Spring Boot

Introducion

Spring Framework est un framework Java open-source puissant et populaire, principalement utilisé pour développer des applications d'entreprise robustes et performantes. Il simplifie le développement en fournissant une architecture modulaire, permettant aux développeurs de choisir les modules spécifiques dont ils ont besoin, sans être obligés d’utiliser tout le framework.

Caractéristiques principales de Spring

1. **Inversion de Contrôle (IoC)** : Permet de déléguer la gestion des objets (leur création, leur configuration, et leur cycle de vie) au conteneur Spring via l’injection de dépendances.
2. **Programmation Orientée Aspect (AOP)** : Facilite la séparation des préoccupations, comme la gestion des transactions, la sécurité, le logging, en les centralisant.
3. **Data Access Layer** : Simplifie l’interaction avec les bases de données, en intégrant et en simplifiant l’utilisation de JPA, Hibernate, et JDBC.
4. **Gestion des Transactions** : Permet une gestion déclarative des transactions, facilitant l’intégrité des opérations et leur gestion via des annotations.
5. **MVC (Modèle-Vue-Contrôleur)** : Fournit une structure MVC pour le développement d’applications web, en séparant les responsabilités entre modèle, vue et contrôleur.

Modularité de Spring

Spring est conçu de manière modulaire, avec des projets spécialisés et des sous-modules pouvant être utilisés indépendamment ou ensemble selon les besoins du projet. Voici un aperçu des principaux modules de Spring :

1. **Spring Core** : Cœur du framework, il inclut le conteneur IoC et l'injection de dépendances.
2. **Spring AOP** : Gère les aspects transversaux tels que le logging, la sécurité, et la gestion des transactions. Cela permet de séparer ces préoccupations sans encombrer le code métier principal.
3. **Spring Data** : Simplifie l'accès aux bases de données en fournissant des abstractions pour JPA, MongoDB, Cassandra, et d'autres technologies de persistance.
4. **Spring MVC** : Permet de construire des applications web en suivant le modèle MVC. Il est très utilisé pour les applications web en entreprise.
5. **Spring Security** : Assure la sécurité des applications, incluant l'authentification et l'autorisation, en prenant en charge divers protocoles de sécurité (comme OAuth et JWT).

6. **Spring Batch** : Conçu pour des applications de traitement par lots, notamment pour les tâches de traitement de données répétitives et planifiées.
7. **Spring Boot** (L'objet de notre étude) : Permet de créer des applications autonomes, prêtes pour la production, et qui nécessitent moins de configuration, grâce à des conventions de configuration.
8. **Spring Cloud** : Fournit des outils pour développer des systèmes distribués et microservices en intégrant les services d'infrastructure nécessaires (comme la découverte de services, la gestion des configurations et le monitoring).
9. **Spring WebFlux** : Un module de programmation réactive pour des applications nécessitant une gestion des flux asynchrones et non-bloquants.

Avantages de la Modularité de Spring

- **Flexibilité** : Les développeurs peuvent choisir les modules selon les exigences du projet, ce qui permet de réduire la complexité et les dépendances inutiles.
- **Réutilisabilité** : Les modules de Spring sont bien découplés, donc ils peuvent être utilisés dans des projets sans dépendance à d'autres parties du framework.
- **Performance et Scalabilité** : Certains modules, comme Spring WebFlux, sont spécifiquement optimisés pour des applications réactives, offrant une meilleure performance dans des environnements hautement concurrentiels.

Les projets Spring

Spring Framework comprend plusieurs projets spécialisés, chacun ayant des fonctionnalités spécifiques pour répondre à divers besoins dans le développement d'applications Java. Voici un aperçu des projets principaux de Spring et leurs applications :

1. **Spring Boot** (L'objet de notre étude)

- **Description** : Simplifie le développement d'applications autonomes prêtes pour la production en Java. Il propose des configurations par défaut qui permettent de démarrer rapidement, sans avoir besoin de gérer manuellement une grande partie de la configuration.
- **Utilisation** : Spring Boot est largement utilisé pour le développement de microservices et d'applications monolithiques. Il inclut un serveur intégré (comme Tomcat) pour exécuter des applications en mode autonome.
- **Caractéristiques** :
    - Démarrage rapide avec des dépendances prêtes à l'emploi.
    - Configurations automatiques pour plusieurs composants.
    - Intégration facile avec d'autres modules Spring.
    - Prise en charge de profils pour gérer des configurations différentes pour divers environnements (développement, test, production).

2. Spring Cloud

- **Description** : Fournit un ensemble d’outils et de bibliothèques pour développer des applications distribuées et des microservices. Il aide à gérer les aspects complexes des systèmes cloud, comme la résilience, la tolérance aux pannes, et la découverte de services.
- **Utilisation** : Principalement utilisé pour les architectures de microservices dans des environnements de cloud, comme AWS, Google Cloud, ou Microsoft Azure.
- **Caractéristiques** :
    - **Service Discovery** : Utilise Eureka, Consul, ou Zookeeper pour enregistrer et découvrir les services.
    - **Configuration distribuée** : Spring Cloud Config permet de gérer des configurations centralisées.
    - **Gestion de la tolérance aux pannes** : Hystrix, Circuit Breaker, et Resilience4j pour gérer les pannes.
    - **API Gateway** : Spring Cloud Gateway pour gérer le routage des demandes entre les services.

3. Spring Data

- **Description** : Simplifie l'accès aux bases de données et fournit des abstractions unifiées pour différents types de stockage de données (SQL, NoSQL).
- **Utilisation** : Utilisé pour faciliter l'interaction avec les bases de données en réduisant la quantité de code nécessaire pour des opérations CRUD et en prenant en charge plusieurs types de stockage, comme MongoDB, Redis, et Cassandra.
- **Caractéristiques** :
    - **Spring Data JPA** : Pour les bases de données relationnelles (utilisant JPA et Hibernate).
    - **Spring Data MongoDB** : Pour les bases de données NoSQL comme MongoDB.
    - **Spring Data Redis** : Pour les opérations en mémoire avec Redis.
    - **Spring Data REST** : Expose les données sous forme de services RESTful.

4. Spring Security

- **Description** : Framework de sécurité qui prend en charge l'authentification, l'autorisation, et d'autres aspects de sécurité.
- **Utilisation** : Utilisé dans des applications qui nécessitent une protection d'accès, avec des fonctionnalités comme l'authentification basée sur des rôles, les connexions multi-facteurs, ou la gestion des sessions.
- **Caractéristiques** :
    - Authentification et autorisation configurables avec les annotations et les règles de sécurité.
    - Prise en charge des protocoles OAuth2 et OpenID Connect pour l'intégration avec des fournisseurs de sécurité tiers.
    - Sécurité pour les applications Web et les API REST.
    - Options de sécurité adaptées aux applications réactives via Spring WebFlux Security.

5. Spring Batch

- **Description** : Fournit des outils pour construire des applications de traitement par lots.
- **Utilisation** : Utilisé pour les applications de traitement par lots nécessitant l'automatisation des tâches répétitives (comme l'importation de données, le traitement de fichiers ou la génération de rapports).
- **Caractéristiques** :
    - Supporte les tâches planifiées et l'exécution de processus batch lourds.
    - Gestion de transactions et de reprise après erreur.
    - Prise en charge du partitionnement, de la distribution et de l'évolutivité des processus batch.

6. Spring Integration

- **Description** : Fournit une infrastructure pour les applications d'entreprise intégrant différents systèmes et processus.
- **Utilisation** : Utile dans les architectures orientées messages pour intégrer différents systèmes ou pour gérer des flux de données.
- **Caractéristiques** :
    - Composants de messagerie pour la communication inter-système.
    - Gestion de l’orchestration de flux de données complexes.
    - Prise en charge des protocoles d'intégration (JMS, AMQP, MQTT, Kafka).

7. Spring WebFlux

- **Description** : Framework de développement réactif pour construire des applications asynchrones et non bloquantes.
- **Utilisation** : Utilisé pour les applications qui nécessitent une haute performance, une faible latence, ou des systèmes qui doivent gérer un grand nombre de connexions simultanées.
- **Caractéristiques** :
    - Basé sur le modèle réactif et les flux de données.
    - Construit sur Netty, un serveur HTTP réactif, pour des performances optimales.
    - Prise en charge de l'approche fonctionnelle et de l'annotation pour les contrôleurs.

8. Spring AMQP

- **Description** : Fournit un modèle d'API pour intégrer des systèmes utilisant le protocole de messagerie AMQP (comme RabbitMQ).
- **Utilisation** : Utile pour les applications nécessitant des files d’attente de messages ou des traitements asynchrones.
- **Caractéristiques** :
    - Supporte les échanges et la gestion des files d'attente dans les systèmes AMQP.
    - Fonctionnalités pour le routage et la répartition des messages.

- Intégration avec Spring Framework pour simplifier la configuration.

9. Spring HATEOAS (Hypermedia as the Engine of Application State)

- **Description** : Facilite le développement d'API RESTful en intégrant des liens hypermedia dans les réponses.
- **Utilisation** : Souvent utilisé dans les architectures RESTful pour fournir des informations de navigation avec les ressources.
- **Caractéristiques** :
  - Génère des liens hypermedia automatiquement dans les réponses.
  - Simplifie la navigation entre les ressources d'API.
  - Prend en charge la mise en œuvre de normes HATEOAS pour une meilleure expérience RESTful.

Documentation de Spring

La documentation de Spring est reconnue pour sa qualité et sa profondeur. Elle offre des guides complets, des références détaillées, et des exemples clairs pour aider les développeurs à comprendre et à utiliser chaque aspect du framework. Organisée par projets (comme Spring Boot, Spring Data, Spring Security, etc.), elle couvre aussi bien les concepts de base que les sujets avancés, facilitant la prise en main et l'approfondissement.

En plus des guides écrits, la documentation de Spring inclut souvent des tutoriels pratiques, des échantillons de code, et des FAQ, ce qui en fait une ressource précieuse pour les développeurs de tous niveaux.Les liens vers la documentation officielle des différents projets du **Spring Framework** et les pages qui fournissent des guides, des références, et des tutoriels pratiques pour chaque module sont les suivants:

1. **Spring Framework**
   Documentation principale du framework Spring.
   Spring Framework Documentation **(**
   https://docs.spring.io/spring-framework/docs/current/reference/html/ **)**
2. **Spring Boot** (L'objet de notre étude)
   Pour démarrer avec Spring Boot et créer des applications prêtes pour la production.
   Spring Boot Documentation (
   https://docs.spring.io/spring-boot/docs/current/reference/html/)
3. **Spring Cloud**
   Documentation pour les outils et bibliothèques permettant de créer des applications distribuées et des microservices.

Spring Cloud Documentation (
https://docs.spring.io/spring-cloud/docs/current/reference/html/)

4. **Spring Data**
   Guides pour l'accès et la gestion de données avec différents types de bases de données (SQL, NoSQL, etc.).
   Spring Data Documentation (
   https://docs.spring.io/spring-data/jpa/docs/current/reference/html/)
5. **Spring Security**
   Documentation complète pour l'authentification, l'autorisation, et la sécurisation des applications.
   Spring Security Documentation (
   https://docs.spring.io/spring-security/site/docs/current/reference/html5/)
6. **Spring Batch**
   Pour les applications de traitement par lots et les tâches automatisées.
   Spring Batch Documentation (
   https://docs.spring.io/spring-batch/docs/current/reference/html/)
7. **Spring Integration**
   Documentation pour intégrer divers systèmes au sein d'une architecture orientée messages.
   Spring Integration Documentation (
   https://docs.spring.io/spring-integration/docs/current/reference/html/)
8. **Spring WebFlux**
   Pour le développement d'applications réactives et non bloquantes.
   Spring WebFlux Documentation (
   https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html)
9. **Spring AMQP**
   Documentation pour l'intégration avec les systèmes de messagerie basés sur le protocole AMQP (ex. RabbitMQ).
   Spring AMQP Documentation (
   https://docs.spring.io/spring-amqp/docs/current/reference/html/)
10. **Spring HATEOAS**
    Pour ajouter des liens hypermedia dans les réponses RESTful.
    Spring HATEOAS Documentation (
    https://docs.spring.io/spring-hateoas/docs/current/reference/html/)

Spring Boot

Spring Boot est un projet du Spring Framework conçu pour simplifier le développement d'applications Java autonomes prêtes pour la production, en

éliminant les configurations fastidieuses. En se basant sur des conventions intelligentes et des configurations automatiques, Spring Boot rend le processus de démarrage rapide et intuitif, surtout pour les microservices et les applications web. Il inclut un serveur embarqué, comme Tomcat ou Jetty, ce qui permet de lancer les applications en standalone sans avoir besoin d'un serveur d'applications externe.

Pourquoi utiliser Spring Boot ?

1. **Démarrage rapide** : Spring Boot configure automatiquement de nombreux composants nécessaires, ce qui permet aux développeurs de se concentrer sur le code métier plutôt que sur la configuration.
2. **Serveur embarqué** : Avec des serveurs intégrés, comme Tomcat, Jetty, ou Undertow, Spring Boot permet d'exécuter des applications en tant qu'exécutables indépendants (via une simple commande java -jar), sans déploiement sur un serveur externe.
3. **Production-ready** : Spring Boot intègre des fonctionnalités prêtes pour la production, comme le monitoring, les métriques, les vérifications de santé et la gestion des applications.
4. **Facilité de configuration** : Grâce aux annotations et aux configurations basées sur les conventions, Spring Boot simplifie la gestion des dépendances et de la configuration.
5. **Support natif pour les microservices** : Spring Boot est souvent choisi pour développer des architectures de microservices, grâce à sa légèreté et à son intégration facile avec Spring Cloud.

Fonctionnalités principales de Spring Boot

1. **Auto-configuration** : Spring Boot propose une auto-configuration intelligente qui détecte les bibliothèques dans le classpath et configure automatiquement les composants requis. Par exemple, si Spring Data JPA et une source de données sont présents, Spring Boot configurera automatiquement JPA pour interagir avec la base de données.

2. **Starter POMs** : Spring Boot propose des "starters" pour simplifier la gestion des dépendances. Un starter est un regroupement de dépendances préconfigurées pour une fonctionnalité spécifique. Par exemple, spring-boot-starter-web inclut les dépendances nécessaires pour créer une application web avec Spring MVC et Tomcat intégré.

3. **Spring Boot Actuator** : Actuator ajoute des **endpoints** pour surveiller et gérer l'application. Ces endpoints permettent de voir l'état de santé de l'application, de vérifier les métriques, de consulter les logs, et d'autres informations utiles pour l'administration.

4. **Profiles et configurations externalisées** : Spring Boot permet de gérer des configurations spécifiques pour différents environnements (développement, test, production) en utilisant des profils. Les fichiers de configuration, comme

application.properties ou application.yml, peuvent être personnalisés pour chaque profil, facilitant le déploiement multi-environnements.

5. **Mise en œuvre rapide de REST API** : Avec Spring Boot et Spring MVC, créer des services REST est simple et rapide. **Spring Boot** gère automatiquement les conventions pour exposer des endpoints RESTful, ce qui est utile dans les architectures microservices.

Démarrage rapide avec Spring Boot

Les étapes pour démarrer avec Spring Boot :

1. **Créer une application Spring Boot** : Vous pouvez créer une application Spring Boot en utilisant Spring Initializr (https://start.spring.io/), un générateur de projets en ligne. Il vous permet de sélectionner vos dépendances, le type de projet (**Maven** ou **Gradle**), et la version de Java.
2. **Structure de base** : Après avoir généré le projet, vous aurez une structure de projet avec un point d'entrée principal (**@SpringBootApplication**), souvent dans une classe nommée Application. Cette classe contient la méthode main et initialise le contexte de l'application.
3. **Configuration minimale** : Les fichiers de configuration **application.properties** ou **application.yml** contiennent les paramètres essentiels, comme le port du serveur, les informations de la base de données, et les configurations de sécurité. Avec Spring Boot, même des configurations complexes peuvent être gérées facilement.
4. **Démarrer l'application** : Vous pouvez démarrer l'application avec la commande suivante :

   mvn spring-boot:run

   ou en exécutant simplement le fichier .jar généré :

   java -jar mon-app.jar

Annotations

Spring Boot utilise une large gamme d'annotations qui simplifient la configuration et le développement des applications. Ces annotations couvrent des aspects variés, comme la configuration, la gestion des composants, la sécurité, l'accès aux données, les services web, et plus encore. Les principales annotations de Spring Boot, organisées par catégories :

1. Annotations de configuration

- **@SpringBootApplication** : C'est l'annotation de base pour démarrer une application Spring Boot. Elle regroupe trois annotations :
  - **@Configuration** : Marque la classe comme source de configurations Spring.

    - **@EnableAutoConfiguration** : Active l'auto-configuration de Spring Boot pour configurer automatiquement les composants.
    - **@ComponentScan** : Indique à Spring de scanner les sous-packages pour détecter et enregistrer les beans annotés.
- **@Configuration** : Déclare une classe comme source de configurations et de définitions de beans.
- **@Bean** : Utilisée dans une classe annotée par **@Configuration**, elle définit un bean que Spring gère dans le conteneur d'injection de dépendances.
- **@PropertySource** : Spécifie un fichier de propriétés externe à charger dans le contexte Spring (ex. **@PropertySource**("classpath:application.properties")).
- **@Value** : Injecte des valeurs de configuration (provenant de application.properties, application.yml, ou d'autres sources) dans les attributs de classe.

2. Annotations pour les composants Spring

- **@Component** : Indique que la classe est un composant géré par Spring. Elle est détectée lors de l'analyse **@ComponentScan** et ajoutée au conteneur Spring.
- **@Service** : Spécialisation de **@Component** pour indiquer une classe de service, utilisée pour la logique métier.
- **@Repository** : Spécialisation de **@Component** pour les classes d'accès aux données (DAO). Elle intègre également la gestion des exceptions spécifiques aux bases de données.
- **@Controller** : Spécialisation de **@Component** pour les classes de contrôleurs, souvent utilisées dans les applications web pour gérer les requêtes.
- **@RestController** : Combinaison de @Controller et **@ResponseBody**. Elle est utilisée pour créer des services RESTful, où chaque méthode renvoie directement une réponse JSON/XML.

3. Annotations pour les endpoints et le routage

- **@RequestMapping** : Utilisée sur les classes ou les méthodes de contrôleurs pour définir les URL de routage. Elle gère plusieurs types de requêtes HTTP.
- **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping**, **@PatchMapping** : Variantes de **@RequestMapping** pour spécifier respectivement les requêtes GET, POST, PUT, DELETE et PATCH.
- **@PathVariable** : Récupère une variable de chemin dans une méthode de contrôleur (ex. @GetMapping("/user/{id}") pour capturer id).
- **@RequestParam** : Utilisée pour extraire les paramètres de requête dans une méthode de contrôleur (ex. @GetMapping("/search") public String search(@RequestParam String query)).
- **@RequestBody** : Lie le corps d'une requête HTTP à un objet Java, souvent utilisé dans les méthodes **@PostMapping** et **@PutMapping** pour les API REST.

- **@ResponseBody** : Indique que le retour d’une méthode de contrôleur doit être directement écrit dans la réponse HTTP sous forme JSON ou XML.
- **@CrossOrigin** : Permet les requêtes CORS (Cross-Origin Resource Sharing) sur des méthodes spécifiques ou au niveau de tout le contrôleur.

4. Annotations pour l’injection de dépendances

- **@Autowired** : Injecte automatiquement une dépendance dans une classe Spring (via le constructeur, le champ, ou les méthodes setter).
- **@Qualifier** : Utilisée avec @Autowired pour résoudre les conflits lorsque plusieurs beans de même type sont disponibles. Elle permet de spécifier le nom du bean à injecter.
- **@Primary** : Lorsqu'il y a plusieurs beans de même type, cette annotation marque un bean comme prioritaire pour l'injection.

5. Annotations de validation

- **@Valid** : Utilisée pour activer la validation sur un objet (par exemple, une requête @RequestBody). Elle nécessite une configuration de validateur (comme Hibernate Validator).
- **@NotNull**, **@NotEmpty**, **@Size**, etc. : Annotations de validation pour restreindre les valeurs des champs, souvent utilisées avec @Valid pour valider des entrées utilisateur.

6. Annotations de persistance (Spring Data)

- **@Entity** : Marque une classe Java en tant que table de base de données (entité) dans JPA (Java Persistence API).
- **@Id** : Indique le champ qui est la clé primaire dans une entité.
- **@GeneratedValue** : Utilisée avec @Id pour spécifier que la clé primaire est générée automatiquement (ex. AUTO, IDENTITY, SEQUENCE, TABLE).
- **@Table** : Spécifie les détails de la table en base de données, comme le nom de la table.
- **@Column** : Spécifie les détails d'une colonne, comme le nom, le type, et les contraintes.
- **@RepositoryRestResource** : Utilisée dans Spring Data REST pour exposer automatiquement des méthodes de repository sous forme de services REST.
- **@Query** : Permet de définir des requêtes JPQL (Java Persistence Query Language) personnalisées dans les méthodes de repository.

7. Annotations de sécurité (Spring Security)

- **@Secured** : Restreint l'accès à une méthode spécifique selon les rôles utilisateur.

- **@PreAuthorize** et **@PostAuthorize** : Permettent d'utiliser des expressions SpEL (Spring Expression Language) pour définir des conditions d'accès avant ou après l'exécution d'une méthode.
- **@RolesAllowed** : Indique que seuls les utilisateurs ayant certains rôles peuvent accéder à la ressource ou à la méthode.
- **@EnableWebSecurity** : Active Spring Security pour l'application et permet de configurer les règles de sécurité.

8. Annotations pour les tâches planifiées

- **@Scheduled** : Permet de planifier l'exécution automatique de méthodes à des intervalles spécifiques (en utilisant cron, fixedRate, fixedDelay, etc.).
- **@EnableScheduling** : Active la planification des tâches, nécessaire pour que @Scheduled fonctionne.

9. Annotations pour le test

- **@SpringBootTest** : Configure un contexte d'application pour les tests d'intégration Spring Boot.
- **@MockBean** : Crée des mocks pour les dépendances des beans Spring dans les tests, en remplaçant le bean dans le contexte de test.
- **@WebMvcTest** : Configure le test d'un contrôleur Spring MVC, en incluant seulement les composants MVC (comme les contrôleurs, les convertisseurs, les validateurs).
- **@DataJpaTest** : Configure un environnement de test pour les repositories JPA. Il crée un contexte de test limité aux opérations JPA.

10. Annotations spécifiques à Spring Boot Actuator

- **@Endpoint** : Déclare un nouveau point de terminaison Actuator.
- **@ReadOperation**, **@WriteOperation**, **@DeleteOperation** : Spécifient des opérations de lecture, d’écriture, et de suppression pour un endpoint personnalisé dans Actuator.

11. Autres annotations utiles

- **@Profile** : Indique le profil pour lequel un bean ou une configuration doit être chargé, permettant des configurations différentes selon l'environnement.
- **@ConditionalOnProperty** : Charge un bean uniquement si une certaine propriété est présente dans la configuration.
- **@ConditionalOnMissingBean** : Charge un bean seulement si aucun autre bean du même type n'est déjà présent.
- **@Import** : Permet d'importer une configuration ou une classe dans le contexte Spring.

Exemple détaillé Spring Data

Les annotations de persistance dans Spring Data, principalement basées sur JPA (Java Persistence API), sont essentielles pour la gestion de la persistance des données. Voici les annotations clés avec leurs attributs détaillés :

1. @Entity

L'annotation @Entity marque une classe Java en tant qu'entité persistante, c'est-à-dire une table en base de données.

- **name** : Spécifie un nom pour l'entité. Si ce champ n'est pas défini, le nom de la classe est utilisé par défaut.

Exemple :
```
@Entity(name = "CustomEntityName")
```

2. @Table

@Table permet de définir des propriétés spécifiques à la table, comme son nom en base de données.

- **name** : Le nom de la table dans la base de données. Par défaut, le nom de la classe est utilisé.
- **catalog** : Définit le catalogue de la base de données dans lequel la table est créée.
- **schema** : Spécifie le schéma dans lequel la table est stockée.
- **uniqueConstraints** : Définit les contraintes d'unicité sur une ou plusieurs colonnes. Utilise l'annotation @UniqueConstraint.
- **indexes** : Crée des index sur les colonnes de la table en utilisant @Index.

Exemple :
```
@Table(
    name = "employees",
    schema = "company",
    uniqueConstraints = {@UniqueConstraint(columnNames = {"email"})},
    indexes = {@Index(name = "idx_employee_name", columnList = "name")}
)
```

3. @Id

Marque un champ comme clé primaire d'une entité. Elle n'a pas d'attributs spécifiques.

Exemple :
```
@Id
private Long id;
```

4. @GeneratedValue

Utilisée avec @Id, elle définit la stratégie de génération de la clé primaire.

- **strategy** : Spécifie la stratégie de génération. Options :
    - **GenerationType.AUTO** : L'implémentation JPA choisit la stratégie (par défaut).
    - **GenerationType.IDENTITY** : Utilise une colonne d'identité dans la base de données.
    - **GenerationType.SEQUENCE** : Utilise une séquence pour générer les valeurs.
    - **GenerationType.TABLE** : Utilise une table spéciale pour générer les valeurs.
- **generator** : Spécifie le nom d'un générateur personnalisé défini avec **@SequenceGenerator** ou **@TableGenerator.**

Exemple :
```
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
```

5. @Column

Personnalise les propriétés d'une colonne dans la base de données.

- **name** : Le nom de la colonne en base de données. Par défaut, le nom de l'attribut Java est utilisé.
- **nullable** : Indique si la colonne accepte les valeurs nulles (par défaut, true).
- **unique** : Indique si la colonne doit être unique (par défaut, false).
- **length** : Définit la longueur de la colonne (utilisé surtout pour les String, avec une valeur par défaut de 255).
- **precision** : Spécifie la précision pour les types numériques (utilisé avec BigDecimal).
- **scale** : Spécifie l'échelle (nombre de décimales) pour les types numériques (utilisé avec BigDecimal).
- **columnDefinition** : Permet de spécifier une définition SQL complète de la colonne.
- **insertable** : Indique si la colonne doit être incluse dans les instructions INSERT (par défaut, true).
- **updatable** : Indique si la colonne doit être incluse dans les instructions UPDATE (par défaut, true).
- **table** : Définit la table à laquelle appartient cette colonne, utile si la classe est mappée sur plusieurs tables.

Exemple :
```
@Column(name = "email", nullable = false, unique = true, length = 150)
private String email;
```

6. @JoinColumn

Utilisée pour définir une clé étrangère dans les relations entre entités (**@OneToOne**, **@OneToMany**, **@ManyToOne**, **@ManyToMany**).

- **name** : Le nom de la colonne de jointure dans la table.
- **referencedColumnName** : La colonne référencée dans l'entité cible.
- **nullable** : Indique si la colonne de jointure peut être nulle (par défaut, true).
- **unique** : Indique si la colonne de jointure doit être unique (par défaut, false).
- **insertable** : Indique si la colonne de jointure est incluse dans les instructions INSERT (par défaut, true).
- **updatable** : Indique si la colonne de jointure est incluse dans les instructions UPDATE (par défaut, true).
- **table** : La table dans laquelle la colonne de jointure est créée (utile dans les associations multi-tables).

Exemple :
@JoinColumn(name = "department_id", referencedColumnName = "id", nullable = false)

7. @OneToOne

Définit une relation un-à-un entre deux entités.

- **mappedBy** : Utilisé dans la classe propriétaire pour spécifier la relation bidirectionnelle.
- **cascade** : Détermine les opérations de cascade (ex. CascadeType.ALL, CascadeType.PERSIST).
- **fetch** : Détermine la stratégie de chargement (par défaut FetchType.EAGER).
- **optional** : Indique si la relation est optionnelle (par défaut, true).

java
Exemple :
@OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
@JoinColumn(name = "address_id")
private Address address;

8. @OneToMany

Définit une relation un-à-plusieurs entre une entité et une collection d'une autre entité.

- **mappedBy** : Définit le côté inverse de la relation (dans une relation bidirectionnelle).
- **cascade** : Détermine les opérations de cascade.
- **fetch** : Détermine la stratégie de chargement (par défaut FetchType.LAZY).
- **orphanRemoval** : Supprime les entités "orphelines" (défaut false).

Exemple :

```
@OneToMany(mappedBy = "department", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Employee> employees;
```

9. @ManyToOne

Définit une relation plusieurs-à-un entre deux entités.

- **cascade** : Détermine les opérations de cascade.
- **fetch** : Détermine la stratégie de chargement (par défaut FetchType.EAGER).
- **optional** : Indique si la relation est optionnelle (par défaut, true).

Exemple :

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "department_id")
private Department department;
```

10. @ManyToMany

Définit une relation plusieurs-à-plusieurs entre deux entités.

- **mappedBy** : Utilisé pour indiquer le côté inverse de la relation dans une relation bidirectionnelle.
- **cascade** : Détermine les opérations de cascade.
- **fetch** : Détermine la stratégie de chargement (par défaut FetchType.LAZY).

Exemple :

```
@ManyToMany
@JoinTable(
    name = "employee_project",
    joinColumns = @JoinColumn(name = "employee_id"),
    inverseJoinColumns = @JoinColumn(name = "project_id")
)
private List<Project> projects;
```

11. @JoinTable

Utilisée dans une relation **@ManyToMany** pour définir la table d'association.

- **name** : Le nom de la table de jointure.
- **joinColumns** : Les colonnes dans la table de jointure qui font référence à la classe propriétaire de la relation.
- **inverseJoinColumns** : Les colonnes dans la table de jointure qui font référence à l'entité associée.

Exemple :

```
@JoinTable(
    name = "employee_project",
    joinColumns = @JoinColumn(name = "employee_id"),
```

```
    inverseJoinColumns = @JoinColumn(name = "project_id")
)
```

12. @Transient

Marque un champ pour qu'il ne soit pas persistant en base de données. Aucun attribut particulier.

Exempe

```
@Transient
private int tempValue;
```

13. @Embeddable

Indique qu'une classe peut être "embarquée" dans une autre entité. Aucun attribut particulier.

Exemple :

```
@Embeddable
public class Address {
    private String city;
    private String state;
}
```

14. @Embedded

Indique qu'une classe est embarquée dans une autre entité, permettant l'inclusion des attributs de la classe @Embeddable.

Exemple :

```
@Embedded
private Address address;
```

15. @AttributeOverride

Permet de remplacer la configuration d'un attribut d'une classe @Embeddable.

- **name** : Le nom de l'attribut à remplacer.
- **column** : La configuration de la colonne (ex. @Column).

  Exemple :

  ```
  @Embedded
  @AttributeOverrides({
      @AttributeOverride(name = "city", column = @Column(name = "home_city")),
      @AttributeOverride(name = "state", column = @Column(name = "home_state"))
  })
  private Address homeAddress;
  ```

Injection de dépendance

En programmation orientée objet en Java, une dépendance forte entre deux classes signifie qu'une classe dépend directement d'une autre classe pour fonctionner, ce qui rend leur lien très étroit. Créons une dépendance forte entre une classe **Command** et une classe **Product**.

Imaginons que la classe **Command** représente une commande effectuée par un client, et que cette commande contient des informations sur un **Product** (produit) associé. Si **Command** a besoin d'instancier un objet **Product** directement en son sein pour fonctionner, cela représente une dépendance forte.Si la classe `Product` subit une modification, cela affectera directement le constructeur de la classe `Command`.

En plus, si on souhaite utiliser une autre implémentation de `Product`, il faudra à chaque fois ajuster la classe `Command` en conséquence 🔧💻 .

Conclusion : l'évolutivité s'envole ! 🚀 ! Avec des centaines ou des milliers de lignes de code, cela rendra notre application très difficile à maintenir à long terme. 🌸🎨🌹🎈🐬🍫

Voici la signification de chaque émoticône :

🌸 **(Fleur)** : Pour le "F", la fleur représente douceur et beauté.

- 🎨 **(Palette)** : Pour le "A", cette palette symbolise créativité et expression artistique.
- 🌹 **(Rose)** : Le "R" est représenté par une rose, qui symbolise la beauté et l'élégance.
- 🎈 **(Ballon)** : Pour le "I", le ballon représente légèreté et joie.
- 🐬 **(Dauphin)** : Le "D" évoque le dauphin, symbole de liberté et d'intelligence.
- 🍫 **(Chocolat)** : Pour le "A" final, le chocolat symbolise la douceur et le plaisir.

Cela donne un style unique et une touche personnelle pour chaque lettre du prénom "**Farida**" ! 😊

"Bon, on a eu notre moment de fun avec les émoticônes 😄, mais revenons aux choses sérieuses pour ne pas perdre le fil ! n'oublions pas qu'on doit aussi aider **Farida** à devenir indépendante et à ne plus dépendre de son mari ! 💪📚 Allez, remettons-nous au travail !"

Exemple avec Dépendance Forte

Dans cet exemple, la classe **Command** crée elle-même une instance de **Product** à partir des paramètres reçus. Cela crée un lien étroit entre **Command** et Product, car **Command** dépend directement de la création de l'objet **Product** en interne.

Classe Product

```
public class Product {

    private String name;

    private double price;
```

```
    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }
}
```

Classe **Command** avec dépendance forte

```
public class Command {
    private Product product;
    private int quantity;

    // Dépendance forte : Command crée une instance de Product en interne
    public Command(String productName, double productPrice, int quantity) {
        this.product = new Product(productName, productPrice);
        this.quantity = quantity;
    }

    public double calculateTotal() {
        return product.getPrice() * quantity;
    }

    public void displayCommandDetails() {
```

```
        System.out.println("Produit: " + product.getName());
        System.out.println("Quantité: " + quantity);
        System.out.println("Prix Total: " + calculateTotal());
    }
}
```

Classe Main pour tester la dépendance forte

```
public class Main {
    public static void main(String[] args) {
        // Création d'une commande avec nom et prix du produit + quantité
        Command command = new Command("Laptop", 1200.00, 3);

        // Affichage des détails de la commande
        command.displayCommandDetails();
    }
}
```

Explications

Ici, la classe **Command** crée un Product en interne, en recevant directement les informations de nom et de prix du produit. Cela crée une **dépendance forte** entre Command et Product, car Command ne peut pas fonctionner sans Product, et toute modification dans Product pourrait nécessiter un changement dans Command.

Exemple avec Dépendance Faible

Dans cet exemple, Command reçoit un objet Product existant en paramètre au lieu de le créer elle-même. Cela rend Command moins dépendante de Product et permet une meilleure flexibilité (par exemple, pour utiliser d'autres objets similaires).

Classe Product (Même code avec dépendance forte)

```
public class Product {
    private String name;
    private double price;

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
```

```
        return name;
    }

    public double getPrice() {
        return price;
    }
}

Classe Command avec dépendance faible
public class Command {
    private Product product;
    private int quantity;

    // Dépendance faible : Command reçoit un Product en paramètre
    public Command(Product product, int quantity) {
        this.product = product;
        this.quantity = quantity;
    }

    public double calculateTotal() {
        return product.getPrice() * quantity;
    }

    public void displayCommandDetails() {
        System.out.println("Product: " + product.getName());
        System.out.println("Quantity: " + quantity);
        System.out.println("Total Price: " + calculateTotal());
    }
}

Classe Main pour tester la dépendance faible
public class Main {
    public static void main(String[] args) {
        // Création d'un produit
        Product product = new Product("Ordinateur portable", 200.00);

        // Création d'une commande avec un produit existant et une quantité
        Command command = new Command(product, 3);

        // Affichage des détails de la commande
        command.displayCommandDetails();
    }
}
```

Explications

Dans ce code, Command ne crée pas l'objet Product elle-même. Elle reçoit un Product existant dans son constructeur. Cela réduit la dépendance directe de Command à Product, permettant plus de flexibilité et facilitant les tests et la maintenance.

A Rétenir

- **Dépendance Forte** : Command crée directement un Product dans son constructeur, ce qui la rend fortement dépendante de Product.
- **Dépendance Faible** (**injection de dépendances**): Command reçoit un Product existant en paramètre, ce qui réduit la dépendance et augmente la flexibilité.

Gestion de dépendance avec Spring

Dans le Spring Framework, **l'injection de dépendances** (ou Dependency Injection, DI) est un concept central pour découpler les classes et favoriser l'inversion de contrôle (Inversion of Control, IoC). Spring gère l'injection de dépendances de manière automatique, en instanciant et en injectant les dépendances des objets sans que ceux-ci aient besoin de les créer eux-mêmes.

Les méthodes principales pour gérer l'injection de dépendances avec Spring sont :

1. **Injection par constructeur**
2. **Injection par setter**
3. **Injection via annotations**

Nous allons revenir plus tard pour mettre en place ces techniques

Création du projet

Starters de dépendances

Les **starters de dépendances** sont des modules prédéfinis utilisés dans des environnements comme **Spring Boot** pour simplifier la gestion des dépendances dans un projet. Ils regroupent une collection de bibliothèques et de configurations nécessaires pour ajouter des fonctionnalités spécifiques, comme la gestion de la sécurité, l'accès aux bases de données, ou les tests. Par exemple, en ajoutant le starter **spring-boot-starter-web**, toutes les bibliothèques nécessaires pour créer des applications web sont incluses automatiquement.

Cela permet de gagner du temps, d'assurer la compatibilité des versions, et de maintenir une structure de projet plus propre et mieux organisée.Ces starters permettent aux développeurs de se concentrer sur la logique métier sans se soucier de la configuration manuelle de chaque dépendance.

Spring Boot propose plusieurs *starters* pour différents besoins de développement, facilitant l'intégration de fonctionnalités spécifiques. Quelques starters courants :

1. **spring-boot-starter** : Base minimaliste contenant les composants fondamentaux de Spring Boot.
2. **spring-boot-starter-web** : Inclut Spring MVC, Jackson pour JSON, et Tomcat pour développer des applications web RESTful.
3. **spring-boot-starter-data-jpa** : Fournit tout ce qu'il faut pour utiliser JPA avec Hibernate comme implémentation par défaut pour la persistance des données.
4. **spring-boot-starter-data-mongodb** : Pour les projets qui utilisent MongoDB comme base de données NoSQL.
5. **spring-boot-starter-security** : Ajoute des fonctionnalités de sécurité, comme l'authentification et l'autorisation.
6. **spring-boot-starter-thymeleaf** : Pour les projets web utilisant Thymeleaf comme moteur de templates.
7. **spring-boot-starter-test** : Inclut JUnit, Mockito, et Spring Test pour des tests unitaires et d'intégration.
8. **spring-boot-starter-actuator** : Ajoute des fonctionnalités de surveillance et de gestion de l'application en production, comme les points de terminaison pour la santé, les métriques, etc.
9. **spring-boot-starter-mail** : Fournit une configuration prête à l'emploi pour envoyer des e-mails avec JavaMail.
10. **spring-boot-starter-websocket** : Facilite l'utilisation de WebSocket pour des applications nécessitant une communication en temps réel.
11. **spring-boot-starter-cache** : Pour activer et configurer la gestion de cache dans une application.
12. **spring-boot-starter-validation** : Fournit des fonctionnalités de validation d'entités avec Hibernate Validator.
13. **spring-boot-starter-amqp** : Pour les applications qui utilisent RabbitMQ pour la messagerie asynchrone.

Projet Mavn

Maven est un outil de gestion de projet et de construction (build) très populaire dans l'écosystème Java. Il simplifie la gestion des dépendances, l'automatisation des tâches de compilation, de test, et de déploiement, et permet d'assurer la cohérence des versions au sein d'un projet.

Maven utilise un fichier de configuration central appelé **pom.xml** (**P**roject **O**bject **M**odel) qui décrit le projet, ses dépendances, les plugins nécessaires, et d'autres configurations. Ce fichier aide à :

1. **Gérer les dépendances** : Maven télécharge automatiquement les bibliothèques nécessaires depuis des dépôts en ligne, évitant ainsi les conflits de versions et facilitant la mise à jour des bibliothèques.
2. **Automatiser les tâches** : Maven peut exécuter des phases comme la compilation, les tests, et le packaging (ex : créer un fichier JAR ou WAR) en une seule commande.
3. **Faciliter la standardisation** : Avec sa structure de projet prédéfinie, Maven encourage les bonnes pratiques et rend le code plus facile à partager ou intégrer.

Déclarer starters dans pom.xml

Pour déclarer les *starters* **Spring Boot** dans le fichier **pom.xml** d'un projet Maven, vous pouvez ajouter les dépendances nécessaires dans la section **<dependencies>**

NB :
Tous les starters sont préfixés par **spring-boot-starter**.

Exemples de starters :

- spring-boot-starter-core ;
- spring-boot-starter-data-jpa ;
- spring-boot-starter-security ;
- spring-boot-starter-test ;
- spring-boot-starter-web

Exemple d'inclusion de Spring Boot Starters dans pom.xml

```
<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
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>com.example</groupId>
   <artifactId>spring-boot-example</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <packaging>jar</packaging>
```

```
    <!-- Déclaration de la version de Spring Boot -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.1.0</version> <!-- Assurez-vous d'utiliser la dernière version
disponible -->
        <relativePath/> <!-- Recherche Spring Boot dans le repository Maven central --
>
    </parent>

    <dependencies>
        <!-- Starter pour les applications web RESTful -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- Starter pour JPA et Hibernate -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <!-- Starter pour la sécurité -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>

        <!-- Starter pour le test -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <!-- Starter pour Actuator (surveillance et gestion) -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
    </dependencies>

    <!-- Configuration du plugin Spring Boot Maven -->
    <build>
        <plugins>
            <plugin>
```

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

Explications

1. **Parent Spring Boot** : La section **<parent>** inclut le spring-boot-starter-parent, qui définit des configurations par défaut pour les versions des dépendances et autres paramètres Maven.
2. **Dépendances Spring Boot** : Chaque <dependency> ajoute un starter pour une fonctionnalité spécifique :
    - spring-boot-starter-web pour les applications web.
    - spring-boot-starter-data-jpa pour JPA et les bases de données relationnelles.
    - spring-boot-starter-security pour les fonctionnalités de sécurité.
    - spring-boot-starter-test pour les tests unitaires et d'intégration.
    - spring-boot-starter-actuator pour la surveillance en production.
3. **Plugin Maven** : Le spring-boot-maven-plugin est nécessaire pour construire et exécuter l'application Spring Boot facilement avec Maven.

En incluant ces starters, Spring Boot configure automatiquement les bibliothèques et les dépendances nécessaires pour chaque fonctionnalité, réduisant ainsi les configurations manuelles.

Il y a deux méthodes pour créer un projet **Spring Boot** : en utilisant **Spring Initializr** ou en passant par **Spring Tool Suite**. Explorons ensemble ces deux approches !

Utilisation de Spring Initializr

Pour créer un projet avec **Spring Initializr** et ajouter le starter de base **spring-boot-starter**, suivez les étapes ci-dessous :

Étape 1 : Accéder à Spring Initializr

1. Ouvrez votre navigateur et allez sur start.spring.io. (https://start.spring.io/)

Étape 2 : Configurer le projet

1. **Project** : Choisissez *Maven Project*.
2. **Language** : Sélectionnez *Java*.
3. **Spring Boot Version** : Assurez-vous de choisir une version stable (par exemple, *3.1.0* ou plus récent).
4. **Project Metadata** : Remplissez les informations du projet :

- ***Group*** : le nom de votre groupe, par exemple **com.ifnti**.
- ***Artifact*** : le nom de votre projet, par exemple **my-spring-project**.
- ***Name*** et ***Description*** : ajustez-les selon vos préférences.
- ***Package Name*** : confirmé automatiquement en fonction de votre **Group** et **Artifact**, mais vous pouvez le modifier.

5. **Packaging** : Choisissez *Jar* (généralement recommandé pour les applications **Spring Boo**t).
6. **Java Version** : Sélectionnez la version de Java que vous utilisez (par exemple *17*).

Étape 3 : Ajouter les dépendances

1. Cliquez sur le bouton **Add Dependencies**. (A droite de l'écran)
2. Recherchez et sélectionnez **Spring Boot Starter** ou tout autre starter dont vous avez besoin, comme **spring-boot-starter-web** pour les applications web, **spring-boot-starter-data-jpa** pour l'accès aux bases de données, etc.

Étape 4 : Générer le projet

1. Cliquez sur **Generate** pour télécharger le projet sous forme d'archive ZIP.
2. Extrayez le fichier ZIP dans votre espace de travail.

Étape 5 : Importer le projet dans votre IDE

1. Ouvrez votre IDE (par exemple **Eclipse**).
2. Importez le projet Maven en sélectionnant le répertoire du projet extrait.
3. Une fois importé, Maven téléchargera automatiquement les dépendances.

Vous avez maintenant un projet **Spring Boot** prêt à l'emploi avec le **spring-boot-starter** de base !

Avec STS

Télécharger Spring Tools Suite (STS)

Pour télécharger et installer **Spring Tool Suite (STS)** sur Windows et Linux, suivez ces étapes :

1. Télécharger Spring Tool Suite

1. **Accédez à la page de téléchargement** : Ouvrez votre navigateur et allez sur https://spring.io/tools.
2. **Sélectionnez STS pour votre système d'exploitation** :
   - Choisissez **Windows** si vous êtes sous Windows.
   - Choisissez **Linux** si vous êtes sous Linux.
3. **Téléchargez l'archive** : Cliquez sur le lien pour télécharger l'archive ZIP ou TAR appropriée.

2. Installation sur Windows

Étape 1 : Extraire le fichier ZIP

1. Une fois le fichier téléchargé, trouvez l'archive ZIP (par exemple, spring-tool-suite-4.x.x.RELEASE.zip) dans votre dossier de téléchargements.
2. Faites un clic droit sur l'archive ZIP, puis sélectionnez **Extraire ici** (ou utilisez un outil comme WinRAR ou 7-Zip pour extraire).

Étape 2 : Démarrer STS

1. Allez dans le dossier extrait, puis ouvrez le dossier principal (par exemple, sts-4.x.x.RELEASE).
2. Double-cliquez sur le fichier **SpringToolSuite4.exe** pour lancer STS.

Étape 3 : Créer un raccourci (facultatif)

1. Pour faciliter l'accès, faites un clic droit sur SpringToolSuite4.exe, sélectionnez **Envoyer vers** > **Bureau (créer un raccourci)**.

3. Installation sur Linux

Étape 1 : Extraire le fichier TAR

1. Trouvez l'archive téléchargée (par exemple, spring-tool-suite-4.x.x.RELEASE.tar.gz).
2. Ouvrez un terminal et utilisez la commande suivante pour extraire le fichier dans un dossier de votre choix (par exemple, le répertoire opt pour les applications) :

   ```
   sudo tar -xvzf ~/Téléchargements/spring-tool-suite-4.x.x.RELEASE.tar.gz -C /opt
   ```

Étape 2 : Lancer STS

1. Allez dans le dossier où STS a été extrait (par exemple, /opt/sts-4.x.x.RELEASE).
2. Exécutez le fichier en lançant la commande suivante :

   ```
   /opt/sts-4.x.x.RELEASE/SpringToolSuite4
   ```

3. Vous pouvez aussi créer un alias dans votre profil ou ajouter STS au menu des applications pour un accès plus facile.

Étape 3 : Créer un raccourci (facultatif)

1. Créez un fichier .desktop pour un lancement rapide. Par exemple :

   ```
   sudo nano /usr/share/applications/spring-tool-suite.desktop
   ```

2. Collez ce contenu, en remplaçant le chemin d'icône si nécessaire :

   ```
   [Desktop Entry]
   Name=Spring Tool Suite
   Comment=Spring Tool Suite 4
   Exec=/opt/sts-4.x.x.RELEASE/SpringToolSuite4
   Icon=/opt/sts-4.x.x.RELEASE/icon.xpm
   Terminal=false
   ```

```
Type=Application
Categories=Development;IDE;
```

3. Enregistrez le fichier. Vous devriez maintenant pouvoir trouver STS dans votre menu d'applications.

Lancement et Configuration Initiale

Lors du premier démarrage, **STS** vous demandera de sélectionner un *workspace* (dossier de travail pour vos projets). Sélectionnez un dossier approprié et commencez à utiliser **Spring Tool Suite** !

Pour créer un projet Spring Boot avec **Spring Tool Suite** (STS) en utilisant le *starter* de base **spring-boot-starter**, suivez les étapes suivantes :

Étape 1 : Ouvrir Spring Tool Suite

1. Lancez **Spring Tool Suite** (STS) sur votre ordinateur.

Étape 2 : Créer un nouveau projet Spring Boot

1. Allez dans le menu **File** > **New** > **Spring Starter Project**.

Étape 3 : Configurer le projet

1. **Name** : Entrez le nom de votre projet, par exemple my-spring-project.
2. **Type** : Sélectionnez *Maven Project*.
3. **Packaging** : Choisissez *Jar* (recommandé pour les applications Spring Boot).
4. **Java Version** : Choisissez la version de Java que vous utilisez (par exemple, *17* ou *20*).
5. **Group** et **Artifact** : Renseignez le *Group* (par exemple, com.example) et l'*Artifact* (nom du projet, par exemple my-spring-project).

Étape 4 : Ajouter les dépendances

1. Cliquez sur le bouton **Next**.
2. Dans la section des dépendances, recherchez et sélectionnez **Spring Boot Starter** ou tout autre starter dont vous avez besoin :
    - **Spring Boot Starter Web** pour une application web.
    - **Spring Boot Starter Data JPA** pour l'accès aux bases de données.
3. Ajoutez les dépendances nécessaires en cochant les cases correspondantes.

Étape 5 : Terminer la configuration et créer le projet

1. Cliquez sur **Finish** pour que STS crée le projet avec les configurations spécifiées.

Étape 6 : Exécuter le projet

1. Une fois le projet créé, vous pouvez le lancer en faisant un clic droit sur le projet dans l'onglet *Project Explorer*, puis en sélectionnant **Run As** > **Spring Boot App**.

Votre projet **Spring Boot** est maintenant configuré et prêt à être exécuté dans **Spring Tool Suite**, avec les dépendances de base automatiquement téléchargées et configurées !

Structure du projet :

Lorsqu'on crée un projet Spring Boot avec Spring Initializr ou Spring Tool Suite, une structure de projet minimale est générée. Les principaux éléments de cette structure et leur utilité :

1. src/main/java

Ce répertoire contient le code source principal de votre application.

- **Classe principale** : Une classe contenant l'annotation **@SpringBootApplication** est créée automatiquement (par exemple, **MySpringProjectApplication.java**). Elle sert de point d'entrée de l'application Spring Boot. Cette classe contient une méthode main() qui lance l'application avec **SpringApplication.run(...)**.
- **Autres packages** : Au sein de **src/main/java,** vous pouvez créer des sous-dossiers ou packages pour organiser le code selon les modules ou composants (ex. **controller**, **service**, **repository**).

2. src/main/resources

Ce dossier contient les fichiers de ressources utilisés par l'application.

- **application.properties** ou **application.yml** : Fichier de configuration de l'application, où vous pouvez définir des paramètres comme les informations de connexion à la base de données, les paramètres de port du serveur, etc.
- **static/** : Dossier réservé aux fichiers statiques, comme les fichiers CSS, JavaScript, et images pour les applications web.
- **templates/** : Utilisé pour les fichiers de templates HTML si vous utilisez des moteurs de templates comme Thymeleaf ou Freemarker.

3. src/test/java

Ce répertoire est destiné aux tests unitaires et d'intégration. Par défaut, une classe de test de base est créée, avec un test contextuel minimal pour s'assurer que l'application peut se charger correctement.

4. pom.xml (ou build.gradle)

Ce fichier est le cœur de la configuration du projet avec Maven (ou Gradle). Il contient :

- **Dépendances** : Les starters Spring Boot et autres dépendances nécessaires pour le projet.
- **Plugins** : Par exemple, le plugin spring-boot-maven-plugin, qui permet d'exécuter et de packager facilement l'application.

- **Propriétés** : Des informations comme la version de Java et des configurations spécifiques.

5. target/ (après compilation)

Ce dossier est généré automatiquement par Maven lors de la compilation du projet. Il contient tous les fichiers de sortie, y compris le fichier exécutable (JAR ou WAR) une fois l'application packagée.

Structure du projet avec Spring Boot

NB : **Not to lose track** ! Je vous ai déjà parler de l'organisation du projet dans Java EE (Rappelez-vous!)

Structurer un projet Spring Boot de manière efficace permet de rendre le code plus lisible, modulaire, et facile à maintenir, surtout pour les projets de grande envergure. Il est important de noter que Spring Boot est souvent utilisé dans le développement d'applications web, bien que ses usages ne se limitent pas uniquement à ce domaine.Ensuite, la plupart des applications doivent interagir avec des sources de données externes, comme des bases de données, d'autres programmes, ou même le système de fichiers.Donc ,même si on ne connaît pas encore les classes qui seront produites , voici une structure commune et recommandée pour un projet Spring Boot, en suivant une organisation basée sur les rôles des composants :

Structure de base

```
src/
├── main/
│   ├── java/
│   │   └── com/
│   │       └── example/
│   │           └── myproject/
│   │               ├── MyProjectApplication.java  // Classe principale
│   │               ├── controller/              // Couches de contrôle (endpoints REST)
```

```
│   │       ├── service/          // Logique métier
│   │       ├── repository/       // Accès aux données (DAO)
│   │       ├── model/            // Modèles de données ou entités JPA
│   │       ├── config/           // Classes de configuration
│   │       ├── dto/              // Objets de transfert de données
│   │       └── exception/        // Gestion des exceptions
│   └── resources/
│       ├── application.properties      // Fichier de configuration principal
│       ├── static/                 // Fichiers statiques (CSS, JS, images)
│       └── templates/              // Modèles HTML (si utilisés)
└── test/                           // Tests unitaires et d'intégration
```

Détail des packages

1. **Classe principale**
    - `MyProjectApplication.java` : C'est la classe de lancement de l'application, annotée avec `@SpringBootApplication`. Elle contient la méthode `main()` et peut aussi contenir des configurations globales si nécessaire.
2. **controller/** (couche de contrôle)
    - Ce package contient les *contrôleurs* qui définissent les *endpoints* REST de l'application.
    - Les classes dans ce package sont annotées avec `@RestController` ou `@Controller`.
    - Exemples : `UserController.java`, `ProductController.java`.
3. **service/** (couche de service ou logique métier)
    - Ce package contient la logique métier de l'application. La couche de service sépare la logique d'affaires des autres couches, comme la couche de contrôle et la couche de persistance.
    - Les classes dans ce package sont annotées avec `@Service`.
    - Exemples : `UserService.java`, `ProductService.java`.
4. **repository/** (couche d'accès aux données)
    - Contient les *repositories* (DAO) pour interagir avec la base de données.
    - Les interfaces dans ce package étendent généralement `JpaRepository` ou `CrudRepository`.
    - Exemples : `UserRepository.java`, `ProductRepository.java`.

5. **model/** (couche de modèle ou entités)
    - Contient les classes qui représentent les modèles de données de l'application, souvent appelées *entités*.
    - Les classes sont annotées avec `@Entity` pour les entités JPA.
    - Exemples : `User.java`, `Product.java`.
6. **config/** (configuration)
    - Ce package contient toutes les classes de configuration spécifiques à l'application, comme les configurations de sécurité, de CORS, ou de persistance.
    - Exemples : `SecurityConfig.java`, `DatabaseConfig.java`.
7. **dto/** (objets de transfert de données)
    - Les *Data Transfer Objects* (DTO) sont utilisés pour transférer des données entre les couches de l'application sans exposer directement les entités.
    - Exemples : `UserDTO.java`, `ProductDTO.java`.
8. **exception/** (gestion des exceptions)
    - Ce package contient les classes pour la gestion centralisée des exceptions.
    - Peut inclure des exceptions personnalisées, ainsi qu'une classe annotée avec `@ControllerAdvice` pour gérer les exceptions globalement.
    - Exemples : `CustomException.java`, `GlobalExceptionHandler.java`.

Ressources et configurations

1. **application.properties** ou **application.yml**
    - Fichier de configuration de l'application Spring Boot, où vous définissez les paramètres, comme l'URL de la base de données, les informations de port, et d'autres configurations de votre application.
2. **static/** et **templates/**
    - `static/` : Contient des fichiers statiques, comme les images, les fichiers CSS, et JavaScript.
    - `templates/` : Contient les templates HTML si vous utilisez des moteurs de template comme Thymeleaf.
3. **test/**
    - Contient les tests unitaires et les tests d'intégration pour chaque couche de l'application.
    - Suivez la même structure que dans `src/main/java`, par exemple, avec des packages comme `controller`, `service`, `repository`.

Conseils pour structurer efficacement le projet

- **Organisez chaque couche** : Séparez bien la logique métier, les contrôleurs et les accès aux données dans des packages distincts.
- **Utilisez les DTO** : Utilisez des DTO pour échanger les données sans exposer directement les entités de la base de données.

- **Centralisez la configuration** : Placez toute configuration complexe dans des classes sous `config/`.
- **Exception handling** : Utilisez un gestionnaire d'exception global pour gérer les erreurs de manière centralisée.

Alors créons ces packags ……

Retour sur le fichier application.propeties

Le fichier `application.properties` de Spring Boot contient les configurations essentielles pour le comportement de l'application. Une liste détaillée de propriétés courantes et de leur utilisation est :

1. **Configuration du serveur**

- **Port du serveur** (`server.port`) :
  - Définit le port sur lequel l'application va écouter les requêtes HTTP. Par défaut, Spring Boot utilise le port 8080.
  - Exemple : `server.port=8081` (l'application écoute sur le port 8081).
- **Chemin de contexte** (`server.servlet.context-path`) :
  - Spécifie le chemin de base pour toutes les URL de l'application. Cela peut être utile pour isoler votre application avec un chemin dédié, comme `/api`.
  - Exemple : `server.servlet.context-path=/api` (toutes les URL seront préfixées par `/api`, ex. `http://localhost:8080/api/...`).
- **SSL** (HTTPS) :
  - **`server.ssl.enabled`** : Active ou désactive SSL (HTTPS). Si activé, l'application acceptera les connexions sécurisées.
  - **`server.ssl.key-store`** : Définit le chemin du keystore contenant le certificat SSL.
  - **`server.ssl.key-store-password`** : Définit le mot de passe pour accéder au keystore.
  - **`server.ssl.key-store-type`** : Spécifie le type de keystore (ex. `JKS` ou `PKCS12`).
  - Exemple :

```
server.ssl.enabled=true
server.ssl.key-store=classpath:keystore.p12
server.ssl.key-store-password=motdepasse
server.ssl.key-store-type=PKCS12
```

2. Configuration de la base de données

- **URL de la base de données** (`spring.datasource.url`) :
  - Spécifie l'URL de connexion à la base de données, dépendant du SGBD (ex. `jdbc:mysql://localhost:3306/nomdelabase` pour MySQL).

- **Nom d’utilisateur et mot de passe** (`spring.datasource.username` et `spring.datasource.password`) :
    - Définit les informations d'authentification pour accéder à la base de données.
- **Pilote de connexion** (`spring.datasource.driver-class-name`) :
    - Définit le pilote JDBC spécifique à utiliser pour se connecter à la base de données (ex. `com.mysql.cj.jdbc.Driver` pour MySQL, `org.postgresql.Driver` pour PostgreSQL).
- **Stratégie de création de schéma** (`spring.jpa.hibernate.ddl-auto`) :
    - Définit la manière dont Hibernate gère la création du schéma. Les options incluent :
        - **`none`** : Ne fait aucune modification de schéma.
        - **`update`** : Modifie les tables pour s’adapter aux entités sans perdre de données existantes.
        - **`create`** : Supprime et recrée le schéma chaque fois que l'application démarre.
        - **`create-drop`** : Supprime le schéma lorsque l'application s'arrête.
- **Dialecte Hibernate** (`spring.jpa.properties.hibernate.dialect`) :
    - Spécifie le dialecte Hibernate, qui est spécifique à chaque SGBD. Cela aide Hibernate à générer des requêtes SQL adaptées au SGBD.
    - Exemple pour MySQL : `spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect`.

3. Configuration JPA et Hibernate

- **Affichage des requêtes SQL** (`spring.jpa.show-sql`) :
    - Si activé, affiche dans la console les requêtes SQL exécutées par Hibernate, ce qui est utile pour le débogage.
    - Exemple : `spring.jpa.show-sql=true`.
- **Formatage des requêtes SQL** (`spring.jpa.properties.hibernate.format_sql`) :
    - Si activé, formate les requêtes SQL de manière lisible dans la console (avec des retours à la ligne et des indentations).
    - Exemple : `spring.jpa.properties.hibernate.format_sql=true`.
- **ddl-auto** (`spring.jpa.hibernate.ddl-auto`) :
    - Permet à Hibernate de gérer la création/modification de schéma pour adapter la base aux entités JPA.
    - Options : `none`, `create`, `create-drop`, `update`.

4. Configuration de la gestion des journaux (logging)

- **Niveau de log global** (`logging.level.root`) :
    - Définit le niveau de log pour toute l'application. Les niveaux disponibles sont :

    - `TRACE` : Niveau le plus bas, avec des informations très détaillées.
    - `DEBUG` : Informations utiles pour le débogage.
    - `INFO` : Messages informatifs standards (niveau par défaut).
    - `WARN` : Avertissements concernant des problèmes potentiels.
    - `ERROR` : Erreurs graves nécessitant une attention immédiate.
- **Niveau de log pour un package spécifique** :
    - Permet de définir le niveau de log pour des packages particuliers.
    - Exemple : `logging.level.com.example=DEBUG`.
- **Fichier de log** (`logging.file.name`) :
    - Définit un fichier où les logs sont enregistrés. Ce fichier se trouve dans le répertoire racine de l'application.
    - Exemple : `logging.file.name=logs/app.log`.

5. Configuration des sessions

- **Durée de la session** (`server.servlet.session.timeout`) :
    - Définit la durée de vie d'une session utilisateur. Accepte des valeurs comme `30m` (30 minutes) ou `2h` (2 heures).
    - Exemple : `server.servlet.session.timeout=30m`.
- **Type de stockage des sessions** (`spring.session.store-type`) :
    - Spécifie où stocker les sessions. Options possibles : `redis`, `jdbc`, `mongodb`, etc.
    - Exemple : `spring.session.store-type=redis`.

6. Configuration de l’envoi d’e-mails

- **Serveur SMTP** :
    - **`spring.mail.host`** : Définit l'adresse du serveur SMTP.
    - **`spring.mail.port`** : Spécifie le port du serveur SMTP.
    - **`spring.mail.username`** et **`spring.mail.password`** : Informations d'authentification pour le serveur SMTP.
- **Paramètres de sécurité** :
    - **`spring.mail.properties.mail.smtp.auth`** : Active l'authentification SMTP.
    - **`spring.mail.properties.mail.smtp.starttls.enable`** : Active le cryptage TLS pour les emails sortants.
    - Exemple :

```
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=utilisateur@gmail.com
spring.mail.password=motdepasse
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

7. Configuration de Spring Security

- **Utilisateur et mot de passe par défaut** :
  - **`spring.security.user.name`** et **`spring.security.user.password`** : Spécifient un utilisateur et un mot de passe par défaut pour l'authentification basique dans Spring Security.
  - Exemple :

```
spring.security.user.name=admin
spring.security.user.password=motdepasse
```

8. Configuration de CORS (Cross-Origin Resource Sharing)

- **Autoriser CORS** :
  - **`spring.web.cors.allowed-origin-patterns`** : Définit quels domaines sont autorisés à accéder aux ressources de l'application via CORS.
  - Exemple pour autoriser tous les domaines : `spring.web.cors.allowed-origin-patterns=*`.

9. Autres paramètres

- **Nom de l'application** (`spring.application.name`) :
  - Définit le nom de l'application, utile pour les journaux ou l'identification dans un système distribué.
  - Exemple : `spring.application.name=MyApp`.
- **Profil actif** (`spring.profiles.active`) :
  - Spécifie quel profil Spring est actif (ex. `dev`, `prod`). Cela permet de charger des configurations différentes selon l'environnement.
  - Exemple : `spring.profiles.active=dev`.

10. Internationalisation (i18n)

- **Langue et format des messages** :
  - **`spring.messages.basename`** : Définit le nom de base pour les fichiers de messages. Par exemple, `messages` chargera `messages_fr.properties`, `messages_en.properties`, etc.
  - **`spring.messages.encoding`** : Définit l'encodage des fichiers de messages (par ex., UTF-8).
  - **`spring.messages.cache-duration`** : Définit la durée en secondes pour laquelle les messages sont mis en cache.

Encore plus de détaille sur les prophéties de la base des données

Dans Spring Boot, le fichier **`application.properties`** permet de configurer divers types de bases de données, qu'elles soient **SQL** (relationnelles) ou **NoSQL** (non relationnelles). Quelques exemples de configurations pour les bases de données **SQL** et **NoSQL** les plus courantes.

Bases de Données SQL

1. MySQL

MySQL est une base de données relationnelle couramment utilisée avec Spring Boot.

Exemples
```
spring.datasource.url=jdbc:mysql://localhost:3306/nomdelabase
spring.datasource.username=utilisateur
spring.datasource.password=motdepasse
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

2. PostgreSQL

PostgreSQL est une autre base de données relationnelle populaire, appréciée pour sa conformité aux standards SQL et ses fonctionnalités avancées.

Exemples
```
spring.datasource.url=jdbc:postgresql://localhost:5432/nomdelabase
spring.datasource.username=utilisateur
spring.datasource.password=motdepasse
spring.datasource.driver-class-name=org.postgresql.Driver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

3. Oracle

Oracle est une base de données commerciale SQL. Spring Boot supporte sa configuration comme suit :

Exemples
```
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:XE
spring.datasource.username=utilisateur
spring.datasource.password=motdepasse
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.OracleDialect
```

4. Microsoft SQL Server

Microsoft SQL Server est une base de données souvent utilisée dans des environnements Windows.

Exemples

```
spring.datasource.url=jdbc:sqlserver://localhost:1433;databaseName=nomdelabase
spring.datasource.username=utilisateur
spring.datasource.password=motdepasse
spring.datasource.driver-class-name=com.microsoft.sqlserver.jdbc.SQLServerDriver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.SQLServerDialect
```

5. H2 (Base de données en mémoire)

H2 est une base de données en mémoire légère, idéale pour les tests.

Exemples

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

spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.h2.console.enabled=true  # Permet d'accéder à la console H2 via /h2-
console
```

Bases de Données NoSQL

1. MongoDB

MongoDB est une base de données NoSQL orientée documents, adaptée aux applications qui gèrent des structures de données variées.

Pour MongoDB, Spring Boot nécessite le starter `spring-boot-starter-data-mongodb`.

Exemples

```
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=nomdelabase
spring.data.mongodb.username=utilisateur
spring.data.mongodb.password=motdepasse
```

2. Cassandra

Cassandra est une base de données NoSQL distribuée orientée colonnes, idéale pour des grandes quantités de données.

Avec Spring Boot, il faut le starter `spring-boot-starter-data-cassandra`.

Exemples

```
spring.data.cassandra.contact-points=localhost
spring.data.cassandra.port=9042
spring.data.cassandra.keyspace-name=nomdukeyspace
spring.data.cassandra.username=utilisateur
spring.data.cassandra.password=motdepasse
spring.data.cassandra.schema-action=CREATE_IF_NOT_EXISTS
```

3. Redis

Redis est une base de données NoSQL en mémoire, orientée clé-valeur. Elle est particulièrement utile pour le caching et le stockage de sessions.

Pour Redis, ajoutez le starter `spring-boot-starter-data-redis`.

Exemples

```
spring.redis.host=localhost
spring.redis.port=6379
spring.redis.password=motdepasse

# Configuration avancée (timeout, pool, etc.)
spring.redis.timeout=60000
spring.redis.jedis.pool.max-active=10
spring.redis.jedis.pool.max-idle=5
spring.redis.jedis.pool.min-idle=1
```

4. Neo4j

Neo4j est une base de données orientée graphes, utilisée pour modéliser des relations complexes entre des données.

Pour Neo4j, utilisez le starter `spring-boot-starter-data-neo4j`.

```
Exemples :
spring.neo4j.uri=bolt://localhost:7687
spring.neo4j.authentication.username=utilisateur
spring.neo4j.authentication.password=motdepasse
```

5. Elasticsearch

Elasticsearch est une base de données NoSQL orientée recherche et texte, idéale pour des recherches complexes sur des volumes de données importants.

Ajoutez le starter `spring-boot-starter-data-elasticsearch`.

Exemples :

```
spring.elasticsearch.uris=http://localhost:9200
spring.elasticsearch.username=utilisateur
spring.elasticsearch.password=motdepasse
```

Affichage de Hello world du projet crée