

Créer un web service REST qui permettra de gérer un annuaire

Le web service devra permettre les opérations suivantes:

- Afficher la liste des personnes inscrites dans l'annuaire
- Afficher une personne spécifique en recherchant par id
- Ajouter une personne dans l'annuaire
- Modifier une personne dans l'annuaire
- Supprimer une personne dans l'annuaire



Créer une classe `Personne` qui contiendra les champs suivant:

- `Id`
- `Nom`
- `Prénom`

Créer une classe Annuaire qui contiendra les champs suivant:

- Une liste de Personne

Cette classe est semblable à un DAO elle devra contenir les méthode pour ajouter, supprimer et modifier des éléments dans la liste de personnes

A la première requête HTTP sur notre service web il faudra créer un annuaire et le stocker dans la session (avec `request.getSession()`)

Il faudra pour chaque méthode récupérer l'annuaire depuis la session



## TP Users API - **AuthFilter**

L'accès à ce web service devra être protégé par une authentification de type Basic Auth (identifiant + mot de passe encodé en Base64).

Pour ce faire, à chaque requête HTTP vous devrez vérifier que le client envoie ses identifiants dans les headers de la requête. La vérification se fera dans un filtre appelé **AuthFilter** afin de centraliser l'authentification.

Pour que Jersey prenne en compte notre filtre "maison" l'annotation `@Provider` est spécifié sur la classe.

L'annotation `@PreMatching` précise le fait que le filtre soit appelé avant que la requête HTTP ne soit match avec une méthode de nos ressources. Ceci nous laisse du choix de la méthode qui sera appelée ensuite.

La classe AuthFilter contiendra l'attribut request HttpServletRequest venant du **Context**

Dans le méthode "filter" vous devrez :

- Récupérer les identifiants encodé dans le header de la requête
- Si les identifiants ne sont pas présents lever l'exception appropriée
- En utilisant la classe BasicAuth, décoder les identifiants
- Si les identifiants décodé ne sont pas conforme lever l'exception appropriée
- Utiliser la méthode checkUser et stocker l'utilisateur connecté dans les attributs de la requête



## TP Users API

Créer un web service REST qui permettra de gérer des utilisateurs

Etape 1

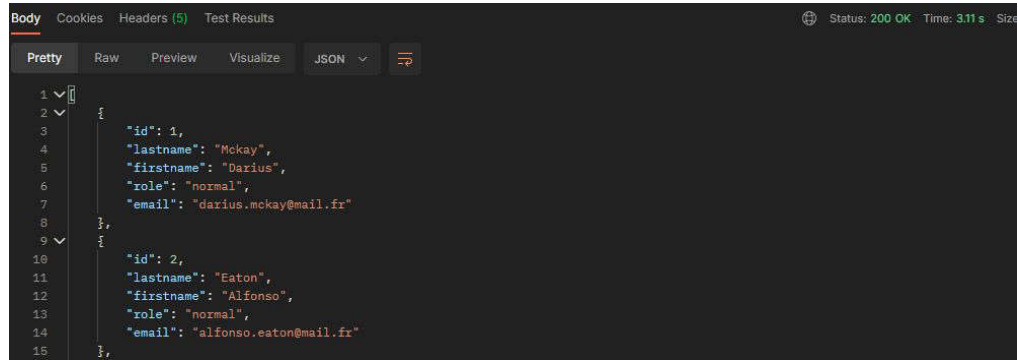
URL: GET /users

Cette route permet de récupérer les informations de tous les utilisateurs de la base de données

Le type de retour doit être un tableau JSON d'objets user

## Etape 1

Exemple de données devant être renvoyées



```
Body Cookies Headers (5) Test Results
Pretty Raw Preview Visualize JSON
1 {
2   "id": 1,
3   "lastname": "Mckay",
4   "firstname": "Darius",
5   "role": "normal",
6   "email": "darius.mckay@mail.fr"
7 },
8 {
9   "id": 2,
10  "lastname": "Eaton",
11  "firstname": "Alfonso",
12  "role": "normal",
13  "email": "alfonso.eaton@mail.fr"
14 },
15 }
```

## Etape 1

Toutes les données concernant les utilisateurs devront être renvoyées sous forme de JSON (sauf le password)

Les réponses possibles sont:

- Les utilisateurs ont bien été récupérés
  - On renvoie le status code: 200
  - On renvoie un tableau JSON avec les données utilisateurs comme sur le screenshot précédent



## Etape 1

- Si l'utilisateur n'est pas du tout connecté
  - On renvoie le status: 401
  - On renvoie le message: You must be connected
  - On ne renvoie pas les données



## TP Users API

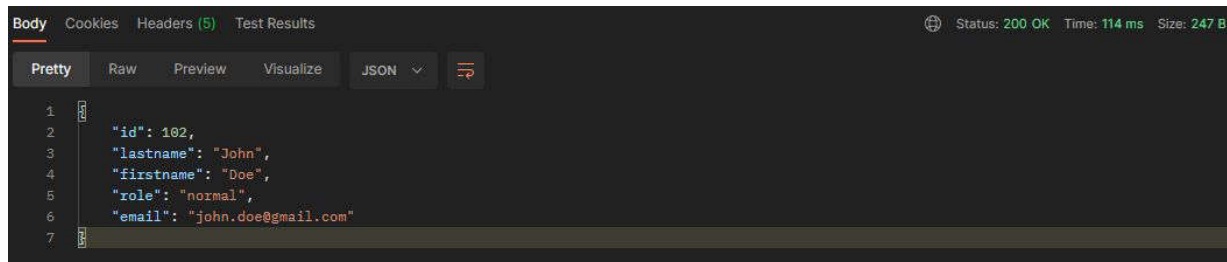
### Etape 2

Route: POST /users

Cette route permet d'ajouter un nouvel utilisateur

## Etape 2

### Exemple de données pouvant être renvoyé



The screenshot shows a REST client interface with a dark theme. The top bar indicates the status is 200 OK, the time is 114 ms, and the size is 247 B. The 'Body' tab is selected, and the response is displayed in a 'Pretty' JSON format. The JSON object contains the following fields: id (102), lastname (John), firstname (Doe), role (normal), and email (john.doe@gmail.com). The interface includes tabs for Body, Cookies, Headers (5), and Test Results, as well as buttons for Pretty, Raw, Preview, Visualize, and a JSON dropdown menu.

```
1 {
2   "id": 102,
3   "lastname": "John",
4   "firstname": "Doe",
5   "role": "normal",
6   "email": "john.doe@gmail.com"
7 }
```

### Etape 2

Aucun champ envoyé ne sera vide, une erreur sera renvoyée si c'est le cas. Les données seront présentées comme dans l'exemple vu précédemment

Les réponses possibles sont:

- L'utilisateur a bien été créé
  - On renvoie le status code: 201
  - On renvoie le message: User successfully created

## Etape 2

- Une erreur s'est produite
  - On renvoie le status code: 400
  - On renvoie le message: An error occurred
  
- Si l'utilisateur n'est pas du tout connecté
  - On renvoie le status code: 401
  - On renvoie le message: You must be connected



### Etape 3

Route: PUT /users/{id}

Cette route permet de modifier un utilisateur

Le paramètre id signifie “user id” c’est l’id de l’utilisateur en base de données

Le paramètre id est obligatoire

## Etape 3

Exemple de données pouvant être envoyé

```
1  curl -X POST http://localhost:3000/users -H 'Content-Type: application/json' -d '{
2    "lastname": "Eclair"
3  }'
```

```
1  curl -X POST http://localhost:3000/users -H 'Content-Type: application/json' -d '{
2    "lastname": "Eclair",
3    "firstname": "Buzz"
4  }'
```

## Etape 3

Les réponses possibles sont:

- L'utilisateur a bien été modifié
  - On renvoie le status code: 200
  - On renvoie le message: User successfully modified
- Une erreur s'est produite
  - On renvoie le status code : 401
  - On renvoie le message: You must be connect



## Etape 3

- Une erreur s'est produite
  - On renvoie le status code: 400
  - On renvoie le message: An error occurred
- Si l'id est invalide (donc l'utilisateur n'existe pas en BDD)
  - On renvoie le status code: 404
  - On renvoie le message: User was not found



### Etape 4

Route: DELETE /users/{id}

Cette route permet de supprimer un utilisateur

Le paramètre id signifie “user id” c’est l’id de l’utilisateur en base de données

Le paramètre id est obligatoire

## Etape 4

Les réponses possibles sont:

- L'utilisateur a bien été supprimé
  - On renvoie le status code: 200
  - On renvoie le message: User successfully deleted
- Une erreur s'est produite
  - On renvoie le status code 400
  - On renvoie le message: An error occurred

## Etape 4

- Si l'utilisateur n'est pas du tout connecté
  - On renvoie le status code: 401
  - On renvoie le message: You must be connected
- Si l'id est invalide (donc que l'utilisateur n'existe pas en BDD)
  - On renvoie le status code 404
  - On renvoie le message: User was not found



### Etape 5

Route: GET /users/{id}

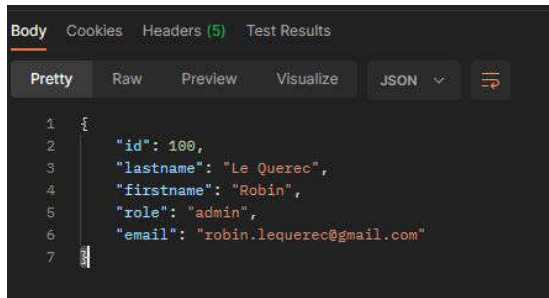
Cette route permet de récupérer les données d'un utilisateur

Le paramètre id signifie "user id" c'est l'id de l'utilisateur en base de données

Le paramètre id est obligatoire

## Etape 5

Exemple de données devant être renvoyées

A screenshot of a REST client interface. The 'Body' tab is selected, and the response is displayed in 'Pretty' format. The JSON data represents a user with the following fields: id, lastname, firstname, role, and email.

```
1 {  
2   "id": 100,  
3   "lastname": "Le Querec",  
4   "firstname": "Robin",  
5   "role": "admin",  
6   "email": "robin.lequerec@gmail.com"  
7 }
```

### Etape 5

Toutes les données concernant un utilisateur devront être renvoyées sous forme de JSON (sauf le password)

Les réponses possibles sont:

- L'utilisateur a bien été récupéré
  - On renvoie le status code: 200
  - On renvoie un objet JSON avec les données de l'utilisateur

## Etape 5

- Une erreur s'est produite
  - On renvoie le status code: 400
  - On renvoie le message: An error occurred
- Si l'utilisateur n'est pas du tout connecté
  - On renvoie le status code: 401
  - On renvoie le message: You must be connected
- Si l'id est invalide (donc que l'utilisateur n'existe pas en BDD)
  - On renvoie le status code: 404
  - On renvoie le message: User was not found



## Etape 6

Route: GET /users/search

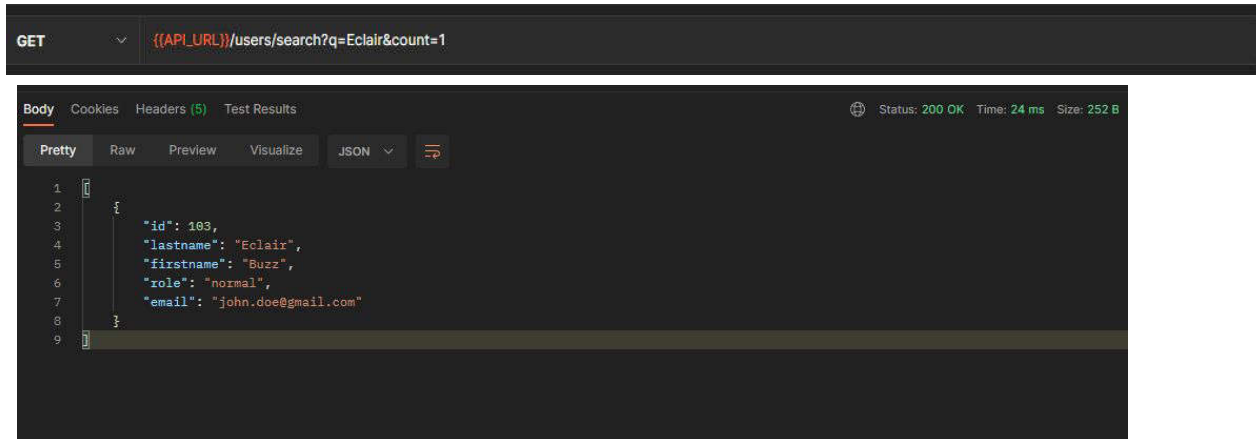
Cette route permet de chercher des utilisateur soit par leur nom de famille (lastname) soit par leur email

Les paramètres (de type GET) de la requête seront:

- q: paramètre obligatoire de type string qui représente l'élément à rechercher
- count: paramètre optionnel qui représente le nombre d'utilisateur maximum à retourner

## Etape 6

### Exemple de données pouvant être renvoyées



The screenshot shows a REST client interface with a GET request to `{{API_URL}}/users/search?q=Eclair&count=1`. The response body is displayed in JSON format, showing a single user object with the following details:

```
1 {
2   "id": 100,
3   "lastname": "Eclair",
4   "firstname": "Buzz",
5   "role": "normal",
6   "email": "john.doe@gmail.com"
7 }
8
9
```

Additional details from the interface: Status: 200 OK, Time: 24 ms, Size: 252 B.



### Etape 6

Les réponses possibles sont:

- On a trouvé des utilisateurs correspondant à la recherche
  - On renvoie le status code: 200
  - On renvoie un tableau JSON avec les données des utilisateurs qui ont été trouvé

## Etape 6

- Une erreur s'est produite
  - On renvoie le status code: 400
  - On renvoie le message: An error occurred
- Si l'utilisateur n'est pas du tout connecté
  - On renvoie le status code: 401
  - On renvoie le message: You must be connected
- Si aucun utilisateur ne correspond à la recherche
  - On renvoie le status code: 404
  - On renvoie le message: No user found



## TP Movie API

Catastrophe ! IMDb a été piraté et le code source a été effacé. Vous avez été engagé par la société pour recréer l'API Web d'IMDb

La première version de l'API devra contenir les fonctionnalités principales et essentielles au bon fonctionnement de l'application web

## Etape 1: Les entités

La base de données est structuré autour de plusieurs entités:

- Movie
- Actor
- User
- Role
- Comment
- Genre

Votre travail sera de créer ces entités et de réfléchir aux relations qu'elles auront entre elles ainsi qu'aux cascades à appliquer



## Etape 1: L'entité Genre

L'entité Genre devra contenir les champs suivant:

- Id (int)
- Name (string)

## Etape 1: L'entité Role

L'entité Role devra contenir les champs suivant:

- Id (int)
- Identifiant (string)
- Description (string)





## Etape 1: L'entité User

L'entité User devra contenir les champs suivant:

- Id (int)
- Firstname (string)
- Lastname (string)
- Nickname (string)
- Email (string)
- Role (Role)



## Etape 1: L'entité Comment

L'entité Comment devra contenir les champs suivant:

- Id (int)
- User (user)
- Content (string)
- Movie (Movie)
- CreationDate (Date)

## Etape 1: L'entité Actor

L'entité Actor devra contenir les champs suivant:

- Id (int)
- Firstname (string)
- Lastname (string)
- birthDate (Date)
- Movies (List<Movie>)

## Etape 1: L'entité Movie

L'entité Movie devra contenir les champs suivant:

- Id (int)
- Name (string)
- ReleaseDate (Date)
- Duration (int)
- Rating (float)
- Synopsis (string)
- Genres (List<Genre>)
- Actors (List<Actor>)
- OriginCountry (string)
- Languages (List<String>)
- Comments (List<Comment>)

## Etape 2: Les DAO

Pour toutes les entités créées il faudra créer les DAO correspondant qui permettent de trouver, ajouter, modifier et supprimer une entité

Pour tester ces DAO vous pouvez créer un main de test

## Etape 3: Les ressources

A présent notre base de données est correctement structurée et l'accès aux données est possible grâce aux DAO

Il faudra maintenant créer les ressources REST pour pouvoir exposer ces données

Nous allons créer une ressource par entité sauf pour l'entité Comment (qui elle est liée à l'entité Movie)

## Etape 3: Les ressources

Chaque ressource devra exposer une URL pour:

- Afficher une liste
- Afficher un élément
- Ajouter un élément
- Modifier un élément
- Supprimer un élément



## Etape 3: Les ressources

Liste des ressources à créer:

- /movies: Pour gérer l'entité Movie
- /actors: Pour gérer l'entité Actor
- /genres: Pour gérer l'entité Genre
- /users: Pour gérer l'entité User
- /roles: Pour gérer l'entité Role



## Etape 4: L'authentification

Afin de pouvoir déterminer l'utilisateur qui fait l'appel aux web services nous allons devoir mettre en place un système d'authentification de type BasicAuth

Ce système devra intercepter la requête avant qu'elle soit envoyée à la ressource et devra vérifier que l'utilisateur existe en BDD et que le mot de passe fourni est le bon

Si c'est le cas l'objet User correspondant devra être ajouté à la requête pour le rendre disponible dans nos ressources

## Etape 5: Fonctionnalités avancées

Les fonctionnalités principales étant déployée, elles ne permettent pas aux utilisateurs de l'application de trouver les films de leur acteur préféré, de rechercher un film par genre ou de recherche un film par titre tout simplement

Nous allons devoir développer ces fonctionnalités avant de pouvoir mettre l'application en production

## Etape 5: Fonctionnalités avancées

Créer les ressource suivantes:

- `/movies/{id}/comments`: Pour ajouter ou supprimer un commentaire sur un film
- `/movies/search`: Pour rechercher un film par titre
- `/movies/getByGenre`: Pour rechercher les films par genre
- `/movies/getByActor`: Pour rechercher les films par acteur (Lastname ou firstname)
- `/movies/top`: Pour voir les films les mieux notés
- `/users/{id}/comments`: Pour voir tous les commentaires rédigé par un utilisateur