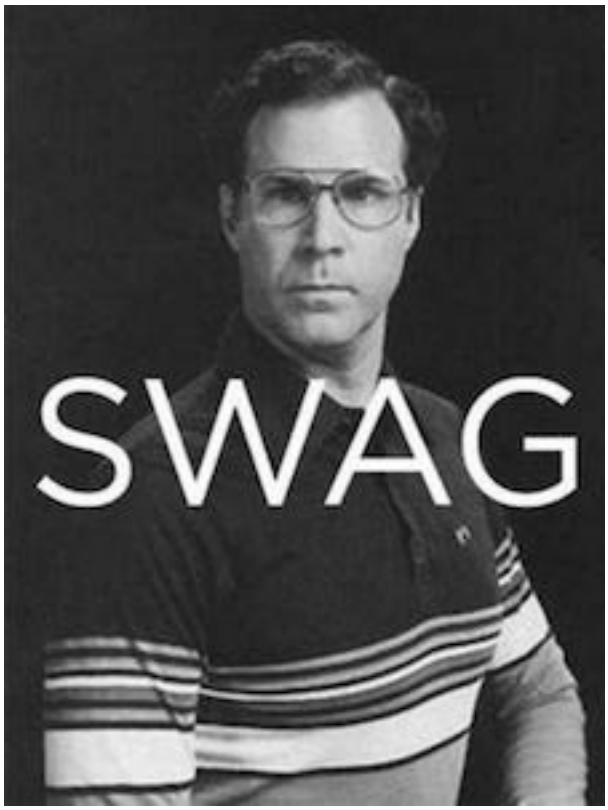
Hapi : Swagger et Joi



Projets de fin de modules notés

À partir de maintenant, tous les Tps que vous allez faire vont vous permettre de réaliser deux projets complets communiquant ensemble :

- **Service Users** : Service de gestion d'utilisateurs. Ce service sera lié à une base de donnée Mongo et utilisera les normes d'API REST :
  - Récupération de tous les utilisateurs
  - Récupération d'un utilisateur donné
  - Ajout d'un utilisateur
  - Modification d'un utilisateur
  - Suppression d'un utilisateur
  - Génération alléatoire de 100 utilisateurs
  - Authentification d'un utilisateur
  - Regénération d'un utilisateur
  - Utilisation du module créé au TP2 pour la génération du mot de passe

- **Service Mail** : Relié via socket.io au premier service, il va vous permettre d'envoyer un email aux utilisateurs lors de l'inscription et de l'oubli du mot de passe.
  - Email de création de l'utilisateur
  - Email de regénération du mot de passe
  - Email de modification de l'utilisateur
- **Bonus** : Migrer Hapi de la v16 à la v17

Pour les deux projets vous devrez respecter les styleguides fournies lors du TP1

Avant de commencer

Avant de commencer ce TP, vous devrez terminer le cours nodeschool `make-me-hapi` (mis à part les deux derniers cours qui sont cookies et authentication) et récupérer dans votre repository git le projet `hapi-boilerplate` servant de base à ce TP et au projet fil rouge. (trouvable à cette adresse : https://github.com/throrin19/hapi-boilerplate)

Swagger

Swagger est un outil permettant de générer la documentation REST de votre service avec, pour chaque route, la possibilité de requêter le serveur depuis la documentation avec les champs correspondant aux données attendues.

Vous trouverez des exemples complets de ce que permet Swagger ici : Demo live Swagger

Mise en place

Afin de mettre en place Swagger, vous devez savoir qu'un plugin hapi est disponible. Il s'agit du module `hapi-swagger`.

Votre travail est d'intégrer ce module à votre projet Hapi de base (basé sur le boilerplate fournit en tp3) et renseigner l'entrypoint de base dans la documentation swagger.

Joi

Comme vous l'avez vu dans `make-me-hapi`, Joi permet de valider des données par rapport à un schéma. Mais Joi peut aussi facilement contrôler dans Hapi les données entrantes au niveau de vos requêtes, que ce soit dans les params GET ou POST et ce directement au niveau de la configuration de vos routes.

Afin de bien ranger tout ça, nous allons rajouter un répertoire `schemas` dans le répertoire `app` afin de stoquer nos différents schémas Joi (un fichier pour un schéma). Ce qui donne l'architecture suivante :

```
Hapi-boilerplate
└── app                 # Partie applicative de l'API
    ├── endpoints       # Entrées des traitements par les requêtes http
    │   └── default.js  # Entrées de toutes les requêtes sur `/`.
    ├── handlers        # Traitement des différentes entrées
    ├── plugins         # Plugins internes au projet
    └── schemas         # Schémas Joi
```

Tips : Vous verrez qu'en intégrant sur vos routes des validations Joi, Swagger va automatiquement remplir ses formulaires en fonction, ce qui vous permettra de tester facilement votre route.

Mise en place

Comme vous avez vu dans `make-me-hapi`, vous devrez utiliser le module `joi` au sein de votre projet.

Votre travail est de créer un enpoint pour une gestion d'utilisateurs. (Pour le moment juste une création sur la méthode `POST`) en vous basant sur la structure suivante :

```
{
    login : "Mon identifiant",
    password : "Mon mot de passe non crypté",
    email : "mon@adresseemailvalide.fr",
    firstname : "Mon prénom requis",
    lastname : "Mon nom",
    company : "Ma société",
    function : "Ma fonction"
}
```

Vous devrez effectuer les tests de validation suivants :

- login : champ requis
- password : champ requis d'au moins 8 caractères.
- email : champ requis et doit être un format d'email valide
- firstname : champ requis
- lastname : champ requis

Le résultat des requêtes de récupération utilisateurs (un ou tous) ne devront pas contenir le champ `password`.

Tips : Pour simplifier la suppression des champs, je vous conseille de regarder ce qu'il est possible de faire via le module `lodash`.

Organisation

- Essayez de toujours utiliser les promesses pour vos fonctions. Ceci vous permettra de facilement capturer une erreur que vous n'avez pas gérer sans faire planter le serveur.
- Les handlers ne doivent servir qu'à appeler différents plugins pour réaliser la tâche demandée. Le squelette des plugins que je vous conseille est le suivant :

```
'use strict';

const Promise   = require('bluebird');

// contient toutes les méthodes privées de votre plugin
const internals = {};

const externals = {
    pubFunc() {
        return new Promise((resolve, reject) => {

        });
    },
    register(server, options, next) {
        internals.server    = server.root;
        internals.settings  = options;

        // à répéter autant de fois
        // que vous avez de méthodes publiques
        server.expose('pubFunc', externals.pubFunc);

        next();
    },
};

externals.register.attributes = {
    name    : 'votrepluginavecunnomunique',
};

module.exports.register = externals.register;
```