Maxime Gagnon-Legault 111 090 082 magal106
François-Joseph Lacroix 111 130 199 fjlac1
Juliette Pelletier

GLO-2005: Rapport technique

Énonciation du problème et des exigences

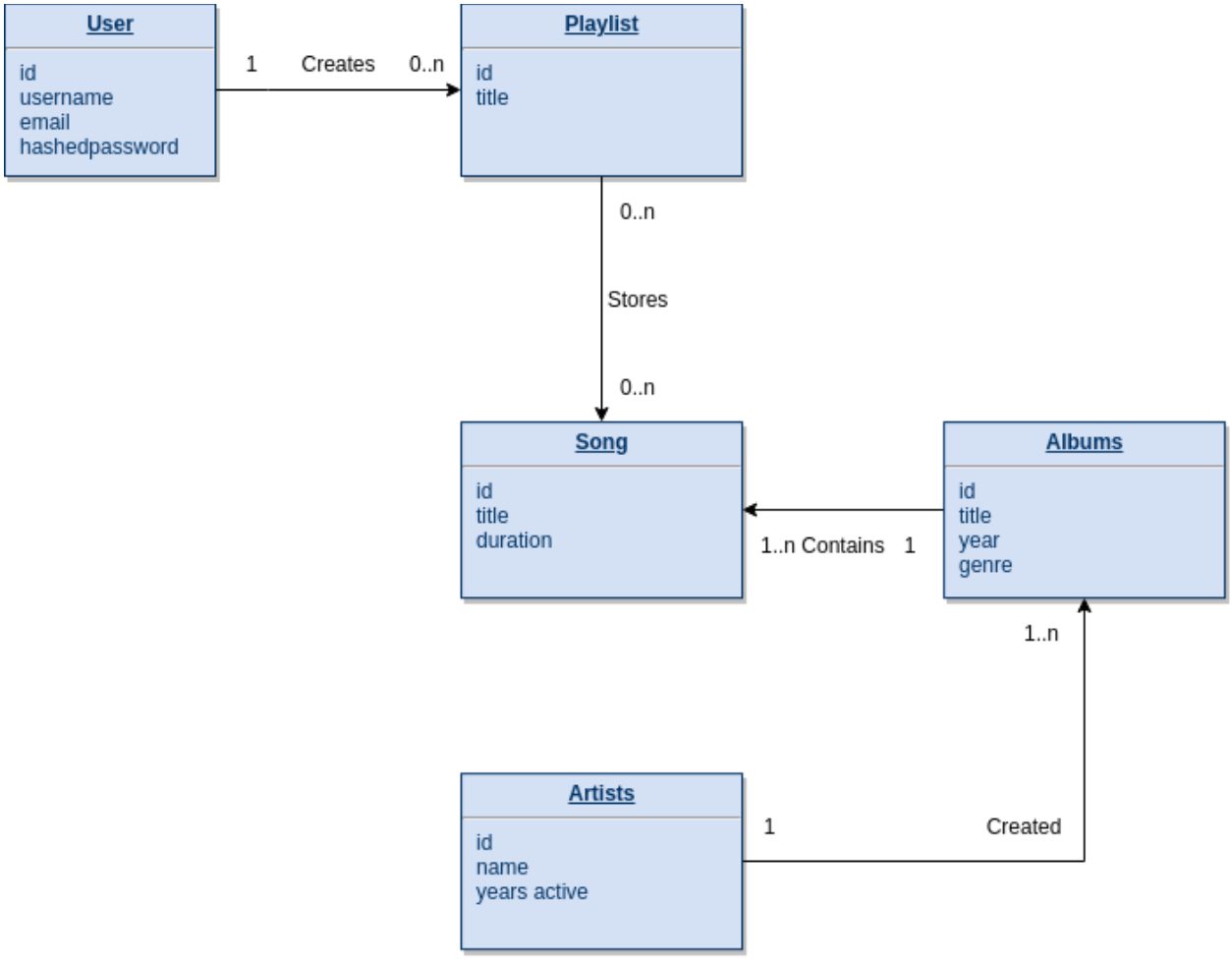
Nous avons décidé d'implémenter une application web pour l'écoute de chansons en ligne dans le style de Spotify ou Google Play Music. On se doit d'offrir de voir et rechercher des artistes, des albums et des chansons. De plus, nous voulons que certaines fonctionnalités soient bloquées derrière le besoin de se faire un compte utilisateur. Ces fonctionnalités sont de pouvoir se créer des listes de chansons personnalisées, aimer des chansons, et pouvoir écouter les chansons.

Spécifications du système et des responsabilités des trois niveaux

Les fonctionnalités de notre application:

- créer un compte client
- pouvoir se connecter avec son compte client
- pouvoir visionner une page pour un artiste particulier avec l'affichage de tous ses albums
- pouvoir visionner une page pour un album particulier avec l'affichage de toutes ses chansons
- pouvoir faire une recherche de chansons, albums ou artistes
- lorsque connecté, pouvoir voir ses playlists
- lorsque connecté, pouvoir modifier ses playlists
- lorsque connecté, pouvoir aimer des chansons

Modèle entités-relations



Modèle relationnel

User	(id, username, email, hashedPassword)
Genre	(genre_id, genre_name)
Artists	(artist_id, artist_name, years_active)
Albums	(album_id, title, year, artist_id, genre_id)
Songs	(song_id, album_id, artist_id, title, duration)
Playlists	(playlist_id, title)
Users_Playlists	(user_id, playlist_id)
Playlist_Songs	(playlist_id, song_id)
Artist_Release	(album_id, artist_id)
Album_Songs	(song_id, album_id)

Le système peut être vu comme étant divisé en trois couches:

Couche serveur de base de données

La couche serveur base de données est au niveau de l'infrastructure et a comme responsabilité d'emmagasiner les données du projet pour une lecture et écriture future. Dans notre cas, la base de données est une base de données Mysql version 8.0. Celle-ci a été configurée avec un seul utilisateur, l'administrateur par défaut root. Ce choix se veut à faciliter le développement correspondant au type de projet prototypage que nous avons. Sa configuration a été laissée au défaut du serveur. Pour faciliter l'assemblage avec les autres partie du projet et comme demandé pour le projet, le serveur Mysql est lancé dans un conteneur Docker disponible seulement sur le réseau de l'ordinateur (le « localhost ») au port 3306, son port par défaut pour prototypage rapide et facile. Ensuite, une Database nommé glo2005 est créée automatiquement au démarrage, puis on lui passe une série de fichier Sql pour la peupler. Les fichiers Sql sont des valeurs bidons générées par un script Python. Cela nous donne une base de données configurée correctement avec les bonnes tables déjà faites et remplies de tuples. Le passage des fichier Sql se fait au moment de l'initialisation du Docker. Cela nous semblait mieux et plus facile que le faire plus tard ou qu'avec un script style python.

La base de données sert principalement à faire des écritures et lectures standard. Notre utilisation en vient à ne faire que des SELECT ou des INSERT d'un élément ou groupe d'éléments. Aucune optimisation n'a été faite au niveau des requêtes à l'intérieur de la base de données. Il existe des index pour certaines valeurs de certains tuples, mais il n'y a pas eu beaucoup de réflexion sur ceux-ci. Cela revient encore au style de projet que nous avons où le prototypage se doit d'être rapide pour le développement ainsi qu'avec le faible volume de données que nous générons et utilisons, l'optimisation se veut prématuré. Dans la même lancé, nous n'avons pas implémenté de routine.

L'entreposage dans le Docker de la base de données s'est fait sans pont avec la machine hôte pour garder les données.

Couche logique d’affaire

La couche de la logique d’affaire se trouve à être un serveur REST API basé sur le framework Flask en Python. Cette couche a comme responsabilité de répondre aux demandes de changement et aux requêtes d’informations de la couche utilisateur.

D’une part, elle se met à la disposition de l’interface utilisateur en offrant des url HTTP qui répondent à des besoins précis de celle-ci. Elle attend les connections des utilisateurs par l’interface sous des appels HTTP de type GET pour la demande d’information et POST pour la création de données. Par exemple, appeler la route «/albums» va renvoyer toutes les albums de la base de données alors que préciser avec «/albums/10000000» renvoit seulement l’album avec l’identifiant 10000000. Pour un post sur /signup avec dans le corps de la requête un nom d’usager, une adresse courriel et un mot de passe va créer un nouvel utilisateur et l’enregistrer dans la base de données.

Certaines routes ne peuvent se faire que si l’utilisateur est inscrit, comme lors de la création de liste de lecture. Dans ces cas, l’interface se doit d’envoyer avec la requête un token de style JWT reçu après une création ou identification réussite. Les token JWT sont produit par la librairie PyJwt. Pour les raisons d'authentification, la sauvegarde des mots de passe se font seulement avec un mot de passe hashé par le plugin Flask-Bcrypt.

Le reste de l’application ressemble à ce qui est déjà discuté plus haut. Le projet est dans le style «CRUD», il n’y a pas beaucoup de logique d’affaire autre que la transformation et déplacement d’information. De l’autre côté de l’application, la gestion des données du serveur de base de données se fait à l’aide du connecteur Mysql pour Python. La plupart des interactions avec cela se veulent des requêtes de données ou l'envoi de données et un peu de réhydratation simple.

Couche interface utilisateur

La couche d’interface utilisateur ou frontend ou site web est la couche que les utilisateurs vont voir et avec laquelle ils pourront interagir. Elle s’occupe d’afficher les données du projet dans le browser du client sous la forme d’une application web. Sa responsabilité est d’afficher le contenu et d’aller demander le contenu voulu par l’utilisateur à la couche API.

Son implémentation a été faite principalement avec le framework javascript Vuejs. Il s'ajoute à cela le gestionnaire de paquets NPM, et d'autre petit librairie comme Axios pour faciliter les appels AJAX et JavaScript Cookie pour la gestion des cookies. Cette application est montée pour rouler dans un conteneur Docker sur le réseau Localhost au port 8080, s'il est disponible, de l'ordinateur le roulant.

Comme mentionné plus haut la fonctionnalité de l'interface utilisateur est d'afficher le contenu du projet à la demande du client. Pour cela, elle possède quelques composantes qui se traduisent en un multitude de pages web différentes en fonction de paramètres qui lui sont passés. On a ainsi pu réutiliser les composantes à plusieurs endroits, ce qui nous a aidé à faire le développement. Quelque exemple de composantes que nous avons sont: «Playlist» qui permet d'afficher des chansons sous différentes pages avec réutilisation, «Albums» et «Artists» qui affichent une page d'album et d'artiste respectivement. Certaines de ces composantes ont la responsabilité de faire des appels AJAX à notre API pour permettre d'accéder à leurs informations. Pour ce faire, Axios est utilisé et nous est retourné un réponse en format JSON correspondant à ce qu'on souhaite afficher.

Sécurité du système

La sécurité du système se joue sur plusieurs points soient: la gestion des mots de passe, l'authentification, la gestion des erreurs du serveur Flask et la protection contre les injections de SQL.

La gestion du mots de passe se fait lorsqu'un utilisateur vient à s'inscrire ou se connecter avec notre application. Dans un premier temps, celui ci va nous envoyer un mot de passe pour garder son compte. Dans notre application, le mot de passe est transféré en simple text dans le corps d'une requête HTTP jusqu'au serveur. Cela représente un grave bris de sécurité en ce moment, car nous n'utilisons pas HTTPS. Normalement, nous aurions dû configurer notre serveur pour qu'il utilise exclusivement le protocole HTTPS avec un certificat confidentiel. À cause de la simplicité du projet et du fait que ce soit un simple projet étudiant nous avons décidé de ne pas faire cela, mais nous sommes conscients que HTTPS est une mesure essentielle de protection. Dans un deuxième temps, nous nous devons de conserver les noms d'utilisateur et mots des usagers pour qu'il puissent se reconnecter. Nous avons choisi de ne pas sauvegarder les mots de passes brut dans notre base de données. Pour ce faire, nous ne gardons que des mots de passe qui ont été hashé et salé par la méthod Bcrypt.

Ainsi, même si la base de données ou les mots de passe qu'on sauvegarde sont compromis, un attaquant n'a pas accès directement aux mots de passe des clients. Ce fut relativement facile à faire dans notre projet avec l'utilisation du plugin Flask-Brcypt pour Flask.

L'authentification après la connexion se fait sans le problème de devoir se reconnecter souvent. En effet, nous utilison des tokens d'accès de type Json Web Token (JWT) pour garder ouvert la session des utilisateurs. Ces tokens gardent avec eux le nom de l'utilisateur et son identifiant unique. Lorsque l'interface se veut d'envoyer une requête à notre API REST sur une route qui demande l'authentification de l'usager, celle-ci se doit d'envoyer aussi son token dans le corps JSON de la requête. La beauté des JWT est qu'il est impossible de recréer un token valide sans avoir le secret de l'application avec lequel il est a été créer. Dans notre cas le secret des tokens n'est pas bien fait. Il se trouve directement écrit dans notre code ce qui est un code smell, mais pour le projet est passable. De plus, si le projet venait à sortir du stade de prototypage, il faudrait s'assurer de mettre un temps d'expiration aux tokens et de faire d'autres actions pour renforcer la sécurité.

La gestion des erreurs dans les applications est normalement très importante. Dans le cours normal de notre projet, il ne devrait pas y avoir d'erreur catastrophique qui brise le fonctionnement des usages courants. Cependant, nous avons laissé des erreurs qui brisent certaines requêtes. Notre but n'était pas d'en laisser, mais nous avons été obligé de laissé des erreurs se propager, surtout dans le serveur Flask et l'application Vuejs, vue de notre manque de temps sur le projet. Cela ne nuit pas trop à l'utilisateur et nous sommes quand même confiant qu'un utilisateur ou un usage normal du projet ne va pas mettre en danger les données et les applications.

L'injection de SQL dans notre application ne semble pas être un problème avec notre implémentation. L'échappement des caractères malicieux du SQL se fait déjà en grande partie par le l'implémentation du connecteur Mysql pour Python. Cependant la création de la base de données à été faite avec un mot de passe simple et l'utilisateur par défaut root. De plus, nous gardons à même dans le code le nom d'utilisateur, le mot de passe et l'adresse de la base de données, ce qui est une très mauvaise pratiques.

L'organisation et la gestion de l'équipe, et division des tâches

Comme vous avez dû le constater, nous avons eu plusieurs problèmes avec l'organisation du projet. Pour commencer, nous nous y sommes pris un peu en retard

pour commencer le projet. Notre première rencontre d'équipe a été faite plusieurs semaines après la création des équipes et quelques semaines après la remise de l'énoncé de projet. Nous avons faite qu'une seul rencontre en équipe pour planifier le travail et celle-ci n'était pas en personne. La communication était très mauvaise, prenant à des endroits plus d'une semaine pour répondre dans notre Slack. La division des tâches était très mal faites. Nous avons mis quelques user stories à faire chacun de notre bord et là aussi il y a eu très peu de discussions entre les membres de l'équipe.

Le code du projet est disponible publiquement à partir du 17 avril 2019 au lien https://github.com/MaximeGLegault/glo2005