Objection.js

`Objection.js` est un ORM (objection relation mapping) léger pour NodeJs. Nous allons l'utiliser avec `schwifty`

`Schwifty` étend les fonctionalités d'`objection` pour permettre l'utilisation du validation `Joi` et `objection` se base lui meme sur `knex` (un constructeur de requetes nodejs).

On pourra le schematisé comme suit:

```
Schwifty -> Objection.js -> Knex
```

Pour intégrer convenablement `Schwifty` et `Objection` au sein de notre projet, nous allons récupérer le flavor `objection` du boilerplate, comme nous avons fait pour `swagger`.

```
git cherry-pick objection
```

Vous aurez dans `server/manifest.js` un nouveau plugin d'ajouter comme ceci:

```
{
                plugin: 'schwifty',
                options: {
                    $filter: 'NODE_ENV',
                    $default: {},
                    $base: {
                        migrateOnStart: true,
                        knex: {
                            client: 'sqlite3',
                            useNullAsDefault: true,         // Suggested for sqlite
                            pool: {
                                idleTimeoutMillis: Infinity // Handles knex v0.12/0
                            },
                            connection: {
                                filename: ':memory:'
                            }
                        }
                    },
                    production: {
                        migrateOnStart: false
                    }
                }
}
```

Vous pouvez donc ici configurer votre base de données. Par default la base de donnée utilise sqlite3. Nous allons faire quelques modifications afin d'utiliser `PostgreSQL`.

Nous devons donc changer la configuration `knex` du plugin `schwifty` comme suit:

```
{
                plugin: 'schwifty',
                options: {
                    $filter: 'NODE_ENV',
```

```
                    $default: {},
                    $base: {
                        migrateOnStart: true,
                        knex: {
                            client: 'pg',
                            connection: {
                                host     : process.env.POSTGRES_HOST || 'localhost
                                port     : process.env.POSTGRES_PORT || 5432 ,
                                user     : process.env.POSTGRES_USER || 'hapi',
                                password : process.env.POSTGRES_PASSWORD || 'hapi'
                                database : process.env.POSTGRES_DATABASE || 'unili
                            }
                        }
                    },
                    production: {
                        migrateOnStart: false
                    }
                }
}
```

> Ce bloc de configuration permet de setter directement les informations de connexion depuis les variables d'environnement. Ce qui permet de lancer le projet sur différents postes ayant, par exemple, des infos de connexion différentes, sans aucune difficulté
> Vous pouvez consulter https://knexjs.org/#Installation-client pour plus d'informations sur la configuration du client.

Vous pouvez utiliser un `docker` pour lancer le server `PostgreSQL`. Vous devrez ensuite configurer vos variables d'environments pour que votre application se connect avec la base de données.

Pour chaque model, vous devrez vous baser sur le modèle suivant dans le répertoire `app/models/` :

```
'use strict';

const { Model } = require('objection');

module.exports = class User extends Model {

  static get tableName() {

    return 'user';
  }

  static get joiSchema() {

    return Joi.object();
  }
```

```
}
```

Migrations

Il se peut que, lors de l'avancée de votre projet, vous ayez des manipulations à faire dans la BDD associée (ajout de nouvelles entrées, modification des indexes, modification de certaines entrées, …).

Si vous êtes seul sur votre projet, et qu'il tourne à un seul endroit, cela peut être vite fait. Mais, vous allez très certainement être avec différents environnements et serveurs où vous devrez relancer manuellement vos scripts de migration. Et, qui dit manipulation humaine dit aussi erreurs humaines possibles.

Pour cela on utilise les migrations de knex

Pour créer un fichier de migration vous pouvez executer la commande suivante:

```
npx knex migrate:make nomDeLaMigration
```

Vous devrez alors voir un fichier qui a été créé dans le dossier `lib/migrations` commencant par un nombre qui correspond à la date.

Il y a deux fonctions dans ce fichier `up` et `down`. `up` est ou vont se retrouver les modification a apporter pour que notre base de données se retrouve dans l'état qu'on souhaite (ajout de

colonne, table…) et `down` est ou se trouve les manipulation pour annuler se qui est fait dans `up` (c'est notament utile si vous voulez revenir en arrière très rappidement lors d'un problème avec une version de votre migration).

La documentation de knex détail toute les possibilités des migrations:
https://knexjs.org/#Schema

Travail à faire

- Vous devrez donc créer le CRUD complet des utilisateurs au sein d'Hapi avec sauvegarde dans une base de donnée Postgres en vous basant sur la structure du TP dernier. Bien entendu, vous devrez respecter la norme REST (codes de réponse HTTP et méthodes d'entrées).
- Vous noterez toutefois que vous devrez gérer les cas des champs uniques (un seul email possible à la fois…).
- Via un script de migration, créer les tables nécessaires ainsi qu'un utilisateur pour que, dès le premier lancement du projet sur un autre poste, on puisse le manipuler via le CRUD.
- Créer une route `/users/generate` qui génère 100 utilisateur a la fois. Vous pouvez vous servir de `faker` ou `felicity` pour générer les utilisateurs si besoin.

Le mot de passe devra être encrypté en utilisant votre module créé en TP 2.

Norme REST

Les API REST utilisent les méthodes d'entrées et les codes HTTP pour donner le status des réponses ainsi que, la plupart du temps, le JSON pour l'envoi des données.

Par exemple, pour une API REST sur les utilisateurs vous aurez les requêtes entrantes suivantes :

- `GET /users` : Récupération de tous les utilisateurs ou en fonction de paramètres GET pour filtrer
- `GET /user/:id` : Récupération d'un utilisateur. Si non trouvé retourne une 404
- `POST /user` : Sauvegarde **un nouvel** utilisateur
- `PUT /user/:id` : Modifie **un utilisateur existant**. Si non trouvé retourne une 404
- `DELETE /user/:id` : Supprime **un utilisateur existant**. Si non trouvé retourne une 404

Pour les retours, aidez vous de Hapi-boom et de hapi-boom-decorator en cas d'erreur, il vous les formatera comme il faut.

En ce qui concerne les codes de retour HTTP, en cas de succès vous aurez les codes suivants :

- **200** : Requête effectuée avec succès. Utilisé que pour des requêtes de récupération d'informations.
- **201** : Contenu créé avec succès. Utilisé lors de la sauvegarde ou la modification d'une donnée sur le serveur.
- **204** : Aucun contenu de retourné. Ce statut est utilisé pour les requêtes `DELETE` si elles se sont passées avec succès.

Récapitulatif des avancées du projet

1. Intégrer Objection
2. Créer le CRUD des utilisateurs avec liaison BDD
3. Créer une route `/users/generate` permettant de générer 100 nouveaux utilisateurs avec la library Faker.

Tips

- Les fonctions de Handler entrantes (celles appelées par les routes) prennent toujours deux paramètres : `request` et `h`.
- Vous pouvez récupérer la variable `server` au travers de `request.server` pour, par exemple, accéder à un plugin spécifique.
- Pour accéder à un model mongoose, vous pouvez de la manière suivante : `const { User } = server.models()` ou `const { User } = request.models()`.