Rapport projet Scala - BeerPass:

Membres du groupe: Antoine Rochat et Benoît Schopfer

URL du repo Github: https://github.com/Nooka10/ProjetScala

URL de l'API Rest: https://beerpass-scala.herokuapp.com

URL du frontend web: https://beerpass-scala-front.herokuapp.com

Description du projet:

Le Beer Pass vous permet d'obtenir une bière gratuite dans tous les établissements partenaires (+100 établissements) sur une année civile. Le prix du pass serait de 49 CHF par année.

Une fois le Beer Pass commandé, les utilisateurs pourront consulter les bons disponibles sur l'application smartphone dédiée. Ils pourront faire valoir leurs offres dans les différents établissements partenaires grâce à un QR Code généré automatiquement par l'application.

Ce projet permettrait aux utilisateurs de découvrir de nouveaux établissements (bars ou brasseries) dans leur région et ceci à un prix attractif.

L'attrait pour les établissements est double. Une telle application leur permettrait de se faire connaître par de nouveaux clients mais aurait également un intérêt financier direct. En effet, il a été prouvé que les clients consommaient davantage après avoir utilisé leur bon (dans les bars) ou passaient commande (dans les brasseries) dans les différents établissements où ils ont fait valoir un bon.

Technologies utilisées:

Pour ce projet, nous avons utilisé:

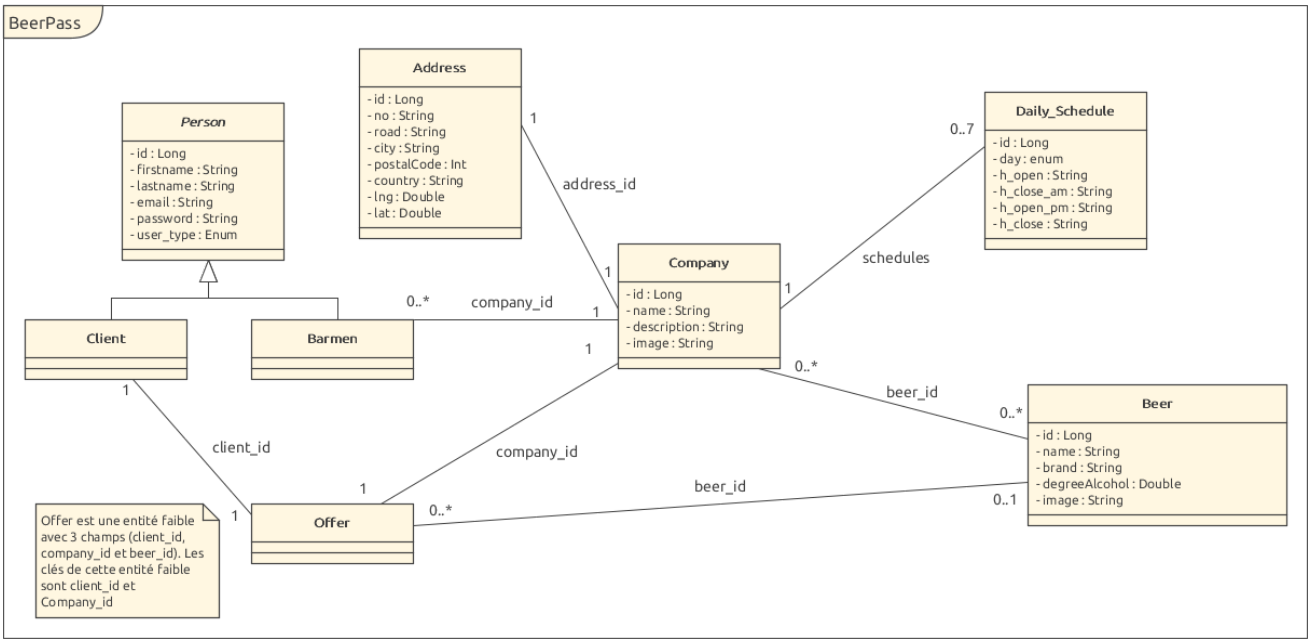
- Scala pour écrire le Backend (une API REST).
- Slick pour faire le lien entre la base de données et le backend.
- React Native et Expo pour créer l'application smartphone, qui permet:
    - Pour les utilisateurs :
        - De consulter les bons disponibles
        - De consulter les bons utilisés

        - De faire valoir un bon en générant un QR Code qui sera scanné par l'établissement partenaire.
    - Pour les établissements :
        - De scanner les QR Code des utilisateurs afin de valider leurs commandes.
- React pour le Frontend Web, qui permet :
    - Consulter les informations sur le Beer Pass
    - Consulter des statistiques (bière la plus vendue et établissement le plus fréquenté)
    - Consulter tous les établissements partenaires sur une carte
    - Consulter les détails de tous les établissements partenaires (Horaires, adresse, ...)
    - Se procurer un Beer Pass (pour l'instant, les Beer Pass sont "bientôt disponibles" donc pas encore commandables)

Description de l'implémentation:

Base de données:

Le schéma UML de notre base de données est le suivant:



Un script SQL est disponible dans le dossier conf/evolution.default/beerPass.sql afin de créer les tables et populer notre base de données avec quelques données de tests.

L'url de connexion (*DB_URL*), l'utilisateur (*DB_USER*) et le mot de passe (*DB_PWD*) sont à définir en temps que variable d'environnement ou en utilisant les mêmes données que ci-dessous:

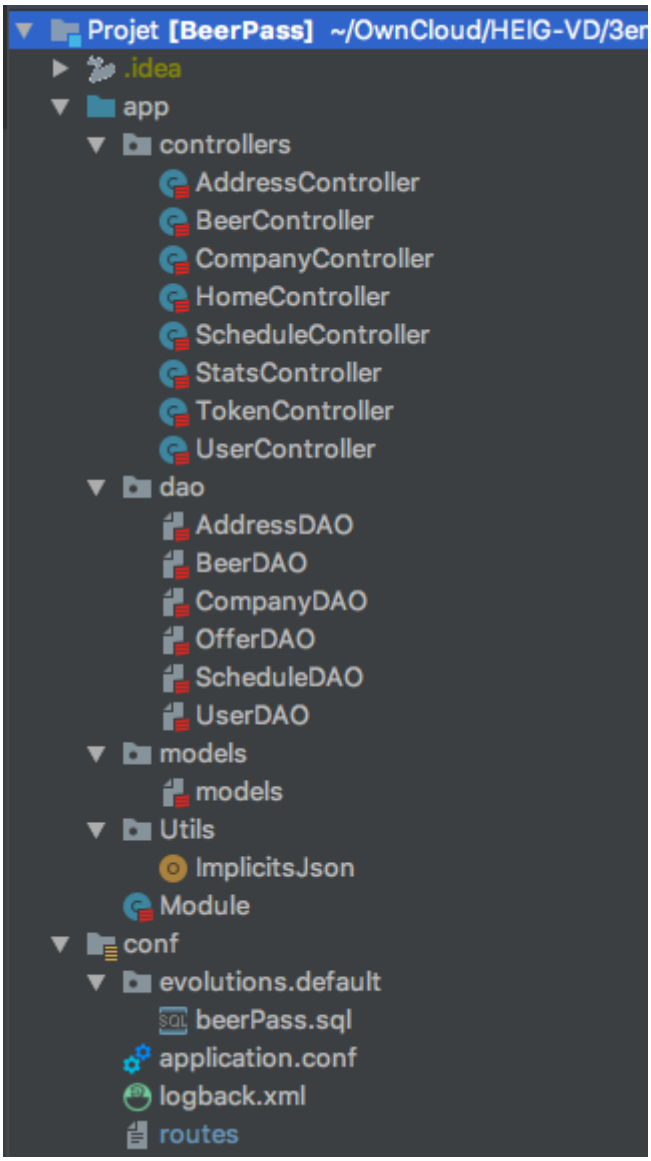
```
slick.dbs.default.profile = "slick.jdbc.MySQLProfile$"
slick.dbs.default.db.driver = "com.mysql.cj.jdbc.Driver"
slick.dbs.default.db.url = "jdbc:mysql://localhost:3306/beerPass"
slick.dbs.default.db.user = "beerPass"
slick.dbs.default.db.password = "beerPass"

// Env variables
slick.dbs.default.db.url = ${?DB_URL}
slick.dbs.default.db.user = ${?DB_USER}
slick.dbs.default.db.password = ${?DB_PWD}
```

Backend:

Architecture générale:

L'architecture de notre backend est très simple et suit un modèle MVC. Cependant, notre backend n'offre aucune vue. En effet, il ne fait que retourner du JSON mais n'affiche jamais de pages HTML ou autre.



Models:

Comme vous pouvez le voir dans la capture ci-dessus, nos modèles se trouvent dans le fichier models du dossier du même nom. Ce fichier contient toutes les case class représentant les objets d'une entrée d'une table de la base de données. Cependant, nous avons plus de modèles dans ce fichier que de tables dans notre base de données car nous utilisons certains de ces modèles pour représenter des objets que nous manipulons dans notre code scala. Par exemple, la case class *Offer* représente la table OFFER de notre base de données, alors que la case class *OfferWithObjects* n'en représente aucune. *OfferWithObjects* est en fait une représentation contenant tout les objets attachés (via une foreignKey) à une *Offer* (qui ne contient que leurs ids). Il en va de même pour la case class *CompanyWithObjects* représentant une *Company* avec l'ensemble de ses objets attachés.

Controllers:

Les contrôleurs gèrent chacune une ou plusieurs des routes proposées par notre API. Dans notre backend, nous avons regroupé dans un même contrôleur toutes les routes agissant avec la même table de notre base de donnée.

Par exemple, toute les routes permettant d'interagir avec une company sont gérées dans le contrôleur *CompanyController*. Ce contrôleur fait lui-même appel à d'autres contrôleurs tels que *AddressController* ou *ScheduleController* pour gérer les horaires et l'adresse d'une company.

Les contrôleurs ne font pas directement les appels à la base de données. En effet, ceux-ci sont fait dans les DAOs que nous décrivons par la suite.

Une requête reçue sur notre API va donc être redirigées vers la méthode du contrôleur adéquat (selon les règles de routage décrites dans le fichier *route*). Cette méthode va effectuer les actions demandées en faisant appel aux DAOs nécessaires, puis déterminera la réponse à retourner en fonction du déroulement des opérations.

Dans le cas de notre API, les retours des contrôleurs sont toujours des objets JSON encapsulés dans des réponses HTTP (Ok, BadRequest, NotFound, ...)

DAOs:

Les DAOs gèrent les appels à la base de données. Dans notre API, nous avons utilisé Slick afin de pouvoir écrire nos requêtes SQL directement en Scala.

Nos DAOs sont découpées en 2 parties. Un trait définissant un component (ex: *CompanyComponent*), ainsi que la DAO héritant de ce trait.

Dans le Component, nous déclarons la table (ex: *CompanyTable*) permettant de convertir un objet Scala en une ligne de la table *COMPANY* de notre base de données et inversement. Nous déclarons également un TableQuery de cette table que nous utiliserons par la suite dans nos DAOs pour interagir avec la table de la base de données.

Dans la DAO (ex: *CompanyDAO*), nous définissons une méthode pour chaque appel à la base de données que nous souhaitons effectuer. Par exemple, la méthode *findById(id: Long)* de notre *CompanyDAO* permet de récupérer dans la base de données la company correspondant à l'id reçu en paramètre, *insert(company: Company)* permet d'ajouter une nouvelle company dans la base de données etc.

Dans ces méthodes, les appels à la base de données sont écrit en Scala et non pas en SQL. Nous avons pu faire cela grâce à l'utilisation de Slick. C'est donc grâce à Slick que nous pouvons écrire la requête suivante, permettant de récupérer la company correspondant à l'id reçu en paramètre :

```
def findById(id: Long): Future[Option[CompanyWithObjects]] = {
    val query = for {
      company <- companies if company.id === id
      address <- company.address
    } yield (company, address)

    db.run(query.result.headOption)
      .map(option => option
        .map(tuple => CompanyWithObjects
          .fromCompany(tuple._1, tuple._2,
Some(Await.result(scheduleDAO.findAllDailySchedulesFromCompanyId(tuple._1.id.get),
Duration.Inf)))))
}
```

En général, comme pour les contrôleurs, une DAO regroupe l'ensemble des requêtes concernant une table de notre base de données. Cependant, certaines de nos DAO utilisent plusieurs tables, tel que *ScheduleDAO.*, qui utilise la table *DAILY_SCHEDULE* et la table *LINK_DAILY_SCHEDULE_COMPANY*.

Nous avons fait cela car ces deux tables sont intimement liées puisque la table *LINK_DAILY_SCHEDULE_COMPANY* est une table de liaison nécessaire pour enregistrer une relation 1-N en MySQL. Ainsi, demander à récupérer tous les horaires d'une company se fait très simplement en appelant la méthode *findAllDailySchedulesFromCompanyId(companyId: Long)*.

Routes:

Notre backend propose une API REST avec les routes suivantes:

```
# REST API endpoints
# ------------------------------------------------------------ USER ENDPOINTS ------------------------------------------------------------
POST        /register                               controllers.UserController.createUser
POST        /login                                  controllers.UserController.login
GET         /users/:id                              controllers.UserController.getUser(id: Long)
PUT         /users                                  controllers.UserController.updateUser
DELETE      /users/:id                              controllers.UserController.deleteUser(id: Long)

# ------------------------------------------------------------ USER'S OFFERS ENDPOINTS ------------------------------------------------------------
GET         /users/:id/offers                       controllers.UserController.getAllOffersOfUser(id: Long)
GET         /users/:id/offers/used                  controllers.UserController.getAllConsumedOffersOfUser(id: Long)
GET         /users/:id/offers/unused                controllers.UserController.getAllUnusedOffersOfUser(id: Long)
POST        /useOffer                               controllers.UserController.useOffer

# ------------------------------------------------------------ COMPANY ENDPOINTS ------------------------------------------------------------
POST        /companies                              controllers.CompanyController.createCompany
GET         /companies                              controllers.CompanyController.getCompanies
GET         /companies/:id                          controllers.CompanyController.getCompany(id: Long)
GET         /companies/:id/employees                controllers.CompanyController.getEmployees(id: Long)
PUT         /companies                              controllers.CompanyController.updateCompany
DELETE      /companies/:id                          controllers.CompanyController.deleteCompany(id: Long)

# ------------------------------------------------------------ COMPANY'S BEERS ENDPOINTS ------------------------------------------------------------
GET         /companies/:companyId/beers             controllers.BeerController.getAllBeersOfCompany(companyId: Long)
GET         /companies/:companyId/beers/:beerId     controllers.BeerController.getOneBeersOfCompany(companyId: Long, beerId: Long)
POST        /companies/:companyId/beers/:beerId     controllers.BeerController.addBeerToDrinkListOfCompany(companyId: Long, beerId: Long)
DELETE      /companies/:companyId/beers/:beerId     controllers.BeerController.removeBeerFromDrinkListOfCompany(companyId: Long, beerId: Long)

# ------------------------------------------------------------ BEER ENDPOINTS ------------------------------------------------------------
GET         /beers                                  controllers.BeerController.getBeers
GET         /beers/:id                              controllers.BeerController.getBeer(id: Long)
POST        /beers                                  controllers.BeerController.createBeer
PUT         /beers                                  controllers.BeerController.updateBeer
DELETE      /beers/:id                              controllers.BeerController.deleteBeer(id: Long)

# ------------------------------------------------------------ STATS ENDPOINTS ------------------------------------------------------------
GET         /stats/mostPopularCompany               controllers.StatsController.getMostPopularCompany
GET         /stats/getMostFamousBeer                controllers.StatsController.getMostFamousBeer
GET         /stats/getMostFamousBeer/:companyId     controllers.StatsController.getMostFamousBeerForCompany(companyId: Long)
```

Nous pouvons effectuer un CRUD sur les company, les users et les bières. Nous pouvons également récupérer et utiliser les offres d'un client ou encore lire quelques statistiques.

Application mobile

L'application que nous avons réalisé n'est pas complète. Il s'agit d'un *Proof Of Concept* et certaines fonctionnalités manquent.

Notre application permet à un utilisateur (client) de consulter ses bons valables et ceux déjà utilisés. S'il décide d'utiliser un de ses bons, il lui suffit de cliquer dessus et de montrer à l'établissement le QR Code généré par l'application. Un employé de l'établissement (un utilisateur connecté en temps qu'employé), doit alors scanner le QR Code. Si le QR code est bien valide, la commande est effectuée, le bon de l'utilisateur est utilisé et il reçoit sa bière gratuitement.

Pour cette application mobile, nous avons utilisé React Native ainsi que Expo afin d'avoir une application cross-plateforme.

Expo permet de simplifier la création d'applications mobiles en permettant notamment de construire des applications Android et iOS sans avoir besoin d'utiliser Android Studio ou XCode. De plus, Expo fournit une série d'outils cross-plateformes bien utiles pour notre application (accès à la caméra, lecture de QR Code, affichage d'une carte, etc).

Écrans

L'écran de login permet à un utilisateur de se connecter à l'application. Il doit pour cela entrer son email ainsi que son mot de passe. En fonction de son rôle (client ou employé d'un établissement partenaire), les écrans disponibles à l'utilisateur ne sont pas les mêmes.

- Pour les clients:
    - Écran avec tous les bons valables et dont il peut profiter. En cliquant sur un bon, il peut accéder aux informations de l'établissement (tels que la description, les horaires, les bières qu'il propose) correspondant et générer le QR Code nécessaire à la validation du bon. Il peut également zoomer sur le QR Code en cliquant dessus.
    - Écran avec les bons déjà utilisés. Sensiblement la même chose que pour les bons utilisés. Utile si l'utilisateur veut retrouver les informations sur un établissement qu'il a visité.
- Pour les employés des établissements:
    - Un écran permettant simplement de scanner un QR Code afin de valider une commande faite par un client

Pour les 2 types d'utilisateurs, un écran *Réglage* leur permettent de se déconnecter de leur compte.

Librairies utilisées

Outre les outils fournit par Expo, nous avons utilisé les librairies React Native suivantes :

- *react-native-snap-carousel* : permet à l'utilisateur de faire défiler les bons valables ou utilisés
- *react-native-qrcode-svg* : permet de générer un QR Code facilement à partir d'une chaîne de caractère fournie
- *react-navigation* : permet de naviguer entre les différentes fenêtres de l'application
- *eslint* : un linter utile en développement afin de renforcer l'uniformité de notre code et le style d'écriture grâce à des règles de codage qui sont vérifiées à chaque compilation.

Frontend Web

En plus de l'application mobile, nous avons implémenté un front end Web en utilisant React afin d'afficher les informations pour un utilisateur qui veut en savoir plus sur le Beer Pass, les établissements partenaires, ou encore se procurer un Beer Pass.

Pages

La page d'accueil présente le concept du Beer Pass et affiche le nombre d'établissements partenaires ainsi que le nombre de bières différentes que l'utilisateur pourrait tester en souscrivant un Beer Pass. Cette page d'accueil contient aussi quelques statistiques permettant de connaître l'établissement qui a le plus de succès ainsi que la bière qui a été choisie par le plus d'utilisateurs.

La page des établissements offre une carte affichant tous les emplacements des établissements partenaires. En cliquant sur un marqueur, cela affiche une pop-up contenant le nom de l'établissement. L'utilisateur peut alors cliquer sur ce nom afin d'afficher les détails de l'établissement sélectionné.

La page de détails des établissements permet de consulter toutes les informations relatives à l'établissement choisi, tels que sa description, son adresse, les bières qu'il propose, ses horaires ainsi que sa position sur une carte.

La page de shopping permettra par la suite de se procurer un Beer Pass. Malheureusement, à l'heure actuelle cette fonctionnalité n'est pas encore disponible mais cela ne devrait plus tarder, restez informé! Lorsque ça sera le cas, le bouton d'ajout au panier sera réactivé...

Librairies utilisées

material-ui : une librairie de composants facilitant et améliorant le style de l'application. De nombreux composants ont été utilisés tels que l'AppBar, des Grid, les Card.

react-leaflet : adaptation de Leaflet avec React permettant d'afficher une carte avec des marqueurs dessus. Service gratuit et open-source, contrairement à Google Maps.

react-router : permet de définir la navigation entre les différents endpoints de l'application

eslint : un linter utile en développement afin de renforcer l'uniformité de notre code et le style d'écriture grâce à des règles de codage qui sont vérifiées à chaque compilation.

Problèmes rencontrés:

Les principaux problèmes rencontrés ont été causés par Slick!

Tout d'abord, il a été passablement ardu de trouver des exemples de codes quelque peu développés et fonctionnels pour Slick!

En effet, l'exemple qui nous avait été fourni ne montrait pas l'utilisation des foreignKeys et les tables étaient très basiques (pas de relations n-n ou 1-n etc)! Malheureusement, la très grande majorité des exemples Slick disponibles sur internet sont tout aussi basiques voir plus simples encore!

Nous avons donc rencontré des difficultés pour réussir à faire fonctionner les foreignKeys. Difficulté que nous avons finalement pu surmonter avec l'aide de la prof et des assistants qui ont complexifié l'exemple pour nous montrer comment faire fonctionner les foreignKeys.

Par la suite, nous avons rencontré des difficultés à faire fonctionner les Enumérations. Plus précisément, il a été compliqué de faire en sorte que Slick également utilise une Enum (comme disponible en MySQL) et non un champ texte.

Finalement, et c'est sans doute ce qui a terminé de nous dégouter de Slick, il s'est avéré impossible d'ajouter des données dans une table ne possédant pas de champ en AutoInc! Or, selon notre modélisation, la table *LINK_DAILY_SCHEDULE_COMPANY* devrait posséder deux clés primaires (CompanyId et dailyScheduleId) qui sont également deux foreignKeys. Elle ne devrait pas posséder de champ en AutoInc! Mais comme nous n'avons pas pu trouver d'exemples fonctionnels sur internet et ne sommes pas parvenu à régler ce problème, nous avons été obligés d'ajouter une clé primaire *id* en autoInc afin que Slick soit heureux. Ce champ est donc enregistré dans notre base de données alors qu'il nous sert en réalité à rien du tout!

Au niveau de l'application mobile et du front end Web, aucun souci particulier n'est à signaler. Le seul problème rencontré a été le manque de temps dû aux autres cours et au travail de Bachelor... C'est pour cela que les deux applications que nous avons développé sont très simplistes et n'implémentent pas toutes les possibilités offertes par le backend (voir *améliorations possibles*).

Améliorations possibles:

De nombreuses améliorations pourraient encore être apportées à notre projet!

Au niveau du Backend, la principale amélioration qu'il faudrait apporter avant une mise en production serait la sécurité! En effet, nous n'avons réalisé qu'un Proof Of Concept. Nous avons mis en place un système de login, mais aucun de nos endpoints ne sont protégés! Autrement dit, n'importe qui peut faire n'importe quoi, rien n'est contrôlé!

D'autres améliorations sont possible, tel que trouver comment faire marcher Slick pour supprimer l'id inutile dans la table *LINK_DAILY_SCHEDULE_COMPANY*, comme expliqué précédemment. Une autre solution possible à se problème serait de recoller le backend pour en faire une application NodeJs codée en TypeScript. Cela permettrait également d'utiliser un vrai ORM, comme, par exemple TypeORM... ;P

On pourrait également ajouter de nouvelles statistiques un peu plus poussées.

Mais le gros des améliorations seraient à apporter à l'application smartphone et au site internet. En effet, les 2 frontends que nous avons codés sont plus des *proof of concept* qu'autre chose... Ils ne proposent que quelques fonctionnalités très simples. Le Backend permet de faire bien plus. Par exemple, l'application smartphone pourrait proposer au client de modifier ses informations ou de supprimer son compte. On pourrait aussi proposer aux employés d'ajouter / enlever / modifier la liste des boissons proposées par l'établissement directement depuis l'application.

Il n'y a pas non plus de possibilité de se créer un compte depuis l'application car nous ne nous sommes pas encore mis d'accord sur la méthode d'inscription d'un utilisateur. Celui-ci devrait d'abord acheter un Beer Pass, ce que fournirait soit un code d'activation à faire valoir avec un code existant, soit directement des identifiants pour se connecter.

Toutes ces fonctionnalités sont disponibles au niveau du Backend, mais nous n'avons pas eu le temps de les implémenter au niveau de l'application smartphone.