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

Félicitation si vous tenez jusque là !!! passons alors aux projets

Projet 1 : Création de l’api avec Spring boot

Enoncé : Créer une API REST de type CRUD avec Spring Boot

Définition :

Une **API REST** (Application Programming Interface - Representational State Transfer) est une interface qui permet à différentes applications de communiquer entre elles via le protocole HTTP, en respectant des principes et des contraintes spécifiques. En simplifiant, une API REST permet à une application cliente (comme

un site web ou une application mobile) de demander des données ou des actions à un serveur de manière standardisée et structurée.

Concepts clés d'une API REST

1. **Ressources** :
    - Dans une API REST, les données (par exemple, des utilisateurs, des produits, des commandes) sont appelées des "ressources".
    - Chaque ressource est identifiée par une URL unique (URI - Uniform Resource Identifier). Par exemple, une ressource "utilisateur" peut être accessible à l'URI /users.
2. **Méthodes HTTP** :
    - Les opérations CRUD (Create, Read, Update, Delete) sont réalisées en utilisant les verbes HTTP standards :
        - **POST** : créer une nouvelle ressource.
        - **GET** : récupérer des informations sur une ou plusieurs ressources.
        - **PUT** ou **PATCH** : mettre à jour une ressource existante.
        - **DELETE** : supprimer une ressource.
    - Par exemple, pour créer un nouvel utilisateur, une application enverra une requête POST à l'URL /users.
3. **Stateless (Sans état)** :
    - Une API REST est sans état, ce qui signifie que chaque requête envoyée par le client au serveur est indépendante. Le serveur ne stocke aucune information sur l'état de la connexion de l'utilisateur entre les requêtes. Toutes les informations nécessaires pour traiter une requête doivent être incluses dans celle-ci.
4. **Utilisation des réponses en JSON** :
    - Les APIs REST échangent souvent des données en format JSON (JavaScript Object Notation), qui est léger, lisible, et facile à manipuler. Par exemple, la réponse d'une API pour obtenir un utilisateur pourrait ressembler à ceci :

```
{
  "id": 1,
  "nom": "toto",
  "email": "toto@example.com"
}
```

5. **Uniformité des URLs et des actions** :
    - Les APIs REST ont des conventions standardisées pour nommer les routes et utiliser les méthodes HTTP, ce qui permet aux développeurs de comprendre facilement comment accéder aux ressources et effectuer des actions spécifiques.

Exemple de fonctionnement d'une API REST

Prenons un exemple simple d'une API REST pour gérer des utilisateurs :

- **POST /users** : Créer un nouvel utilisateur.
- **GET /users** : Récupérer tous les utilisateurs.
- **GET /users/{id}** : Récupérer un utilisateur spécifique par son identifiant.
- **PUT /users/{id}** : Mettre à jour les informations d'un utilisateur spécifique.
- **DELETE /users/{id}** : Supprimer un utilisateur spécifique.

Etapes à suivre et le choix des bons starters

Pour créer une API REST de type CRUD avec Spring Boot, il est essentiel de suivre une série d'étapes afin de structurer correctement l'architecture de l'application et d'assurer que les fonctionnalités CRUD (Create, Read, Update, Delete) soient bien implémentées. Un exemple de guide détaillé pour chaque étape :

1. Initialisation du projet Spring Boot

- **Utilisation de Spring Initializr** : On peut créer un projet Spring Boot en utilisant Spring Initializr. Ce service permet de sélectionner les dépendances nécessaires pour notre projet. Pour une API REST CRUD, les dépendances importantes sont :
    - **Spring Web** pour le développement de services web et d'API REST.
    - **Spring Data JPA** pour faciliter l'interaction avec la base de données.
    - **Base de données** : Vous pouvez choisir une base de données intégrée (comme H2) ou une base de données relationnelle telle que PostgreSQL ou MySQL.

2. Configuration de l'application

- **Fichier application.properties ou application.yml** : Dans ce fichier de configuration, on définit les paramètres de la base de données (URL, nom d'utilisateur, mot de passe, etc.). Cela permet à l'application Spring Boot de se connecter à la base de données et d'y stocker ou récupérer des données.
  NB : Je vous ai déjà montré comment le faire
- **Configurer JPA** : Le framework Spring Data JPA s'appuie sur des configurations spécifiques pour gérer les entités et les tables de la base de données. En activant la création et la mise à jour automatique des tables, JPA peut générer la structure de base de données à partir de nos classes d'entité.

3. Création des entités

- **Classe entité** : Une entité est une classe Java qui représente une table dans la base de données. Chaque champ de la classe correspond à une colonne de la table. On utilise les annotations JPA comme @Entity, @Id et @GeneratedValue pour indiquer qu'il s'agit d'une entité et pour spécifier la clé primaire.

- **Gestion des relations** : Si les données sont liées entre elles (par exemple, un utilisateur peut avoir plusieurs articles), on configure les relations (ex. : @OneToMany, @ManyToOne) en fonction du modèle de la base de données.

4. Création du repository

- **Interface Repository** : Spring Data JPA fournit des interfaces comme JpaRepository qui permettent de manipuler les entités sans écrire de code SQL. En créant une interface qui étend JpaRepository, on obtient immédiatement des méthodes pour effectuer des opérations CRUD de base (comme save, findById, findAll, deleteById).
- **Méthodes personnalisées** : Il est aussi possible de définir des requêtes personnalisées dans le repository en utilisant des conventions de nommage ou des annotations comme @Query.

5. Création des services

- **Service métier** : La couche service contient la logique métier de l'application. Elle utilise le repository pour accéder aux données, traite les données si nécessaire, puis les retourne au contrôleur.
- **Gestion des transactions** : Les services peuvent également gérer les transactions en utilisant l'annotation @Transactional, ce qui garantit que toutes les opérations sur la base de données s'exécutent de manière atomique.

6. Création des contrôleurs REST

- **Contrôleur** : La couche contrôleur est la porte d'entrée de l'API. Elle reçoit les requêtes HTTP, interagit avec le service, et retourne des réponses HTTP.
- **Annotations REST** : En utilisant @RestController, @GetMapping, @PostMapping, @PutMapping et @DeleteMapping, on peut facilement créer des points de terminaison pour chaque opération CRUD :
    - **Create** (POST) : pour créer une nouvelle ressource.
    - **Read** (GET) : pour lire les données (soit toutes les données, soit une donnée spécifique).
    - **Update** (PUT ou PATCH) : pour mettre à jour une ressource existante.
    - **Delete** (DELETE) : pour supprimer une ressource.

7. Gestion des erreurs

- **Gestion des exceptions** : Pour garantir que les erreurs sont bien gérées et que des messages clairs sont renvoyés, on peut utiliser @ExceptionHandler dans le contrôleur ou créer un gestionnaire d'exceptions global avec @ControllerAdvice.
- **Personnalisation des réponses d'erreur** : En personnalisant les réponses d'erreur, on rend l'API plus conviviale pour les utilisateurs. Par exemple, on peut renvoyer un code 404 pour une ressource non trouvée, ou 400 pour une requête mal formulée.

8. Documentation et tests

- **Documentation avec Swagger** : Swagger est un outil qui génère automatiquement une documentation interactive pour l'API, permettant de tester les points de terminaison directement depuis un navigateur.
- **Tests** : L'écriture de tests unitaires et d'intégration est essentielle pour vérifier que chaque composant fonctionne correctement. On peut utiliser JUnit et MockMvc pour tester les contrôleurs, services, et autres composants.

Résumé du flux de données dans une API CRUD avec Spring Boot

1. **Requête** : L'utilisateur envoie une requête HTTP (GET, POST, PUT ou DELETE) vers le point de terminaison de l'API.
2. **Contrôleur** : Le contrôleur reçoit la requête, la transmet au service, et attend une réponse.
3. **Service** : Le service exécute la logique métier, interagit avec le repository pour manipuler les données, puis retourne les résultats au contrôleur.
4. **Repository** : Le repository effectue les opérations CRUD sur la base de données.
5. **Réponse** : Le contrôleur renvoie une réponse HTTP à l'utilisateur, incluant les données demandées ou un message de confirmation.

Les starters

Pour créer un projet API REST avec Spring Boot, l'utilisation de Spring Initializr est une méthode simple et efficace. Cela permet de configurer le projet avec les bons starters nécessaires pour une API REST robuste et performante. Les starters appropriés et leur utilité dans le projet sont :

1. Accéder à Spring Initializr

Pour démarrer, rendez-vous sur Spring Initializr ou utilisez votre IDE (comme Eclipse ou Spring Tools Suite) qui intègre souvent Spring Initializr pour la création de projets.

2. Configuration de base

Sur Spring Initializr :

- **Project** : Choisissez **Maven** (ou Gradle si vous préférez).
- **Language** : Sélectionnez **Java**.
- **Spring Boot Version** : Optez pour la dernière version stable de Spring Boot.
- **Group** et **Artifact** : Donnez un nom unique à votre application.
- **Packaging** : Sélectionnez **Jar** pour une API REST.
- **Java Version** : Choisissez une version compatible avec la dernière version de Spring Boot (Java 17 ou plus est recommandé).

3. Choix des dépendances (starters)

Pour une API REST CRUD, sélectionnez les starters suivants :

a. Spring Web

- **Dépendance principale** pour la création d'une API REST. Elle inclut Spring MVC, nécessaire pour construire des contrôleurs REST et gérer les requêtes HTTP.
- Ce starter permet de créer des points de terminaison HTTP et de manipuler des requêtes GET, POST, PUT, et DELETE.

b. Spring Data JPA

- Permet d'accéder à une base de données de manière simple et efficace en utilisant l'ORM (Object-Relational Mapping) JPA.
- Inclut JpaRepository, qui facilite l'interaction avec la base de données pour les opérations CRUD sans avoir à écrire beaucoup de code SQL.

c. Base de données :

- Sélectionnez la dépendance en fonction de la base de données que vous souhaitez utiliser :
    - **H2 Database** : Une base de données embarquée utile pour le développement rapide et les tests. Elle ne nécessite pas de configuration externe.
    - **MySQL Driver** : Pour utiliser une base de données MySQL.
    - **PostgreSQL Driver** : Pour utiliser une base de données PostgreSQL.

d. Spring Boot DevTools (optionnel mais recommandé)

- Ajoute le rechargement automatique des modifications et une expérience de développement améliorée.
- Lorsqu'une modification est effectuée dans le code, le serveur redémarre automatiquement, permettant un développement plus rapide.

e. Lombok (optionnel mais recommandé)

- Simplifie le code en générant automatiquement des méthodes getter et setter, des constructeurs, et d'autres éléments courants à l'aide d'annotations.
- Réduit le code répétitif, facilitant la lisibilité et la maintenance.

f. Spring Security (optionnel, pour les API sécurisées)

- Permet d'ajouter facilement des mécanismes d'authentification et d'autorisation.
- Si vous souhaitez protéger votre API, ce starter peut être utile pour mettre en place des stratégies de sécurité, comme l'authentification par JWT (JSON Web Token).

4. Générer et télécharger le projet

Une fois les dépendances sélectionnées :

1. Cliquez sur **Generate** pour télécharger le projet.
2. Décompressez l'archive téléchargée.
3. Importez le projet dans votre IDE favori.

5. Configuration du projet

Dans le fichier application.properties ou application.yml (dans le dossier src/main/resources), configurez les paramètres de la base de données. Par exemple :

```
//Paramètres de la base de données (exemple pour PostgreSQL)
spring.datasource.url=jdbc:postgresql://localhost:5432/nom_de_base
spring.datasource.username=nom_utilisateur
spring.datasource.password=mot_de_passe

//Configuration JPA
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Ces configurations permettent à Spring Boot de se connecter à la base de données et d'utiliser JPA pour créer automatiquement les tables si elles n'existent pas déjà.

6. Lancer le projet

Dans votre IDE, vous pouvez maintenant exécuter l'application en lançant la classe principale (celle annotée avec @SpringBootApplication). Spring Boot démarrera le serveur intégré (par défaut, Tomcat sur le port 8080) et exposera l'API.

Vous avez maintenant une base solide pour développer une API REST CRUD avec Spring Boot !

Retour sur la configuration de JPA

Ces configurations sont des propriétés de Spring Boot, spécifiquement pour le module Spring Data JPA. Elles permettent de configurer le comportement de Hibernate, l'ORM (Object-Relational Mapping) utilisé par défaut par Spring Data JPA pour interagir avec une base de données relationnelle. Le détail de chaque propriété et leur utilité :

1. spring.jpa.hibernate.ddl-auto=update

Cette propriété contrôle le comportement de **création et mise à jour du schéma de base de données** par Hibernate. Voici ce que signifie chaque option courante :

- **none** : Désactive toute action de gestion du schéma par Hibernate. La base de données reste inchangée.
- **validate** : Hibernate vérifie si le schéma de la base de données correspond aux entités JPA, mais ne modifie pas la base. Cela peut être utile pour s'assurer que les entités et le schéma de la base sont cohérents sans faire de changements.
- **update** : Hibernate modifie le schéma de la base de données pour correspondre aux entités, en ajoutant ou modifiant les tables et colonnes si nécessaire. Cette option est utile en développement car elle permet d'éviter des mises à jour manuelles de la base.

- **create** : Hibernate supprime toutes les tables existantes et crée un nouveau schéma basé sur les entités. Cela supprime les données à chaque redémarrage, donc à éviter en production.
- **create-drop** : Similaire à create, mais en plus, le schéma est supprimé à la fin de la session. Pratique pour les tests automatisés.

Exemple d'utilisation :

```
spring.jpa.hibernate.ddl-auto=update
```

Dans cet exemple, update permet de garder la base de données à jour avec les entités sans effacer les données. Cela est pratique pour le développement, mais en production, une option comme validate est souvent préférable.

2. spring.jpa.show-sql=true

Cette propriété contrôle si les requêtes SQL générées par Hibernate doivent être affichées dans la console. Si elle est activée, chaque requête SQL envoyée à la base de données sera visible dans la console, ce qui est très utile pour **déboguer** et **vérifier** les requêtes.

Exemple d'utilisation :

```
spring.jpa.show-sql=true
```

En activant cette option, vous verrez toutes les requêtes SQL exécutées par votre application dans la console. Cela permet de vérifier que les requêtes correspondent à ce que vous souhaitez, et peut aider à optimiser les performances.

3. spring.jpa.properties.hibernate.format_sql=true

Cette propriété permet de **formater** les requêtes SQL affichées dans la console pour qu'elles soient **plus lisibles**. Par défaut, les requêtes sont affichées sur une seule ligne, ce qui peut être difficile à lire, surtout pour les requêtes complexes. En activant cette option, les requêtes SQL seront **indiquées avec des retours à la ligne** et une **indentation**, ce qui facilite l’analyse des requêtes générées.

Exemple d'utilisation :

```
spring.jpa.properties.hibernate.format_sql=true
```

En activant cette option, chaque requête SQL sera affichée dans un format plus lisible, ce qui permet d'analyser la structure et la logique des requêtes.

Et si on parle de notre starter magique : Lombok

Lombok est une bibliothèque Java qui facilite l'écriture de code en réduisant le code répétitif, notamment en générant automatiquement certaines méthodes. C’est particulièrement utile en Java, où les classes peuvent vite devenir longues et verbeuses en raison de nombreux **getters**, **setters**, **constructeurs**, et **autres** méthodes utilitaires.

1. Fonctionnalités principales de Lombok

Lombok utilise des **annotations** pour générer dynamiquement du code lors de la compilation. Les annotations les plus couramment utilisées et leur fonction :

- **@Getter et @Setter** : En ajoutant ces annotations au-dessus des champs d'une classe, Lombok génère automatiquement les méthodes **getter** et **setter** pour ces champs.
    - Par exemple, au lieu d'écrire manuellement les méthodes pour obtenir et modifier la valeur d'un attribut, il suffit d'ajouter @Getter et @Setter. Cela simplifie énormément le code, surtout dans des classes avec plusieurs attributs.
- **@NoArgsConstructor, @AllArgsConstructor, et @RequiredArgsConstructor** : Ces annotations permettent de générer automatiquement des constructeurs.
    - **@NoArgsConstructor** : Crée un constructeur sans arguments.
    - **@AllArgsConstructor** : Crée un constructeur qui prend tous les champs comme paramètres.
    - **@RequiredArgsConstructor** : Crée un constructeur pour les champs marqués comme final (ou ceux annotés **@NonNull**). C'est utile pour les classes où certains champs doivent être initialisés à la création de l'objet.
- **@ToString** : Génère automatiquement la méthode **toString()**, qui renvoie une chaîne de caractères contenant les valeurs de tous les champs de l'objet. Cela facilite le débogage et l’affichage des objets.
- **@EqualsAndHashCode** : Génère les méthodes **equals()** et **hashCode()**, ce qui est important pour comparer les objets de manière précise. Cela est particulièrement utile pour les structures de données, comme les collections, qui dépendent de ces méthodes.
- **@Data** : Cette annotation combine plusieurs annotations (@Getter, @Setter, @ToString, @EqualsAndHashCode, et @RequiredArgsConstructor) pour les classes qui ont des attributs de base et où toutes ces méthodes sont généralement nécessaires.

2. Comment Lombok améliore la lisibilité et la maintenance du code

En utilisant Lombok, le code devient beaucoup plus lisible et facile à entretenir. Au lieu de pages entières de code pour les méthodes d'accès, les constructeurs, et les méthodes de comparaison, seules quelques lignes d'annotations suffisent. Voici comment cela facilite la maintenance :

- **Moins de code à lire** : Les développeurs peuvent se concentrer sur la logique métier sans être distraits par des méthodes accessoires, rendant la classe plus lisible.
- **Réduction des erreurs humaines** : En générant automatiquement ces méthodes, Lombok minimise les erreurs dues à des fautes de frappe ou à des oublis dans l'implémentation de méthodes de base.

- **Évolutivité** : Si vous ajoutez ou retirez des attributs, Lombok s’assure que les méthodes générées sont toujours à jour, éliminant la nécessité de modifier manuellement les méthodes d'accès, les constructeurs, etc.

3. Exemple d'utilisation de Lombok

Imaginons une classe **Utilisateur** sans Lombok :

```
public class Utilisateur {
    private String nom;
    private String email;

    public Utilisateur() {}

    public Utilisateur(String nom, String email) {
        this.nom = nom;
        this.email = email;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @Override
    public String toString() {
        return "Utilisateur{" +
                "nom='" + nom + '\'' +
                ", email='" + email + '\'' +
                '}';
    }
}
```

Cette classe est assez simple, mais déjà plusieurs lignes de code sont consacrées aux getters, setters, constructeurs, et méthodes toString(). Avec **Lombok**, la même classe serait bien plus concise :

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Utilisateur {
    private String nom;
    private String email;
}
```

Soka, tu ne vois pas que ce code est magique 😊 ,en tout cas **Sankara** a vient de confirmer?

Ici, l'annotation **@Data** remplace tous les getters, setters, toString, equals, et hashCode. Les annotations @NoArgsConstructor et @AllArgsConstructor ajoutent automatiquement les constructeurs nécessaires. Résultat : une classe plus lisible et moins de code à maintenir.

Haha, d'accord ! On a suffisamment théorisé, passons à l'action. 🚀 Prêt à coder maintenant ? 😄

Étape 1 : Créer un projet Spring Boot

Utilisez Spring Initializr pour créer un nouveau projet avec les bonnes dépendances

Je vous laisse faire car vous êtes capables !!!!!

Étape 2 : Configurer la base de données

Dans le fichier application.properties (ou application.yml), configurez les paramètres de votre base de données PostgreSQL.

Je compte sur vous pour cà car nous en avons parler à maintes fois !!!

Étape 3 : Créer une entité

Définition et Explication :

Une **entité** en Java, dans le contexte de la **persistante** des données avec JPA (Java Persistence API), représente une classe qui est mappée à une table de base de données. Chaque objet de cette classe correspond à une ligne dans la table, et les attributs de l'objet correspondent aux colonnes de la table.

1. Définition d'une entité

En Java, une classe peut être marquée comme **entité** en utilisant l'annotation @Entity. Elle indique à JPA que cette classe est mappée à une table de base de données.

Exemple d'entité

Prenons l'exemple d'une classe User :

```
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Column;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Table;
import javax.persistence.OneToMany;
import java.util.List;

@Entity
@Table(name = "users")  // Nom de la table dans la base de données
```

```
public class User {

    @Id  // Désigne l'attribut comme clé primaire de la table
    @GeneratedValue(strategy = GenerationType.IDENTITY)  // Stratégie pour générer les valeurs de la clé primaire (ici auto-incrément)
    private Long id;

    @Column(name = "name", nullable = false)  // Indique que cet attribut est mappé à une colonne 'name' dans la table
    private String name;

    @Column(name = "email", unique = true, nullable = false)  // 'email' doit être unique et ne peut pas être null
    private String email;

    @OneToMany(mappedBy = "user")  // Relation One-to-Many avec l'entité 'Order'
    private List<Order> orders;

    // Getters et Setters
}
```

2. Explication détaillée des annotations utilisées

Voici les annotations les plus courantes dans une entité et leur signification :

@Entity

- **Description** : Indique que la classe est une entité JPA et doit être persistée dans la base de données.
- **Exemple** : @Entity

@Table

- **Description** : Spécifie le nom de la table à laquelle l'entité est mappée. Si cette annotation n'est pas utilisée, le nom de la classe (en minuscules) est pris comme nom de la table par défaut.
- **Exemple** : @Table(name = "users") indique que cette classe sera mappée à la table users dans la base de données.

@Id

- **Description** : Marque un champ comme étant la **clé primaire** de l'entité. Chaque entité doit avoir un champ unique identifié par @Id.
- **Exemple** : @Id private Long id;

@GeneratedValue

- **Description** : Indique que la valeur de la clé primaire doit être générée automatiquement, soit par un mécanisme de base de données (comme un auto-increment).
- **Exemple** :

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

    - GenerationType.IDENTITY : Indique que la base de données va gérer l'auto-incrément de la clé primaire.

@Column

- **Description** : Permet de configurer les détails de la colonne dans la base de données à laquelle un attribut est mappé.
    - name : Spécifie le nom de la colonne dans la table.
    - nullable : Indique si la colonne peut avoir une valeur nulle.
    - unique : Spécifie que la colonne doit avoir des valeurs uniques dans la base de données.
- **Exemple** :

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

@OneToMany

- **Description** : Déclare une relation **un-à-plusieurs** entre deux entités. Cette annotation est utilisée dans la classe "un" pour indiquer que cette entité a plusieurs instances de l'entité "plusieurs".
- **Exemple** :

```
@OneToMany(mappedBy = "user")
private List<Order> orders;
```

    - mappedBy : Spécifie le côté "inverse" de la relation, c'est-à-dire que l'entité Order a un champ user qui fait référence à l'entité User.

@ManyToOne

- **Description** : Déclare une relation **plusieurs-à-un**. C'est l'opposée de @OneToMany. Elle est utilisée dans la classe "plusieurs" pour indiquer qu'un objet de cette classe fait référence à un seul objet de la classe "un".
- **Exemple** :

```
@ManyToOne
@JoinColumn(name = "user_id")  // Spécifie la colonne qui fait référence à la clé primaire de l'entité "un"
private User user;
```

@ManyToMany

- **Description** : Déclare une relation **plusieurs-à-plusieurs** entre deux entités.
- **Exemple** :

```
@ManyToMany
@JoinTable(
  name = "user_roles",
  joinColumns = @JoinColumn(name = "user_id"),
  inverseJoinColumns = @JoinColumn(name = "role_id")
)
private Set<Role> roles;
```

- Cette relation nécessite une table de jointure (ici user_roles) pour stocker les associations.

@JoinColumn

- **Description** : Spécifie la colonne qui est utilisée pour faire référence à une autre table dans une relation. Elle est souvent utilisée avec @ManyToOne, @OneToMany ou @ManyToMany.
- **Exemple** :

```
@JoinColumn(name = "user_id")
private User user;
```

@Transient

- **Description** : Indique que cet attribut **ne doit pas être persisté dans la base de données**. Il est ignoré par JPA et n'est pas mappé à une colonne de la table.
- **Exemple** :

```
@Transient
private int temporaryValue;  // Cet attribut ne sera pas mappé en base de données.
```

@Lob

- **Description** : Indique que l'attribut doit être stocké sous forme de **"Large Object"**, c'est-à-dire qu'il peut contenir de grandes quantités de données (par exemple des fichiers, des images).
- **Exemple** :

```
@Lob
private String description;
```

@Version

- **Description** : Permet de mettre en œuvre une stratégie de **contrôle de version** pour gérer la concurrence optimiste. Il est généralement utilisé pour les entités qui nécessitent un suivi des modifications.
- **Exemple** :

```
@Version
private Long version;
```

Exemple complet d'entité avec annotations

Voici une classe d'entité User complète avec différentes annotations :

```
import javax.persistence.*;
import java.util.List;

@Entity
@Table(name = "users")
public class User {

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

```
    @Column(name = "name", nullable = false)
    private String name;

    @Column(name = "email", unique = true, nullable = false)
    private String email;

    @OneToMany(mappedBy = "user")
    private List<Order> orders;

    @Transient
    private int temporaryValue;  // Cet attribut ne sera pas persistant

    @Version
    private Long version;  // Pour gérer le contrôle de version (concurrence optimiste)

    // Getters et Setters
}
```

Appliquons Entity pour notre projet API REST

Définissons une entité pour représenter les données dans la base de données. Dans cet exemple, nous allons créer une entité User.

Sans la technique de Lombok

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

@Entity // si pas @Table ,alors le nom de la table sera le nom de la classe
@Table(name = "users", schema = "apiTables") // Nom de la table et du schéma
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

@Column(name = "username", nullable = false, unique = true) // Personnalisation de la colonne

/**
```

name : Spécifie le nom de la colonne dans la table (par défaut, le nom de l'attribut est utilisé).

nullable : Définit si la colonne peut accepter des valeurs nulles (par défaut true).

unique : Définit si la colonne doit être unique (par défaut false).

```
**/
    private String name;
    private String email;

    // Getter pour id
    public Long getId() {
        return id;
    }
```

```
    // Setter pour id
    public void setId(Long id) {
        this.id = id;
    }

    // Getter pour name
    public String getName() {
        return name;
    }

    // Setter pour name
    public void setName(String name) {
        this.name = name;
    }

    // Getter pour email
    public String getEmail() {
        return email;
    }

    // Setter pour email
    public void setEmail(String email) {
        this.email = email;
    }
}
```

Avec Lombok

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.Getter;
import lombok.Setter;

@Entity
@Table(name = "users", schema = "apiTables")
@Getter
@Setter
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
@Column(name = "username", nullable = false, unique = true)
    private String name;
    private String email;
}
```

Étape 4 : Créer un Repository

Définition et EXPLICATION

Un **repository** est une interface ou une classe qui s'occupe des interactions avec la base de données. Dans le cadre de Spring Boot et JPA, un repository est généralement une interface qui étend JpaRepository ou CrudRepository, fournissant des méthodes pour

effectuer des opérations CRUD (Create, Read, Update, Delete) sans que le développeur n'ait besoin d'écrire de code SQL.

Fonction principale :

- **Gestion des données** : Le repository sert d'interface pour accéder, modifier, sauvegarder et supprimer des données dans la base de données.

En créant un repository, vous déléguez à Spring Data JPA la gestion des requêtes SQL nécessaires aux opérations de base sur les données.

Exemple

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
    // Vous pouvez ajouter des méthodes personnalisées ici si besoin, par exemple :
    Optional<User> findByEmail(String email);
}
```

Ici, UserRepository est une interface qui hérite de JpaRepository<User, Long>, ce qui signifie qu'elle permet de manipuler des entités User identifiées par un Long (l'ID de l'utilisateur). Sans aucune implémentation supplémentaire, elle hérite de méthodes comme save, findAll, findById, delete, etc.

Un mot sur JpaRepository

CrudRepository

CrudRepository est une interface de base de Spring Data JPA qui fournit des méthodes essentielles pour les opérations CRUD (Create, Read, Update, Delete). Elle est générique et peut être utilisée pour manipuler n'importe quel type d'entité.

Les principales méthodes de CrudRepository incluent :

- save(S entity): Sauvegarde ou met à jour une entité.
- findById(ID id): Récupère une entité par son identifiant.
- findAll(): Récupère toutes les entités.
- deleteById(ID id): Supprime une entité par son identifiant.
- delete(S entity): Supprime une entité spécifiée.

2. JpaRepository

JpaRepository est une interface qui **étend** CrudRepository et ajoute des fonctionnalités supplémentaires spécifiques à JPA (Java Persistence API). Elle fournit des méthodes supplémentaires pour des opérations avancées sur les données et inclut également des options de pagination et de tri.

Les méthodes supplémentaires de JpaRepository incluent :

- findAll(Pageable pageable): Récupère toutes les entités avec pagination.

- findAll(Sort sort): Récupère toutes les entités avec tri.
- saveAll(Iterable<S> entities): Sauvegarde plusieurs entités en une seule opération.
- flush(): Force l'application des modifications en base de données.
- saveAndFlush(S entity): Sauvegarde une entité et force l'application immédiate en base de données.

Différence entre CrudRepository et JpaRepository

Aspect	CrudRepository	JpaRepository
Fonctions	Opérations CRUD de base	CRUD + pagination, tri, flush
Spécificité JPA	Générique, pas spécifique à JPA	Spécifique aux fonctionnalités JPA
Utilisation commune	Applications basiques CRUD	Applications avec besoins avancés en données

Exemple d'utilisation

Si vous voulez utiliser seulement les fonctions CRUD de base, **CrudRepository** peut suffire. Si votre application nécessite des fonctionnalités supplémentaires, comme la pagination ou le tri, **JpaRepository** est plus adapté.

Appliquons repository pour notre projet API REST

Utilisons JpaRepository pour gérer les opérations CRUD de User.

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
}
```

Explication du Code

Dans cet exemple, on définit une interface **UserRepository** qui hérite de **JpaRepository**. Cette interface est utilisée pour manipuler des entités **User** (le nom de l'entité qu'on vient de définir) dans la base de données en profitant des fonctionnalités de **Spring Data JPA**.

Décomposons ce code ligne par ligne par ligne :

```
import org.springframework.data.jpa.repository.JpaRepository;
```

- Cette ligne importe l'interface **JpaRepository** de Spring Data JPA, qui fournit une collection de méthodes pour les opérations de base sur la base de données. JpaRepository étend l'interface CrudRepository, donc elle hérite des méthodes

CRUD de base (Create, Read, Update, Delete) et y ajoute des fonctionnalités avancées telles que la pagination et le tri.

```
public interface UserRepository extends JpaRepository<User, Long> {
```

- Ici, nous définissons UserRepository comme une interface, pas une classe. Cette interface **hérite de JpaRepository**, ce qui signifie qu'elle va automatiquement disposer de toutes les méthodes que JpaRepository offre, comme save, findAll, findById, delete, etc.
- JpaRepository<User, Long> : Nous indiquons à JpaRepository deux éléments :
    - **L'entité gérée** : User, qui est probablement une classe annotée avec @Entity pour représenter une table dans la base de données.
    - **Le type de l'ID** : Long, indiquant que l'identifiant unique pour chaque User est de type Long (généralement une clé primaire).

Que fait UserRepository ?

Cette interface ne nécessite pas de code d'implémentation pour fonctionner. En l'héritant de JpaRepository, Spring Data JPA fournit automatiquement les implémentations pour les opérations de base sur les entités User. Cela permet d'interagir avec la base de données sans écrire de code SQL.

Exemple d'utilisation des méthodes JpaRepository

Dans votre application, vous pourriez injecter UserRepository et utiliser ses méthodes pour effectuer des opérations sur la base de données :

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;
import java.util.Optional;

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public List<User> getAllUsers() {
        return userRepository.findAll(); // Récupère tous les utilisateurs
    }

    public Optional<User> getUserById(Long id) {
        return userRepository.findById(id); // Récupère un utilisateur par ID
    }

    public User createUser(User user) {
        return userRepository.save(user); // Sauvegarde un nouvel utilisateur
    }

    public void deleteUser(Long id) {
        userRepository.deleteById(id); // Supprime un utilisateur par ID
    }
}
```

Pourquoi utiliser JpaRepository ?

Utiliser JpaRepository vous permet :

- D'accéder aux fonctionnalités de CRUD et de gestion des transactions sans écrire de code supplémentaire.
- D'ajouter des méthodes personnalisées si nécessaire. Par exemple, vous pourriez ajouter une méthode pour trouver un utilisateur par email :

  Optional<User> findByEmail(String email);

Spring Data JPA générera automatiquement le code pour cette méthode en fonction de son nom.

Étape 5 : Créer un Service

Définition

En architecture logicielle, et particulièrement dans les applications Spring Boot, un **service** représente une couche logique qui contient la **logique métier** de l'application. La couche de service fait le lien entre les **contrôleurs** (qui gèrent les requêtes HTTP) et les **repositories** (qui gèrent l'accès aux données). On appelle la **logique métier** (ou **logique d'affaires** en français) représente l'ensemble des règles et processus qui définissent comment les données de l'application doivent être manipulées et traitées pour répondre aux besoins de l'entreprise ou du domaine d'activité. Elle décrit ce que fait l'application en termes de fonctionnalités et de transformations de données, indépendamment des aspects techniques (comme l'interface utilisateur ou le stockage des données).

Exemple de logique métier

Supposons une application de commerce en ligne. Voici quelques exemples de logique métier courante :

- **Calcul des remises** : En fonction du type de client ou du montant total de la commande, une remise peut être appliquée.
- **Validation d'inventaire** : Lorsqu'un client passe une commande, l'application doit vérifier si les articles sont en stock.
- **Validation des informations utilisateur** : Lors de l'inscription, l'application doit s'assurer que les informations fournies par le client (comme l'adresse e-mail ou le mot de passe) respectent certaines règles.
- **Calcul des frais de livraison** : En fonction de l'adresse de livraison, du poids et des dimensions du colis, l'application calcule les frais d'expédition.

Exemple de logique métier dans un service

Disons que nous avons une application de vente qui doit appliquer une remise de 10 % pour les commandes dont le total dépasse 100 $.

> **Classe de service** : La logique pour appliquer la remise serait placée dans une classe de service, qui encapsule cette règle.

```
package com.example.demo.service;


import com.example.demo.model.Order;
import org.springframework.stereotype.Service;


@Service
public class OrderService {


    public double calculateTotalPrice(Order order) {
        double total = order.getSubTotal();


        // Logique métier pour appliquer la remise
        if (total > 100) {
            total *= 0.9;  // Applique une remise de 10%
        }


        return total;
    }
}
```

A**ppliquons service pour notre projet API REST**

Création du service pour gérer la logique métier. Le service interagit avec le repository.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
```

```
import java.util.Optional;

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    public Optional<User> getUserById(Long id) {
        return userRepository.findById(id);
    }

    public User createUser(User user) {
        return userRepository.save(user);
    }

    public User updateUser(Long id, User userDetails) {
        return userRepository.findById(id).map(user -> {
            user.setName(userDetails.getName());
            user.setEmail(userDetails.getEmail());
            return userRepository.save(user);
        }).orElseThrow(() -> new RuntimeException("User not found"));
    }

    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }
}
```

Allô ! **Moustabchirath**!!! tu es perdue dans le code ?

Analysons ce code ligne par ligne.

import org.springframework.beans.factory.annotation.Autowired;

Cette ligne importe l'annotation **@Autowired** de Spring. **@Autowired** est utilisée pour injecter des dépendances automatiquement dans une classe. Dans notre exemple, elle est utilisée pour injecter une instance de **UserRepository** dans la classe **UserService**.

import org.springframework.stereotype.Service;

Cette ligne importe l'annotation **@Service** de Spring. **@Service** indique que cette classe fait partie de la couche service, c'est-à-dire qu'elle contient la logique métier de l'application. Spring détecte cette annotation et gère cette classe comme un "**bean**", permettant l'injection de dépendances.

import java.util.List;
import java.util.Optional;

Ces lignes importent les classes **List** et **Optional** de Java. **List** est utilisée pour stocker et gérer des collections d'objets (dans ce cas, des objets **User**). **Optional** est un conteneur qui peut contenir une valeur ou être vide, permettant de gérer plus facilement les valeurs nulles.

```
@Service
public class UserService {
```

@Service marque la classe **UserService** comme un service, un composant de Spring qui gère la logique métier. **UserService** est une classe publique, ce qui signifie qu'elle est accessible depuis d'autres classes.

```
    @Autowired
    private UserRepository userRepository;
```

- @Autowired injecte une instance de **UserRepository** dans **UserService**. Cela permet à **UserService** d'accéder aux méthodes de UserRepository pour interagir avec la base de données.
- **UserRepository** est un repository qui interagit directement avec la base de données pour la gestion des utilisateurs. Il est déclaré private pour limiter son accès à l'intérieur de UserService.

Méthode getAllUsers()

```
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }
```

Cette méthode publique getAllUsers() retourne une liste de tous les utilisateurs (List<User>). Elle appelle findAll() sur **userRepository**, une méthode fournie par JpaRepository qui récupère tous les enregistrements d'utilisateurs de la base de données.

Méthode getUserById()

```
    public Optional<User> getUserById(Long id) {
        return userRepository.findById(id);
    }
```

La méthode getUserById(Long id) retourne un utilisateur spécifique basé sur son identifiant (id). findById(id) est une méthode de JpaRepository qui retourne un Optional<User> :

- Si l'utilisateur est trouvé, Optional contiendra l'utilisateur.
- Si non, Optional sera vide, permettant ainsi de gérer facilement les cas où l'utilisateur est introuvable.

Méthode createUser()

```
    public User createUser(User user) {
        return userRepository.save(user);
    }
```

createUser(User user) prend un objet User en paramètre et l'enregistre dans la base de données. save(user) est une méthode de JpaRepository qui ajoute l'utilisateur dans la base si c'est un nouvel enregistrement ou le met à jour s'il existe déjà. Elle retourne l'objet User qui a été enregistré.

Méthode updateUser()

```
    public User updateUser(Long id, User userDetails) {
```

```
        return userRepository.findById(id).map(user -> {
            user.setName(userDetails.getName());
            user.setEmail(userDetails.getEmail());
            return userRepository.save(user);
        }).orElseThrow(() -> new RuntimeException("User not found"));
    }
```

- La méthode updateUser(Long id, User userDetails) permet de mettre à jour un utilisateur existant en fonction de son id.
- userRepository.findById(id) cherche d'abord l'utilisateur par son id.
- Si l'utilisateur est trouvé (map() est exécuté sur l'utilisateur), on met à jour ses attributs name et email avec les valeurs de userDetails, puis on enregistre les modifications avec save(user).
- Si l'utilisateur n'est pas trouvé (Optional est vide), orElseThrow() lève une exception RuntimeException avec le message "User not found".

Méthode deleteUser()

```
    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }
```

- deleteUser(Long id) supprime un utilisateur de la base de données en fonction de son id.
- deleteById(id) est une méthode de JpaRepository qui effectue cette suppression. Si l'identifiant n'existe pas, rien n'est supprimé et aucune exception n'est levée.

Étape 6 : Créer un contrôleur REST

Terminons en créant le contrôleur un contrôleur pour exposer les **endpoints CRUD**.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserService userService;

    // GET tous les utilisateurs
    @GetMapping
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    // GET utilisateur par ID
    @GetMapping("/{id}")
```

```
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        return userService.getUserById(id)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }

    // POST pour créer un nouvel utilisateur
    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.createUser(user);
    }

    // PUT pour mettre à jour un utilisateur existant
    @PutMapping("/{id}")
    public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody User userDetails) {
        try {
            User updatedUser = userService.updateUser(id, userDetails);
            return ResponseEntity.ok(updatedUser);
        } catch (RuntimeException e) {
            return ResponseEntity.notFound().build();
        }
    }

    // DELETE pour supprimer un utilisateur
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        userService.deleteUser(id);
        return ResponseEntity.noContent().build();
    }
}
```

Explications du code

Ce code est une classe contrôleur **REST** Spring Boot qui gère les opérations CRUD pour les entités User. Voici une explication détaillée ligne par ligne.

Importations de classes

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
```

Ces lignes importent les classes nécessaires pour le fonctionnement du contrôleur :

- **Autowired** : pour injecter des dépendances de manière automatique.
- **ResponseEntity** : pour manipuler et personnaliser les réponses HTTP.
- **RestController** et les autres annotations de Spring (GetMapping, PostMapping, etc.) : pour créer des points d'entrée REST et mapper les requêtes HTTP.
- List : pour gérer des collections d'utilisateurs.

Déclaration de la classe UserController

```
@RestController
```

```
@RequestMapping("/api/users")
public class UserController {
```

- **@RestController** : indique que cette classe est un contrôleur REST et que chaque méthode renverra une réponse JSON.
- **@RequestMapping("/api/users")** : mappe les requêtes dont l'URL commence par /api/users à cette classe, définissant ainsi le chemin de base pour toutes les routes.

Injection de dépendance pour UserService

```
@Autowired
private UserService userService;
```

- **@Autowired** injecte une instance de UserService pour que UserController puisse accéder aux opérations définies dans ce service.

Méthode getAllUsers - Récupérer tous les utilisateurs

```
@GetMapping
public List<User> getAllUsers() {
    return userService.getAllUsers();
}
```

- **@GetMapping** : cette méthode est accessible via une requête HTTP GET sur /api/users.
- **Retourne une liste d'utilisateurs** récupérée via userService.getAllUsers().

Méthode getUserById - Récupérer un utilisateur par ID

```
@GetMapping("/{id}")
public ResponseEntity<User> getUserById(@PathVariable Long id) {
    return userService.getUserById(id)
        .map(ResponseEntity::ok)
        .orElse(ResponseEntity.notFound().build());
}
```

- **@GetMapping("/{id}")** : cette méthode est accessible via une requête HTTP GET sur /api/users/{id}, où {id} est un identifiant d'utilisateur dynamique.
- **@PathVariable Long id** : lie le {id} de l'URL au paramètre id de la méthode.
- **userService.getUserById(id)** : tente de récupérer un utilisateur par son id. Retourne un ResponseEntity.ok(user) si l'utilisateur est trouvé, sinon un ResponseEntity.notFound() si absent.

Méthode createUser - Créer un nouvel utilisateur

```
@PostMapping
public User createUser(@RequestBody User user) {
    return userService.createUser(user);
}
```

- **@PostMapping** : cette méthode est accessible via une requête HTTP POST sur /api/users.
- **@RequestBody User user** : lie le corps de la requête JSON au paramètre user.
- **userService.createUser(user)** : utilise le service pour créer un nouvel utilisateur et le renvoie dans la réponse.

Méthode updateUser - Mettre à jour un utilisateur existant

```
@PutMapping("/{id}")
public ResponseEntity<User> updateUser(@PathVariable Long id, @RequestBody User userDetails) {
    try {
        User updatedUser = userService.updateUser(id, userDetails);
        return ResponseEntity.ok(updatedUser);
    } catch (RuntimeException e) {
        return ResponseEntity.notFound().build();
    }
}
```

- **@PutMapping("/{id}")** : cette méthode est accessible via une requête HTTP PUT sur /api/users/{id}, où {id} est l'ID de l'utilisateur.
- **@PathVariable Long id** : lie le {id} de l'URL au paramètre id.
- **@RequestBody User userDetails** : lie le corps de la requête JSON contenant les nouvelles informations de l'utilisateur au paramètre userDetails.
- **Essaye de mettre à jour l'utilisateur** via userService.updateUser(id, userDetails). Si l'utilisateur n'est pas trouvé, il renvoie une réponse 404 Not Found.

Méthode deleteUser - Supprimer un utilisateur

```
@DeleteMapping("/{id}")
public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
    userService.deleteUser(id);
    return ResponseEntity.noContent().build();
}
```

- **@DeleteMapping("/{id}")** : cette méthode est accessible via une requête HTTP DELETE sur /api/users/{id}.
- **@PathVariable Long id** : lie le {id} de l'URL au paramètre id.
- **Supprime l'utilisateur** via userService.deleteUser(id).
- **Retourne ResponseEntity.noContent()** indiquant une réponse 204 No Content si la suppression est réussie.

Étape 7 : Tester l'API

Pour tester cette API REST, vous pouvez utiliser des outils comme Postman, cURL, ou encore Swagger pour effectuer des requêtes HTTP et vérifier que l'API fonctionne comme prévu. Voici les étapes pour chaque type de test :

1. Utiliser Postman pour tester l'API

Installation : Téléchargez Postman depuis postman.com et créez un compte si nécessaire.

Tester chaque point de terminaison

- **GET /api/users** - Récupérer tous les utilisateurs
    1. Ouvrez Postman, créez une nouvelle requête GET.
    2. Saisissez l'URL : http://localhost:8080/api/users.
    3. Cliquez sur "Send".
    4. Vous devriez voir une réponse avec une liste d'utilisateurs en JSON (vide au début, si vous n'avez pas encore ajouté d'utilisateurs).
- **GET /api/users/{id}** - Récupérer un utilisateur par ID
    1. Créez une requête GET avec l'URL http://localhost:8080/api/users/1 (remplacez 1 par un ID existant).
    2. Cliquez sur "Send".
    3. Vous verrez les détails de l'utilisateur avec cet ID. Si l'utilisateur n'existe pas, la réponse devrait être une erreur 404 Not Found.
- **POST /api/users** - Créer un nouvel utilisateur
    1. Créez une requête POST avec l'URL http://localhost:8080/api/users.
    2. Dans l'onglet "Body", sélectionnez raw et JSON.
    3. Entrez les données de l'utilisateur dans le format JSON. Exemple :

```
{
  "name": "toyi",
  "email": "toyi@example.com"
}
```

    4. Cliquez sur "Send". Vous devriez recevoir les détails de l'utilisateur créé avec un nouvel id.
- **PUT /api/users/{id}** - Mettre à jour un utilisateur existant
    1. Créez une requête PUT avec l'URL http://localhost:8080/api/users/1 (remplacez 1 par l'ID de l'utilisateur à mettre à jour).
    2. Dans "Body" > raw > JSON, entrez les nouvelles informations :

```
{
  "name": "toyi Updated",
  "email": "toyi.updated@example.com"
}
```

3. Cliquez sur "Send". Si l'utilisateur existe, vous devriez voir les informations mises à jour ; sinon, un message d'erreur 404 Not Found.

- **DELETE /api/users/{id}** - Supprimer un utilisateur

1. Créez une requête DELETE avec l'URL http://localhost:8080/api/users/1.
2. Cliquez sur "Send".
3. Si l'utilisateur est supprimé avec succès, vous obtiendrez une réponse 204 No Content.

2. Utiliser cURL pour tester l'API

Si vous préférez la ligne de commande, vous pouvez utiliser cURL.

- **GET tous les utilisateurs** :

```
curl -X GET http://localhost:8080/api/users
```

- **GET utilisateur par ID** :

```
curl -X GET http://localhost:8080/api/users/1
```

- **POST pour créer un utilisateur** :

```
curl -X POST http://localhost:8080/api/users -H "Content-Type: application/json" -d '{"name": "toyi", "email": "toyi@example.com"}'
```

- **PUT pour mettre à jour un utilisateur** :

```
curl -X PUT http://localhost:8080/api/users/1 -H "Content-Type: application/json" -d '{"name": "toyi Updated", "email": "toyi.updated@example.com"}'
```

- **DELETE pour supprimer un utilisateur** :

```
curl -X DELETE http://localhost:8080/api/users/1
```

3. Configurer Swagger pour une documentation interactive

Si vous avez intégré Swagger dans votre projet (en ajoutant la dépendance springdoc-openapi-ui), vous pouvez utiliser Swagger pour tester facilement chaque point de terminaison dans une interface web.

1. Ajoutez la dépendance suivante dans votre fichier pom.xml :

```
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-ui</artifactId>
   <version>1.5.9</version>
</dependency>
```

2. Lancez votre application Spring Boot.
3. Ouvrez un navigateur et allez à http://localhost:8080/swagger-ui.html.

4. Vous verrez une documentation interactive de votre API où vous pourrez tester chaque point de terminaison.

4. Tester avec JUnit et Mockito

Vous pouvez également écrire des tests automatisés avec **JUnit** et **Mockito** pour vérifier le bon fonctionnement de votre API. Cela permet d'assurer la fiabilité de l'API lors de modifications futures.

Un exemple de test pour la méthode getAllUsers pourrait ressembler à ceci :

```
@SpringBootTest
@AutoConfigureMockMvc
public class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testGetAllUsers() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.get("/api/users"))
                .andExpect(status().isOk())
                .andExpect(content().contentType("application/json"));
    }
}
```

Cela effectue une requête GET et vérifie que le statut est bien 200 OK et que le type de contenu est JSON.

Déploiement de l'application

Pour déployer une application Spring Boot, plusieurs options s'offrent à vous, en fonction de votre environnement et des outils disponibles. Voici les étapes générales pour différents types de déploiement :

1. Déploiement local avec un fichier JAR

1. **Générez un fichier JAR** : Utilisez Maven ou Gradle pour créer le fichier JAR de l'application. Par exemple, avec Maven :

```
mvn clean package
```

Le fichier JAR se trouvera généralement dans le dossier target.

- **Exécutez le fichier JAR** : Une fois le JAR généré, exécutez-le avec Java :

```
java -jar target/nom-de-votre-application.jar
```

Cela lancera l'application sur le port défini dans application.properties ou application.yml (par défaut, le port 8080).

2. Déploiement sur un serveur de conteneurs (Tomcat, Jetty, etc.)

Si votre application n'est pas un fichier JAR exécutable mais un fichier WAR, vous pouvez la déployer sur un serveur de conteneurs.

1. **Générez un fichier WAR** : Modifiez la configuration dans pom.xml pour générer un fichier WAR. Puis générez le WAR :

mvn clean package

Déployez le WAR : Copiez le fichier WAR dans le dossier de déploiement de votre serveur de conteneurs (par exemple, webapps pour Apache Tomcat). Le serveur chargera automatiquement le fichier WAR et démarre l'application.

3. Déploiement sur un service Cloud (Heroku, AWS, Google Cloud, Azure)

- **Heroku** : Poussez le projet sur Heroku en utilisant Git. Assurez-vous d'avoir un fichier Procfile contenant :

web: java -jar target/nom-de-votre-application.jar

AWS Elastic Beanstalk : Créez un fichier .zip du projet et utilisez la CLI AWS pour déployer ou déployez directement depuis la console.

- **Google App Engine** : Créez un fichier app.yaml et déployez l'application en utilisant gcloud app deploy.
- **Microsoft Azure** : Utilisez Azure App Service ou Azure Spring Apps pour déployer l'application.

Configuration du projet Spring Boot pour générer un fichier WAR au lieu d'un JAR

Pour modifier la configuration de votre projet Spring Boot pour générer un fichier WAR au lieu d'un JAR, vous devez ajuster le fichier pom.xml et votre classe principale Spring Boot. Voici les étapes à suivre :

Étape 1 : Modifier le packaging dans le pom.xml

Dans votre fichier pom.xml, changez le type de packaging de jar à war :

<packaging>war</packaging>

Étape 2 : Ajouter la dépendance pour le support Tomcat en tant que conteneur

Par défaut, Spring Boot utilise Tomcat en tant que serveur intégré. Pour permettre un déploiement en tant que fichier WAR, il est conseillé d'ajouter cette dépendance et de marquer spring-boot-starter-tomcat comme fourni (provided) pour éviter de le dupliquer dans le conteneur.

Dans les dépendances, ajoutez :

```
<dependencies>
    <!-- Autres dépendances... -->

    <!-- Tomcat en mode "provided" -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

Étape 3 : Modifier la classe principale pour étendre SpringBootServletInitializer

Pour rendre votre application compatible avec un conteneur de servlets (comme Tomcat ou Jetty), votre classe principale Spring Boot doit étendre **SpringBootServletInitializer** et implémenter la méthode configure.

Exemple :

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

@SpringBootApplication
public class MonApplication extends SpringBootServletInitializer {
```

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

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(MonApplication.class);
    }
}
```

Étape 4 : Compiler le projet pour générer le fichier WAR

Une fois ces modifications apportées, vous pouvez compiler le projet pour générer le fichier WAR en exécutant :

```
mvn clean package
```

Le fichier **.war** sera créé dans le dossier **target** du projet et pourra être déployé sur un serveur de conteneurs comme Tomcat, Jetty ou autre.

Lancement automatique de l’application Spring boot déployer en local

Chercher vous même !!!!!!!!!!!

Projet 2

Dans ce projet , nous allons passer à une vitesse supérieur jusqu'à ralentir au niveau du contrôleur car la création des **Entity**,**repository**,**service** n'ont plus de secret pour vous.Même le contrôleur lui-même n'a plus de secret mais il y a une légère différence d'implémentation puisqu'ici nous voulons retourner les vues et non de JSON .Prêt ? Partons

Énoncé du projet

Créer une application **CRUD** de gestion de produits en utilisant **Spring Boot** avec les starters **Spring Web**, **Spring Data JPA**, **PostgresSQL**, **Thymeleaf** et **Bootstrap**. Ce projet comprendra un modèle des utilisateurs avec les champs suivants :

name,email,age et utilisera **Thymeleaf** pour les vues.

Avant projet : parlons de Thymeleaf

1. Qu'est-ce que Thymeleaf ?

Thymeleaf est un moteur de templates Java qui permet de créer des vues HTML (ou autres types de documents) dynamiques pour des applications web. Il est souvent intégré avec Spring Boot, car il simplifie la génération de contenu côté serveur en liant directement le HTML aux objets Java.

2. Pourquoi utiliser Thymeleaf ?

Thymeleaf a été conçu pour être à la fois convivial pour les développeurs et les designers :

- **Simplicité** : La syntaxe est intuitive et se rapproche de l'HTML.
- **Prévisualisation directe** : Les pages Thymeleaf peuvent être affichées en tant que fichiers HTML statiques, facilitant ainsi la visualisation pour les designers avant intégration.
- **Intégration avec Spring** : Thymeleaf fonctionne très bien avec Spring Boot, facilitant l'injection de données dans les vues.
- **Sécurisé et puissant** : Il propose des expressions puissantes pour manipuler les données et est conçu pour être sécurisé par défaut (contre les attaques XSS par exemple).

3. Configurer Thymeleaf avec Spring Boot

Avec **Spring Boot**, Thymeleaf est généralement intégré par défaut. Il suffit d'ajouter la dépendance suivante dans le fichier pom.xml :

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

Spring Boot configurera automatiquement Thymeleaf comme moteur de rendu pour les vues.

4. Structure d'un projet Thymeleaf

Dans un projet Spring Boot, Thymeleaf s’attend à trouver les templates dans le dossier src/main/resources/templates.

5. Syntaxe de base de Thymeleaf

Thymeleaf utilise des attributs HTML spécifiques pour intégrer les expressions Java. Les attributs les plus courants sont :

- **th:text** : Pour afficher du texte.
- **th:href** : Pour définir un lien.
- **th:src** : Pour définir la source d’une image.
- **th:each** : Pour les boucles.
- **th:if / th:unless** : Pour les conditions.

Exemple de template Thymeleaf basique (index.html) :

<!DOCTYPE html>

<html **xmlns:th="http://www.thymeleaf.org"**>

<head>

<title>Thymeleaf </title>

</head>

<body>

</body>

</html>

suite ……

Créations du projet

Etape 1 : Configuration du Projet Spring Boot

Utilisez Spring Initializr pour générer votre projet avec les dépendances suivantes :

- **Spring Web** : pour créer l'API REST et gérer les requêtes HTTP.
- **Spring Data JPA** : pour l'intégration avec la base de données.
- **PotgresSQL Driver** : pour la connexion à PostgresSQL ou **MySQL Driver** : pour la connexion à MySQLou
- **Thymeleaf** : pour le rendu des vues côté serveur.
- **Lombok** : pour simplifier le code des entités (exemple. : **getters/setters** automatiques).

Étape 2. Configuration de la Base de Données MySQL ou PostgresSQL

Je vous fais confiance pour la configuration

NB : N'oublier pas de créer des packages

Étape 3. Créer l'Entité User

Je vous fais confiance ……...

Étape 4. Créer le Repository **User**Repository

Je vous fais confiance…….

Étape 4. Créer du service

Je vous fais aussi confiance mais voici un exemple :

```
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
```

```java
import org.springframework.stereotype.Service;

import com.ifnti.gestion.models.User;
import com.ifnti.gestion.reposotories.UserRepository;

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    public User getUserById(Long id) {
        return userRepository.findById(id).orElse(null);
    }

    public User saveUser(User user) {
        return userRepository.save(user);
    }

    public void deleteUser(Long id) {
        userRepository.deleteById(id);

    }
}
```

Etape 5 : Création du controller

```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 org.springframework.web.bind.annotation.ModelAttribute;
```

```java
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;



@Controller
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping
    public String listUsers(Model model) {
        model.addAttribute("users", userService.getAllUsers());
        return "users";
    }

    @GetMapping("/add")
    public String addUserForm(Model model) {
        model.addAttribute("user", new User());
        return "add-user";
    }

    @PostMapping("/add")
    public String saveUser(@ModelAttribute("user") User user) {
        userService.saveUser(user);
        return "redirect:/users";
    }

    @GetMapping("/edit/{id}")
    public String editUserForm(@PathVariable Long id, Model
model) {
        model.addAttribute("user", userService.getUserById(id));
        return "edit-user";
    }

    @PostMapping("/edit/{id}")
```

```
    public String updateUser(@PathVariable Long id,
@ModelAttribute("user") User user) {
        user.setId(id);
        userService.saveUser(user);
        return "redirect:/users";
    }

    @GetMapping("/delete/{id}")
    public String deleteUser(@PathVariable Long id) {
        userService.deleteUser(id);
        return "redirect:/users";
    }
}
```

Avant l'explication du code : classe Model

Dans Spring Boot, la classe Model est utilisée pour transférer des données entre le contrôleur et la vue. Elle fait partie du package org.springframework.ui et est principalement utilisée pour fournir des données de modèle à la vue, permettant d'afficher dynamiquement du contenu en fonction des informations récupérées ou générées dans le contrôleur.

Voici les éléments clés de la classe Model en Spring Boot :

1. Rôle de la classe Model

La classe Model sert d'intermédiaire pour ajouter des attributs qui seront accessibles dans la vue (comme les pages HTML Thymeleaf, JSP, etc.). Lorsque vous ajoutez un attribut à Model, il sera disponible pour être utilisé et affiché dans la vue associée.

2. Utilisation de la classe Model dans un contrôleur

Dans un contrôleur Spring Boot, vous pouvez injecter la classe Model en paramètre de méthode, ce qui vous permet d'ajouter des données directement.

Exemple :

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class MyController {

    @GetMapping("/home")
    public String home(Model model) {
        model.addAttribute("message", "Bienvenue sur notre page d'accueil!");
        model.addAttribute("user", "DAZRO NAZIFOU");
        return "home"; // nom de la vue (par exemple, home.html avec Thymeleaf)
    }
```

}

Dans cet exemple, les attributs message et user sont ajoutés à l'objet Model. Ces données seront accessibles dans la vue home.html (ou une autre technologie de rendu) et pourront être affichées dynamiquement.

3. Accéder aux données du modèle dans la vue

Si vous utilisez Thymeleaf, vous pouvez accéder aux attributs du modèle dans la vue en utilisant la syntaxe ${}. Par exemple :

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
   <title>Accueil</title>
</head>
<body>
   <h1 th:text="${message}">Message par défaut</h1>
   <p>Utilisateur : <span th:text="${user}">Nom de l'utilisateur</span></p>
</body>
</html>
```

Dans cet exemple, **${message}** et **${user}** afficheront les valeurs associées ajoutées dans le contrôleur.

4. Avantages de la classe Model

- **Simplicité** : Elle permet de transmettre facilement des données d'un contrôleur à une vue.
- **Séparation des préoccupations** : Elle aide à maintenir la logique de traitement dans le contrôleur et le rendu dans la vue.
- **Support des templates** : Fonctionne bien avec les moteurs de template comme Thymeleaf, FreeMarker, et JSP.

Explication du code

Explication du Code

```
@Controller
@RequestMapping("/users")
public class UserController {
```

- **@Controller** : Cette annotation indique que cette classe est un contrôleur Spring MVC, chargé de gérer les requêtes HTTP.
- **@RequestMapping("/users")** : Cette annotation spécifie que toutes les requêtes commençant par /users seront traitées par ce contrôleur. Elle sert de préfixe pour toutes les URL des méthodes de cette classe.

```
   @Autowired
```

```
private UserService userService;
```

- **@Autowired** : Cette annotation demande à Spring d'injecter automatiquement une instance de UserService, qui contient la logique métier pour manipuler les données utilisateur.
- **private UserService userService;** : Déclare une instance de UserService, qui sera utilisée pour appeler les méthodes CRUD sur l'entité User.

```
@GetMapping
public String listUsers(Model model) {
    model.addAttribute("users", userService.getAllUsers());
    return "users";
}
```

- **@GetMapping** : Cette annotation indique que cette méthode doit être appelée pour les requêtes HTTP GET sur l'URL /users.
- **public String listUsers(Model model)** : Méthode pour afficher la liste des utilisateurs.
- **model.addAttribute("users", userService.getAllUsers());** : Ajoute à l'objet model un attribut nommé "users", contenant la liste de tous les utilisateurs obtenue via userService.getAllUsers(). Ce modèle est ensuite accessible depuis la vue users.html.
- **return "users";** : Retourne le nom de la vue users.html, qui affiche la liste des utilisateurs.

```
@GetMapping("/add")
public String addUserForm(Model model) {
    model.addAttribute("user", new User());
    return "add-user";
}
```

- **@GetMapping("/add")** : Cette méthode est appelée pour les requêtes GET sur /users/add.
- **public String addUserForm(Model model)** : Méthode pour afficher le formulaire d'ajout d'utilisateur.
- **model.addAttribute("user", new User());** : Ajoute un nouvel objet User vide au modèle, pour être utilisé dans le formulaire d'ajout dans add-user.html.
- **return "add-user";** : Retourne le nom de la vue add-user.html, qui affiche le formulaire d'ajout d'utilisateur.

```
@PostMapping("/add")
public String saveUser(@ModelAttribute("user") User user) {
    userService.saveUser(user);
    return "redirect:/users";
}
```

- **@PostMapping("/add") :** Cette méthode est appelée pour les requêtes POST sur /users/add.
- **public String saveUser(@ModelAttribute("user") User user)** : Méthode pour enregistrer un nouvel utilisateur dans la base de données.

- **@ModelAttribute("user") User user** : Permet de lier les données du formulaire à un objet User.
- **userService.saveUser(user);** : Enregistre l'utilisateur en appelant saveUser dans UserService.
- **return "redirect:/users";** : Redirige vers la liste des utilisateurs après l'ajout, en rechargeant la page d'affichage de tous les utilisateurs.

```
@GetMapping("/edit/{id}")
public String editUserForm(@PathVariable Long id, Model model) {
    model.addAttribute("user", userService.getUserById(id));
    return "edit-user";
}
```

- **@GetMapping("/edit/{id}") :** Cette méthode est appelée pour les requêtes GET sur /users/edit/{id}, où {id} est un identifiant utilisateur dynamique.
- **@PathVariable** Long id : L'annotation @PathVariable indique que le paramètre id dans l'URL sera lié à l'argument id de la méthode.
- **model.addAttribute**("user", userService.getUserById(id)); : Ajoute l'utilisateur, correspondant à l'id, au modèle en utilisant userService.getUserById(id).
- **return "edit-user";** : Retourne la vue edit-user.html, qui contient le formulaire de modification de l'utilisateur.

```
@PostMapping("/edit/{id}")
public String updateUser(@PathVariable Long id, @ModelAttribute("user") User user) {
    user.setId(id);
    userService.saveUser(user);
    return "redirect:/users";
}
```

- **@PostMapping("/edit/{id}")** : Cette méthode est appelée pour les requêtes POST sur /users/edit/{id}.
- @PathVariable Long id : Lie l'id de l'URL à l'argument id de la méthode.
- **@ModelAttribute("user") User user** : Lie les données du formulaire à un objet User.
- **user.setId(id);** : Associe l'id existant à l'objet User pour assurer la mise à jour de l'enregistrement existant.
- **userService.saveUser(user)**; : Met à jour l'utilisateur dans la base de données.
- **return "redirect:/users";** : Redirige vers la liste des utilisateurs après la mise à jour.

```
@GetMapping("/delete/{id}")
public String deleteUser(@PathVariable Long id) {
    userService.deleteUser(id);
    return "redirect:/users";
}
}
```

- **@GetMapping("/delete/{id}")** : Cette méthode est appelée pour les requêtes GET sur /users/delete/{id}.
- **@PathVariable Long id :** Lie l'id de l'URL à l'argument id de la méthode.

- **userService.deleteUser(id);** : Supprime l'utilisateur correspondant à l'id fourni.
- **return "redirect:/users";** : Redirige vers la liste des utilisateurs après la suppression.

Etape 8 :Créations des vues

users.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Liste des utilisateurs</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/
bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div class="container mt-5">
    <h1>Liste des utilisateurs</h1>
    <a href="/users/add" class="btn btn-primary mb-3">Ajouter un
utilisateur</a>
    <table class="table">
        <thead>
            <tr>
                <th>ID</th>
                <th>Email</th>
                <th>Nom</th>
                <th>Âge</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="user : ${users}">
                <td th:text="${user.id}"></td>
                <td th:text="${user.email}"></td>
                <td th:text="${user.nom}"></td>
                <td th:text="${user.age}"></td>
                <td>
```

```
                <a th:href="@{/users/edit/{id}(id=${user.id})}"
class="btn btn-warning">Modifier</a>
                <a th:href="@{/users/delete/{id}(id=${user.id})}"
class="btn btn-danger">Supprimer</a>
            </td>
        </tr>
    </tbody>
  </table>
</div>
</body>
</html>
```

Explication du code

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
```

- <!DOCTYPE html> : Indique le type de document HTML.
- <html xmlns:th="http://www.thymeleaf.org"> : Définit l'espace de noms th pour Thymeleaf, permettant d'utiliser ses attributs dans le document.

```
<head>
  <title>Liste des utilisateurs</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
```

- <title>Liste des utilisateurs</title> : Définit le titre de la page, qui apparaît dans l'onglet du navigateur.
- <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css" rel="stylesheet"> : Charge la bibliothèque CSS de Bootstrap 5.3 pour styliser la page, avec des classes telles que container, btn, table, etc.

```
<body>
<div class="container mt-5">
  <h1>Liste des utilisateurs</h1>
  <a href="/users/add" class="btn btn-primary mb-3">Ajouter un utilisateur</a>
```

- <body> : Début du corps de la page.
- : Crée un conteneur centré avec une marge supérieure (mt-5).
- <h1>Liste des utilisateurs</h1> : Affiche un en-tête de niveau 1 avec le titre "Liste des utilisateurs".
- Ajouter un utilisateur : Lien vers le formulaire d'ajout d'utilisateur avec les classes Bootstrap btn btn-primary pour le style du bouton et mb-3 pour une marge inférieure.

```
    <table class="table">
      <thead>
        <tr>
          <th>ID</th>
          <th>Email</th>
          <th>Nom</th>
          <th>Âge</th>
          <th>Actions</th>
        </tr>
      </thead>
```

- <table class="table"> : Crée une table stylisée par Bootstrap.
- <thead> : Début de l'en-tête de la table.
- <tr> : Début d'une ligne de la table.
- Les balises <th> : Chaque <th> représente une colonne de l'en-tête, affichant les noms des champs ID, Email, Nom, Âge, et Actions.

```
      <tbody>
        <tr th:each="user : ${users}">
          <td th:text="${user.id}"></td>
          <td th:text="${user.email}"></td>
          <td th:text="${user.nom}"></td>
          <td th:text="${user.age}"></td>
          <td>
            <a th:href="@{/users/edit/{id}(id=${user.id})}" class="btn btn-warning">Modifier</a>
            <a th:href="@{/users/delete/{id}(id=${user.id})}" class="btn btn-danger">Supprimer</a>
          </td>
        </tr>
      </tbody>
    </table>
</div>
</body>
</html>
```

- <tbody> : Contient les données dynamiques de la table.
- <tr th:each="user : ${users}"> : Cette boucle th:each de Thymeleaf itère sur chaque utilisateur de la liste ${users}, créant une ligne <tr> pour chaque utilisateur.
- <td th:text="${user.id}"></td> : Remplit la cellule avec l'id de l'utilisateur en utilisant th:text pour insérer la valeur dynamique.
- <td th:text="${user.email}"></td> : Affiche l'email de l'utilisateur.
- <td th:text="${user.nom}"></td> : Affiche le nom de l'utilisateur.
- <td th:text="${user.age}"></td> : Affiche l'âge de l'utilisateur.
- <a th:href="@{/users/edit/{id}(id=${user.id})}" class="btn btn-warning">Modifier : Lien pour modifier l'utilisateur avec un bouton stylisé en orange (btn-warning). th:href crée un lien dynamique vers /users/edit/{id}, remplaçant {id} par user.id.
- <a th:href="@{/users/delete/{id}(id=${user.id})}" class="btn btn-danger">Supprimer : Lien pour supprimer l'utilisateur avec un bouton rouge (btn-danger), dynamique sur le même principe.

add-user.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Ajouter un utilisateur</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/
bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div class="container mt-5">
    <h1>Ajouter un utilisateur</h1>
    <form th:action="@{/users/add}" th:object="${user}"
method="post">
        <div class="mb-3">
            <label for="email" class="form-label">Email</label>
            <input type="email" th:field="*{email}" class="form-control"
required />
        </div>
        <div class="mb-3">
            <label for="nom" class="form-label">Nom</label>
            <input type="text" th:field="*{nom}" class="form-control"
required />
        </div>
        <div class="mb-3">
            <label for="age" class="form-label">Âge</label>
            <input type="number" th:field="*{age}" class="form-control"
required />
        </div>
        <button type="submit" class="btn btn-
primary">Enregistrer</button>
        <a href="/users" class="btn btn-secondary">Annuler</a>
    </form>
</div>
</body>
</html>
```

Explication du code

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
```

- <!DOCTYPE html> : Indique le type de document HTML.
- <html xmlns:th="http://www.thymeleaf.org"> : Définit l'espace de noms th pour Thymeleaf, permettant d'utiliser ses attributs dans le document.

```
<head>
    <title>Ajouter un utilisateur</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
```

- <title>Ajouter un utilisateur</title> : Définit le titre de la page, qui apparaît dans l'onglet du navigateur.
- <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css" rel="stylesheet"> : Charge la bibliothèque CSS de Bootstrap 5.3 pour styliser la page, avec des classes telles que container, btn, form-control, etc.

```
<body>
<div class="container mt-5">
    <h1>Ajouter un utilisateur</h1>
```

- <body> : Début du corps de la page.
- : Crée un conteneur centré avec une marge supérieure (mt-5).
- <h1>Ajouter un utilisateur</h1> : Affiche un en-tête de niveau 1 avec le titre "Ajouter un utilisateur".

```
    <form th:action="@{/users/add}" th:object="${user}" method="post">
```

- <form th:action="@{/users/add}" th:object="${user}" method="post"> :
    - th:action="@{/users/add}" : Définit l'URL de l'action du formulaire pour l'envoi des données vers /users/add en méthode POST.
    - th:object="${user}" : Associe le formulaire à un objet user, permettant de lier les champs de saisie aux attributs de cet objet.
    - method="post" : Indique que le formulaire utilise la méthode POST pour envoyer les données.

```
        <div class="mb-3">
            <label for="email" class="form-label">Email</label>
            <input type="email" th:field="*{email}" class="form-control" required />
        </div>
```

- : Crée une division avec une marge inférieure (mb-3).
- <label for="email" class="form-label">Email</label> : Étiquette pour le champ de saisie de l'email, stylisée avec form-label.

- <input type="email" th:field="*{email}" class="form-control" required /> :
    - type="email" : Spécifie que le champ attend une adresse email.
    - th:field="*{email}" : Lie ce champ de saisie à l'attribut email de l'objet user.
    - class="form-control" : Applique le style Bootstrap pour les champs de formulaire.
    - required : Rend ce champ obligatoire.

```
<div class="mb-3">
  <label for="nom" class="form-label">Nom</label>
  <input type="text" th:field="*{nom}" class="form-control" required />
</div>
```

- <label for="nom" class="form-label">Nom</label> : Étiquette pour le champ de saisie du nom.
- <input type="text" th:field="*{nom}" class="form-control" required /> :
    - type="text" : Spécifie que le champ attend du texte (nom de l'utilisateur).
    - th:field="*{nom}" : Lie ce champ de saisie à l'attribut nom de l'objet user.
    - class="form-control" : Applique le style Bootstrap pour les champs de texte.
    - required : Rend ce champ obligatoire.

```
<div class="mb-3">
  <label for="age" class="form-label">Âge</label>
  <input type="number" th:field="*{age}" class="form-control" required />
</div>
```

- <label for="age" class="form-label">Âge</label> : Étiquette pour le champ de saisie de l'âge.
- <input type="number" th:field="*{age}" class="form-control" required /> :
    - type="number" : Spécifie que le champ attend un nombre (l'âge de l'utilisateur).
    - th:field="*{age}" : Lie ce champ de saisie à l'attribut age de l'objet user.
    - class="form-control" : Applique le style Bootstrap pour les champs de type nombre.
    - required : Rend ce champ obligatoire.

```
  <button type="submit" class="btn btn-primary">Enregistrer</button>
  <a href="/users" class="btn btn-secondary">Annuler</a>
 </form>
</div>
</body>
</html>
```

- <button type="submit" class="btn btn-primary">Enregistrer</button> : Bouton pour soumettre le formulaire, stylisé avec Bootstrap (btn btn-primary).
- Annuler : Lien vers la page de liste des utilisateurs (/users), stylisé comme un bouton avec btn btn-secondary pour permettre d'annuler l'opération.

edit-user.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Ajouter un utilisateur</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/
bootstrap.min.css" rel="stylesheet">
</head>
<body>
<div class="container mt-5">
    <h1>Ajouter un utilisateur</h1>
    <form th:action="@{/users/edit/{id}(id=${user.id})}" th:object="$
{user}" method="post">
        <div class="mb-3">
            <label for="email" class="form-label">Email</label>
            <input type="email" th:field="*{email}" class="form-control"
required />
        </div>
        <div class="mb-3">
            <label for="nom" class="form-label">Nom</label>
            <input type="text" th:field="*{nom}" class="form-control"
required />
        </div>
        <div class="mb-3">
            <label for="age" class="form-label">Âge</label>
            <input type="number" th:field="*{age}" class="form-control"
required />
        </div>
        <button type="submit" class="btn btn-
primary">Enregistrer</button>
        <a href="/users" class="btn btn-secondary">Annuler</a>
    </form>
</div>
</body>
</html>
```

Explication du code

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
```

- <!DOCTYPE html> : Indique le type de document HTML.
- <html xmlns:th="http://www.thymeleaf.org"> : Définit l'espace de noms th pour Thymeleaf, permettant d'utiliser ses attributs dans le document.

```
<head>
    <title>Ajouter un utilisateur</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css"
rel="stylesheet">
</head>
```

- <title>Ajouter un utilisateur</title> : Définit le titre de la page qui apparaît dans l'onglet du navigateur.
- <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha1/dist/css/bootstrap.min.css" rel="stylesheet"> : Charge la bibliothèque CSS Bootstrap pour styliser la page.

```
<body>
<div class="container mt-5">
    <h1>Ajouter un utilisateur</h1>
```

- <body> : Début du corps de la page.
- : Crée un conteneur centré avec une marge supérieure (mt-5).
- <h1>Ajouter un utilisateur</h1> : Affiche un en-tête de niveau 1 avec le titre "Ajouter un utilisateur".

```
    <form th:action="@{/users/edit/{id}(id=${user.id})}" th:object="${user}" method="post">
```

- <form th:action="@{/users/edit/{id}(id=${user.id})}" th:object="${user}" method="post"> :
    - th:action="@{/users/edit/{id}(id=${user.id})}" : Définit l'URL de l'action pour l'envoi des données vers /users/edit/{id}, avec {id} remplacé dynamiquement par user.id pour spécifier l'ID de l'utilisateur à modifier.
    - th:object="${user}" : Associe le formulaire à l'objet user, permettant de lier les champs de saisie aux attributs de cet objet.
    - method="post" : Indique que le formulaire utilise la méthode POST pour envoyer les données.

```
        <div class="mb-3">
            <label for="email" class="form-label">Email</label>
            <input type="email" th:field="*{email}" class="form-control" required />
        </div>
```

- : Crée une division avec une marge inférieure (mb-3).

- <label for="email" class="form-label">Email</label> : Étiquette pour le champ de saisie de l'email.
- <input type="email" th:field="*{email}" class="form-control" required /> :
    - type="email" : Spécifie que le champ attend une adresse email.
    - th:field="*{email}" : Lie ce champ de saisie à l'attribut email de l'objet user.
    - class="form-control" : Applique le style Bootstrap pour le champ de formulaire.
    - required : Rend ce champ obligatoire.

```
<div class="mb-3">
  <label for="nom" class="form-label">Nom</label>
  <input type="text" th:field="*{nom}" class="form-control" required />
</div>
```

- <label for="nom" class="form-label">Nom</label> : Étiquette pour le champ de saisie du nom.
- <input type="text" th:field="*{nom}" class="form-control" required /> :
    - type="text" : Spécifie que le champ attend du texte (nom de l'utilisateur).
    - th:field="*{nom}" : Lie ce champ de saisie à l'attribut nom de l'objet user.
    - class="form-control" : Applique le style Bootstrap pour le champ de formulaire.
    - required : Rend ce champ obligatoire.

```
<div class="mb-3">
  <label for="age" class="form-label">Âge</label>
  <input type="number" th:field="*{age}" class="form-control" required />
</div>
```

- <label for="age" class="form-label">Âge</label> : Étiquette pour le champ de saisie de l'âge.
- <input type="number" th:field="*{age}" class="form-control" required /> :
    - type="number" : Spécifie que le champ attend un nombre (l'âge de l'utilisateur).
    - th:field="*{age}" : Lie ce champ de saisie à l'attribut age de l'objet user.
    - class="form-control" : Applique le style Bootstrap pour le champ de formulaire.
    - required : Rend ce champ obligatoire.

```
  <button type="submit" class="btn btn-primary">Enregistrer</button>
  <a href="/users" class="btn btn-secondary">Annuler</a>
 </form>
</div>
</body>
</html>
```

- <button type="submit" class="btn btn-primary">Enregistrer</button> : Bouton pour soumettre le formulaire, stylisé avec Bootstrap (btn btn-primary).
- Annuler : Lien vers la page de liste des utilisateurs (/users), stylisé comme un bouton pour annuler l'opération.