**SpringBoot**

1. **Spring et SpringBoot**

Spring est un framework open-source pour le développement d'applications Java. Il fournit une infrastructure complète pour la création d'applications Java d'entreprise, en offrant des fonctionnalités telles que l'inversion de contrôle (IoC), l'injection de dépendances, la gestion de transactions, la gestion des données, la sécurité, etc. Spring permet aux développeurs de créer des applications Java de manière modulaire et flexible, en favorisant une conception orientée objet et en facilitant l'intégration avec d'autres technologies.

Spring Boot, d'autre part, est un projet au sein de l'écosystème Spring qui simplifie le processus de création et de déploiement d'applications Spring. Il fournit des conventions de configuration par défaut et une automatisation des tâches pour réduire la complexité du développement. Avec Spring Boot, les développeurs peuvent créer des applications Java autonomes qui sont auto-configurées dans une large mesure. Il offre également des intégrations faciles avec des serveurs d'applications, des bases de données, des outils de surveillance, etc., ce qui permet aux développeurs de se concentrer davantage sur le développement des fonctionnalités métier de leurs applications. En résumé, Spring Boot facilite la création rapide d'applications Spring prêtes à être déployées.

Spring Boot est un projet au sein de l'écosystème Spring qui vise à simplifier le processus de création, de configuration et de déploiement d'applications Spring. Il fournit une approche convention-over-configuration, ce qui signifie qu'il préconfigure automatiquement de nombreuses fonctionnalités de Spring et fournit des valeurs par défaut, réduisant ainsi la quantité de code et de configuration nécessaire de la part des développeurs.

Les principales caractéristiques et avantages de Spring Boot sont les suivants :

1. **Configuration automatique :** Spring Boot utilise la détection automatique pour configurer les fonctionnalités de Spring et les bibliothèques tierces, ce qui réduit la nécessité de configurer manuellement chaque composant.
2. **Embarquement des serveurs d'application :** Il permet de créer des applications autonomes qui intègrent un serveur d'application intégré tel que Tomcat, Jetty ou Undertow, ce qui rend le déploiement et l'exécution des applications plus simples.
3. **Dépendances starters :** Spring Boot fournit des dépendances starters préconfigurées pour différentes fonctionnalités courantes telles que la connectivité de base de données, la sécurité, la création de web services, etc. Ces starters incluent toutes les bibliothèques nécessaires et la configuration de base pour démarrer rapidement.
4. **Actuator :** Spring Boot Actuator fournit des fonctionnalités pour surveiller et gérer l'application en cours d'exécution, telles que les endpoints HTTP pour la surveillance de la santé, les métriques, les informations sur les configurations, etc.
5. **Simplicité :** Spring Boot réduit la quantité de code boilerplate en utilisant des annotations et des conventions par défaut, ce qui permet aux développeurs de se concentrer sur la logique métier de leurs applications.

Dans l'ensemble, Spring Boot facilite le développement d'applications Spring en réduisant la complexité de la configuration et du déploiement, tout en offrant une grande flexibilité et une intégration transparente avec d'autres technologies.

1. **JDK**

Un JDK, ou Java Development Kit, est un ensemble d'outils de développement logiciel fourni par Oracle Corporation pour créer des applications Java. Il contient tout ce dont un développeur a besoin pour développer, déboguer et exécuter des programmes Java, y compris :

1. **JRE (Java Runtime Environment) :** Le JDK comprend une version du JRE qui est utilisée pour exécuter des applications Java sur une machine. Le JRE contient la machine virtuelle Java (JVM) et les bibliothèques de classes standard nécessaires à l'exécution des programmes Java.
2. **Compilateur Java :** Le JDK comprend le compilateur Java, appelé "javac", qui est utilisé pour traduire le code source Java en bytecode exécutable par la JVM.
3. **Outils de débogage :** Le JDK fournit des outils de débogage tels que "jdb" (Java Debugger) pour aider les développeurs à identifier et corriger les erreurs dans leur code Java.
4. **Outils de documentation :** Le JDK inclut des outils de documentation tels que "javadoc" qui permettent de générer de la documentation à partir des commentaires du code source Java.
5. **Autres utilitaires :** Le JDK comprend divers autres utilitaires et outils de ligne de commande pour aider les développeurs à gérer et à analyser leur code Java.

En résumé, le JDK est un ensemble complet d'outils et de composants nécessaires au développement d'applications Java. Il est essentiel pour tout développeur Java qui souhaite créer des logiciels Java, que ce soit des applications de bureau, des applications web, des applications mobiles, des services web, etc.

1. **Maven**
2. **Maven**

Maven est un outil de gestion de projet largement utilisé dans le développement logiciel, principalement pour les projets Java. Il fournit un moyen cohérent de gérer les dépendances, de construire, de tester et de déployer des applications logicielles. Voici quelques-unes de ses principales fonctionnalités :

1. **Gestion des dépendances :** Maven permet de déclarer les dépendances d'un projet dans un fichier de configuration appelé "pom.xml" (Project Object Model). Maven télécharge automatiquement les dépendances nécessaires à partir d'un référentiel central, ce qui simplifie la gestion des bibliothèques tierces.
2. **Cycle de vie de construction :** Maven définit un cycle de vie de construction standard composé de phases telles que "compile", "test", "package", "install" et "deploy". Chaque phase correspond à une série de tâches prédéfinies, ce qui permet d'exécuter facilement des opérations de construction standardisées.
3. **Gestion des plugins :** Maven utilise des plugins pour étendre ses fonctionnalités. Ces plugins fournissent des objectifs spécifiques qui peuvent être exécutés pendant différentes phases du cycle de vie de construction. Par exemple, le plugin "maven-compiler-plugin" est utilisé pour compiler le code source Java.
4. **Convention sur configuration :** Maven favorise une approche convention-over-configuration, ce qui signifie qu'il fournit des configurations par défaut pour de nombreuses tâches. Cela permet de réduire la quantité de configuration nécessaire et encourage les bonnes pratiques de développement.
5. **Réutilisabilité des projets :** Maven encourage la réutilisation des projets grâce à la gestion des dépendances et à la possibilité de créer des modules (projets enfants) dans une structure de projet multi-modules.

En résumé, Maven est un outil puissant et largement utilisé dans l'écosystème Java pour automatiser les tâches de gestion de projet et de construction logicielle, ce qui permet aux développeurs de se concentrer davantage sur le développement de leurs applications.

1. **Pom.xml**
2. <?xml version="1.0" encoding="UTF-8"?>  
   <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">  
    <modelVersion>4.0.0</modelVersion>  
    <parent>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-parent</artifactId>  
    <version>3.2.4</version>  
    <relativePath/> <!-- lookup parent from repository -->  
    </parent>  
    <groupId>com.springboot</groupId>  
    <artifactId>springboot</artifactId>  
    <version>0.0.1-SNAPSHOT</version>  
    <name>springboot</name>  
    <description>Demo project for Spring Boot</description>  
    <properties>  
    <java.version>17</java.version>  
    </properties>  
    <dependencies>  
    <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
    </dependency>  
     
    <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-devtools</artifactId>  
    <scope>runtime</scope>  
    <optional>true</optional>  
    </dependency>  
    <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-test</artifactId>  
    <scope>test</scope>  
    </dependency>  
    </dependencies>  
     
    <build>  
    <plugins>  
    <plugin>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-maven-plugin</artifactId>  
    </plugin>  
    </plugins>  
    </build>  
     
   </project>

Ce fichier XML est un fichier de configuration POM (Project Object Model) pour un projet Maven, souvent appelé "pom.xml". Il décrit les détails du projet et spécifie comment Maven doit gérer le projet, y compris ses dépendances, ses plugins et sa construction. Voici une explication détaillée des différentes parties de ce fichier :

1. **<?xml version="1.0" encoding="UTF-8"?>** : Cette ligne spécifie la version XML et l'encodage utilisé dans le fichier.
2. **<project>** : C'est l'élément racine du fichier POM.
3. **xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"** : Ces attributs définissent l'espace de noms XML utilisé dans le fichier.
4. **xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"** : Cet attribut spécifie l'emplacement du schéma XML utilisé pour valider le fichier.
5. **<modelVersion>4.0.0</modelVersion>** : Il spécifie la version du modèle de projet Maven utilisée.
6. **<parent>** : Cet élément spécifie le POM parent de ce projet, héritant ainsi de sa configuration. Dans ce cas, le projet hérite de la configuration définie par le parent "spring-boot-starter-parent".
7. **<groupId>com.springboot</groupId>** : C'est l'identifiant de groupe du projet, utilisé pour identifier de manière unique le projet parmi d'autres projets.
8. **<artifactId>springboot</artifactId>** : C'est l'identifiant d'artefact du projet, utilisé pour identifier de manière unique l'artefact généré par ce projet.
9. **<version>0.0.1-SNAPSHOT</version>** : C'est la version actuelle du projet.
10. **<name>springboot</name>** : C'est le nom du projet.
11. **<description>Demo project for Spring Boot</description>** : C'est la description du projet.
12. **<properties>** : Cet élément permet de définir des propriétés qui peuvent être utilisées dans tout le fichier POM.
13. **<java.version>17</java.version>** : C'est une propriété définissant la version de Java utilisée dans ce projet.
14. **<dependencies>** : Cet élément contient la liste des dépendances du projet. Chaque dépendance est spécifiée à l'aide des éléments **<dependency>**.
15. **<build>** : Cet élément contient la configuration liée à la construction du projet.
16. **<plugins>** : Cet élément contient la liste des plugins Maven utilisés pour la construction du projet.
17. **<plugin>** : Cet élément spécifie un plugin Maven.
18. **<groupId>org.springframework.boot</groupId>** : C'est l'identifiant de groupe du plugin.
19. **<artifactId>spring-boot-maven-plugin</artifactId>** : C'est l'identifiant d'artefact du plugin.

En résumé, ce fichier POM spécifie les détails du projet ainsi que les dépendances et les plugins nécessaires à sa construction à l'aide de Maven.

1. **La structure d'un projet Spring Boot avec Maven (mvn)**

La structure d'un projet Spring Boot avec Maven (mvn) suit généralement une organisation standard qui facilite le développement, la gestion des dépendances et la construction de l'application. Voici une structure typique de projet Spring Boot avec Maven :

**Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement**

Explications des principaux éléments de la structure :

* **src/main/java :** C'est le répertoire principal où se trouve le code source Java de l'application. Il contient généralement les packages et les classes Java de l'application, y compris les contrôleurs, les modèles, les services, etc.
* **src/main/resources :** C'est le répertoire où se trouvent les ressources de l'application, telles que les fichiers de configuration (comme application.properties), les fichiers statiques (CSS, JS, etc.), les modèles HTML (dans le répertoire "templates"), etc.
* **src/test :** C'est le répertoire où se trouve le code source des tests unitaires de l'application.
* **pom.xml :** C'est le fichier de configuration Maven pour le projet. Il contient les informations sur le projet, les dépendances, les plugins, les propriétés, etc.
* **target :** C'est le répertoire où sont générés les fichiers binaires compilés et les artefacts de l'application lors de la construction.
* **README.md :** C'est un fichier contenant des informations sur le projet, telles que des instructions d'installation, des guides de démarrage rapide, etc.

Cette structure de projet est une convention courante pour les projets Spring Boot avec Maven, mais elle peut varier en fonction des besoins spécifiques du projet et des préférences de l'équipe de développement.

1. **Create a springBoot Application**

package com.houssam;  
  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class Main {  
  
 public static void main(String[] args){  
 SpringApplication.*run*(Main.class,args);  
 }  
  
}

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

ce code est un point d'entrée typique d'une application Spring Boot. Il utilise l'annotation **@SpringBootApplication** pour activer la configuration automatique et démarrer l'application en utilisant **SpringApplication.run()**.

1. **Configuring Embedded Web Server**

To configure the embedded web server in a Spring Boot application using a YAML (YAML Ain't Markup Language) file, you typically use the **application.yml** or **application.yaml** file in the **src/main/resources** directory of your project. Here's an example of how you can configure various properties of the embedded web server using YAML:

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

1. **API**
2. **C’est quoi un API**

Une API (Interface de Programmation Applicative) est un ensemble de règles, de protocoles et d'outils qui permettent à différentes applications logicielles de communiquer entre elles. Elle définit les méthodes et les formats de données que les applications peuvent utiliser pour demander et échanger des informations. Les APIs sont utilisées pour permettre l'interaction entre différents systèmes logiciels, services ou composants, leur permettant de travailler ensemble et de partager des données.

Il existe différents types d'APIs, notamment :

1. **APIs Web :** Ce sont des APIs auxquelles on accède via le web en utilisant les protocoles HTTP ou HTTPS. Elles sont couramment utilisées pour intégrer des services basés sur le web ou pour accéder à des données depuis des serveurs distants. Les APIs Web retournent souvent des données dans des formats tels que JSON (JavaScript Object Notation) ou XML (eXtensible Markup Language).
2. **APIs de Bibliothèque :** Ce sont des APIs fournies par des bibliothèques logicielles ou des frameworks pour permettre aux développeurs d'utiliser leur fonctionnalité. Les APIs de bibliothèque définissent des classes, des fonctions et des structures de données que les développeurs peuvent utiliser dans leur code pour effectuer des tâches spécifiques sans avoir à les implémenter à partir de zéro.
3. **APIs d'Exploitation Système :** Ce sont des APIs fournies par les systèmes d'exploitation pour permettre aux applications d'interagir avec les ressources matérielles et logicielles sous-jacentes du système. Les APIs d'exploitation système permettent des tâches telles que l'E/S de fichier, le réseau, la gestion de processus et le contrôle de périphériques.
4. **APIs Distantes :** Ce sont des APIs fournies par des services ou des plateformes distantes qui permettent aux développeurs d'accéder à leur fonctionnalité et à leurs données sur un réseau. Les APIs distantes sont couramment utilisées pour intégrer des services tiers, tels que des plateformes de médias sociaux, des passerelles de paiement ou des services cloud.

Les APIs jouent un rôle crucial dans le développement logiciel moderne, car elles permettent l'interopérabilité et l'intégration entre différents systèmes et services. Elles abstraient la complexité des systèmes sous-jacents et fournissent une interface standardisée pour que les développeurs puissent interagir. Les APIs sont fondamentales pour la création d'architectures logicielles évolutives, modulaires et maintenables.

1. **Restful API**

Une API RESTful (Representational State Transfer) est une architecture de conception d'API basée sur les principes du web et du protocole HTTP. Les API RESTful permettent aux clients de communiquer avec des serveurs de manière cohérente et uniforme en utilisant des opérations standardisées telles que GET, POST, PUT, DELETE, etc. Voici quelques caractéristiques clés des API RESTful :

1. **Utilisation de ressources :** Les API RESTful exposent des ressources, qui représentent des objets ou des données, via des URI (Uniform Resource Identifiers). Chaque ressource est identifiée par une URI unique.
2. **Manipulation des ressources via des méthodes HTTP :** Les opérations CRUD (Create, Read, Update, Delete) sont effectuées sur les ressources en utilisant des méthodes HTTP standardisées. Par exemple, GET pour récupérer des données, POST pour créer de nouvelles ressources, PUT pour mettre à jour des ressources existantes, DELETE pour supprimer des ressources, etc.
3. **Utilisation de représentations :** Les ressources sont généralement représentées en utilisant des formats de données standard tels que JSON (JavaScript Object Notation) ou XML (eXtensible Markup Language). Les clients peuvent spécifier le format de représentation souhaité à l'aide des en-têtes HTTP.
4. **Statelessness (Sans état) :** Les serveurs RESTful sont sans état, ce qui signifie qu'ils ne conservent pas l'état de session des clients entre les requêtes. Chaque requête doit contenir toutes les informations nécessaires pour traiter cette requête, ce qui rend les API RESTful facilement scalable et évolutives.
5. **HATEOAS (Hypermedia as the Engine of Application State) :** Ce principe stipule que les réponses des API RESTful doivent inclure des liens hypertexte permettant aux clients de naviguer de manière autonome à travers les différentes ressources. Cela permet une découverte dynamique des fonctionnalités de l'API sans nécessiter de connaissances préalables.

En résumé, les API RESTful sont un style d'architecture web populaire pour la création d'APIs, en raison de leur simplicité, de leur évolutivité, et de leur compatibilité avec les standards du web. Elles sont largement utilisées pour le développement d'applications web, mobiles et IoT, ainsi que pour les services web et les intégrations entre systèmes.

1. **JavaObject to simple Object**

Jackson est inclus par défaut dans les projets Spring Boot. Spring Boot utilise Jackson comme bibliothèque par défaut pour la sérialisation et la désérialisation des objets Java en JSON (JavaScript Object Notation) et vice versa.

Jackson est une bibliothèque très populaire et largement utilisée dans l'écosystème Spring. Spring Boot la configure automatiquement pour vous dès que vous incluez les dépendances de base telles que **spring-boot-starter-web**, qui sont souvent incluses dans les nouveaux projets Spring Boot.

Grâce à l'intégration automatique de Jackson dans Spring Boot, vous pouvez facilement travailler avec des objets Java et JSON sans avoir à configurer manuellement Jackson. Vous pouvez annoter vos classes Java avec des annotations Jackson telles que **@JsonProperty**, **@JsonFormat**, **@JsonIgnore**, etc., pour contrôler la sérialisation et la désérialisation des objets.

Par exemple, vous pouvez créer un contrôleur Spring MVC qui renvoie un objet Java en tant que réponse HTTP, et Spring Boot utilisera automatiquement Jackson pour le convertir en JSON avant de l'envoyer au client. De même, lors de la réception d'une requête HTTP contenant des données JSON, Spring Boot utilisera Jackson pour les convertir en objets Java appropriés.

En résumé, Jackson est inclus par défaut dans Spring Boot pour faciliter le travail avec JSON dans les applications Spring Boot, et il est automatiquement configuré pour vous par Spring Boot.

1. **Creating a simple API**
   1. **class**

package com.houssam;  
  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@SpringBootApplication  
@RestController  
public class Main {  
  
 public static void main(String[] args){  
 SpringApplication.*run*(Main.class,args);  
 }  
  
 @GetMapping  
 private User getUser(){  
 return new User(1L,"houssam");  
 }  
  
 private class User{  
  
 private Long id;  
 private String name;  
  
 public User(Long id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 public Long getIdUtilisateur() {  
 return id;  
 }  
  
 public String getNameUtilisateur() {  
 return name;  
 }  
 }  
  
}

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

Dans la classe **User**, les méthodes **getName()** et **getId()** sont des méthodes d'accès, également appelées "getters". Elles sont utilisées pour récupérer les valeurs des attributs **name** et **id** respectivement. Voici ce que ces méthodes font :

1. **getName() :**
   * Cette méthode retourne la valeur de l'attribut **name** de l'objet **User**.
   * Dans le contexte de cet exemple, **getName()** est utilisée pour récupérer le nom de l'utilisateur représenté par l'objet **User**.
2. **getId() :**
   * Cette méthode retourne la valeur de l'attribut **id** de l'objet **User**.
   * Dans le contexte de cet exemple, **getId()** est utilisée pour récupérer l'identifiant de l'utilisateur représenté par l'objet **User**.

Ces méthodes sont des éléments clés de l'encapsulation en programmation orientée objet. Elles permettent d'accéder aux données encapsulées dans un objet de manière contrôlée, tout en masquant les détails d'implémentation internes. En fournissant des méthodes d'accès plutôt qu'un accès direct aux attributs, vous pouvez contrôler comment les données sont manipulées et garantir l'intégrité des données en appliquant des règles de validation ou des traitements supplémentaires si nécessaire.

Dans le contexte d'une application Spring Boot, ces getters sont également utilisés implicitement par le mécanisme de sérialisation JSON pour extraire les valeurs des attributs de l'objet **User** lors de la conversion en JSON. C'est pourquoi il est important de fournir ces getters lorsque vous souhaitez que les données de votre objet soient sérialisées correctement lors de la communication avec d'autres composants de l'application, tels que les contrôleurs REST.

* + 1. **JsonIgnore And JsonProperty**

Les annotations **@JsonIgnore** et **@JsonProperty** sont utilisées dans le contexte de la sérialisation et de la désérialisation d'objets Java en JSON, souvent utilisées avec des frameworks tels que Jackson dans les applications Spring Boot.

Voici leur traduction en français :

1. **@JsonIgnore** : Cette annotation indique à Jackson (ou tout autre framework de sérialisation JSON) d'ignorer une propriété lors de la sérialisation ou de la désérialisation d'un objet. En d'autres termes, lorsque cette annotation est placée sur un champ ou une méthode d'accesseur (getter), la propriété correspondante ne sera pas incluse dans la représentation JSON de l'objet.

Traduction en français : **@IgnorerDansLaSerializationJSON**

1. **@JsonProperty** : Cette annotation est utilisée pour spécifier explicitement le nom d'une propriété dans la représentation JSON de l'objet. Par défaut, Jackson utilise le nom du champ ou de la méthode d'accesseur (getter) dans la représentation JSON. En utilisant **@JsonProperty**, vous pouvez spécifier un nom différent à utiliser dans le JSON.

Traduction en français : **@ProprieteJSON**

Ces annotations sont utiles pour contrôler précisément comment les objets Java sont convertis en JSON et vice versa, en définissant les noms des propriétés et en excluant certaines propriétés de la sérialisation JSON selon les besoins de l'application.

public class Personne {  
  
 private String nom;  
 private int age;  
  
 public Personne(String nom, int age) {  
 this.nom = nom;  
 this.age = age;  
 }  
  
 @JsonProperty("name") // Définir explicitement le nom de la propriété dans JSON en donnant name a la place de nom  
 public String getNom() {  
 return nom;  
 }  
  
 @JsonIgnore // Ignorer cette propriété lors de la sérialisation en JSON  
 public int getAge() {  
 return age;  
 }  
}

En utilisant l'annotation **@JsonProperty** de Jackson, vous pouvez spécifier différents types d'accès à une propriété JSON. Voici les principaux types d'accès avec des exemples :

1. **READ\_ONLY** : Accès en lecture uniquement à la propriété JSON. La propriété ne peut pas être modifiée lors de la désérialisation.

Exemple :

**Une image contenant texte, Police, capture d’écran

Description générée automatiquement**

1. **WRITE\_ONLY** : Accès en écriture uniquement à la propriété JSON. La propriété ne sera pas incluse lors de la sérialisation, mais elle peut être définie lors de la désérialisation.

Exemple :

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

1. **READ\_WRITE** : Accès en lecture et en écriture à la propriété JSON. La propriété peut être lue et modifiée lors de la sérialisation et de la désérialisation.

Exemple :

**Une image contenant texte, Police, Logiciel multimédia, capture d’écran

Description générée automatiquement**

1. **READ\_ONLY, WRITE\_ONLY combiné** : Vous pouvez également combiner les annotations pour spécifier différents types d'accès en lecture et en écriture.

Exemple :

**Une image contenant texte, Police, capture d’écran

Description générée automatiquement**

En utilisant ces annotations, vous pouvez contrôler finement la manière dont les propriétés JSON sont traitées lors de la sérialisation et de la désérialisation dans votre application.

* 1. **Nommage et versionnage d’un API**

La "nomenclature" (ou "naming") et la "versionisation" des API sont des aspects cruciaux du développement d'applications, y compris dans le contexte de Spring Boot. Voici une explication de ces concepts en français :

1. **Nomenclature des API** : La nomenclature des API se réfère à la manière dont les endpoints (points de terminaison) et les ressources sont nommés dans une API REST. Il est important de choisir des noms descriptifs et cohérents pour les endpoints afin de faciliter la compréhension et l'utilisation de l'API par les développeurs clients.

Par exemple, au lieu d'avoir des URLs ambiguës comme **/getdata** ou **/fetch**, il est recommandé d'utiliser des noms significatifs comme **/utilisateurs** pour récupérer une liste d'utilisateurs ou **/commandes** pour obtenir les commandes d'un utilisateur.

1. **Versionisation des API** : La versionisation des API consiste à gérer les différentes versions d'une API au fil du temps. À mesure que votre application évolue, il peut être nécessaire d'apporter des modifications à l'API existante, ce qui peut potentiellement entraîner des incompatibilités avec les clients existants. La versionisation permet de gérer ces changements de manière transparente pour les clients de l'API.

Il existe différentes approches pour versionner une API, telles que l'ajout d'un numéro de version dans l'URL (par exemple, **/v1/utilisateurs**), l'utilisation de paramètres de requête (par exemple, **/utilisateurs?version=1**), ou même l'utilisation de l'en-tête de la requête HTTP pour spécifier la version.

En utilisant une nomenclature claire et une versionisation appropriée, vous pouvez garantir que votre API est facilement compréhensible, extensible et compatible avec les clients existants.

En pratique avec Spring Boot, la nomenclature des endpoints est généralement définie à l'aide des annotations dans les contrôleurs (par exemple, **@GetMapping("/utilisateurs")**). Pour la versionisation, vous pouvez choisir une méthode qui convient le mieux à vos besoins et utiliser des annotations ou des configurations pour gérer les versions d'API dans votre application Spring Boot.

* 1. **Record**

En Java, à partir de la version 14, les "records" sont une nouvelle fonctionnalité du langage qui introduit une syntaxe concise pour la définition de classes qui servent principalement à stocker des données. Les records simplifient la création de classes immuables avec un ensemble de champs de données, ainsi que la génération automatique de méthodes telles que les getters, les equals, les hashCode et les toString. Voici un exemple de la syntaxe d'un record en Java :

**Une image contenant texte, capture d’écran, Police, Logiciel multimédia

Description générée automatiquement**

Dans cet exemple, **Person** est un record qui contient deux composants : **name** de type **String** et **age** de type **int**. Lorsqu'un record est déclaré de cette manière, le compilateur génère automatiquement les éléments suivants :

* Un constructeur pour initialiser les champs.
* Des méthodes de getters pour accéder aux champs (**getName()** et **getAge()**).
* Les méthodes **equals**, **hashCode** et **toString** sont également générées automatiquement en utilisant les valeurs des champs pour fournir une implémentation équivalente aux objets standards.

Voici un exemple d'utilisation d'un record :

**Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement**

Les records sont souvent utilisés pour représenter des données immuables dans les applications Java modernes, notamment dans les services RESTful, la sérialisation/désérialisation JSON, la manipulation de données dans les applications de traitement des données, etc. Ils offrent une syntaxe concise et claire pour définir des classes simples dont le principal objectif est de stocker des données.

package com.houssam;  
  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
import java.time.Period;  
import java.util.List;  
  
@SpringBootApplication  
@RestController  
public class Main {  
  
 public static void main(String[] args){  
 SpringApplication.*run*(Main.class,args);  
 }  
  
 @GetMapping("/greet")  
 public Greet greet(){  
 return new Greet("Hello",List.*of*("Java","Golang","Javascript"),new Person("Houssam" , 30 , 15));  
 }  
  
 record Person(String name , int age , double savings ){}  
  
 record Greet(  
 String green,  
 List<String> favProgramingLanguages,  
 Person person  
 ){}  
  
 @GetMapping("/user")  
 private User getUser(){  
 return new User(1L,"houssam");  
 }  
  
 private class User{  
  
 private Long id;  
 private String name;  
  
 public User(Long id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 public Long getIdUtilisateur() {  
 return id;  
 }  
  
 public String getNameUtilisateur() {  
 return name;  
 }  
 }  
  
}

****

1. **Dependency Injection**

L'injection de dépendances en Spring Boot est utilisée pour gérer les dépendances entre les différents composants d'une application. Plutôt que d'instancier directement les dépendances dans une classe, l'injection de dépendances permet de déléguer cette responsabilité à un conteneur Spring, qui s'occupe de créer et d'injecter les dépendances appropriées au moment de l'exécution.

L'utilisation de l'injection de dépendances présente plusieurs avantages :

1. **Flexibilité** : Vous pouvez facilement changer les dépendances sans modifier le code de votre application. Cela rend l'application plus modulaire et permet d'adopter plus facilement de nouvelles fonctionnalités ou de remplacer des composants obsolètes.
2. **Facilité de test** : En utilisant l'injection de dépendances, vous pouvez facilement substituer les dépendances par des objets factices (mocks) ou des doubles (stubs) lors des tests unitaires. Cela facilite le test des différentes parties de votre application de manière isolée.
3. **Réutilisabilité** : Les composants de votre application deviennent plus réutilisables car ils sont indépendants les uns des autres. Ils peuvent être utilisés dans différentes parties de l'application ou même dans d'autres projets sans modification.
4. **Séparation des préoccupations (SoC)** : L'injection de dépendances favorise une meilleure séparation des préoccupations en encourageant une conception orientée objet où chaque classe a une responsabilité unique. Cela rend le code plus lisible, maintenable et évolutif.

En ce qui concerne les annotations, en plus de l'utilisation des annotations d'injection de dépendances telles que **@Autowired**, **@Inject**, ou **@Resource**, Spring Boot propose également d'autres annotations pour la configuration et la gestion des composants, tels que :

* **@Component**, **@Service**, **@Repository**, et **@Controller** : Ces annotations sont utilisées pour marquer les classes en tant que composants Spring gérés par le conteneur IoC.
* **@Qualifier** : Cette annotation est utilisée pour spécifier quelle implémentation d'une interface doit être injectée lorsqu'il y a plusieurs implémentations disponibles.
* **@Value** : Cette annotation est utilisée pour injecter des valeurs provenant de propriétés de configuration ou d'expressions Spring EL.

En résumé, l'injection de dépendances en Spring Boot permet de simplifier la gestion des dépendances entre les composants de l'application, ce qui conduit à un code plus flexible, testable et réutilisable.

Haut du formulaire

* + - **Changement d’implementation**

Pour changer l'implémentation d'une interface tout en utilisant l'injection de dépendances dans Spring Boot, vous pouvez suivre quelques étapes simples :

1. **Définissez l'interface** : Tout d'abord, définissez une interface qui représente le comportement que vous souhaitez avoir dans différentes implémentations. Par exemple :

**Une image contenant texte, capture d’écran, Police, Graphique

Description générée automatiquement**

**Implémentez différentes versions** : Créez différentes implémentations de cette interface en fonction des besoins. Par exemple :

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

1. **Sélectionnez l'implémentation à utiliser** : Pour sélectionner l'implémentation à utiliser, vous pouvez soit changer l'annotation **@Component** pour indiquer explicitement quelle implémentation doit être utilisée, soit configurer cela dynamiquement à l'aide d'une propriété de configuration ou d'une condition.

Par exemple, en utilisant l'annotation **@Qualifier** pour sélectionner explicitement une implémentation :

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

Cela indique à Spring d'injecter l'implémentation nommée "implementation1" de l'interface **MonInterface**.

En suivant ces étapes, vous pouvez facilement changer l'implémentation de votre interface sans changer le code de la classe qui l'utilise, ce qui rend votre application plus flexible et facile à maintenir.

**This is another example**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement**

**Une image contenant capture d’écran, texte, ligne

Description générée automatiquement**

1. **Beans and annotation**
2. **Annotation**

En Spring Boot, les annotations jouent un rôle crucial dans la configuration et le fonctionnement des applications. Voici quelques-unes des annotations les plus couramment utilisées en Spring Boot :

1. **@SpringBootApplication** : Cette annotation est utilisée pour marquer la classe principale de l'application Spring Boot. Elle combine trois annotations importantes : @Configuration, @EnableAutoConfiguration et @ComponentScan. Cela signifie que la classe annotée avec @SpringBootApplication est à la fois une classe de configuration, active l'auto-configuration de Spring Boot et active la recherche des beans annotés dans le package de l'application.
2. **@RestController** : Cette annotation est utilisée pour définir un contrôleur REST dans une application Spring Boot. Elle indique que les méthodes de la classe annotée seront automatiquement découvertes par Spring et exposées en tant que points de terminaison REST.
3. **@Autowired** : Cette annotation est utilisée pour injecter des dépendances dans les beans de l'application. Lorsqu'elle est placée sur un champ, un constructeur ou une méthode setter, Spring effectuera l'injection de dépendances automatiquement lors de la création du bean.
4. **@RequestMapping** : Cette annotation est utilisée pour mapper les requêtes HTTP aux méthodes de contrôleur dans une application Spring Boot. Elle permet de spécifier le chemin de l'URL et la méthode HTTP à laquelle la méthode annotée répondra.
5. **@Service**, **@Repository**, **@Component** : Ces annotations sont utilisées pour marquer les classes comme des composants de l'application. Ils sont utilisés pour activer la détection automatique des beans par Spring dans les packages spécifiés.
6. **@Configuration** : Cette annotation est utilisée pour marquer une classe comme une source de définitions de bean. Les méthodes annotées avec @Bean à l'intérieur d'une classe annotée avec @Configuration seront traitées comme des méthodes de création de bean.
7. **@EnableScheduling** : Cette annotation est utilisée pour activer la planification des tâches dans une application Spring Boot. Elle permet d'utiliser les annotations de planification, telles que @Scheduled, pour exécuter des tâches à intervalles réguliers.
8. **@Transactional** : Cette annotation est utilisée pour indiquer que les méthodes d'une classe doivent être exécutées dans une transaction. Spring Boot gère automatiquement la gestion des transactions lorsqu'une méthode annotée avec @Transactional est appelée.

Ces annotations sont quelques-unes des nombreuses fonctionnalités fournies par Spring Boot pour simplifier le développement d'applications. En utilisant ces annotations de manière appropriée, vous pouvez configurer et développer rapidement des applications robustes et extensibles.

1. **@Bean and configuration**

**@Bean** est une annotation utilisée dans le framework Spring pour indiquer à Spring qu'une méthode de configuration dans une classe Java doit être traitée comme une méthode de création de bean. Un "bean" dans le contexte de Spring est simplement un objet géré par le conteneur IoC (Inversion of Control) de Spring. Les beans sont des éléments fondamentaux de toute application Spring.

Lorsqu'une méthode est annotée avec **@Bean**, Spring utilise cette méthode pour créer et initialiser un bean, et elle est appelée automatiquement lorsque le bean est demandé pour la première fois. La méthode peut effectuer des opérations de configuration ou de personnalisation sur le bean avant de le renvoyer.

**@Bean** offre une grande flexibilité dans la configuration des beans dans une application Spring. Par exemple, vous pouvez créer des beans qui dépendent d'autres beans, et Spring s'occupera automatiquement de l'injection des dépendances. Vous pouvez également spécifier des options de configuration avancées pour les beans, telles que la portée, les dépendances, les paramètres d'initialisation, etc.

En résumé, **@Bean** est une annotation essentielle dans le framework Spring qui permet de déclarer explicitement les beans et de les configurer dans une application Spring.

import org.springframework.beans.factory.annotation.Value;  
import org.springframework.boot.CommandLineRunner;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
  
@Configuration  
public class CustomerConfiguration {  
  
 @Value("${app.useFakeCustomerRepo:false}")  
 private Boolean useFakeCustomerRepo;  
 @Bean  
 CommandLineRunner commandLineRunner(){  
 return args ->{  
 System.*out*.println("Command line runner hooray");  
 };  
 }  
  
  
}

1. **Java Bean Validation**

En Spring Boot, vous pouvez facilement intégrer la validation des beans Java en utilisant l'API de validation de Bean Validation, qui est une spécification standard pour la validation des objets en Java. Voici comment procéder :

1. **Ajouter les dépendances** : Assurez-vous que votre projet inclut les dépendances nécessaires pour la validation des beans Java. Cela inclut généralement la dépendance **spring-boot-starter-validation**, qui est généralement incluse par défaut dans les projets Spring Boot :

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Annoter vos beans** : Annotez les champs de vos classes de modèle avec les annotations de validation de Bean Validation telles que **@NotNull**, **@Size**, **@Email**, etc., pour définir les règles de validation :

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, capture d’écran

Description générée automatiquement**

**Activer la validation dans vos contrôleurs** : Dans vos contrôleurs Spring, vous pouvez activer la validation en ajoutant l'annotation **@Valid** à votre objet de modèle. Cela déclenche la validation lors de la liaison des données dans vos points de terminaison :

**Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement**

1. **Gérer les erreurs de validation** : Si une validation échoue, Spring Boot générera automatiquement une exception **MethodArgumentNotValidException**. Vous pouvez gérer cette exception en utilisant un gestionnaire d'exceptions personnalisé ou en ajoutant un intercepteur pour intercepter les erreurs de validation.

En suivant ces étapes, vous pouvez facilement intégrer la validation des beans Java dans votre application Spring Boot, ce qui garantit que les données soumises par les utilisateurs ou reçues par votre application sont conformes aux règles de validation spécifiées. Cela aide à garantir l'intégrité des données et à améliorer la robustesse de votre application.

1. **Exception Handling**

Dans une application Spring Boot, vous pouvez gérer les exceptions de différentes manières pour garantir une gestion robuste des erreurs.

* 1. **Enable properties to view exception details**

En Spring Boot, vous pouvez activer des propriétés pour afficher des informations détaillées sur les exceptions dans la réponse HTTP. Par défaut, Spring Boot renvoie un message d'erreur générique et le code de statut HTTP lorsqu'une exception se produit. Pour activer les informations détaillées sur les erreurs, vous pouvez définir certaines propriétés dans votre fichier **application.properties** ou **application.yml**.

server:  
 port: 3030  
 error:  
 include-binding-errors: *always* include-stacktrace: *never* include-message: *always*

**Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement**

* 1. **HTTP Status Codes**

Les codes de statut HTTP sont des nombres à trois chiffres envoyés par un serveur HTTP en réponse à une requête effectuée par un client HTTP. Ils indiquent l'état du traitement de la requête. Voici les cinq catégories principales de codes de statut HTTP :

1. **1xx (Informationnel)** : Ces codes indiquent que la requête a été reçue et comprise par le serveur, mais que le traitement est toujours en cours.

Exemples :

* + 100 (Continue) : Indique que le serveur est prêt à recevoir la requête.
  + 101 (Changement de protocole) : Indique que le serveur a accepté de changer le protocole demandé par le client.

1. **2xx (Succès)** : Ces codes indiquent que la requête a été reçue, comprise et acceptée par le serveur.

Exemples :

* + 200 (OK) : La requête a réussi.
  + 201 (Créé) : La requête a été traitée avec succès et un nouveau contenu a été créé.

1. **3xx (Redirection)** : Ces codes indiquent que le client doit prendre une action supplémentaire pour compléter la requête.

Exemples :

* + 301 (Déplacé définitivement) : La ressource demandée a été déplacée de manière permanente vers une nouvelle URL.
  + 302 (Trouvé) : La ressource demandée a été temporairement déplacée vers une autre URL.

1. **4xx (Erreur du client)** : Ces codes indiquent que la requête contient une erreur de la part du client.

Exemples :

* + 400 (Bad Request) : La syntaxe de la requête est incorrecte.
  + 404 (Non trouvé) : La ressource demandée n'a pas été trouvée sur le serveur.

1. **5xx (Erreur du serveur)** : Ces codes indiquent que le serveur a rencontré une erreur lors du traitement de la requête.

Exemples :

* + 500 (Internal Server Error) : Le serveur a rencontré une erreur interne et n'a pas pu terminer la requête.
  + 503 (Service Unavailable) : Le serveur n'est pas disponible pour répondre à la requête en raison d'une surcharge temporaire ou de la maintenance du serveur.

Ces catégories de codes de statut HTTP aident à communiquer efficacement l'état des requêtes entre les serveurs et les clients HTTP.

* 1. **Custom Exception and @ResposeStatus**

Les exceptions personnalisées avec des codes de statut de réponse personnalisés sont souvent utilisées dans les applications Spring Boot pour fournir des réponses HTTP spécifiques lorsqu'une erreur se produit. Voici comment créer une exception personnalisée avec un code de statut de réponse personnalisé en français :

1. **Création d'une exception personnalisée :**

package com.houssam.Exception;  
  
import org.springframework.http.HttpStatus;  
import org.springframework.web.bind.annotation.ResponseStatus;  
  
@ResponseStatus(HttpStatus.*NOT\_FOUND*)  
public class NotFoundException extends RuntimeException {  
 public NotFoundException(String message) {  
 super(message);  
 }  
  
}

public Optional<Customer> getCustomerById(Integer id){  
  
 if(!this.customerRepo.findById(id).isPresent())  
 throw new NotFoundException( "Customet with :"+ id+" N=not found");  
  
 return this.customerRepo.findById(id);  
}

* 1. **ExceptionHandler**

L'annotation **@ExceptionHandler** est utilisée dans Spring Boot pour gérer les exceptions spécifiques ou génériques levées dans les contrôleurs. Elle permet de définir des méthodes de gestion des exceptions à l'intérieur du contrôleur lui-même. Voici comment l'utiliser :

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

package com.houssam.Exception;  
  
import org.springframework.http.HttpStatus;  
  
import java.time.ZonedDateTime;  
  
public class ApiException {  
  
 private final String message;  
 private final Throwable throwable;  
 private final HttpStatus httpStatus;  
 private final ZonedDateTime zonedDateTime;  
  
 public ApiException(String message, Throwable throwable, HttpStatus httpStatus, ZonedDateTime zonedDateTime) {  
 this.message = message;  
 this.throwable = throwable;  
 this.httpStatus = httpStatus;  
 this.zonedDateTime = zonedDateTime;  
 }  
  
 public String getMessage() {  
 return message;  
 }  
  
 public Throwable getThrowable() {  
 return throwable;  
 }  
  
 public HttpStatus getHttpStatus() {  
 return httpStatus;  
 }  
  
 public ZonedDateTime getZonedDateTime() {  
 return zonedDateTime;  
 }  
  
 @Override  
 public String toString() {  
 return "ApiException{" +  
 "message='" + message + '\'' +  
 ", throwable=" + throwable +  
 ", httpStatus=" + httpStatus +  
 ", zonedDateTime=" + zonedDateTime +  
 '}';  
 }  
}

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

Ce code représente un gestionnaire d'exceptions global pour votre application Spring Boot. Voici ce qu'il fait :

* Il est annoté avec **@ControllerAdvice**, ce qui signifie qu'il s'agit d'un composant global qui peut intercepter les exceptions levées dans n'importe quel contrôleur de votre application.
* Il contient deux méthodes annotées avec **@ExceptionHandler**, chacune gérant une exception spécifique :
  1. La méthode **handleApiRequestException** gère les exceptions de type **ApiRequestException**. Elle crée une instance d'**ApiException** avec le message d'erreur, l'exception d'origine, le code de statut HTTP (ici **HttpStatus.BAD\_REQUEST**), et la date et l'heure actuelles. Ensuite, elle renvoie une réponse **ResponseEntity** contenant cette **ApiException** avec le code de statut HTTP correspondant.
  2. La méthode **handleNotFoundException** gère les exceptions de type **NotFoundException**. Elle fonctionne de la même manière que **handleApiRequestException**, mais avec un code de statut HTTP différent (ici **HttpStatus.NOT\_FOUND**).
* Les classes **ApiRequestException** et **NotFoundException** doivent être des classes d'exceptions définies par l'utilisateur. Dans cet exemple, elles ne sont pas incluses, mais elles doivent étendre **RuntimeException** ou une de ses sous-classes.
* La classe **ApiException** est également utilisée dans ce code pour encapsuler les détails de l'exception. Elle n'est pas fournie dans le code fourni, mais elle pourrait être une classe que vous avez définie pour représenter des exceptions spécifiques dans votre application.

1. **Databases && spring Data JPA**
2. **N Tier Diagram**

**Une image contenant texte, capture d’écran, diagramme, conception

Description générée automatiquement**

Un diagramme N-Tier est un diagramme architectural qui représente la structure d'une application logicielle en plusieurs couches ou niveaux (appelés "tiers"). Chaque niveau est responsable de différentes tâches et possède une responsabilité claire dans l'application. Les architectures N-Tier sont couramment utilisées dans le développement logiciel pour créer des applications modulaires, évolutives et faciles à maintenir. Voici une explication des principaux niveaux typiquement présents dans un diagramme N-Tier :

1. **Présentation (Presentation Tier) :**
   * Ce niveau représente l'interface utilisateur de l'application, qui interagit directement avec les utilisateurs finaux.
   * Il est généralement composé de composants tels que des pages web, des interfaces graphiques utilisateur (GUI), des applications mobiles, etc.
   * Son rôle est de présenter les données de manière conviviale à l'utilisateur et de capturer les entrées utilisateur.
2. **Logique Métier (Business Logic Tier) :**
   * Ce niveau contient la logique métier de l'application, également appelée "couche métier".
   * Il traite les opérations métier de l'application, telles que les calculs, les validations, les transformations de données, etc.
   * Il est généralement implémenté à l'aide de classes ou de services qui encapsulent la logique métier de manière à être réutilisables et testables.
3. **Accès aux Données (Data Access Tier) :**
   * Ce niveau est responsable de l'accès aux données stockées dans une base de données ou tout autre système de stockage de données.
   * Il interagit avec la couche métier pour récupérer et enregistrer les données nécessaires aux opérations de l'application.
   * Il peut être mis en œuvre à l'aide de classes d'accès aux données, d'ORM (Object-Relational Mapping), de services web, etc.
4. **Infrastructure (Infrastructure Tier) :**
   * Ce niveau fournit l'infrastructure et les services nécessaires au bon fonctionnement de l'application.
   * Il comprend des composants tels que des serveurs d'application, des systèmes de gestion de base de données, des services de messagerie, etc.
   * Son rôle est de fournir un environnement d'exécution fiable et sécurisé pour l'application.

Voici un exemple simplifié d'un diagramme N-Tier :

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

Dans une architecture N-Tier bien conçue, chaque niveau est indépendant des autres niveaux et communique uniquement avec les niveaux adjacents. Cela favorise la modularité, la réutilisabilité et la maintenabilité de l'application. De plus, l'architecture N-Tier permet souvent une évolutivité horizontale, où chaque niveau peut être mis à l'échelle indépendamment pour répondre à la demande croissante.

1. **Model**

**Une image contenant texte, capture d’écran, diagramme, conception

Description générée automatiquement**

En Spring Boot, un "Modèle" fait généralement référence à une classe Java qui représente les données transférées entre les différentes couches de l'application, telles que la couche Contrôleur et la couche Vue (dans l'architecture MVC) ou entre différents composants dans un système plus complexe.

Voici comment vous utilisez généralement un modèle dans une application Spring Boot :

1. **Créer une Classe Modèle :** Vous créez une classe Java qui représente la structure des données avec lesquelles vous souhaitez travailler. Cette classe se compose généralement de champs privés représentant les attributs des données et de méthodes getters et setters publics pour accéder et modifier ces attributs.

Exemple :

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Utiliser le Modèle dans les Contrôleurs :** Dans vos contrôleurs Spring MVC, vous utilisez les classes de modèle pour transférer des données entre le client et le serveur. Vous peuplez des instances des classes de modèle avec des données reçues des requêtes HTTP ou de la base de données, puis les passez aux vues pour qu'elles soient rendues.

Exemple :

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Sérialiser en JSON :** Spring Boot sérialise automatiquement vos objets de modèle en JSON (ou d'autres formats comme XML) lorsqu'ils sont retournés à partir des méthodes de contrôleur annotées avec **@RestController** ou **@ResponseBody**.

Exemple :

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

1. Dans cet exemple, l'objet **Utilisateur** retourné par la méthode **getUtilisateurParId** sera automatiquement sérialisé en JSON avant d'être envoyé en tant que réponse HTTP.
2. **Utiliser le Modèle dans les Vues (Optionnel) :** Si vous utilisez le rendu côté serveur avec des technologies comme Thymeleaf ou JSP, vous pouvez passer des objets de modèle aux modèles de vue pour rendre dynamiquement du contenu HTML en fonction des données.

En résumé, un "Modèle" en Spring Boot est une classe Java utilisée pour représenter des données au sein de l'application. Il joue un rôle crucial dans le transfert de données entre différents composants et couches de l'application.

* **Example for an MVC architecture**
* @SpringBootApplication   
  public class Main {  
    
   public static void main(String[] args){  
   SpringApplication.*run*(Main.class,args);  
   }

import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.stereotype.Repository;  
  
  
@Repository  
public interface CustomerRepo extends JpaRepository<Customer,Integer> {  
}

import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.stereotype.Service;  
import org.springframework.web.bind.annotation .PathVariable;  
  
import java.util.List;  
import java.util.Optional;  
  
@Service  
public class CustomerService {  
  
 @Autowired  
 @Qualifier("fake")  
 public CustomerRepo customerRepo;  
  
// record NewCustomerRequest(String name,String email,Integer age){}  
  
 public List<Customer> getCustomer(){  
 return this.customerRepo.findAll();  
 }  
  
 public void addCustomer(Customer request){  
  
 Customer customer=new Customer();  
 customer.setName(request.getName());  
 customer.setEmail(request.getEmail());  
 customer.setAge(request.getAge());  
  
 this.customerRepo.save(customer);  
  
 }  
  
 public Optional<Customer> updateCustomer(Customer request, @PathVariable("CustomerId") Integer id) {  
  
 return this.customerRepo.findById(id).map(  
 customer -> {  
 customer.setAge(request.getAge());  
 customer.setName(request.getName());  
 customer.setEmail(request.getEmail());  
 return this.customerRepo.save(customer);  
 }  
 );  
  
 }  
  
 public void deleteCustomer(Integer id){  
 this.customerRepo.deleteById(id);  
 }  
  
  
}

import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
import java.util.Optional;  
  
@RestController  
@RequestMapping("api/v1/customer")  
public class CustomerController {  
  
 @Autowired  
 private CustomerService CustomerService;  
  
 @GetMapping  
 public List<Customer> getCustomer(){  
 return this.CustomerService.getCustomer();  
 }  
  
 @PostMapping  
 public void addCustomer(@RequestBody Customer request){  
  
 this.CustomerService.addCustomer(request);  
  
 }  
  
 @PutMapping("{CustomerId}")  
 public Optional<Customer> replaceEmployee(@RequestBody Customer request, @PathVariable("CustomerId") Integer id) {  
  
 return this.CustomerService.updateCustomer(request,id);  
  
 }  
  
 @DeleteMapping("{CustomerId}")  
 public void deleteCustomer(@PathVariable("CustomerId") Integer id){  
 this.CustomerService.deleteCustomer(id);  
 }  
}

1. **DB & JPA**

**Une image contenant texte, capture d’écran, diagramme, logiciel

Description générée automatiquement**

* 1. **JPA**

JPA (Java Persistence API) est une spécification Java qui définit une interface de programmation pour la gestion de la persistance des données dans les applications Java. Elle fournit une méthode standard pour que les développeurs Java interagissent avec les bases de données relationnelles à l'aide d'objets Java.

Voici quelques points clés à retenir sur JPA :

1. **Abstraction de la couche de persistance :** JPA permet aux développeurs de travailler avec des objets Java plutôt qu'avec des requêtes SQL directes. Il offre une abstraction de la couche de persistance, ce qui permet de manipuler les données de la base de données en utilisant des entités Java plutôt que des requêtes SQL explicites.
2. **Mapping Objet-Relationnel (ORM) :** JPA utilise le concept de mapping objet-relationnel pour mapper les objets Java aux tables de la base de données et leurs attributs aux colonnes. Cela permet de manipuler les données de manière transparente en utilisant des objets Java et d'ignorer les détails de la structure de la base de données sous-jacente.
3. **Annotations et XML :** JPA peut être configuré à l'aide d'annotations Java ou de fichiers de configuration XML. Les annotations sont souvent utilisées car elles permettent une configuration plus concise et sont intégrées directement dans le code source des entités Java.
4. **API de requête :** JPA fournit une API de requête (par exemple, JPQL - Java Persistence Query Language) qui permet d'écrire des requêtes orientées objet pour interagir avec les entités persistantes. Ces requêtes sont indépendantes de la base de données et peuvent être portables entre les différents fournisseurs de persistance JPA.
5. **Gestion du cycle de vie des objets :** JPA gère le cycle de vie des objets persistants, ce qui signifie qu'il gère automatiquement la création, la mise à jour et la suppression des objets dans la base de données en fonction des opérations effectuées sur les objets Java.
6. **Intégration avec les frameworks :** JPA est souvent utilisé en combinaison avec d'autres frameworks Java EE, tels que Spring Framework et Java EE, pour faciliter le développement d'applications d'entreprise complètes.

En résumé, JPA fournit une interface de programmation standard pour la gestion de la persistance des données dans les applications Java, permettant aux développeurs de travailler avec des objets Java plutôt que des requêtes SQL directes. Elle offre une abstraction de la couche de persistance et facilite le développement d'applications robustes et évolutives.

* 1. **SptingDataJpa**

Spring Data JPA est un sous-projet de Spring Data qui simplifie le développement d'applications reposant sur JPA (Java Persistence API). Il fournit une couche d'abstraction supplémentaire au-dessus de JPA pour faciliter la mise en œuvre de la persistance des données dans les applications Java Spring.

Voici quelques points clés à retenir sur Spring Data JPA :

1. **Abstraction de la couche d'accès aux données :** Spring Data JPA offre une abstraction de haut niveau pour la couche d'accès aux données en utilisant des interfaces de référentiel (repository) pour interagir avec la base de données. Ces interfaces de référentiel fournissent des méthodes prédéfinies pour effectuer des opérations courantes telles que l'insertion, la mise à jour, la suppression et la recherche d'entités.
2. **Réduction du code boilerplate :** En utilisant les interfaces de référentiel, Spring Data JPA élimine une grande partie du code boilerplate nécessaire pour interagir avec la base de données. Les développeurs peuvent définir des interfaces de référentiel avec des méthodes déclaratives, et Spring Data JPA générera automatiquement l'implémentation correspondante à l'exécution.
3. **Personnalisation des requêtes :** Spring Data JPA offre plusieurs options pour personnaliser les requêtes, y compris l'utilisation de méthodes de nommage conventionnelles, de requêtes JPQL (Java Persistence Query Language), de requêtes dérivées basées sur la signature des méthodes, de critères de requête et de spécifications.
4. **Pagination et tri :** Spring Data JPA facilite la pagination et le tri des résultats de requête en fournissant des méthodes prédéfinies pour ces opérations dans les interfaces de référentiel.
5. **Intégration avec d'autres modules Spring :** Spring Data JPA s'intègre parfaitement avec d'autres modules Spring tels que Spring MVC, Spring Boot et Spring Security, ce qui permet de développer des applications web robustes et évolutives.
6. **Support pour différents fournisseurs JPA :** Spring Data JPA prend en charge plusieurs fournisseurs JPA, tels que Hibernate, EclipseLink, OpenJPA, etc., ce qui permet aux développeurs de choisir le fournisseur qui convient le mieux à leurs besoins.

En résumé, Spring Data JPA simplifie le développement d'applications basées sur JPA en fournissant une abstraction de haut niveau pour la couche d'accès aux données et en réduisant le code boilerplate. Il offre des fonctionnalités avancées pour personnaliser les requêtes, paginer les résultats et s'intègre harmonieusement avec d'autres modules Spring pour créer des applications Java robustes et efficaces.

* 1. **Create and configure our DB**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

**Une image contenant texte, Appareils électroniques, capture d’écran, ordinateur

Description générée automatiquement**

Ce code Java définit une classe **Customer** qui est une entité persistante dans une application Java utilisant JPA (Java Persistence API).

Analysons le code ligne par ligne :

1. Les annotations **@Entity**, **@Id**, **@SequenceGenerator**, et **@GeneratedValue** sont des annotations de JPA utilisées pour mapper la classe **Customer** à une table dans la base de données et spécifier la génération de la clé primaire.
2. **@Entity** : Cette annotation indique que la classe **Customer** est une entité JPA, ce qui signifie qu'elle sera mappée à une table dans la base de données.
3. **@Id** : Cette annotation spécifie que le champ **id** de la classe **Customer** est la clé primaire de l'entité.
4. **@SequenceGenerator** : Cette annotation définit un générateur de séquence pour générer les valeurs de la clé primaire. Elle est utilisée en conjonction avec **@GeneratedValue** pour spécifier comment la valeur de la clé primaire doit être générée.
5. **@GeneratedValue** : Cette annotation spécifie la stratégie de génération de la valeur de la clé primaire. Dans cet exemple, la valeur de la clé primaire sera générée à l'aide d'une séquence spécifiée par **@SequenceGenerator**.
6. Les champs **name**, **email**, et **age** sont les attributs de la classe **Customer** qui seront mappés aux colonnes de la table dans la base de données.
7. Le constructeur **Customer** est utilisé pour créer des instances de la classe **Customer** avec les valeurs spécifiées pour chaque attribut. Notez que le champ **id** n'est pas initialisé dans ce constructeur, car sa valeur sera générée automatiquement par JPA lors de la persistance de l'entité.

En résumé, ce code définit une classe **Customer** en tant qu'entité JPA avec une clé primaire générée par une séquence spécifiée. Cette classe peut être utilisée pour représenter et manipuler des enregistrements de clients dans une base de données relationnelle à l'aide de JPA.

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

Ce code définit une interface **CustomerRepository** qui étend l'interface **JpaRepository** fournie par Spring Data JPA.

Analysons le code ligne par ligne :

1. **package com.houssam;** : Cette ligne déclare le package dans lequel se trouve l'interface **CustomerRepository**.
2. **import org.springframework.data.jpa.repository.JpaRepository;** : Cette ligne importe l'interface **JpaRepository** fournie par Spring Data JPA. Cette interface fournit de nombreuses méthodes prédéfinies pour effectuer des opérations CRUD (Create, Read, Update, Delete) sur une entité JPA.
3. **public interface CustomerRepository extends JpaRepository<Customer,Integer>** : Cette ligne définit l'interface **CustomerRepository** qui étend **JpaRepository**. La syntaxe **extends JpaRepository<Customer, Integer>** indique que **CustomerRepository** est un repository pour l'entité **Customer**, où **Customer** est la classe d'entité, et **Integer** est le type de la clé primaire de l'entité **Customer**.

En étendant **JpaRepository**, l'interface **CustomerRepository** hérite de nombreuses méthodes prédéfinies pour effectuer des opérations CRUD sur l'entité **Customer**, telles que **save()**, **findById()**, **findAll()**, **deleteById()**, etc. Ces méthodes sont utilisées pour interagir avec la base de données et manipuler les enregistrements de la table **Customer**.

En résumé, l'interface **CustomerRepository** fournit un moyen pratique d'effectuer des opérations CRUD sur l'entité **Customer** en utilisant des méthodes prédéfinies fournies par Spring Data JPA, ce qui permet de simplifier le code et d'accéder facilement à la base de données à partir de l'application Spring.

1. **CRUD API**

**Une image contenant texte, capture d’écran, diagramme, ligne

Description générée automatiquement**

Une API CRUD expose des points de terminaison (endpoints) qui permettent aux clients d'effectuer ces opérations sur les données. Ces points de terminaison sont généralement accessibles via HTTP, et chaque opération CRUD est généralement associée à une méthode HTTP spécifique :

* Create : POST
* Read : GET
* Update : PUT ou PATCH
* Delete : DELETE

Une API CRUD bien conçue offre une interface simple et cohérente pour interagir avec les données, ce qui facilite le développement d'applications et de systèmes qui manipulent des données persistantes.

import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
import java.util.Optional;  
  
@SpringBootApplication  
@RestController  
@RequestMapping("api/v1/customer")  
public class Main {  
  
  
 @Autowired  
 private CustomerRepository customerRepository;  
 public static void main(String[] args){  
 SpringApplication.*run*(Main.class,args);  
 }  
  
 @GetMapping  
 public List<Customer> getCustomer(){  
 return this.customerRepository.findAll();  
 }  
  
 record NewCustomerRequest(String name,String email,Integer age){}  
  
 @PostMapping  
 public void addCustomer(@RequestBody NewCustomerRequest request){  
  
 Customer customer=new Customer();  
 customer.setName(request.name);  
 customer.setEmail(request.email);  
 customer.setAge(request.age);  
  
 this.customerRepository.save(customer);  
  
 }  
  
 @PutMapping("{CustomerId}")  
 public Optional<Customer> replaceEmployee(@RequestBody NewCustomerRequest request, @PathVariable("CustomerId") Integer id) {  
  
 return this.customerRepository.findById(id).map(  
 customer -> {  
 customer.setAge(request.age);  
 customer.setName(request.name);  
 customer.setEmail(request.email);  
 return this.customerRepository.save(customer);  
 }  
 );  
  
 }  
  
 @DeleteMapping("{CustomerId}")  
 public void deleteCustomer(@PathVariable("CustomerId") Integer id){  
 this.customerRepository.deleteById(id);  
 }

}