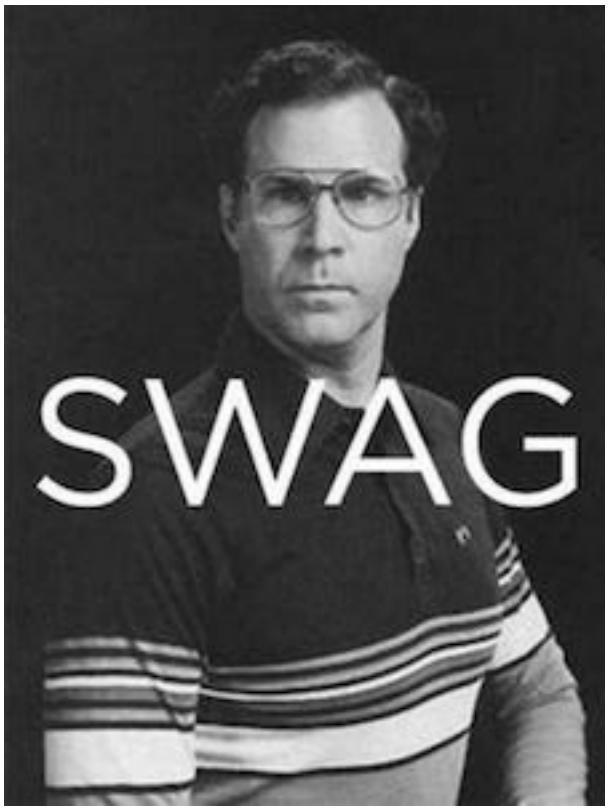
Hapi : Swagger et Joi



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 :

- **API 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

- **API Mail** : Relié via RabbitMQ 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

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 initialisé une nouveau projet hpal servant de base à ce TP et au projet fil rouge. (trouvable à cette adresse : https://github.com/hapipal/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`.

Pour intégrer ce module à partir du boilerplate hapipal, vous avez quelques commandes a effectuer pour ajouter des "falvors" (gôuts) à votre projet:

```
git fetch pal --tags
```

Afin de récupérer les "flavors" proposé par Hapipal

```
git cherry-pick swagger
```

Pour prendre le "flavor" swagger prédéfini

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 `lib` afin de stoquer nos différents schémas Joi (un fichier pour un schéma). Ce qui donne l'architecture suivante :

```
Hapi-boilerplate
└── lib                 # Partie applicative de l'API
    ├── routes          # Les routes de l'application
    ├── services        # différents services
    ├── models          # models de base de données
    ├── plugins         # Configuration de plugins / ou plugins internes du projet
    └── schemas         # Schémas Joi
```

Vous ne verrez pas tout les dossiers au début, nous allons en rajouter dans les prochains TP's

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 la suppression de champs penser au "destructuring" (décomposition). Cette téchnique peut éviter d'inclure une librairie de style `lodash` unutilement. A vous de voir!

Organisation

- Essayez de toujours utiliser async/await . Ceci vous permettra de facilement capturer une erreur dans le `catch` que vous n'avez pas géré sans faire planter le serveur.
- Les routes ne doivent servir qu'à appeler différents services pour réaliser la tâche demandée.

Services

Utilisant `Schmervice` de hapipal.

- Server methods
- Caching intégré

Example

Dans le dossier `services` créer un fichier appelé `user.js` contenant le code suivant:

```
'use strict';
```

```
const { Service }        = require('schmervice');

module.exports = class UserService extends Service {

  async initialize(){ // CALLED ON SERVER INITIALIZATION (onPreStart)

    // set up stuff here
  }

  async teardown(){ // CALLED ON SERVER STOP (OnPostStop)

   // tear down stuff here
  }

  hello(user){

    return `Hello ${user.firstName}`
  }

}
```

Vous pouvez accéder a vos service depuis `request` et `server` la method `.services`

Par example pour récupérer notre service utilisateur depuis une route nous aurons le code qui suit:

```
 (request, h) => {

   const { userService } = request.services();

   return userService.hello({ firstName: 'John'});
 }
```