Rendu du projet de création d'une plateform de gestion des joueurs et des résultats

Réalisé par
IBEGHOUCHENE Nadir
SIMONIAN Sergio

Encadré par
Grégory Galli

Au démarrage de l'application:

Notre application utilise notre base Mysql et serveur Apache à distance trouvable à l'adresse 82.255.166.104 (c'est une rasberry pi qui tourne chez nous en permanence).
Pour lancer notre application avec succès il est donc nécessaire d'avoir une connection réseau et également assurer que le port local 8090 est utilisable.
Si notre application ne se lance toujours pas il est également possible que le parefeux de votre routeur bloque l'accès à notre domaine (cela nous est arrivé avec le réseau universitaire Unice-HotSpot) , dans ce cas une solution serait de changer de routeur ou passer par le réseau mobile.
Une fois que le projet est importé correctement et lancé, la page d'accueil de l'application s'ouvre sur l'adresse : localhost:8090/mbds
A partir de la page d'accueil on peut soit:

- Se connecter autant que user/admin (les logins et mots de passes sont trouvable dans projetGrails/grails-app/init/mbds/BootStrap)
- Créer un nouveau compte user
- Lire la documentation de l'api rest

Réalisation:

Partie 1: Création d'une plateforme de gestion de joueurs et résultats

Cette première partie est développée sous le framework **Grails.**

Cette plateforme gère deux niveau d'accès via des rôles qui seront attribués aux utilisateurs

Rôle 1: Administrateur
Un seul administrateur possède l'accès au backoffice permettant d'opérer toutes les fonctions de CRUD sur le modèle de donnée existant.
Rôle 2: Utilisateur
Le rôle utilisateur correspond à celui d'un joueur ayant comme seule possibilité de soumettre et récupérer des informations sur l'API (partie 2)

Les fonctionnalités de l'administrateur:

L'administrateur peut

- créer / modifier / voir / supprimer des Users
- créer / modifier / voir / supprimer des Résultats
- créer / modifier / voir / supprimer des Messages

Les fonctionnalités du joueur (utilisateur):

- Les Joueurs pourront envoyer des messages, ces derniers sont forcément adressés à un autre Joueur (envoyer des messages à soi même et egalement possible)
- Les Joueurs s'inscriront en saisissant un username, un mot de passe et ont la possibilité de fournir une photo de profil
- Les Messages maintient un « flag » pour savoir s'ils ont été lu (pour lire le contenue d'un message il faut cliquer sur celui ci, cela déclenche une requête en ajax pour la marquer comme lu)

Bonus :

1. La possibilité d'uploader les fichiers en Ajax en faisant un simple drag'n'drop du fichier sur le champ en question i**mplémenté dans l'espace utilisateur(joueur) uniquement (pas dans celui de admin)**

2. Un cron fonctionnel pour purger les messages, toutes les nuits à 4 heures du matin, ce dernier ira effacer tous les messages lus.
   Nous nous avons servie du plugin quartz 2.0.13. Le cron se trouve dans : projetGrails/grails-app/jobs/mbds/MessagePurgeJob.

Partie 2: API REST

Notre Api rest traite des operation GET / POST / DELETE / PUT sur les 3 entités User , DeadMatch (équivalent de Match) et Message.
(Nous avons renommé la domaine Match en DeadMatch puisque nous utilisons une base des donnees Mysql et le mot “match” est un mot réservé.)

Spécifications et documentation:

(la spécification et documentations est également disponible dans le projet sur la page localhost:8090/mbds/doc)

Avant d'envoyer des requêtes :

- 1) Envoyer une requête POST a http://localhost:8090/mbds/api/login
  avec Content-Type:application/json et un Body en JSON contenant le nom de l'utilisateur (username) et le mot de passe (password)
  Exemple: {"username":"VotreNom","password":"motdepasseSecret"}
  2) Vous recevrez une réponse exemple : {"username": "VotreNom","roles": ["ROLE_ADMIN"],"access_token": "4hoieugu60lpcthh6cv5d7m7aagski6e"}
  3) Mettez par la suite consécutivement la ligne X-Auth-Token: {votre access token} dans le header de vos requêtes

Requêtes sur l'utilisateurs :

- Url: http://localhost:8090/mbds/api/user/{id}
  Requête GET sans id : renvoye la liste de tous les utilsateurs en JSON (status 200)
  Requête GET avec id valide/existant: renvoye l'utilisateur en JSON (status 200)
  Requête GET avec id invalide/inexistant: réponse "user not found" (status 404)
  Requête POST avec ou sans id contenant les champs nécessaires et valides:
  response "http://localhost:8090/mbds/api/user/{id}" (status 201)
  Requête POST avec ou sans id ne contenant pas les champs nécessaires et valides:
  response "Failed to add user - not valid champs" (status 400)
  Requête DELETE sans id : renvoie "for user delete/put use /api/user/{Your user ID}" (status 400)
  Requête DELETE avec id valide/existant: réponse vide - l'utilisateur est supprimé (status 200)

Requête DELETE avec id invalide/inexistant: réponse "user not found" (status 404)
Requête PUT sans id : renvoyé "for user delete/put use /api/user/{Your user ID}" (status 400)
Requête PUT avec id valide/existant : réponse http://localhost:8090/mbds/api/user/{id} -
l'utilisateur est mis à jour, les champs invalides seront ignorées (status 200)
Requête PUT avec id invalide/inexistant: réponse "user not found" (status 404)

Requêtes sur les DeadMatchs :

- Url: http://localhost:8090/mbds/api/deadmatch/{id}
  Requête GET sans id : renvoie la liste de tous les deadmatchs en JSON (status 200)
  Requête GET avec id valide/existant: renvoie le deadmatch en JSON (status 200)
  Requête GET avec id invalide/inexistant: réponse "deadmatch not found" (status 404)
  Requête POST avec ou sans id contenant les champs nécessaires et valides:
  response "http://localhost:8090/mbds/api/deadmatch/{id}" (status 201)
  Requête POST avec ou sans id ne contenant pas les champs nécessaire et valides:
  response "Failed to add deadmatch - not valid champs" (status 400)
  Requête DELETE sans id : renvoie "for deadmatch delete/put use /api/deadmatch/{Your deadmatch ID}" (status 400)
  Requête DELETE avec id valide/existant: réponse vide - le deadmatch est supprimé (status 200)
  Requête DELETE avec id invalide/inexistant: réponse "deadmatch not found" (status 404)
  Requête PUT sans id : renvoie "for deadmatch delete/put use /api/deadmatch/{Your deadmatch ID}" (status 400)
  Requête PUT avec id valide/existant : réponse http://localhost:8090/mbds/api/deadmatch/{id} -
  le deadmatch est mis à jour, les champs invalides seront ignorées (status 200)
  Requête PUT avec id invalide/inexistant: réponse "deadmatch not found" (status 404)

Requêtes sur les Messages :

- Url: http://localhost:8090/mbds/api/deadmatch/{id}
  Requête GET sans id : renvoie la liste de tous les messages en JSON (status 200)
  Requête GET avec id valide/existant: renvoie le messages en JSON (status 200)
  Requête GET avec id invalide/inexistant: reponse "messages not found" (status 404)
  Requête POST avec ou sans id contenant les champs nécessaires et valides:
  response "http://localhost:8090/mbds/api/message/{id}" (status 201)
  Requête POST avec ou sans id ne contenant pas les champs nécessaire et valides:
  response "Failed to add message - not valid champs" (status 400)
  Requête DELETE sans id : renvoyé "for message delete/put use /api/message/{Your message ID}" (status 400)
  Requête DELETE avec id valide/existant: réponse vide - le message est supprimé (status 200)
  Requête DELETE avec id invalide/inexistant: réponse "message not found" (status 404)
  Requête PUT sans id : renvoyé "for message delete/put use /api/message/{Your message ID}" (status 400)
  Requête PUT avec id valide/existant : réponse http://localhost:8090/mbds/api/deadmatch/{id} -
  le message est mis à jour, les champs invalides seront ignorées (status 200)
  Requête PUT avec id invalide/inexistant: réponse "message not found" (status 404)

Quand tous les requêtes sont envoyées :

- Envoyer une requête GET a http://localhost:8090/mbds/api/logout
  avec dans le header X-Auth-Token: {votre token}
  le token sera alors plus utilisable

Implementation

Dans notre projet nous avons fait le choix de concentrer notre api autour d'un seul contrôleur - ApiController avec 3 méthodes user, deadmatch et message. Ces 3 méthodes partagent 2 autres méthodes génériques comme leur comportement est assez similaire.
Notre api est capable de repondre a 18 requetes 2 GET 1 POST 1 DELETE et 1 PUT sur les 3 entités.

Procédure de Test

Nous avons créé une collection Postman regroupant nos requêtes (nous avons commencé avec Curl mais à cause des problèmes d'inter compatibilité Linux / Windows nous sommes passés à Postman). La collection Postman et les requêtes curl se situent dans le dossier projetGrails/grails-app/utils.

Sécurité

Nous avons choisi d'utiliser l'extension Spring-security-rest pour la gestion de la sécurité (des token) de notre api et spring-security-rest-gorm pour le stockage des tokens sur notre base mysql. Pour executer des requetes avec succès il faut mettre dans le header des requêtes un token valide (X-Auth-Token) qui est récupérable en envoyant une requête POST avec nom et mot de passe à l'adresse localhost:8090/mbds/api/login. Par la suite le token a un temps d'expiration de 20 min.