigences incontournables. Les applications web modernes sont exposées à une multitude de risques : injections SQL, attaques XSS, vols de session, CSRF, déni de service, phishing, qui peuvent compromettre la confidentialité, l'intégrité et la disponibilité des systèmes.

Comprendre les bases de la sécurité web, maîtriser les concepts d'authentification et d'autorisation, et adopter des pratiques robustes (HTTPS, encodage, gestion des sessions, validation des entrées...) sont des étapes essentielles pour renforcer la résilience d'une application.

Dans ce contexte, les frameworks modernes offrent aux développeurs des mécanismes complets et centralisés pour gérer la sécurité. C'est précisément le rôle de **Spring Security**, un composant majeur de l'écosystème Spring, conçu pour répondre efficacement aux besoins de protection des applications Java EE et, plus particulièrement, des applications Spring Boot.

6.1 Qu'est ce que Spring Security?

Spring Security est un framework puissant, flexible et hautement personnalisable destiné à assurer la sécurité des applications Java. Il fournit un ensemble complet de mécanismes pour gérer l'authentification (vérification de l'identité de l'utilisateur) et l'autorisation (contrôle des permissions et des accès).

Initialement développé comme *Acegi Security*, il est aujourd'hui devenu la solution standard pour sécuriser les applications Spring.

Contrairement à un pare-feu ou un système de surveillance réseau, Spring Security n'a pas pour rôle de bloquer les attaques au niveau infrastructurel ; il agit au niveau applicatif en filtrant les requêtes HTTP, en protégeant les méthodes sensibles, en gérant les sessions, l'encodage des mots de passe, et en offrant une protection native contre des attaques courantes (CSRF, XSS, fixation de session…).

Il peut s'intégrer à différents modes d'authentification : base de données, LDAP, authentification par formulaires, OAuth2, SSO, JAAS, tokens JWT, etc., ce qui en fait un outil polyvalent et adapté aux architectures modernes, y compris les microservices.

6.2 Comment Spring Boot simplifie l'utilisation de Spring Security?

Sans Spring Boot, la configuration de Spring Security nécessite plusieurs fichiers XML ou des classes Java complexes pour définir les filtres, déclarer les beans, gérer les sessions, etc.

Spring Boot simplifie radicalement ce processus :

- Il **auto-configure** la chaîne de sécurité (Security Filter Chain).
- Il génère un **utilisateur par défaut** avec un mot de passe temporaire.
- Il protège automatiquement **toutes les URLs** de l'application.
- Il permet de personnaliser facilement la sécurité avec des annotations ou une simple classe Java.

6.3 Fonctionnement général

Avant d'entrer dans les détails techniques, il est essentiel de comprendre le fonctionnement global de Spring Security dans une application Spring Boot. Lorsqu'un utilisateur envoie une requête HTTP, celle-ci ne parvient pas directement aux contrôleurs. Elle traverse d'abord un ensemble de filtres spécialisés organisés dans ce qu'on appelle la ***Security Filter Chain***. Ces filtres analysent la requête, déterminent si l'utilisateur est authentifié, vérifient ses permissions et appliquent plusieurs protections de sécurité (CSRF, sessions, en-têtes sécurisés…).

Spring Security agit donc comme une couche de défense placée avant la logique métier. Au lieu d'éparpiller la sécurité dans toute l'application, le framework centralise la vérification des identités, des rôles et des droits d'accès. Ce modèle est comparable à un checkpoint : aucune requête ne passe sans être inspectée.

L'intérêt de cette architecture réside dans son efficacité et sa modularité : chaque filtre est responsable d'un aspect spécifique (authentification, autorisation, gestion des sessions…), ce qui permet au développeur de personnaliser la sécurité sans complexifier le code métier.

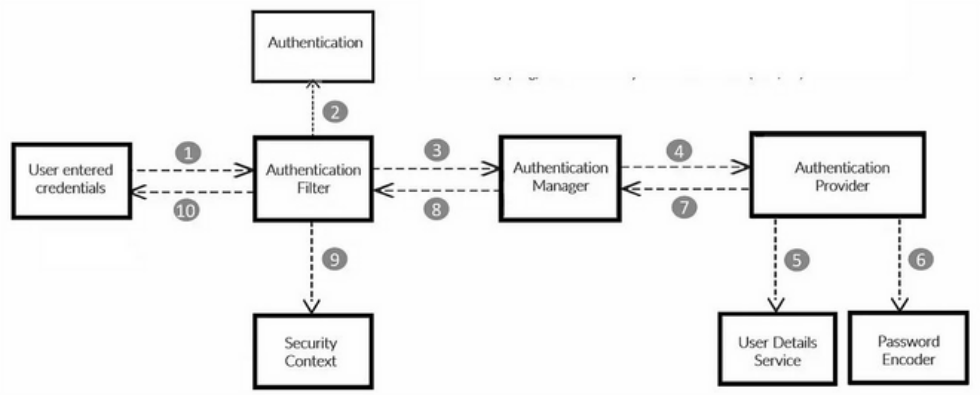


Figure 6-1. *Spring Security Flow*

Ce schéma illustre de manière simplifiée le processus complet d'authentification dans Spring Security. Lorsque l'utilisateur saisit ses identifiants (1), ceux-ci sont d'abord interceptés par un Authentication

Filter. Ce filtre transforme les informations fournies en un objet Authentication ,(2) puis les transmet à l'**AuthenticationManager** (3). L'**AuthenticationManager** délègue ensuite la vérification à un ou plusieurs **AuthenticationProvider** (4), responsables de comparer les informations reçues avec celles stockées dans le système.

L'**AuthenticationProvider** fait appel au **UserDetailsService** (5), qui récupère depuis la base de données l'utilisateur correspondant, ainsi qu'au **PasswordEncoder** (6), qui vérifie que le mot de passe saisi correspond au mot de passe haché enregistré. Si l'authentification est réussie, le provider renvoie un objet Authentication complet contenant les rôles et autorisations de l'utilisateur (7–8). Cet objet est alors stocké dans le **SecurityContext** (9), qui représente l'état de sécurité actif pour la requête et les suivantes. Enfin, le filtre renvoie la main au navigateur ou au contrôleur approprié (10).

Le schéma montre que Spring Security suit un processus structuré, modulable et entièrement basé sur l'enchaînement de composants spécialisés, permettant une authentification sécurisée sans disperser la logique dans l'application.

6.4 Concepts Fondamentaux de Spring Security

Spring Security s'appuie sur plusieurs concepts essentiels qui lui permettent d'assurer une sécurité fine et modulaire :

Security Filter Chain

La sécurité fournie par Spring Security repose sur un mécanisme fondamental du monde Java : les **filtres Servlet**. Dans une application classique, chaque requête HTTP envoyée par l'utilisateur transite d'abord par le conteneur web, puis par les Servlets chargés de la traiter.

Les filtres se situent précisément entre ces deux éléments : ils interceptent les requêtes et les réponses, et peuvent effectuer des opérations avant et après le traitement du Servlet. Par exemple, un filtre peut afficher un message avant l'exécution d'un Servlet, laisser passer la requête via filterChain.doFilter(), puis effectuer un traitement final après la réponse. Ce fonctionnement permet d'ajouter proprement des comportements transversaux comme la journalisation, la vérification d'accès ou la transformation des données.

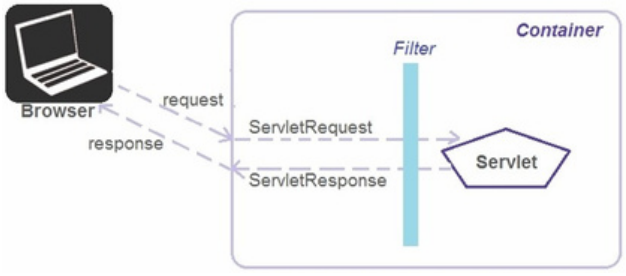


***Figure 6-2**. Servlet Request and Response Flow*

Considérons un filtre simple, nommé FilterA, qui est mappé à toutes les URL (/*) et dont la méthode doFilter() exécute la logique suivante :

1. Afficher un message **"Starting Filter"** (avant filterChain.doFilter()).
2. Laisser passer la requête au reste de l'application (via filterChain.doFilter()).
3. Afficher un message **"Ending Filter"** (après filterChain.doFilter() et le traitement du Servlet).

Le diagramme ci-dessous illustre le flux de traitement pour une requête HTTP GET /home interceptée par ce filtre : Comme le montre l'exemple, lorsqu'une requête arrive, le conteneur l'intercepte et invoque la méthode doFilter() du FilterA. L'exécution du filtre commence, affiche **"Starting Filter"**, puis **filterChain.doFilter()** est appelé. La requête est ensuite traitée par la ressource demandée (home.jsp), qui renvoie la réponse. L'exécution revient au FilterA, qui affiche **"Ending Filter"** avant que la réponse finale ne soit renvoyée au navigateur.

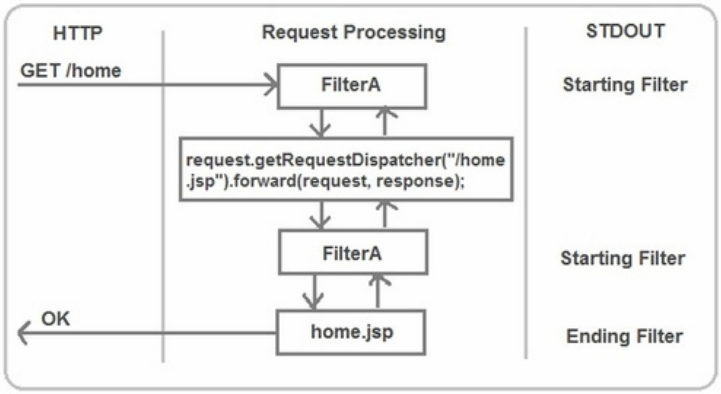


***Figure 6-3**.Java Servlet Filter Chaining with RequestDispatcher*

Spring Security exploite ce principe pour mettre en place sa propre chaîne de sécurité : la **Security Filter Chain** . Celle-ci est composée d'un ensemble de filtres Java implémentant l'interface `javax.servlet.Filter`, exécutés dans un ordre strict défini par le framework.

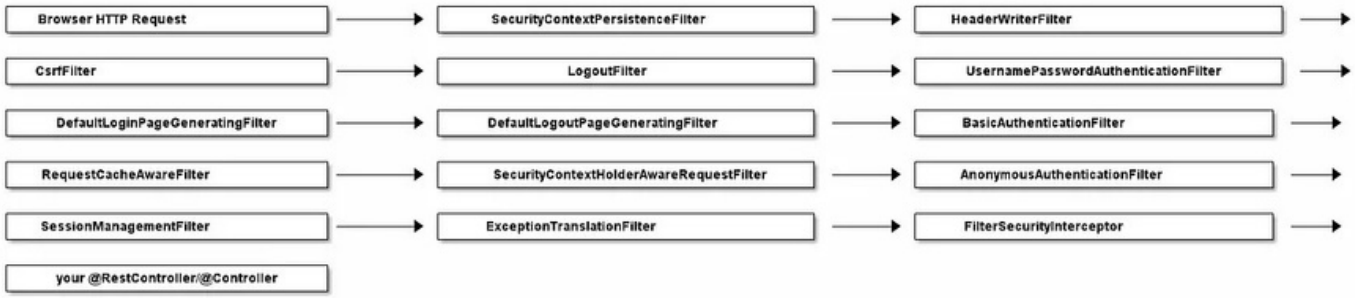


***Figure 6-4**.Order of Spring Security Filters*

Le premier filtre qui décide si une requête doit être sécurisée est le **SecurityContextPersistenceFilter**, chargé de restaurer le contexte de sécurité de l'utilisateur (rôles, informations d'authentification…). Ensuite, selon le type de requête, différents filtres s'activent.

Chaque filtre joue un rôle bien précis dans la sécurisation de l'application. Certains, comme le **UsernamePasswordAuthenticationFilter**, traitent l'authentification via formulaire ; d'autres, comme le **BasicAuthenticationFilter**, gèrent les en-têtes HTTP. D'autres filtres contrôlent encore les sessions, appliquent la protection CSRF, ou vérifient les permissions et les rôles associés à l'utilisateur.

Lorsqu'une requête arrive, elle traverse successivement chaque filtre de la chaîne. Si l'utilisateur n'est pas authentifié ou tente d'accéder à une ressource non autorisée, un des filtres peut bloquer la requête et retourner une réponse adaptée (par exemple, une redirection vers la page de connexion ou une erreur 403). Si tout est conforme, la requête est transmise aux contrôleurs pour exécuter la logique métier. La Security Filter Chain agit donc comme une barrière protectrice : elle examine chaque requête, valide l'identité de l'utilisateur, contrôle ses droits, et applique diverses protections avant que l'application ne commence réellement à traiter la requête.

Avec Spring Boot, cette chaîne est configurée automatiquement, mais elle reste entièrement personnalisable à travers la classe SecurityConfig. Le développeur peut choisir quels chemins laisser publics, quels filtres activer ou désactiver, ou encore modifier la manière dont les utilisateurs sont authentifiés. Grâce à ce système organisé et flexible, Spring Security peut analyser et sécuriser efficacement chaque requête, qu'il s'agisse d'accéder à une page, d'effectuer un envoi de formulaire ou d'interagir avec une API REST.

Authentification

L'Authentification est le processus qui consiste à identifier et vérifier qu'un utilisateur est bien celui qu'il prétend être. Elle combine l'identification (fourniture d'un nom d'utilisateur) et la vérification (fourniture d'un mot de passe ou d'une preuve similaire).

- **Mécanismes de Vérification :** Spring Security est conçu pour prendre en charge une grande variété de méthodes, y compris la vérification des identifiants stockés en mémoire, dans une base de données (JDBC), via des annuaires d'entreprise (LDAP), ou par des systèmes de connexion unique (CAS, OAuth2, JWT).
- **Sécurité des Mots de Passe :** Pour assurer l'intégrité des données, Spring Security impose l'utilisation d'un **PasswordEncoder** qui garantit que les mots de passe sont stockés sous forme hachée (encodée) et jamais en clair.
- **Gestion des Utilisateurs :** Le framework utilise les interfaces clés UserDetails (représentant les données d'un utilisateur, y compris ses rôles) et UserDetailsService (chargée de récupérer ces données depuis n'importe quelle source, telle qu'une base de données ou un service externe) pour charger et manipuler les utilisateurs lors du processus de connexion.
- **Post-Authentification :** Après une vérification réussie, Spring Security prend en charge la gestion de la session utilisateur de manière sécurisée (souvent via des cookies ou des tokens).

PasswordEncoders

La gestion sécurisée des mots de passe est un pilier essentiel de toute application. Spring Security impose l'utilisation d'un *PasswordEncoder* pour éviter le stockage des mots de passe en clair, une pratique extrêmement dangereuse. En effet, si un attaquant accède à la base de données, il pourrait immédiatement utiliser les mots de passe volés pour se connecter aux comptes des utilisateurs ou les tester sur d'autres sites.

Un PasswordEncoder applique un algorithme de hachage, parfois combiné à un sel cryptographique et à un nombre d'itérations. Les méthodes les plus courantes sont :

- **BCrypt** : basé sur l'algorithme Blowfish, il inclut un salt automatique et est résistant aux attaques par force brute même sur du matériel moderne.
- **SCrypt** : spécialement conçu pour être coûteux en mémoire, rendant inefficace l'usage de GPU pour craquer les mots de passe.
- **PBKDF2** : utilise un grand nombre d'itérations pour ralentir les attaques, souvent utilisé dans des environnements industriels.

Spring Security utilise BCrypt comme option par défaut, car il offre le meilleur compromis entre sécurité, performance et compatibilité. De plus, son coût adaptable permet d'augmenter la résistance au fur et à mesure que le matériel devient plus puissant.

OAuth2 et JWT

En plus des mécanismes classiques basés sur les sessions, Spring Security prend également en charge des méthodes d'authentification modernes largement utilisées dans les architectures distribuées et les applications mobiles : OAuth2 et JWT. **OAuth2** est un protocole d'autorisation permettant à une application d'accéder à des ressources au nom d'un utilisateur, sans jamais connaître son mot de passe. Ce mécanisme est aujourd'hui la base des connexions via Google, Facebook ou GitHub, et repose sur l'obtention d'un « access token » délivré par un serveur d'autorisation. Dans les applications plus légères ou les API REST, **JWT** (JSON Web Token) s'impose comme un format de token compact, signé et auto-contenu : il ne nécessite pas de session côté serveur, car toutes les informations (identité, rôles, date d'expiration) sont inscrites et sécurisées directement dans le token. Spring Security intègre nativement ces deux approches, permettant ainsi de construire des systèmes d'authentification adaptés aux microservices, aux applications mobiles ou aux API stateless. Ce modèle offre des performances élevées et une excellente scalabilité, tout en restant compatible avec les bonnes pratiques modernes de sécurité.

Autorisation

L'Autorisation est le processus qui survient après l'authentification et qui détermine les actions et les ressources auxquelles l'utilisateur authentifié est autorisé à accéder. C'est le mécanisme de contrôle d'accès.

L'objectif est d'assurer que l'utilisateur n'a accès qu'aux ressources (pages web, API, méthodes de service) pour lesquelles il possède les droits.

Exemple concret : Dans une application de paie RH, l'Autorisation permet de s'assurer que seuls les employés ayant le rôle HR peuvent accéder à la section de l'application gérant les salaires, tandis que la consultation des bulletins de paie est autorisée à tous les employés (rôle EMPLOYEE).

- **Modèles d'Accès :** Spring Security implémente principalement le modèle **Role-Based Access Control (RBAC)**, où les permissions sont regroupées et attribuées à des Rôles définis (ex: ADMIN, USER, HR). Cependant, d'autres modèles peuvent être mis en œuvre, comme la vérification de permissions spécifiques.

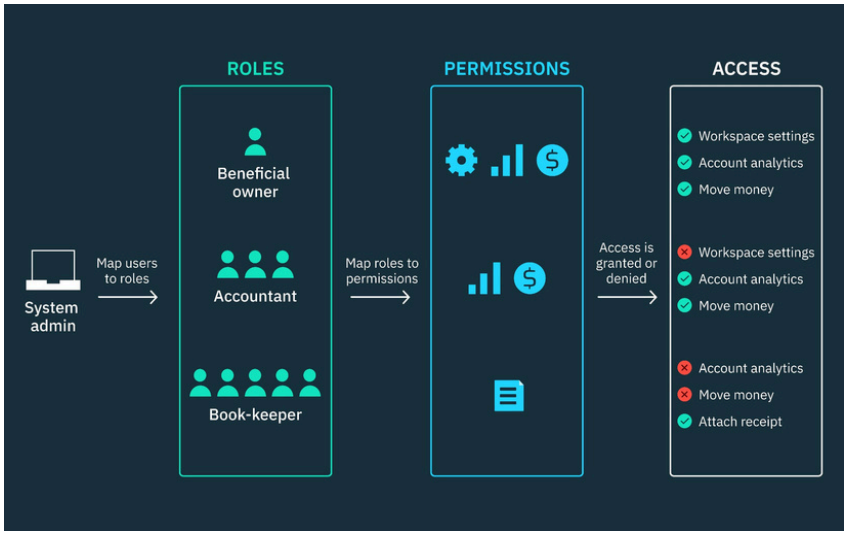


***Figure 6-5**.RBAC*

- **Contrôle Granulaire :** L'Autorisation peut être appliquée à différents niveaux de l'application :
  - **Au niveau des URL/endpoints** (par exemple, autoriser l'accès à /admin/** uniquement aux utilisateurs avec le rôle ADMIN).
  - **Auniveaudesméthodes** (par exemple, empêcher l'exécution d'une méthode de service critique si l'utilisateur n'a pas la permission adéquate).

Sécurisation des URLs et des Méthodes

La sécurisation des ressources dans Spring Security s'effectue principalement à deux niveaux complémentaires : au niveau des URLs et au niveau des méthodes. D'abord, la configuration **HttpSecurity** permet de contrôler l'accès aux différentes routes HTTP de l'application. Grâce à cette configuration, il est possible de définir quelles URLs sont publiques, lesquelles nécessitent une authentification, et lesquelles sont réservées uniquement à certains rôles d'utilisateurs. Par exemple, on peut autoriser librement l'accès aux pages de connexion ou d'inscription, tout en protégeant les pages

d’administration. Cette approche garantit qu’une requête envoyée à l’application ne pourra atteindre un contrôleur sensible que si l’utilisateur possède les permissions requises.

En complément, Spring Security propose la sécurisation au niveau du code grâce aux annotations telles que *@Secured*, *@PreAuthorize* et *@PostAuthorize*. Ces annotations permettent de contrôler l’accès directement sur les méthodes Java, offrant une sécurité plus fine et liée à la logique métier.

Premièrement, l'annotation *@Secured* est la plus simple : elle est placée sur une méthode pour indiquer quels rôles spécifiques (ex: ROLE_ADMIN) sont autorisés à l'exécuter. Si l'utilisateur n'a pas l'un de ces rôles, l'exécution est bloquée. Deuxièmement, *@PreAuthorize* est l'outil le plus flexible car elle utilise le langage d'expression **SpEL** (Spring Expression Language) pour évaluer des conditions complexes avant l'appel de la méthode. Cela permet de vérifier non seulement le rôle de l'utilisateur, mais aussi des conditions basées sur les données passées à la méthode (par exemple, s'assurer que l'utilisateur n'essaie de modifier que son propre compte). Enfin, *@PostAuthorize* est une annotation rare qui évalue une expression SpEL après l'exécution de la méthode, permettant de vérifier la validité ou l'accessibilité du résultat retourné (par exemple, autoriser ou non la lecture d'un objet si l'utilisateur est son propriétaire). L'utilisation de ces annotations assure un contrôle d'accès précis, directement intégré à la logique métier de l'application.

L’association de la sécurisation des URLs via **HttpSecurity** et de la sécurisation méthodologique via ces annotations assure une protection complète, cohérente et flexible de l’application.

Protection contre les attaques courantes

Au-delà des mécanismes d'authentification et d'autorisation, Spring Security renforce la posture de sécurité d'une application en intégrant nativement des défenses automatiques contre les vulnérabilités web largement reconnues, souvent via sa propre chaîne de filtres. Ceci est un avantage essentiel : ces protections sont activées par défaut et ne demandent aucune implémentation manuelle de la part du développeur, ce qui assure une base de sécurité robuste dès le démarrage de l'application.

Une protection fondamentale est la gestion des **jetons CSRF (Cross-Site Request Forgery)**. Pour contrer cette attaque qui force l'utilisateur à exécuter des actions non désirées à son insu, Spring Security fait en sorte que chaque formulaire ou requête importante reçoive un code secret unique (le jeton). Le serveur vérifie que ce code secret est bien envoyé avec la requête. Si la requête provient d'un site tiers malveillant, elle sera bloquée faute de ce jeton valide. Ce mécanisme est comparable à l'utilisation d'un mot de passe unique à usage unique pour chaque transaction importante.

Le framework met également en œuvre la **prévention de la fixation de session** (*Session Fixation*). Cette défense empêche un attaquant d'imposer un identifiant de session connu à un utilisateur avant que celui-ci ne se connecte. Spring Security résout ce problème par la **rotation de la session** : lorsqu'un utilisateur se connecte avec succès, le framework invalide immédiatement l'ancienne session temporaire et en crée une nouvelle avec un identifiant complètement différent et secret. Cela est similaire à changer la serrure de sa chambre d'hôtel après avoir prouvé son identité.

Enfin, le composant de sécurité HTTP assure la sécurisation des en-têtes de réponse. Ces petites informations envoyées au navigateur sont configurées pour se défendre contre d'autres attaques :

- Contre le **XSS (Cross-Site Scripting)**, l'en-tête **Content-Security-Policy (CSP)** agit comme un ensemble de règles strictes, ordonnant au navigateur de n'autoriser le chargement de scripts que depuis des sources fiables (le domaine de l'application).
- Contre le **Clickjacking** (qui piège l'utilisateur en plaçant la page dans un cadre transparent), l'en-tête **X-Frame-Options** (ou des directives CSP) interdit l'affichage de la page dans une *iframe* sur un site étranger.
- Pour forcer le **HTTPS**, l'en-tête **Strict-Transport-Security (HSTS)** demande au navigateur de se souvenir de toujours n'utiliser que le protocole sécurisé pour toutes les communications futures avec le site.

L'ensemble de ces mesures renforce considérablement l'intégrité et la confidentialité des échanges de données.

Gestion des sessions

Spring Security assure également une gestion avancée des sessions afin de protéger l'application contre plusieurs attaques liées à l'usurpation ou au détournement des sessions utilisateur. D'abord, le framework inclut une protection contre la **session fixation**, déjà décrite précédemment, en régénérant systématiquement l'ID de session après authentification. Ensuite, il prend en charge l'**expiration automatique des sessions**, permettant de définir une durée maximale d'inactivité. Lorsque la session expire, l'utilisateur doit se reconnecter, renforçant ainsi la sécurité des zones sensibles.

Un autre aspect important est la **gestion de la concurrence des sessions** (*session concurrency control*). Spring Security peut empêcher un utilisateur d'ouvrir plusieurs sessions simultanées, ou limiter leur nombre (par exemple, une seule session active par utilisateur). Cette fonctionnalité est essentielle dans des contextes où le partage de comptes est interdit ou lorsque l'application manipule des données sensibles.

Grâce à ces mécanismes, la gestion des sessions dans Spring Security apporte une couche supplémentaire de protection tout en permettant de configurer des comportements adaptés selon les besoins de l'application.

6.5 Intégration avec Spring Boot

L'un des principaux avantages de Spring Boot est sa capacité à simplifier la configuration de Spring Security grâce à son mécanisme d'auto-configuration. En effet, l'ajout de la dépendance **spring-boot-starter-security** suffit pour activer immédiatement une sécurité de base dans l'application. Dès le premier démarrage, Spring Boot génère automatiquement un utilisateur par défaut, généralement nommé *user*, accompagné d'un mot de passe temporaire affiché dans la console. Cette configuration préétablie protège également l'ensemble des endpoints de l'application : toutes les URLs nécessitent une authentification, et un formulaire de connexion standard est mis à disposition sans qu'aucun code supplémentaire ne soit nécessaire.

Une fois cette configuration automatique en place, le développeur peut personnaliser la sécurité selon les besoins du projet. Cela se fait généralement en créant une classe de configuration dédiée (SecurityConfig), où il devient possible de définir les règles d'accès aux ressources, de créer des utilisateurs personnalisés, d'intégrer une base de données pour la gestion des comptes, ou encore de modifier le comportement du formulaire de connexion. Grâce à cette approche, Spring Boot combine simplicité et flexibilité, permettant aux débutants de démarrer rapidement tout en offrant aux développeurs avancés un haut niveau de contrôle sur la sécurité de leur application.

La création de la classe `SecurityConfig` est annotée avec *@Configuration* et *@EnableWebSecurity*, dans laquelle on redéfinit le bean SecurityFilterChain.
Cette classe permet de définir quelles pages sont publiques, quelles pages nécessitent une authentification, quels rôles sont autorisés à accéder à certains endpoints, ou encore quel formulaire de connexion utiliser. Voici un exemple simple de configuration :

```
@Configuration
@EnableWebSecurity
publicclassSecurityConfig {
    @Bean
    publicSecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception{
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/login", "/register").permitAll()
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
            .formLogin(form -> form
                .loginPage("/login")
                .defaultSuccessUrl("/home", true)
            )
            .logout(logout -> logout
                .logoutUrl("/logout")
                .logoutSuccessUrl("/login?logout")
            );
        returnhttp.build();
    }
    @Bean
    publicPasswordEncoder passwordEncoder() {
        returnnew BCryptPasswordEncoder();
    }
}
```

***Code 6-1**.SecurityConfig*

Cette configuration illustre les principaux mécanismes de Spring Security dans une application Spring Boot, comment permettre l'accès libre à certaines pages, protéger les pages administratives, personnaliser

le formulaire d'authentification, comment Spring Security centralise la gestion des rôles, des pages protégées et du processus d'authentification grâce à une approche simple et modulable.

Le bean SecurityFilterChain permet de définir les règles d'accès aux différentes URL : dans cet exemple, les pages /login et /register sont publiques, tandis que toutes les autres requêtes exigent une authentification. De plus, les routes commençant par /admin/** sont réservées aux utilisateurs possédant le rôle ADMIN. La section formLogin() permet de personnaliser le formulaire d'authentification, en indiquant la page de connexion et la destination après un login réussi. De même, logout() configure l'URL de déconnexion ainsi que la page affichée après la sortie. Enfin, le bean PasswordEncoder utilise l'algorithme BCrypt pour hacher les mots de passe, garantissant un stockage sécurisé.

6.6 Bonnes pratiques

Pour garantir un niveau de sécurité élevé dans une application Spring Boot, il est essentiel d'adopter un ensemble de bonnes pratiques complémentaires aux mécanismes fournis par Spring Security. Tout d'abord, les mots de passe ne doivent jamais être stockés en clair : ils doivent toujours être hachés à l'aide d'un algorithme robuste comme BCrypt. L'utilisation systématique du protocole HTTPS est également indispensable afin de protéger les données en transit contre les interceptions et les attaques de type « Man-in-the-Middle ». Il est recommandé d'appliquer le principe du moindre privilège en attribuant aux utilisateurs uniquement les rôles nécessaires à leurs actions, tout en évitant de coder les rôles ou permissions directement dans le code source. Pour les API REST, il est important de désactiver la protection CSRF, tout en adoptant une authentification stateless basée sur des tokens (JWT). Enfin, la journalisation des tentatives d'accès, la surveillance des anomalies, ainsi que la mise en place d'une expiration de session et de limites sur les connexions simultanées permettent de renforcer la sécurité globale de l'application.

Chapitre 7. Tests et Qualité du Code

Les tests jouent un rôle essentiel dans le développement logiciel, car ils permettent de vérifier la justesse et la fiabilité des fonctionnalités. Dans le monde Java, JUnit et TestNG comptent parmi les frameworks de test les plus utilisés. Une pratique répandue est le Test Driven Development (TDD), qui consiste à écrire d'abord les tests puis à développer uniquement le code nécessaire pour les faire réussir. On distingue généralement plusieurs types de tests : les tests unitaires, qui évaluent un composant isolé, et les tests d'intégration, qui valident le comportement d'un ensemble de composants. Lors de tests d'intégration, il est souvent nécessaire de simuler des dépendances externes comme des appels à des services web tiers ou des interactions avec la base de données ; pour cela, des bibliothèques comme Mockito, PowerMock ou jMock permettent de créer des objets simulés. L'injection de dépendances, principe fondamental de Spring, facilite grandement l'écriture de code testable puisqu'elle permet d'injecter des implémentations factices pendant les tests et des implémentations réelles en production. Spring, en tant que conteneur IoC, propose d'ailleurs une excellente prise en charge de différents scénarios de test. Dans le contexte de Spring Boot, il existe des outils spécialement conçus pour tester des parties précises de l'application : par exemple, @WebMvcTest pour tester les contrôleurs web, @DataJpaTest pour tester les repositories JPA, ou encore @JdbcTest pour vérifier les interactions via JDBC. Ce chapitre montre ainsi comment tester efficacement les composants d'une application Spring Boot en isolant chaque couche selon son rôle spécifique.

7.1 Tests des Applications Spring Boot

L'une des principales raisons de la popularité du framework Spring est son excellent support pour les tests. Spring fournit SpringRunner, un exécuteur JUnit personnalisé qui permet de charger automatiquement le Spring ApplicationContext grâce à l'annotation @ContextConfiguration(classes=AppConfig.class).
Un test Spring typique ressemble à ce qui suit :

Listing 7-1. *Test JUnit Spring typique*

```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes=AppConfig.class)
publicclass UserServiceTests
{
    @Autowired
    UserService userService;

    @Test
    public void should_load_all_users()
    {
        List<User> users = userService.getAllUsers();
        assertNotNull(users);
        assertEquals(10, users.size());
    }
}
```

Une application Spring Boot n'est finalement rien d'autre qu'une application Spring, donc vous pouvez utiliser toutes les fonctionnalités de test de Spring dans une application Spring Boot. Cependant, certaines fonctionnalités propres à Spring Boot comme le chargement automatique des propriétés externes ou la configuration du logging ne sont disponibles que si l'ApplicationContext est créé avec la classe SpringApplication, utilisée dans la classe d'entrée de l'application :

```
@SpringBootApplication
publicclass SpringbootTestingDemoApplication
{
    public static void main(String[] args)
    {
        SpringApplication.run(SpringbootTestingDemoApplication.class, args);
    }
}
```

Spring Boot propose alors l'annotation @SpringBootTest , qui utilise SpringApplication en interne pourcharger l' ApplicationContext, garantissant ainsi que toutes les fonctionnalités Spring Boot restent accessibles pendant les tests.

Listing 7-2. *Test JUnit typique avec Spring Boot*

```
@RunWith(SpringRunner.class)
@SpringBootTest
publicclass SpringbootTestingDemoApplicationTests
{
    @Autowired
    UserService userService;

    @Test
    public void should_load_all_users()
    {
        ...
        ...
    }
}
```

Avec @SpringBootTest, il est possible de fournir des classes de configuration Spring, des fichiers XML ou d'autres types de configuration, mais dans une application Spring Boot, on utilise généralement la classe d'entrée principale.

Le starter spring-boot-starter-test inclut JUnit, Spring Test et Spring Boot Test, ainsi que plusieurs bibliothèques essentielles pour les tests, notamment :

- **Mockito** : framework de mock Java
- **Hamcrest** : bibliothèque de matchers pour les assertions
- **AssertJ** : bibliothèque d'assertions fluides

- **JSONassert** : assertions sur des données JSON
- **JsonPath** : équivalent XPath pour JSON

7.2 Tests avec des Implémentations Mock

Lors des tests unitaires, il est souvent nécessaire de simuler les appels à des services externes, comme les accès à la base de données ou les appels à des web services. Pour cela, deux approches sont possibles : créer manuellement des implémentations mock utilisées uniquement dans les tests, ou utiliser une bibliothèque de mocking pour générer automatiquement des objets simulés. Créer des mocks manuellement consiste à écrire soi-même des classes qui imitent le comportement des dépendances réelles. Cette approche fonctionne, mais elle devient rapidement lourde et fastidieuse dès que les cas à couvrir se multiplient.

Pour éviter cette complexité, il est beaucoup plus pratique d'utiliser une bibliothèque de mocking. L'une des plus populaires en Java est **Mockito**, qui s'intègre parfaitement avec JUnit. Mockito permet de créer des objets mock sans devoir écrire de classes supplémentaires, et de définir précisément les comportements attendus.

Par exemple, si votre service appelle un web service externe et que vous devez tester la logique de *retry* en cas d'erreur, il serait difficile de provoquer réellement une panne de communication. Avec Mockito, vous pouvez facilement simuler une exception, forcer l'échec de l'appel, et vérifier que votre code tente bien trois essais avant d'abandonner. De même, supposez que vous importez des données utilisateurs depuis un service tiers comme dans l'exemple du Listing 7-3.

Listing 7-3. Users Importer.java

```
@Service
public class UsersImporter
{
public List<User> importUsers() throws UserImportServiceCommunicationFailure
{
List<User> users = new ArrayList<>();
//get users by invoking some web service
//if any exception occurs throw UserImportServiceCommunicationFailure
//dummy data users.add(new User());
users.add(new User());
users.add(new User());
return users;
}
}
```

Vous pouvez utiliser @Mock pour créer un objet mock et @InjectMocks pour injecter automatiquement ces mocks dans les dépendances de la classe testée.

Vous pouvez également utiliser @RunWith(MockitoJUnitRunner.class) pour initialiser les objets mock, ou déclencher cette initialisation manuellement en appelant MockitoAnnotations.initMocks(this) dans une méthode @Before de JUnit.

Listing 7-4. *Tests utilisant des objets mock Mockito*

```
import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.BDDMockito.*;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;
import com.apress.demo.exceptions.UserImportServiceCommunicationFailure;
import com.apress.demo.model.UsersImportResponse;
 @RunWith(MockitoJUnitRunner.class) public class UsersImportServiceMockitoTest
{
@Mock
private UsersImporter usersImporter;
@InjectMocks
private UsersImportService usersImportService;
@Test
public void should_retry_3times_when_UserImportServiceCommunicationFailure_occured()
{
given(usersImporter.importUsers()).willThrow(new      UserImportServiceCommunication
Failure());
UsersImportResponse response = usersImportService.importUsers();
 assertThat(response.getRetryCount()).isEqualTo(3);
 assertThat(response.getStatus()).isEqualTo("FAILURE");

}
}
```

Ici, vous simulez une condition d'échec lors de l'importation des utilisateurs via le service web grâce à l'instruction suivante : given(usersImporter.importUsers()).willThrow(new UserImportServiceCommunicationFailure());

Ainsi, lorsque vous appelez userService.importUsers() et que l'objet mock usersImporter déclenche l'exception UserImportServiceCommunicationFailure, la méthode sera réessayée trois fois avant d'abandonner.

Spring Boot fournit également l'annotation @MockBean, qui permet de définir un nouveau mock Mockito en tant que bean Spring, ou de remplacer un bean Spring existant par un mock, puis de l'injecter automatiquement dans les composants qui en dépendent.

Les mock beans sont automatiquement réinitialisés après chaque méthode de test.

Voir le Listing 7-5.

Listing 7-5. *Tests utilisant le mock* @MockBean *de Spring Boot*

```
import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.BDDMockito.*;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.test.context.junit4.SpringRunner;
import com.apress.demo.exceptions.UserImportServiceCommunicationFailure;
import com.apress.demo.model.UsersImportResponse;
@RunWith(SpringRunner.class)
@SpringBootTest
public class UsersImportServiceMockitoTest
{
@MockBean
private UsersImporter usersImporter;
@Autowired
private UsersImportService usersImportService;
@Test
public void should_retry_3times_when_UserImportServiceCommunicationFailure_occured()
{
given(usersImporter.importUsers()).willThrow(new UserImportServiceCommunication
Failure());
UsersImportResponse response = usersImportService.importUsers();
 assertThat(response.getRetryCount()).isEqualTo(3);
 assertThat(response.getStatus()).isEqualTo("FAILURE");
}
}
```

Ici,SpringBootvacréer un objet mock pour *UsersImporter* et l' injecter dans le bean *UsersImportService* .

7.3 Tester des tranches de l'application à l'aide des annotations @*Test

Lors du test des différents composants de l'application, vous pouvez souhaiter charger uniquement un sous-ensemble de beans du Spring ApplicationContext, ceux qui sont liés au *sujet testé (SUT)*. Par exemple, lorsque vous testez un contrôleur Spring MVC, vous pouvez vouloir charger uniquement les composants de la couche MVC (couche présentation) et fournir des beans simulés (mock) de la couche service en tant que dépendances. Spring Boot fournit des annotations comme @WebMvcTest, @DataJpaTest, @DataMongoTest, @JdbcTest et @JsonTest pour tester des tranches spécifiques de l'application.

Tester les Contrôleurs Spring MVC Avec @WebMvcTest

Spring Boot fournit l'annotation @WebMvcTest, qui va auto-configurer les composants de l'infrastructure Spring MVC et charger uniquement les éléments suivants : @Controller, @ControllerAdvice, @JsonComponent, Filter, WebMvcConfigurer, et HandlerMethodArgumentResolver.

Les autres beans Spring (annotés avec @Component, @Service, @Repository, etc.) ne seront pas scannés lorsque vous utilisez cette annotation.

Vous allez maintenant voir comment créer un contrôleur qui ajoute des données au modèle et rend une vue Thymeleaf. Voir Listing 15-14.

Listing 7-6. TodoController.java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import com.apress.demo.repositories.TodoRepository;
 @Controller
public class TodoController
{
@Autowired
TodoRepository todoRepository;
@GetMapping("/todolist")
public String showTodos(Model model)
{
model.addAttribute("todos", todoRepository.findAll());
return "todos";
}
}
```

Listing *7-7* montre comment écrire un test pour TodoController en utilisant @WebMvcTest.

Listing 7-7. *Tester un contrôleur Spring MVC avec MockMvc*

```
import static org.mockito.BDDMockito.*;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;
import static org.hamcrest.Matchers.*; import java.util.Arrays; import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
```

```
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.test.web.servlet.MockMvc;
import com.apress.demo.entities.Todo;
import com.apress.demo.repositories.TodoRepository;
 @RunWith(SpringRunner.class)
@WebMvcTest(controllers= TodoController.class)
public class TodoControllerTests
{
@Autowired
private MockMvc mvc;
@MockBean
private TodoRepository todoRepository;
@Test
public void testShowAllTodos() throws Exception
{
Todo todo1 = new Todo(1, "Todo1",false);
Todo todo2 = new Todo(2, "Todo2",true);
 given(this.todoRepository.findAll()).willReturn(Arrays.asList(todo1, todo2));
 this.mvc.perform(get("/todolist")
.accept(MediaType.TEXT_HTML))
.andExpect(status().isOk())
.andExpect(view().name("todos"))
.andExpect(model().attribute("todos", hasSize(2)))
;
verify(todoRepository, times(1)).findAll();
}
}
```

Vous avez annoté le test avec @WebMvcTest(controllers = TodoController.class) en spécifiant explicitement quel contrôleur vous testez. Comme @WebMvcTest ne charge pas les autres beans Spring classiques et que TodoController dépend de TodoRepository, vous avez fourni un bean mock en utilisant l'annotation @MockBean.L'annotation @WebMvcTest configure automatiquement MockMvc, qui peut être utilisé pour tester les contrôleurs sans démarrer un véritable conteneur servlet.

Dans cette méthode de test, vous définissez le comportement attendu de todoRepository.findAll(), afin qu'il renvoie une liste de deux objets Todo. Ensuite, vous effectuez une requête GET vers "/todolist" et vous vérifiez plusieurs éléments dans la réponse.

Tester les Composants de la Couche de Persistance Avec `@DataJpaTest` et `@JdbcTest`

Vous pouvez souhaiter tester les composants de la couche de persistance de votre application, ce qui ne nécessite pas le chargement de nombreux autres composants comme les contrôleurs, la configuration de sécurité, etc. Spring Boot fournit les annotations @DataJpaTest et @JdbcTest pour tester les beans Spring qui interagissent avec des bases de données relationnelles.L'annotation @DataJpaTest permet de tester les composants de la couche persistance en auto-configurant des bases de données embarquées en mémoire et en scannant les classes annotées @Entity ainsi que les repositories Spring Data JPA. Cette annotation ne charge pas les autres beans Spring (@Component, @Controller, @Service, etc.) dans le ApplicationContext.

Vous allez maintenant voir comment tester les repositories Spring Data JPA dans une application Spring Boot. Pour cela, créez un projet Maven Spring Boot avec les starters Data-JPA et Test.

Ensuite,vous créez une entité JPA appelée User, représentant les utilisateurs dans la base de données, ainsi qu'un repository Spring Data JPA appelé UserRepository pour gérer les opérations CRUD sur cette entité. L'entité contient des champs tels que l'identifiant, le nom, l'email et le mot de passe, avec des contraintes appropriées (unicité, non nullité, etc.). Pour initialiser la table USERS de la base de données, vous pouvez ajouter des données statiques via un fichier data.sql situé dans src/main/resources, ce qui vous permet de disposer immédiatement d'un jeu de données de test lors de l'exécution de l'application.

Vous pouvez maintenant tester UserRepository en utilisant l'annotation @DataJpaTest, comme illustré dans Listing 7-8 :

Listing 7-8. *Tester les Spring Data JPA Repositories avec* @DataJpaTest

```
@RunWith(SpringRunner.class)
@DataJpaTest
publicclass UserRepositoryTests
{
    @Autowired
    private UserRepository userRepository;

    @Test
    public void testFindByEmail() {
        User user = userRepository.findByEmail("admin@gmail.com");
        assertNotNull(user);
        assertEquals(1, user.getId());
        assertEquals("admin", user.getName());
    }
```

```
}
```

Lorsque vous exécutez UserRepositoryTests, Spring Boot auto-configurera automatiquement une base de données embarquée en mémoire H2 (si le driver H2 est présent dans le classpath) et exécutera les tests. Si vous souhaitez effectuer les tests sur la base de données réelle configurée, vous pouvez annoter le test avec@AutoConfigureTestDatabase(replace=Replace.NONE).

Cela utilisera la DataSource enregistrée au lieu d'une datasource en mémoire. Vous pouvez également utiliser Replace.AUTO_CONFIGURED pour remplacer la DataSource auto-configurée, ou Replace.ANY (valeur par défaut) pour remplacer toute datasource bean auto-configurée ou définie explicitement.Les tests avec @DataJpaTest sont transactionnels et les modifications sont annulées à la fin de chaque test par défaut.Vouspouvez désactiver ce comportement de rollback pour un test spécifique ou pour une classe de test entière en utilisant @Transactional(propagation = Propagation.NOT_SUPPORTED).

Demanière similaire à l'annotation @DataJpaTest, vous pouvez utiliser @JdbcTest pour tester des méthodes liées au JDBC en utilisant JdbcTemplate. L'annotation @JdbcTest auto-configure également des bases de données embarquées en mémoire et exécute les tests de manière transactionnelle.

Vous allez maintenant créer un JdbcUserRepository pour effectuer des opérations sur la base de données en utilisant JdbcTemplate, comme montré dans Listing 7-9 :

Listing 7-9. *JdbcUserRepository.java*

```
publicclass JdbcUserRepository
{
    private JdbcTemplate jdbcTemplate;

    public JdbcUserRepository(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public List<User> findAll() {
        ....
        ....
    }
}
```

Listing 7-10 montre comment tester les méthodes de JdbcUserRepository en utilisant @JdbcTest :

Listing 7-10*. Tester les opérations JDBC avec* @JdbcTest

```
@RunWith(SpringRunner.class)
@JdbcTest
```

```
publicclass JdbcUserRepositoryTests
{
    @Autowired
    private JdbcTemplate jdbcTemplate;

    private JdbcUserRepository userRepository;

    @Before
    public void init()
    {
        userRepository       =       new       JdbcUserRepository(jdbcTemplate);
        jdbcTemplate.execute("create table people(id int, name varchar(100))");
        jdbcTemplate.execute("insert into people(id, name) values(1, 'John')");
        jdbcTemplate.execute("insert into people(id, name) values(2, 'Remo')");
        jdbcTemplate.execute("insert into people(id, name) values(3, 'Dale')");
    }

    @Test
    public void testFindAllUsers() throws Exception
    {
        List<User> users = userRepository.findAll();
        assertThat(users.size()).isEqualTo(3);
    }
}
```

Comme @JdbcTest ne charge aucun bean Spring régulier annoté @Component, cet exemple crée manuellement l'instance de `JdbcUserRepository` en utilisant le `bean JdbcTemplate` auto-configuré.

De manière similaire à `@DataJpaTest` et `@JdbcTest`, Spring Boot fournit d'autres annotations pour tester des parties spécifiques de l'application, comme @DataMongoTest, @DataNeo4jTest, `@JooqTest, @JsonTest et @DataLdapTest.`

Annexes

AnnexeA : Annotations Spring les plus courantes

Annotation	Description	Utilisation
@Controller	Indique qu'une classe est un contrôleur MVC	Couche Web (retourne une vue)
@RestController	Combinaison de @Controller et @ResponseBody	API REST (retourne JSON/XML)
@Service	Indique une classe de logique métier	Couche Service Couche d'accès BD,
@Repository	Indique une classe d'accès aux données (DAO)	gère les exceptions
@Component	Composant générique Spring	Classe à injecter (si aucune autre annotation spécialisée ne correspond)
@Autowired	Injection automatique de dépendance	Constructeur, attribut, setter
@Qualifier("name")	Précise quel bean utiliser quand il y en a plusieurs du même type	Injection ciblée
@Value("${prop}")	Injecte une valeur depuis le fichier `application.properties`	Variables configurables
@Bean	Déclare manuellement un bean dans une classe @Configuration	Configurations avancées
@SpringBootApplication	Active l'auto-configuration, le scan de composants, la config Spring Boot	Classe principale
@EnableAutoConfiguration	Laisse Spring Boot configurer les beans automatiquement	Inclus dans @SpringBootApplication
@ComponentScan	Indique à Spring quelles packages scanner	Inclus dans SpringBootApplication
@Configuration	Indique que la classe contient des beans Java-based	Fichiers de configuration
@GetMapping("/path")	Mappe une requête HTTP GET	Contrôleur REST
@PostMapping("/path")	Mappe une requête HTTP POST	Formulaires, création
@PutMapping("/path")	Mappe une requête HTTP PUT	Mise à jour
@DeleteMapping("/path")	Mappe une requête HTTP DELETE	Suppression

@RequestParam	Paramètres dans l'URL ?name=value	Requêtes GET
@PathVariable	Paramètre dans le chemin /users/{id}	REST API
@RequestBody	Récupère le JSON envoyé par le client	POST, PUT
@ResponseBody	Indique que la méthode retourne directement des données	Automatique avec @RestController
@Entity	Représente une table dans la BD	Classe modèle
@Table(name="...")	Spécifie le nom de la table	Mapping BD
@Id	Clé primaire	Identifiant
@GeneratedValue	Auto-incrémentation	PK
@Column	Propriétés d'une colonne	Taille, nullability, nom
@OneToMany / @ManyToOne / @ManyToMany	Relations entre tables	Mapping relationnel
@JoinColumn	Colonne de jointure	Relations JPA
@Valid	Valide un objet reçu	Contrôleurs
@NotNull	Champ obligatoire	Validation form
@NotBlank	Texte non vide	String
@Email	Vérification email	Formulaires
@Min, @Max	Contraintes numériques	Champs int/float
@Size(min, max)	Taille minimale/maximale	String ou listes
@EnableWebSecurity	Active Spring Security	
@Configuration	Définit une classe de sécurité	
@PreAuthorize("hasRole('ADMIN')")	Autorisation au niveau des méthodes	
@Secured("ROLE_ADMIN")	Autorisation basique	
@Transactional	Indique que la méthode (ou la classe) utilise une transaction	Couche service / DAO
@EnableTransactionManagement	Active la gestion des transactions	Fichier de configuration

Annexe B : Bibliographie et références

Spring Start Here: Learn what You Need and Learn it Well par *Laurentiu Spilca*

Spring developing java applications for the enterprise par *Ravi Kant Soni, Amuthan Ganeshan Rajesh RV*

Java Spring Boot From Beginner to Pro A Comprehensive Guide to Modern Java Development 3 Books in 1 par *Darren Green*

Beginning Spring Boot 2 Applications and Microservices with the Spring Framework par *K. Siva Prasad Reddy*

https://www.geeksforgeeks.org/advance-java/spring-boot/

https://www.geeksforgeeks.org/springboot/spring-boot-rest-example/

https://docs.spring.io/spring-framework/reference/

https://docs.spring.io/spring-framework/reference/web/webmvc.html

https://gayerie.dev/docs/spring/spring/spring_mvc_intro.html

https://cloud.tencent.com/developer/article/2571626

https://openclassrooms.com/fr/courses/6900101-creez-une-application-java-avec-spring-boot

https://www.sfeir.dev/back/comprendre-les-annotations-dans-spring-boot/