# Springboot, ce qu’il faut savoir

Dans le contexte des API (Interfaces de Programmation d'Application), une **route** fait référence à un chemin spécifique dans une application web ou serveur API qui correspond à une URL particulière. Les routes définissent les points d'accès auxquels les clients peuvent accéder pour effectuer certaines opérations ou obtenir des informations spécifiques.

Une route est généralement associée à une méthode HTTP spécifique (comme GET, POST, PUT, DELETE, etc.) et est utilisée pour définir comment une ressource ou une opération particulière doit être traitée. Les routes sont une composante fondamentale de la structure d'une **API RESTful** (Representational State Transfer), qui est un style architectural commun pour la conception d'API web.

Par exemple, dans une API de gestion des utilisateurs, vous pourriez avoir les routes suivantes :

* **GET /utilisateurs** : Récupérer la liste de tous les utilisateurs.
* **GET /utilisateurs/1** : Récupérer les détails de l'utilisateur avec l'ID 1.
* **POST /utilisateurs** : Créer un nouvel utilisateur.
* **PUT /utilisateurs/1** : Mettre à jour les informations de l'utilisateur avec l'ID 1.
* **DELETE /utilisateurs/1** : Supprimer l'utilisateur avec l'ID 1.

Chaque route est associée à une fonction ou un ensemble de fonctionnalités spécifiques dans le backend (généralement le contrôleur) de l'application qui détermine comment répondre à la demande du client associée à cette route.

Dans ce projet, l’architecture choisie se présente comme suit :

**Route -> Controller -> Payload (Requests & Responses) -> Service -> Service Impl -> JPARepository -> Entity**

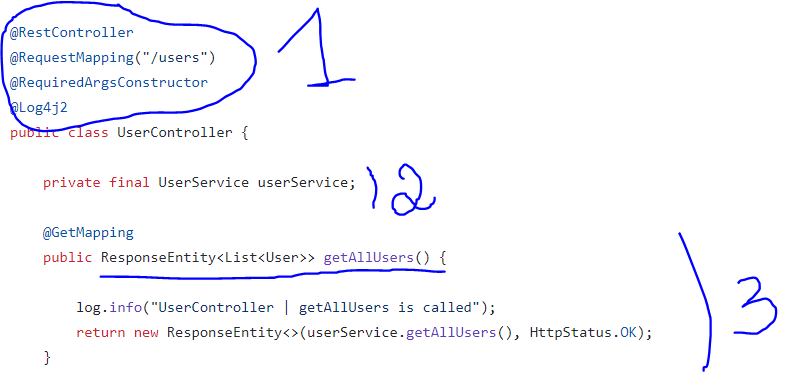
Expliquons de manière pratique.

NB : Nous partons du principe que vous savez ce que veut dire chacun de ces termes. Ce document ne parcoureras donc pas les détails futiles.

Prenons un exemple simple : **Je veux récupérer la liste des utilisateurs via l’API**.

## Lecture de la route et action

Les actions concernant les utilisateurs se passent dans le fichier **UserController.java** qui se trouve dans le dossier **controller.**



Explications :

Tout d’abord, les **@...** sont ce que nous appelons des **annotations**. Elles sont généralement utilisées pour réduire la charge de code. Vous comprendrez plus loin.

1. Explication des annotations.

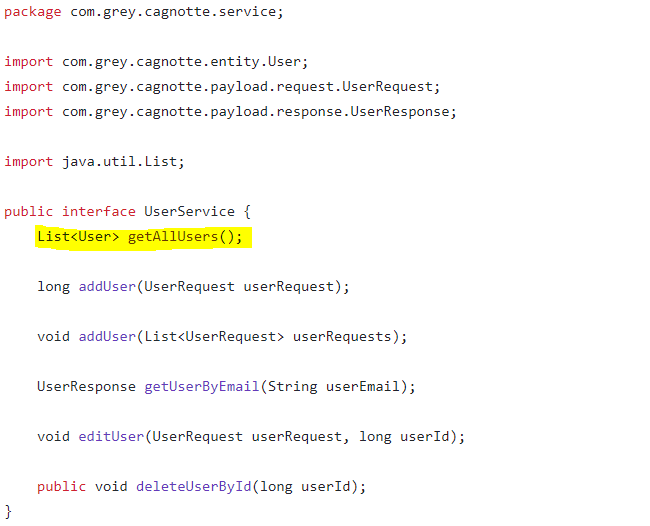
* **@RestController**: simplifie la création de services web RESTful. Il s'agit d'une annotation pratique qui combine **@Controller** et **@ResponseBody**, ce qui élimine le besoin d'annoter chaque méthode de traitement des requêtes de la classe du contrôleur avec l'annotation @ResponseBody.
* **@RequestMapping("/users") :** Il définit le mappage d'URL de base pour le contrôleur. Cela signifie que toutes les requêtes commençant par « /users » seront traitées par le contrôleur.
* **@RequiredArgsConstructor :** est une annotation **Lombok** qui génère automatiquement des constructeurs pour tous les champs finaux et non nuls. Plus besoin de coder des getters ou setters.
* **@Log4j2 :** Plus besoin de coder une logique de logging, c’est gérer automatiquement. Il suffit d’appeler la fonction requise comme fait sur la partie 3 de la figure 1.

1. On appelle juste le service qui contient les actions spécifiques.
2. La fonction qui nous intéresse. Elle renvoie la liste des utilisateurs sous forme d’objet Json.

* **@GetMapping :** Cette fonction peut prendre des parametres ou non. Lorsqu’il n’y a pas de paramètre comme sur cet exemple, elle est mappée sur l’url définie dans @RequestMapping un peut plus haut. Concrètement, cela veut dire que si je veux due :  
  - **@GetMapping = /users**- **@GetMapping("/juste-pour-voir") = /users/juste-pour-voir**
* **log.info(…) :** Vous vous rappelez de @Log4j2 ? Eh bien, voilà un exemple de son utilisation.
* **List<User>** représente l’objet à renvoyer. Ce qui implique que **userService.getAllUsers()** qui est une fonction dans le service renvoie **List<User>**.

## Il se passe quoi dans le service ?

La requête est maintenant au niveau de **UserService.java** dans le dossier **service**.



Ne cherchons pas loin, nous savons qu’une interface ne s’exécute pas directement, allons donc voir son implémentation **UserServiceImpl.java** dans le dossier **service/impl.**

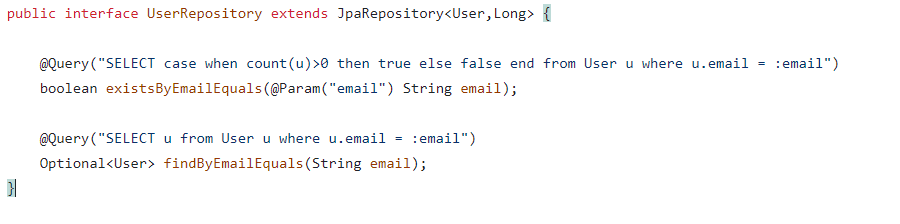


Le corps de la méthode est simple, tout se passe dans le tibia (le Repository je veux dire).

## Repository ? C’est quoi ?

**UserRepository.java** dans **repository**.

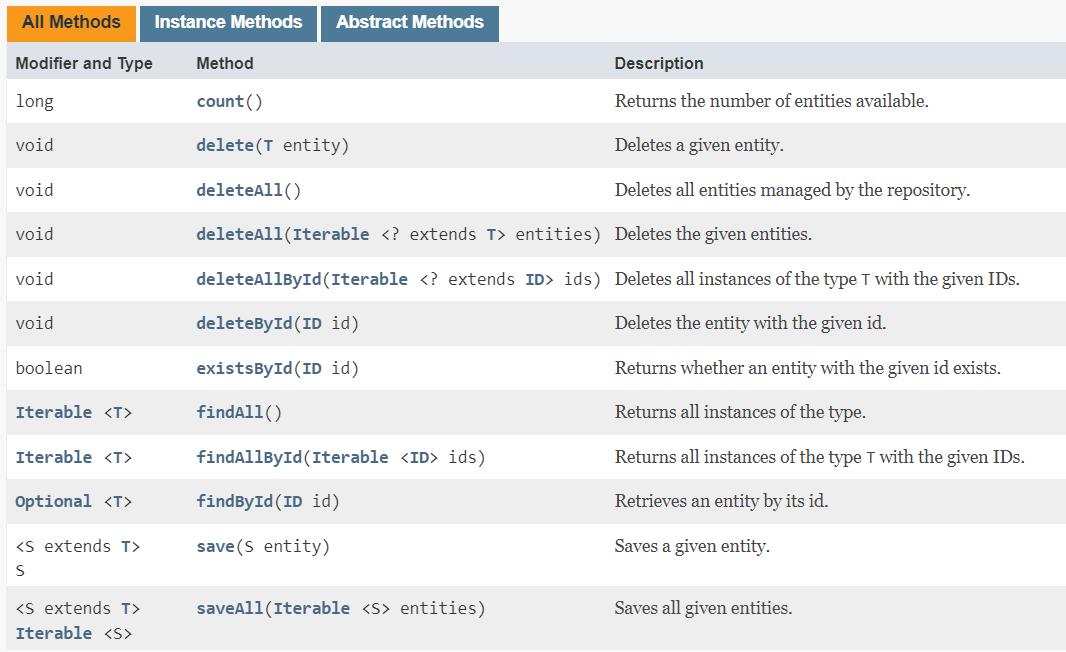
Un **repository** dans Spring Boot est utilisé pour effectuer les différentes activités sur les objets (entity), ces opérations incluent la **suppression**, le **stockage**, la **récupération**, le stockage de la liste en une seule fois, ainsi que la recherche dans la base de données en fonction de critères particuliers.



**Mais où est la fonction findAll() ?**

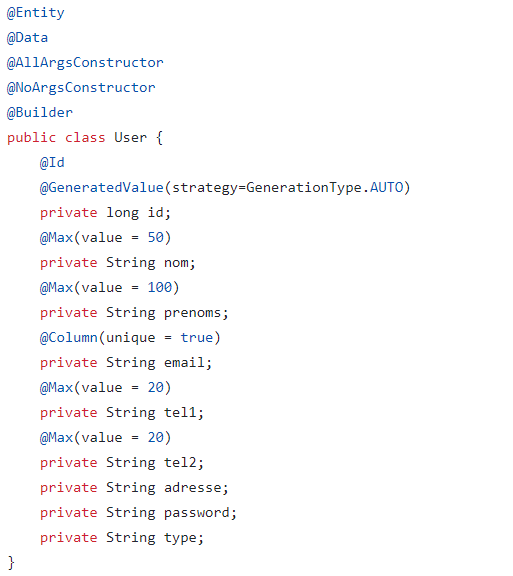
Elle est bien là, mais elle est contenue dans **JpaRepository<User, Long>.**

Dans l’approche classique, nous créons un fichier de connexion, on s’assure que le Driver est bien chargé, puis que la connexion à la base de données est bien établie…(Souvenirs de JDBC). Tout cela prend du temps, Nous avons donc dans Springboot des utilitaires qui nous permettent de réduire les actions et se concentrer sur l’essentiel.

* **JpaRepository** est un assemblage de CrudRepository et PagingAndSortingRepository. Ils contient toutes les fonctions nécessaires pour le CRUD et la pagination.  
  
* **User** est mis pour indiquer le modèle d’où il prend ses données
* **Long** représente le type de la clé primaire de l’entity User.
* **Optional<User> findByEmailEquals(String email)**: En règle générale, le mot-clé **findBy** ou **findAllBy** est utilisé pour indiquer au moteur Springboot que nous voulons faire une recherche et avoir 0 ou plusieurs objets en retour. (Dans notre cas, c’est 0 ou 1, d’où le mot-clé **Optional**).  
  Etant donné que **email** est un champ de la classe **User**, il suffit d’ajouter el lowerUpperCase le nom du champ et le tout est joué, le moteur génère automatiquement la requête correspondante.  
  En soit : **Optional<User> findByEmailEquals(String email)** = **"SELECT u from User u where u.email = :email"**.
* **Si c’est généré automatiquement, pourquoi avoir écris la requête en haut ?**  
  C’était à titre d’exemple. Vous voudriez peut-être faire des actions ou requêtes un peu complexes ou tout simplement vous assurer que la requête est correcte et éviter les surprises, pour cela, vous utilisez tout simplement l’annotation **@Query().  
   :email** représente une variable externe, elle provient toujours de la fonction en bas.
* IL peut aussi arriver que le nom du paramètre soit trop long, vous pouvez raccourcir au besoin.  
  

## L’entity, il se présente comment ?

**User.java** dans **entity**.



* **@Entity** : Permet de dire au moteur que cette classe a une représentation dans la base de données. Vous pouvez utiliser **@Table(name=)** pour donner un nom particulier à la table liée.
* **@Data** est une annotation Lombok qui créer les Getters, les Setters,.. et la méthode toString() automatiquement (très pratique)
* **@AllArgsConstructor** : crée un constructeur avec tous les champs de la classe.
* **@NoArgsConstructor** : crée le constructeur par défaut.
* **@Builder** : permet d’instancier la classe
* **@Id** et **@Generated**… vous l’avez compris, il s’agit de la clé primaire
* **@Max** : utilisé pour limiter la longueur d’une colonne dans la base de donnée.
* Très souvent, nous utilisons des variables nommées suivant le lowerUpperCase (**maVariable**). Or dans la base de données, cela devrait s’afficher **ma\_variable**. Nous avons donc l’annotation **@Column** qui peut nous permettre de palier à ça quoique elle a d’autre utilisations : @Column(name=…)

Voilà, c’est tout, je vous laisse découvrir le reste par vous-même, suivez le chemin indiqué :

**Route -> Controller -> Payload (Requests & Responses) -> Service -> Service Impl -> JPARepository -> Entity**

La documentation est accessible sur le **/swagger-ui.html**. Les packages sont gérés dans le **pom.xml** et la connexion à la base données dans **application.yml**. La base de données est en ligne, vous n’avez donc pas à vous souciez de ça.

Pour la conception, je vous recommande l’ordre :

**Entity -> Payload (Request & Response) -> Repository -> Service -> Service Impl -> Controller.**

*Je vous remercie.*

*Cordialement,  
Grey,  
+228 93721124.*