Rapport de Stage

Samuel Monroe

16 mai 2016

Table des matières

Remerciements

Je voudrais tout d'abord remercier Virginie Van Den Schrieck et Marie-Noël Vroman, qui m'a poussé à postuler chez Belighted, pour le rôle qu'elles ont tenu dans le projet d'intégration du premier quadrimestre, en nous ayant responsabilisés quant à la gestion de notre projet et donné carte blanche pour la réalisation de celui-ci. C'est donc grâce à elles que j'ai fait mes premiers pas dans le monde du Ruby on Rails et du développement web moderne en général, et ainsi avoir eu l'opportunité de faire ce stage chez Belighted.

Je remercie Nicolas Jacobeus, CEO de chez Belighted, de m'avoir offert cette place de stage pendant ces quinze semaines où j'ai énormément progressé et pu apprendre de ce métier que je veux faire plus tard, ainsi que tous les membres de l'équipe pour leur accueil, leurs conseils et leur aide tout au long du stage.

Je voudrais également remercier Yves Delvigne, qui par son cours de programmation multimédia nous a inculqué les bases du développement web, et qui a donc grandement contribué au chemin que j'ai entrepris de suivre depuis la deuxième année.

Mes remerciements vont enfin à tous les autres professeurs de l'EPHEC qui m'ont accompagné tout au long de ce cursus de bachelier en technologie de l'informatique et qui ont, quelle que soit la matière enseignée, contribué à me transmettre un panel de connaissances varié qui m'est et me sera utile pour le reste de ma vie. Merci donc à Christian Lambeau, Claude Masson, Michel de Vleeschouwer,Youcef Bouterfa, Arnaud Dewulf, Stéphane Faulkner, Véronique Leclerq, Thomas Thiry, Maxime Vanlerberghe, Jean-François Depasse, Marc Deherve et Laurent Hirsoux.

Chapitre 1

Introduction

Point d'orgue de ces trois années de bachelier en technologie de l'informatique, la perspective de ce stage en entreprise est en même temps excitante, motivante, mais aussi un peu effrayante.
C'est le moment où nos acquis académiques sont confrontés à la réalité du terrain, celui du premier vrai pas vers la sortie de classes de l'Ephec.

C'est en direction de Belighted que j'ai fait ce pas, qui m'a accueilli grâce à une petite expérience déjà acquise en Ruby on Rails.
Lors de nos entretiens avant le stage, l'objectif et le travail que j'allais accomplir n'étaient pas clairement définis, néanmoins il était très clair qu'ils seraient de contribuer à un ou plusieurs projets au sein de l'équipe en faisant évoluer les compétences que j'avais acquises.
L'objectif qu'on m'a confié lors du premier jour de stage, que je décrirai plus en détail dans les pages suivantes, était de mettre sur pied un Software as a Service qui offrirait une gestion des avatars très simple à d'autres applications web, leur évitant tout besoin de mettre en place un espace de stockage dédié et d'implémenter un système d'avatars.

Concernant mes attentes personnelles pour ce stage, j'étais en premier lieu motivé par la perspective d'apprendre un maximum sur le développement web. Je voulais vivre une expérience dans une PME travaillant dans le domaine, découvrir les bonnes pratiques suivies lors du développement ainsi que les étapes du cycle de vie d'une application web.
Plus encore, je voulais avoir l'opportunité d'être très content de me lever chaque matin pour aller faire quelque chose d'agréable et de passionnant dans un cadre lui-même agréable.

Ce rapport de stage commencera par introduire la société Belighted, je parlerai ensuite de manière assez concise du travail que j'ai accompli lors de ces quinze semaines.
Ensuite viendra une explication plus précise de points techniques que j'ai rencontrés lors de ce travail et des expériences vécues au cours de celui-ci.

Chapitre 2

À propos de Belighted

Belighted est une PME active depuis 2008 sur le marché du développement d'applications web et mobiles.
L'équipe est composée d'un peu plus de dix personnes, jeunes et passionnées, toutes travaillant dans le même open-space. Tout ceci donne une ambiance sûrement plus proche de la startup que de l'entreprise classique.

L'entreprise s'est spécialisée dans le développement en Ruby on Rails, faisant d'elle et de son équipe des experts et la plus grosse boîte de développement en Rails de Belgique.
Cette expertise se fonde sur une recherche de la qualité, en suivant notamment des principes Agiles et par conséquent des pratiques telles que le développement dirigé par les tests ainsi qu'une relation proche au client.

Belighted étant composée d'une demi-douzaine de développeurs, il n'y a pas pour ainsi dire de structure vraiment définie et distincte au sein du personnel.
Tous sont des développeurs avec un minimum d'expérience en Ruby on Rails et JavaScript, certains étant plutôt spécialisés dans les technologies mobiles et le frontend JavaScript, d'autres vraiment spécialisés dans le backend de Rails.
Cette diversité dans les compétences leur permet de s'entre-aider et d'apporter des solutions à des problèmes survenant sur des projets dont ils ne font pas partie, et ainsi accroître la connaissance globale de l'entreprise.

Il y a donc tout de même une structure interne à l'entreprise. Au niveau de la direction Nicolas Jacobeus est le *Chief Executive Officer* et fondateur de Belighted, accompagné d'Aurélien Vecchiato qui est le *Chief Operating Officer*.
Pour ce qui est des développeurs front-end et designers on retrouve Maxime Nguyen et Simon Henrotte.
Les développeurs backend Ruby on Rails sont Philippe Van Eerdenbrugghe, Kathleen Clacens, Philippe Bourez et Dominique Lebrun.
Michaël est analyste business et développeur.

C'est donc Philippe Van Eerdenbrugghe qui a été chargé de m'accompagner dans la réalisation de mon SaaS, étant un développeur senior en Rails et ayant de très grandes connaissances et compétences. Il m'a énormément appris et surtout conditionné au développement dirigé par les tests et à l'application de spécifications et de bonnes pratiques dans le développement logiciel.

Etant installé entre Kathleen et Philippe tout au long du stage, Kathleen a également pu m'aider sur certains points en Rails, ainsi que tous les membres de l'équipe qui m'ont conseillé et orienté sans hésitation lorsque j'avais la moindre question.

Chapitre 3

Nature du stage et modalités de réalisation

Etant novice dans le développement Rails et JavaScript, Nicolas m'a confié un projet interne qui me permettrait d'une part de faire énormément progresser mes compétences dans ces domaines, et d'autre part de me confronter à des problèmes et à des questions que pose la conception d'un *Software as a Service* pour contribuer à la connaissance de l'entreprise dans ce domaine.
J'ai également été chargé de rédiger de manière régulière des *blogposts*, afin de contribuer à l'activité du blog de Belighted et de partager cette expérience que j'aurais à mener.

Ce SaaS a été nommé **Pravatar**, pour *Private Avatar*, et l'idée était de développer un service qui se chargerait de toute la gestion d'avatars à la place d'applications clientes.
Une application cliente désireuse d'implémenter un système d'avatars n'aurait alors plus qu'à souscrire à Pravatar, et dans l'idéal plus qu'à insérer un tag HTML *<script>* pour importer un script JavaScript permettant la gestion des avatars.
Pravatar aurait alors pu faire économiser à l'équipe un certain nombre de jours de développement par projet en matière de gestion d'avatars, mais potentiellement se vendre à d'autres applications.

Ma mission était donc la suivante :

— Explorer dans quelles mesures un service de ce type est faisable.
— Mettre en lumière les contraintes rencontrées lors du développement.
— Apporter des éléments de solutions à ces contraintes.
— Au terme du stage, avoir mis sur pieds un service qui permette à Belighted de déterminer si la solution est viable, et maintenable par l'équipe si tel était le cas.
— Le test final était d'implémenter Pravatar sur Scale, un SaaS de resource management développé en interne par Belighted.

Je vais au long des sections qui suivent décrire de manière assez concise l'entièreté du travail accompli sur ce SaaS au cours de mon stage, les détails plus techniques et qui ont réellement contribué à l'entreprise seront détaillés dans le chapitre suivant.

1 Les bases du service

1.1 Mise en place du backend

La première étape de mon travail sur Pravatar fut de mettre en place le bon environnement de travail et de poser les bases saines du backend en Ruby on Rails.
Philippe, mon mentor lors du stage, en a surtout profité pour m'inculquer les bonnes pratiques à suivre et la façon dont l'équipe s'y prend pour commencer un nouveau projet.

J'ai donc commencé par configurer tout l'environnement, ainsi que d'inclure un service d'administration de l'application via la gemme *ActiveAdmin*.
Une fois installée, j'ai pu commencer le développement de la partie administration en **Behaviour Dri-**

ven Development avec RSpec.

Le but était de me former dans la démarche d'écrire des tests de features avant toute chose, écris à la manière des *User Stories* propres aux méthodes Agiles, et seulement enfin d'écrire le code qui couvrait ces tests, suivant le cycle Red-Green-Refactor. Ce fut une vraie mise en pratique des concepts théoriques enseignés lors du cours d'intégration de Mr.Thiry.

Une fois habitué à cette manière de travailler, j'ai pu commencer le développement d'une **RESTapi** sur le backend qui serait chargée de fournir les endpoints pour le script tiers à inclure chez les clients.
Encore une fois, ce développement s'est fait dans une démarche *TDD*, où j'écrivais les tests de requêtes avant même d'écrire cette API.

1.2 Le script tiers

Les détails de conception du script ont été placés sous mon entière responsabilité, c'était justement un des points d'interrogation de Belighted au sujet de la réalisation de ce type de Software as a Service. Cette solution se devait d'être simple à installer et ne pas entrer en conflit avec la configuration du client, peu importe les technologies utilisées par celle-ci.

J'ai donc pris le parti de développer ce script en pur JavaScript/jQuery, en faisant en sorte que ce script soit fourni avec sa propre version de jQuery afin de ne pas interférer avec une possible version préinstallée chez le client.
Ce script était chargé de parser le code HTML présent chez le client pour y cibler des éléments comportant des *data-type* spécifiques précisant l'upload ou l'affichage d'un avatar, et d'ensuite interagir avec mon API afin de récupérer ou mettre à jour les avatars.

Philippe m'a bien évidemment chargé de mettre au point une solution de testing de ce script tiers, qui était fourni par l'application Rails mais qui existait en tant qu'élément distinct du backend.
Le framework de testing **Jasmine** a fourni une solution très proche de RSpec dans la manière syntaxique d'écrire les tests, et l'utilisation de **Karma** en tant que test-runner a permis la gestion entière du script au sein même de l'application Rails.

2 Améliorations de Pravatar

Les bases du service ayant été posées et ayant pu faire fonctionner celui-ci, j'ai dû améliorer l'ensemble qui était encore relativement basique.

2.1 Améliorations du backend

La première amélioration a consisté en l'insertion d'une quatrième couche à l'architecture MVC. En effet, la tendance d'un débutant en Rails est de surcharger la responsabilité des contrôleurs, alors qu'en principe un contrôleur n'est chargé que de récupérer une ressource et de répondre à une requête avec celle-ci.
La phrase Rails visant à pallier à ce problème annonce *"Fat models, skinny controllers"*.
La surcharge des modèles n'est en réalité pas la meilleure façon qui soit de concevoir l'application, et c'est pour cela que j'ai été chargé par Philippe d'insérer des **interacteurs**, qui sont des objets chargés d'encapsuler la logique business d'une application, et de laisser aux contrôleurs la façon spécifique de répondre aux requêtes.

J'ai dû ensuite mettre en place du stockage des images sur une instance Amazon S3.
Cette configuration du stockage Amazon S3 est justement ce que Pravatar devrait absorber sur les autres projets de chez Belighted.

Une importante mise à jour du service fut sa mise en conformité par rapport à la spécification *JSON :API*. En effet, une API JSON peut en pratique répondre ou recevoir des requêtes comportant n'importe quelle structure JSON.

Cette liberté peut néanmoins se révéler problématique selon la vision des développeurs quant à la définition des ressources qui doivent être échangées sur cette API.
J'ai donc dû transformer celle de Pravatar afin de suivre la spécification JSON :API, qui précise une manière "standard" de représenter ces ressources.

2.2 Améliorations du front-end

Au fur et à mesure du développement du script tiers, travaillant avec du Javacript pur, je me suis rapidement retrouvé avec un amas de fonctions difficilement maintenables.
J'ai pu grâce à **Browerify** opérer une découpe de mon code en modules indépendants et définis selon leurs responsabilités.
Le code du script est devenu beaucoup plus compréhensible, et évidemment beaucoup plus maintenable du fait de cette découpe en modules.
Une réécriture du code en Coffeescript m'a également été demandée par Philippe.

Suite à cette réécriture en Coffeescript et à l'indépendance de l'environnement du script, la compilation et le packaging de celui-ci n'étaient évidemment plus assurés par l'asset pipeline de Rails.
J'ai donc pu mettre en place un système d'automatisation de compilation et de configuration avec l'outil **Gulp**, ce qui a permis un travail fluide sur le script plus ou moins comme on aurait travaillé sur Rails.
Ceci était impératif, puisque le script devait résider au sein du projet Rails, mais vivre dans son propre écosystème.

2.3 Gem

Afin de faciliter l'implémentation du service sur des projets Ruby on Rails, j'ai entamé le projet de créer une gem qui pourrait être incluse dans les projets clients et qui rendrait le service extrêmement facile à mettre en place.

Cette gem était un **engine** Rails, fournissant à l'application deux méthodes très simples et concises afin de déclarer soit l'affichage d'un avatar, soit la définition d'une zone d'upload pour mettre à jour l'avatar.
Le code HTML généré par ces méthodes était personnalisable via des générateurs, qui dupliquaient les templates fournis par la gemme chez le client et l'autorisaient à totalement modifier le rendu.

2.4 Sécurité

A ce stade avancé du projet, la question de la sécurité du système des avatars s'est posée. Il fallait pouvoir empêcher des uploads non désirés sur Pravatar.
Cette sécurisation a été accomplie via l'utilisation de **JSON Web Token**, qui permet une signature de données JSON via une clé secrète et est décrite dans la *RFC 7519*.

3 Finalisation du projet

3.1 Fin du développement

Afin de pouvoir tester mon travail avec le backend, le script et la gemme, j'ai créé un petit projet Rails dans l'unique but d'implémenter le service Pravatar.
L'utilisation de la gem et du script s'est révélée être extrêmement simple, et le tout fonctionnait très bien.

L'étape finale de la mise à l'épreuve de Pravatar fut son implémentation sur Scale, un des projets internes de chez Belighted.
Scale fonctionnant avec un stack AngularJS/Rails, je me suis rapidement retrouvé dans l'impossibilité d'utiliser la gemme et le script sur ce projet à cause de la single-page app écrite en Angular.
J'ai donc eu l'opportunité d'apprendre AngularJS et de commencer l'implémentation du service en pur Angular en utilisant l'API de Pravatar.
Le résultat final s'est avéré vraiment gratifiant pour ma part, me lancer dans une nouvelle technologie telle qu'Angular n'ayant pas été forcément très facile dès le début.

3.2 Dernier travaux

Les derniers jours de mon travail au sein de l'entreprise ont consisté en la rédaction de mon compte-rendu sur les découvertes et éléments importants que j'avais pu retirer de cette expérience, et qui étaient très importants pour Belighted et pour leurs futurs travaux sur des SaaS.

Chapitre 4

Expériences vécues et objet d'étude

1 Expériences vécues au sein de Belighted

D'un naturel assez timide et réservé, j'espère et j'estime cependant m'être assez bien intégré au sein de l'équipe de chez Belighted.

Au niveau des relations avec les collègues, j'étais principalement en contact avec Kathleen et Philippe puisque je travaillais entre eux deux, mais j'ai pu également pas mal discuter avec l'ensemble de l'équipe, que ce soit à propos de sujets techniques ou totalement personnels.

Chaque lundi midi se tenait un *weekly*, sorte de stand-up meeting, mais où toute l'équipe se réunissait pour manger et où chacun parlait de ce qu'il faisait actuellement.
Je pense que ce weekly était surtout utile pour Nicolas afin qu'il ait une vision globale d'où en étaient tous les projets.
De vrais stand-up meetings me semblaient difficiles à tenir puisque maximum deux employés travaillaient sur le même projet en même temps.

J'ai eu aussi l'occasion d'aller manger à l'extérieur avec l'équipe, faire une sortie badminton, mais aussi d'aller quelques fois boire un verre hors des heures de bureau une fois la journée terminée.
C'était vraiment agréable et je ne sais pas si j'aurais pu avoir l'occasion de faire ce genre de choses dans une plus grosse entreprise.

2 Compte rendu sur l'expérience du développement d'un SaaS

Ce compte-rendu de mon expérience reprend avec plus de détails tous les éléments de réponses et découvertes qui seront utiles à l'entreprise suite à l'accomplissement du projet Pravatar.

2.1 Création d'un script tiers en pur JS/jQuery

Le premier challenge du service Pravatar était le suivant :
"Comment créer un third-party widget à intégrer dans d'autres applications pour utiliser le service de Pravatar, sans altérer la configuration du client ?"

La solution qui présentait le moins de contraintes et également l'approche la plus simple pour moi s'est avérée celle de créer un snippet **JavaScript** important sa propre version de **jQuery**, de sorte qu'il n'interfère en aucun cas avec l'application du client peu importe sa configuration.

2.1.1 Browserify

jQuery ayant été sélectionné pour accomplir le travail d'échange avec Pravatar, et donc d'affichage et d'upload chez le client, la question de **conflit** de version s'est posée.

Browserify couvrait cette question en proposant l'option de packager des éléments JavaScript entre eux afin d'en ressortir un *bundle* indépendant des technologies présentes chez le client.

Ceci a permis via un système de "require()", de spécifier le requirement de jQuery dans le snippet et donc de l'intégrer dans une application cliente de manière transparente et non conflictuelle.

JavaScript tel quel ne propose pas réellement de solution pour séparer le code selon les **responsabilités** couvertes.
Du fait de mon expérience un peu trop fraîche sur le JavaScript au moment de créer ce script, celui-ci est rapidement devenu un gros fichier plein de fonctions s'appelant les unes les autres et dont les liens devenaient un peu difficiles à identifier.

J'étais cependant passé à côté d'un avantage qu'offrait **Browserify** et que j'aurais pu utiliser afin de régler ce problème.

Le système de requiremenents fourni par Browserify pour intégrer d'autres librairies au script peut aussi être utilisé au sein du script lui-même pour décomposer le code et créer un arbre logique de sous-composants selon leurs **dépendances** et **responsabilités**.
Cette décomposition force une certaine rigueur dans le développement de ces composants, d'autant que deux composants qui se "require()" mutuellement provoquent une dépendance circulaire empêchant le code de fonctionner.

Enfin, encore un avantage proposé par cette décomposition en **sous-modules** via Browserify était lié au testing.
En effet, certains éléments concernant le drag'n'drop des images pour upload aurait été extrêmement compliqués à tester.
Cependant, en incluant le sous-module responsable de l'upload via un require() dans le test associé, il était possible de tester de manière isolée l'upload d'une image, que celle-ci soit faite en drag'n'drop ou via un file input.

2.1.2 Utilisation du script chez le client

Le script étant bundlé et prêt à l'emploi, les options d'exécution du code chez le client n'étaient pas innombrables.

Les options qui ont été trouvées pour ce faire sont les suivantes :

— Attacher à l'objet "window", un objet "pravatar" qui expose les fonctions utilisables par le client pour générer l'affichage et l'upload des avatars.
— Paramétrer le lancement des fonctions sur l'évènement $document.ready() afin de parser la page et générer les éléments dans le DOM.

La solution de l'évènement "ready" s'est avérée relativement fonctionnelle sur une application pure Rails stateless, où les actions rafraîchissaient la page et accomplissaient bien le travail attendu.

Dans l'implémentation de Pravatar sur Scale, l'utilisation du script s'est révélée **impossible**, une implémentation directe en Angular utilisant les entrées d'API sur service Pravatar était beaucoup plus efficace et élégante.

2.2 Testing du script tiers fourni par Rails

La mise en place d'une solution de testing pour ce snippet JavaScript fourni par l'application Rails de Pravatar était indispensable.

2.2.1 Jasmine et RSpec

Étant donné la syntaxe type scénario utilisée pour tester l'application Rails avec RSpec, il fallait dans une optique de consistance construire l'environnement de test JavaScript sur cette même syntaxe.

Jasmine s'est avéré être un très bon framework de test pour ce but, et permet d'avoir des déclarations de tests qui suivent ce schéma :

```
describe("A suite", function() {
  it("contains spec with an expectation", function() {
    expect(true).toBe(true);
  });
});
```

Un autre gros avantage de Jasmine est la possibilité de lui lier des modules de testing pour assurer des fonctionnalités plus spécialisées.
Dans le cas du snippet de chez Pravatar, les fonctionnalités à tester étaient principalement liées au **jQuery**.
L'inclusion du package jasmine-jquery a grandement couvert ce besoin en fournissant des matchers tels que toBeChecked(), grandement utiles pour tester les changements dans le DOM.

2.2.2 Test Runner : Karma

Une fois le framework de test sélectionné, il fallait un moyen de pouvoir faire tourner ces tests Jasmine au sein du projet.

Le script étant réellement une fonctionnalité séparée de l'application Rails, le test runner devait également vivre dans son propre éco-système au sein du projet.

Installé via npm, *Karma* est un test runner qui va exécuter les tests Jasmine sur un browser spécifié dans la configuration.
De nombreuses options de configurations peuvent être paramétrées sur Karma, c'est là également que se retrouvent listés les autres modules dont les tests dépendent, tels que jasmine-jquery.

2.3 Automatisation des tâches sur le script client

En plaçant l'environnement du script client dans son propre écosystème au sein de l'application Rails, un autre problème a dû être résolu.

En effet, les sous-modules du script ayant été réécrits en **Coffeescript**, ils n'étaient plus servis et donc compilés par l'asset pipeline de Rails.
Un moyen était donc nécessaire pour pouvoir travailler confortablement sur ce script, sans devoir à chaque fois recompiler le tout et relancer les tests manuellement, ce qui aurait été un gros frein au développement du script.

Ce moyen s'est avéré être **Gulp**.
Gulp m'a permis d'automatiser ces tâches de compilation/testing en créant des tâches gulp qui résument en une commande une série d'actions à effectuer.
Le fichier Gulp de Pravatar permet de :

- Lancer les tests une fois, ou en mode surveillance à chaque changement des sources
- Lancer la compilation et le bundling des sources en une fois, ou également en mode surveillance afin de ne pas devoir relancer celle-ci à chaque changement dans le code.

Gulp a donné un moyen de travailler dans un worklow très proche de celui de Rails, où les tests sont aisément lancés et où il n'est pas nécessaire de déporter son attention du code pour continuer le travail.

2.4 Signature de données échangées via l'API

Il a fallu un moment donné se pencher sur la question de la sécurité à propos de Pravatar.
Comment empêcher un utilisateur de modifier les valeurs du formulaire d'upload, afin de par exemple modifier l'avatar d'un autre utilisateur ?

2.4.1 Solution maison

Ma première réaction a été d'imaginer un système de sécurité "maison".

Ce système de sécurité ajoutait aux *params* de la requête un payload de sécurité garantissant la validité de l'upload, il était établi de la manière suivante.

Au niveau client :

— Un hash était créé en chiffrant les données utiles via OpenSSL en utilisant l'API key du projet associé.
— Un digest base64 était ensuite généré à partir du hash, et placé au côté des données utiles avant envoi au serveur.

Au niveau serveur :

— Le digest en base64 était décodé.
— Un hash était recréé à partir des données en clair.
— Le hash reçu était comparé à celui recréé pour validation.

Ce système, bien que fonctionnel, était extrêmement lourd et rien ne me permettait d'affirmer l'inviolabilité de mon système.
De plus le chiffrement via OpenSSL était complètement inutile.

2.4.2 JWT, la bonne solution

Rétrospectivement, j'avais mal évalué les réels besoins du système de sécurité à implémenter.

L'**essentiel** du problème était uniquement de s'assurer que :

— L'upload ne doit pas être rejouable par un utilisateur.
— L'utilisateur ne peut pas cibler un autre en modifiant les données d'upload.

L'url de l'API indique de toute façon l'id du profil et le nom du projet, et les données échangées n'ont absolument rien de *confidentiel*.

Le système de **JSON Web Token** permet de créer un ensemble de données comprenant un **header**, le **payload** avec les données utiles et une **vérification de signature**.
Ce système permet d'échanger des données signées avec une clé, et la signature prouve qu'aucune donnée n'a été altérée.

Les avantages à utiliser JWT au lieu d'une solution customisée sur Pravatar étaient les suivants :

— JWT est décrit dans la RFC 7519 et est donc standardisé et ouvert, inutile d'élaborer un système de signature incertain et lourd.
— La complexité du code de l'upload sur l'API a été considérablement réduite, l'encodage/décodage se fait via une simple instruction. Un décodage échouera sur la moindre donnée altérée et déclenchera une erreur, simplifiant encore l'implémentation.

2.5 Communication entre l'app cliente et Pravatar

Par défaut Rails empêchait l'intervention du third-party widget sur son API.

Le problème était lié aux CORS (Cross-origin resource sharing), les requêtes provenant d'un autre domaine que celui de l'app Rails étaient tout simplement refusées.

Ceci a été réglé via la mise en place de la gem rack-cors et en spécifiant dans le fichier de configuration application.rb les méthodes HTTP autorisées ainsi que les origines autorisées.

2.6 Conclusion de l'expérience

La conclusion qui est ressortie au terme de ce travail sur Pravatar était que ce SaaS demanderait peut-être plus de jours/homme à être maintenu que ceux qu'il ferait économiser sur les divers projets.

Un gros problème pour les applications JavaScripts était que la clé API posait des soucis de sécurité, en effet sur Scale, un provider Pravatar était initialisé avec la clé API et était forcément visible dans les fichiers sources reçus par le client.

Cependant, le service pourrait avoir un intérêt réel pour sa mise en place chez d'autres clients.

Chapitre 5

Conclusion

Je suis très heureux d'avoir eu l'opportunité de faire mon stage au sein de l'équipe de chez Belighted.

Tout au long de ces quinze semaines, j'ai énormément appris et fait progresser mes compétences dans le développement web.
Je peux le constater rien qu'en regardant à nouveau notre premier projet en Rails accompli pour le cours d'intégration des technologies.

Je ressors de cette aventure avec des compétences que j'avais avant le stage beaucoup plus affûtées, mais également avec des tas de nouvelles autres.
Cette expérience aura également été l'accomplissement pratique de nombreux concepts théoriques qui m'ont été donnés lors de cours au sein de l'Ephec, ne serait-ce qu'en matière de méthodologie Agile et de toutes les bonnes pratiques que celle-ci met en avant.

Ce stage conclut ma troisième année de bachelier à l'Ephec m'aura apporté énormément et me conforte plus encore dans l'idée du chemin que je veux poursuivre.

Annexes

Development of a SaaS as an IT trainee

2016/02/08

Introduction

My name is Samuel Monroe, I'm a 23 years old student at EPHEC Louvain-la-Neuve.

As I'm currently in my last year of bachelor in IT, I had to find a company which would give me the opportunity of an internship during 15 weeks.

I then applied to Belighted, which was offering an internship among a young team of passionated and motivated web developers, working on various and interesting projects. It seemed to me that it would be the best place to acquire great professional experience and programming skills, but also to really enjoy these 15 weeks.

Here's where my story with Belighted begins.

My background

EPHEC is more focused on networking stuff than development, moreover I started quite lately to learn my first programming language.

However, I went through Pascal, Java, C and PHP during my studies.
I wanted to learn other languages and framework to achieve some projects, which led me to **Ruby on Rails** and basic **Javascript**.

The SaaS Project

Pravatar's goal is to provide web applications with a simple and efficient way to manage their users' avatars, allowing each user to have a different and private avatar on each Pravatar-enabled app they use, just with a single line of javascript.

This will also be a solution to avoid setting up a specific storage service for every app created by Belighted only for avatars, delegating this to the Pravatar service.

The back-end would be running on a **Rails** app, providing an API and a way to manage the accounts.
The front-end would be a Javascript third-party widget, which could be included in a client's app with a `<script>` tag requesting the script from the rails app.

My First Week

On the Back-end

First steps

I spent my first day setting up the **Rails** environment and getting a bit used to **RSpec** testing.

Philippe asked me to install *ActiveAdmin* and play with it, writing some test in a User-Story way.

Bases of the Pravatar service

As I'll describe later, the client side will rely on a script (included in the client app) which will interact with the app's API.

So my first objectives for the back-end were to find a way to provide the script, build a basic API and write test on it.

Providing the script

My main problem here was to ensure that the script would still be available by its name in production, like `http://app.io/assets/script.js`, and avoid a digest on the filename produced by the assets pipeline.

This could have been an irritating problem, but I found a gem allowing me to white-list assets which would be available with and without the digest.
The gem is `non-stupid-digest-assets`.

Building the API

Kathleen introduced me to the **api-versions** gem to manage an API, then I created a controller called `profile_widgets_controller` inheriting from the `profile_controller` and controlling my profile resources.

This single controller in the API just provides a path to the *avatar* from a received *profile_id*.

Testing the API

The test is pretty simple, creating a fake account and a fake profile, stubbing the request with a valid ID. Then verifying the response has a valid **200** status, and containing a well formatted body.

On the Front-end

Choosing the technology

My objective here was first to define which technology to use in order to develop the **third-party widget** that will be included in the client's app.

This inclusion solution meant:

- The widget should be simple to install
- As it would be written in some *Javascript* framework, the whole package should not interfere with the client configuration.

As the available WebComponents technologies seemed to cause version conflict for now, the solution I chose after some discussions with Simon and Maxime was to bundle my javascript code with his whole jQuery version using **Browserify**.

The Widget istelf

For now the widget is pretty simple but correctly fulfill my first goal, fetching the pravatars by:

- Parsing the DOM searching for a div with the specific `data-type`
- Making a request to the API using a key value
- Appending an `img` tag to the div using the API response

The process, was launched by the `ready` event, so wrapped in a `$('document').ready` event listener (*This was going to get me into some trouble later*).
Finally, Philippe asked me to rewrite it in **Coffeescript**, never wrote any lines in Coffee before but I quickly get into it.

Testing

This is the most **painful** part of my first week.

I set up my test environment with Karma and Jasmine (*Jasmine to keep the BDD syntax of RSpec*).
Also, I had to include `jasmine-jquery`, `mock-ajax`, and `jquery` to Karma.

My main problems were:
- Understanding how Karma works
- Beeing aware that I could not test the script without stubbing my API
- Finding a solution to test and control something that was waiting for the DOM ready event, without knowing exactly when this event was occuring

So, this is how I got the whole test working (for now) with the precious help of Michael and Philippe :

1. In a `beforeEach` section, I **appended** my special pravatar div into the DOM, and I **stubbed** the API to intercept the calls and provide a specific response to any of these calls.

2. As the snippet code was wrapped into the `$('document').ready` listener, I had to create a way to call this code when I wanted to.
   This has been achieved by creating a `window` object, and giving him a single method called `init()`.
   Then, the `$('document').ready` listener would trigger the `window.object.init()` method.
   (*Sounds so simple now...*)

Conclusion

This first week as an intern was really great, Philippe gave me a lot of advice and taught me a lot of useful things.

Managing to have my Javascript tests running after so much struggle made me feel great and even if I'm sure I'll encounter great difficulties again, I'm really motivated to achieve great work.

I enjoy the fact I'll be working at Belighted every day, and that's exactly what I expected of my internship.

Development of a SaaS as an IT trainee

2016/02/19

Introduction

This is my second blog post about Pravatar and my third week among the Belighted team is already over, time flies so fast!

In this one, I'll share with you what seemed to be the most interesting things to talk about for me.

In order not to lose you with my explanations nor to overload you with details about the SaaS, I'll begin with a little summary of the main features of Pravatar at the time of writing, then I'll talk about the back-end and the front-end.

Pravatar

Pravatar now has a default page where guests can register to the service, manage their accounts and set default avatar.

Client snippet now allows an app to request different versions of an avatar, to upload an avatar for a profile and also to create accounts on the fly.

On the Back-end

The main tasks on the back-end were to continue building the API and adapting the app in order to suit the client needs, but it led me to make some major changes to the app.

Interactor

One says :

> Fat models skinny controllers.

But Philippe taught me that putting all the logic in models is not the solution either.
What we want is to keep in the models their own specialized logic, but we can extract the business logic in a fourth layer which is called an **Interactor**, and which reflects (as the creators of the gem say) what our application *does*.

I built an interactor to represent the *registration* process, it simply creates a `user` and its `account`

with given parameters, and returns a context with a success or a failure.
This *interactor* is then called by a *controller*, which will additionaly perform its more specific actions.

AWS S3

Despite how obscure it sounded to me, setting up S3 to store the *avatars* was quite fun.

The **fog** gem provides a cloud services library, which can easily be used with **CarrierWave** to uploads your files to S3.
My requirements were to keep the local storage for *testing*, and to target three different buckets for *development*, *staging* and *production*.

This was simply done by giving **fog** my AWS credentials, make **CarrierWave** use *file* or *fog* storage depending on the current environment, and define the right *buckets* for the non-testing environments.

Once configured, no changes were needed in the avatars implementation, **CarrierWave** works like a charm.

On the Front-end

Gulp : Comfy Testing & Workflow

Lastly, my javascript tests were located in the `spec` folder which normally concerns the RSpec tests on the Rails app only.
The project has two distinct sides but we wanted to keep the client one close the the server one, in order to easily maintain the whole package and simply because the **js snippet** is provided by the Rails app.

So I created a dedicated directory on the app's root for all the front-end app, and gave it the miraculous tool which is **Gulp**.

Gulp revealed itself a **freaking great** solution to a whole bunch of current or even future problems :

- How to easily launch tests while working on the app
- How to manage the bundling with jQuery everytime the snippet source change?
- As the tests and sources are written in coffeescript, how to avoid the early solution which was to use the rails assets precompiler?

The workflow on the widget now goes like this:

1. Launch gulp to `watch` for any changes in my .coffee sources, to compile them if it occurs
2. Write the tests
3. Write the snippet code, which will be compiled and bundled on the fly, and also automatically tested by gulp & karma with the tests I just wrote.

Sorry for hating you two weeks ago, testing.

Improvements on the Widget

Fetching the avatars

First of all, in order to move towards what the project will be later, the avatars are now fetched by the combination of an **account id** and an **external id** referencing the profile.
This means an application can register profiles for their users providing Pravatar their "*primary key*", no matter what they are (mails, usernames, ...).

Upload and default avatar

Account owners are now able to define a default avatar for their profiles, which will automatically be displayed if a requested profile doesn't exist or doesn't have its avatar.

An upload functionnality has been added to the widget, a pravatar `div` can now be used to define an upload zone for a requested user, existing or not.
If an upload occurs on a non-existing profile, it will be created on the fly by Pravatar on the related account.
This is moving towards the **plug'n'play** characteristic of Pravatar, an easy-to-use service and no constraints for the app which will use it.

Conclusion

Step by step, increment by increment, the SaaS is progressively taking shape.

It's only the beginning but I really enjoy working on it, and also learning valuable things every day and working with all the people at Belighted.

Development of a SaaS as an IT trainee : part 3

2016/03/29

Introduction

Day after day, Pravatar is growing.
My work these days is more about details and refactoring than brand new features, yet it might be the most important part of a good project.

NO Like my previous posts, let's talk about the back-end and front-end of Pravatar.

Backend

JSON:API

In order to build Pravatar on good and maintainable bases for the Belighters who will continue the project at the end of my internship, Philippe asked me to change my API to make it compliant to the **JSON:API** specification.
This specification is pretty simple and clear, describing what your API **must**, **may** or **should** ask and answer.

However, these changes made me reconsider this API and helped me to *increase* the code *quality*, but also made me *learn* a lot of new useful things.

ActiveModel Serializer

As you may know, one of the first things the JSON:API stands is that your API **must** respond or be requested with a **single resource object** containing at least its *id* and *type*.

Although it sounds logical to me now, considering I want to follow a **RESTful** architecture, I was making some mistakes :

- My API responses and requests weren't correctly composed of a single (or an array of) resource object.
- I was building the JSON responses in the API controller.

As the responses must represent a *single resource object*, the simplest way to move the representation responsibility out the controller and to be compliant to the specification is to use **ActiveModel Serializer**.
This gem lets you define serializers basically defining the "JSON representation" of a resource. These

serializers will be used when calling :
`render json: your_resource`

This being done, I achieved to move out pieces of code which weren't supposed to be under the responsibility of the controller and to correctly follow the JSON:API specification.
Moreover, when my API now calls `render json: profile`, it really does what it means.
It's not anymore about rendering some JSON I build previously, is about rendering the *resource* the client requested.
The intent is *clear* and *precise*.

JSON Web Tokens

Pravatar was pretty simple back then, but a question fastly came up after I implemented the upload : *How to ensure an upload is legit ?*

When I first considered the problem, I came up with a homemade solution.
That solution was actually working fairly good, but I later discovered **JSON Web Tokens** which was the right choice to solve the problem, for these reasons :

- It's an effective way to communicate a small payload of information between two sources via an unsecured transport
- It can be sent inside a HTTP header
- It is signed with a secret (our API keys in that case)

This allowed me to follow the JSON:API, and to send meta informations related to the request and not directly to the resource inside the *JWT*.
Putting these fields (like the upload's datetime) inside the JWT's payload and signing it with the API key was the simplest way to ensure the upload validity.

JWT also provides a solution for **authentication** via APIs.

Frontend

Browserify

At some point of the project, a terrible thing happened: my javascript went all *spaghetti*; the snippet was a single code file made of multiple functions calling each other.
I overstate it a little bit, but I can be sure it would have been a total mess with the code growth.
This was the time to **take action**.

Funny thing is that I already was including the right tool to achieve this from the beginning: the great Browserify.

I divided the code in *modules*, each one beeing a class and having its own responsibility.
Then, like I require *jquery* in the main module for the bundling, I can `require` my other modules, providing me with an instance of the class with its public functions.

Organizing your code with *Browserify* **forces** you to correctly dispatch the responsibilities between your modules.
Indeed, if your module **A** require the module **B** to run but **B** also require **A**, you create a *circular dependency* that will prevent your script to run.

Engine

As long as there was no security feature in Pravatar, client could just use the javascript snippet and write html tags to fetch and upload the avatars.

Implementing the security system meant finding a way to make the client generates the **JWT** with its *API Key*.

I achieved this by building a gem for a Rails client app. This gem is an engine, providing the client with:

- Helpers method to create fetch and upload tags
- A generator to copy the html templates of the generators and allow the clients to modify them following their own taste.

The client just has to install the gem and to set its API Key in `secret.yml`, then he's ready to use Pravatar.

Conclusion

I remember thinking how far I was from completing this projet during my first weeks at Belighted.
Actually, I'm quite impressed to see what I've learned during these seven weeks and what I'm able to achieve today.
I'm very grateful to Belighted for having taken me as a trainee and made me progress that much.

There is still six weeks of internship to enjoy !

Development of a SaaS as an IT trainee : final part

Introduction

It's already the end of my internship at Belighted, last weeks passed amazingly quickly.

I'd like to thank so much all the Belighters for their particularly warm welcome, and all the things and advice they gave to me.
Especially Philippe who was my internship mentor and taught me a huge amount of numerous things about good practises, design pattern, RESTful architecture, testing (a lot about testing), etc ...

It's kinda funny to take a look again at the code of my first school project in Rails. This is making me realize how much Belighted made me progress.

Let's now talk about what I did and learnt during these last weeks.

Implementing Pravatar on Scale

Scale is one of Belighted's project.
It's a resource managment system SaaS designed for digital agencies.

Scale has no avatar management except Gravatar, this is what led to the idea of Pravatar.
That idea is providing Scale with a separate avatar system, then re-use it on other projects needing avatars, and eventually making it available for external projects.

So once I made Pravatar working, and tested it on an other project including the dedicated **gem** and javascript **snippet**, it was the time to make the final test, running Pravatar on Scale !

The available tools

During this work on Pravatar, I built these tools in order to make clients able to use the services :

- **API** : An API following the JSON API specification.
- **Script** : Included in a `<script>` tag by the client, which can then use Pravatar feature simply by using some `data-purpose` attributes in the HTML.
- **Gem** : Working alongside the script, the gem is simplifying the process a step further, providing two simple, concise and configurable helpers for displaying and uploading Pravatar features.

Equipped with these tools, next step was to discover Scale codebase and figure out how to set up Pravatar.

Scale analysis

At first glance, Scale quite frightened me because it was something I never saw before.
I was expecting some simple Rails structure with just an app folder containing the `mvc` folders and nothing more, so I could just insert the snippet in the classic `application.html.erb` file, then put the gem in the gemfile and raise my hand telling I was finished with this.
How naive was I ! However, looking at it today, I'm **so much** more satisfied that it hasn't been that easy.

So, Scale was in fact an *AngularJS* single-page app served by the Rails backend app.
I didn't understand how exactly AngularJS is running with Rails in Scale, but I quickly understood that I would have to work only with Angular. So I took some days to learn Angular after analyzing the code.

The tools and Scale

Equipped with my new knowledge of AngularJS, I was able to review the usability of my tools inside the project.

Javascript Snippet

The main problem with my snippet was that I based all the functionnalities on the `$document.ready()` event, which was fine with a "stateless" web-app like my Rails demo, but totally **not** with an Angular "stateful" one !

So I updated that script by providing my functionnality inside a *Pravatar* object attached to the `window` by the script.
But it didn't worked well anyway, moreover it seemed quite ugly to me to mix up my script with angular and brutalize its logical process with my functions.

In conclusion, the snippet wasn't designed for that kind of project.
Even if I would have been able to make it work with Angular, I'm still not sure about how good it would have been.

The Rails Gem

My failure with the snippet was implying that I could not use the gem, because it was designed to work alongside it !

Anyway, even if the snippet would have been totally fine, the gem would have been constituing a source of issues.

The Pravatar gem has `JWT gem` as a dependency.
Trying to `bundle install` Pravatar gem on Scale led to a monumental amount of **sub-dependecies versions conflicts**, solving these conflicts by upgrading these dependencies could have made Scale totally unstable for whichever gem version update.

Pravatar API

This was the only option left to make Pravatar works on Scale, and probably the **best**.

If I managed to build a script using this API to run the service, the thing was totally achievable by using that API with AngularJS despite my lack of experience and my brand-new newbie knowledge about it.

Implementation

The only parts of Scale that I had to focus onto were the avatars.

As the avatars were only displayed (no update/upload options), I rapidly found out that all of the avatar displays on Scale were achieved by the same `directive`.

In order to use my Pravatar API, I had to build a `PravatarProvider` which would be initialized with the *apiKey*, the *projectName* and the *pravatarUrl*.
This provider would then expose two functions, a `load` one for loading the avatar from a profile id, and an `upload` one for uploading a new avatar with base64 image data and the profile's id.

As the display zone for avatars was perfectly designed, the result of a fetched Pravatar was marvelously fitting in it.
As the upload function wasn't implemented before, it took me a bit of time to design something intuitive for the users.
But with the advice of Simon, and re-using some *loading-effect* element of Scale, I managed to achieve a stylish upload feature on Scale.

Like usual, it seems really simple now, but it took me some time to figure out how Angular was working and become confident with it.

Conclusion

These fifteen weeks were really awesome, so much new experience and stuff to learn, some beers with the Belighters, being happy to go there every day !

I really hope that the Pravatar stuff will be useful to Belighted in some ways and that I achieved good work for them, as they gave me a lot in counterpart.

That experience confirmed my will to pursue in the web development area, and I feel really lucky to have made that internship in such a good company wich such nice people.

Thank you again Belighted !

Pravatar : Developper's Guide

Création d'un compte utilisateur

Pour pouvoir utiliser Pravatar sur une autre application, il est nécéssaire de posséder :

- Un projet sur Pravatar
- La clé `API` associée à ce projet

La création de projet passe par la création d'un compte utilisateur sur le site web de Pravatar.
Une fois un compte utilisateur créé, la section `My Account` permet la création de multiple projets et de la clé **API** associée.

Regénération de la clé API

En cas de clé API compromise, il est possible de générer une nouvelle clé API pour un projet donné dans cette section **My Account**.

Avatar par défaut

Un avatar par défaut peut-être défini pour un projet donné via l'option `Edit Project`, cet avatar par défaut sera utilisé par le service pour tout profil ne possédant pas d'avatar.

Implémentation de Pravatar

Via la gemme et le snippet

Des applications web simples permettent l'utilisation du snippet javascript, et une application Rails peut inclure la gemme pour encore plus de facilité.

L'inclusion du snippet via le placement d'un tag `<script src="http://pravatar.url/pravatar.js"></script>` dans les headers html de l'application.

La gemme nécéssite le placement de la clé API du projet et le nom du projet sous les constantes *pravatar_key* et *pravatar_project* dans `secrets.yml` . Elle permet ensuite l'utilisation de deux helpers pour générer les zones d'affichage et d'upload des avatars :

- `pravatar_display(external_id, options = {})` : Permet la définition d'une zone d'affichage via l'id utilisateur et permet le placement d'options pour ajouter des attributs au tag

`<img>` généré.
- `pravatar_upload(external_id)` : Permet la définition d'une zone d'upload via l'id d'un utilisateur.

Via l'API

Des solutions utiliant des frameworks tels qu'Angular ne sont pas adaptées pour l'utilisation de la gemme et du snippet javascript.

Néamoins l'utilisation de l'API permet d'utiliser assez simplement le service de Pravatar sur l'application.

Display

L'affichage d'un avatar se fait sur l'entrée **GET** `/api/project/:name/profile/:id` de Pravatar.

Celle-ci requiert uniquement le nom du projet et l'id du profil dans l'url, et renvoie la réponse suivante :

```
data = {
        'type' => avatar_type (profile or project_default),
        'attributes' => {
            'avatars' => {
                'thumbnail' =>  thumbnail_avatar_link,
                'normal' => normal_avatar_link,
                'large' => large_avatar_link,
                'original' => avatar_link,
            }
        }
    }
```

Upload

L'upload d'un avatar se fait sur l'entrée **POST** `/api/project/:name/profile/:id` de Pravatar

Cet appel requiert un *header* http `Authorization` conçu avec la libraire `JWT`, formé des éléments suivants :

- **payload** : JSON object forme d'un `external_id` qui est l'id du profil ciblé, un `project_name` qui est le nom du projet utilisant Pravatar et d'un `datetime` formé avec le Time.now() du serveur de l'application.
- **key** : L'API key du projet en question.
- **algorithm** : HS256

Les données d'images qui doivent être envoyées en paramètres d'une requête d'`update` suivent la spécification JSON:API et se présentent sous la forment suivante :

```
'data' => {
              'type' => 'profile',
              'attributes' => {
                  'avatar' => {
                      'data' => image_b64_data,
                      'name' => image_name
                  }
              }
          }
```

Suite à un appel réussi sur cette entrée, la réponse émise par le serveur est typiquement celle reçue par un `show`, ce qui permet de mettre à jour l'affichage du profil immédiatement après upload en utilisant cette réponse :

```
data = {
        'type' => avatar_type (profile or project_default),
        'attributes' => {
            'avatars' => {
                'thumbnail' =>  thumbnail_avatar_link,
                'normal' => normal_avatar_link,
                'large' => large_avatar_link,
                'original' => avatar_link,
            }
        }
    }
```

Développement sur Pravatar

Snippet Javascript

Le snippet client est localisé dans le directory `client`.

Les technologies principales utilisées pour ce script sont :

- Browserify: pour la décomposition du script en modules et l'intégration isolée de jQuery.
- Karma/Jasmine pour le testing dont la configuration se trouve dans le `karma.conf.js`
- Gulp pour l'automatisation de tâches. Le fichier de config est `gulpfile.js`

Sources

Les sources du snippet se trouvent dans le dossier `src/pravatar/coffee`, ces fichiers coffeescript sont séparés en modules intégrés via Browserify.

Les tests sont également écrits en coffeescript et intègrent des modules de Pravatar pour certains éléments

de tests spécifiques, ils couvrent les fonctionnalités d'upload et de download des avatars. Ils se trouvent dans le dossier `src/test`.

Compilation et testing

La compilation et le bundling Browserify sont simplifiés via des commandes Gulp présentes dans le `gulpfile.js` :

- `make_snip` : Compile les sources coffeescript du snippet et lance Browersify sur le résultat Javascript, l'output final est ensuite renvoyé dans le dossier public sous le nom `pravatar.js`.
- `make_test` : Compile les sources coffeescript des tests pour pouvoir les faire tourner par Karma/Jasmine.
- `watch` : Lance une surveillance de l'ensemble des sources coffee pour lancer une compilation/bundling sur chaque changement.
- `test` : Lance la suite de test Karma une fois
- `tdd` : Lance la suite de tests à chaque changements des sources.

Backend Rails

Le backend Rails comporte trois facettes différentes.

User Interface

L'interface utilisateur est celle qui permet aux clients de s'enregistrer et de créer des projets afin d'implémenter le service Pravatar.

Elle est conçue comme une application Rails classique, un `interacteur` a été ajouté en plus sur le MVC.

Administration

Le panel d'administration est conçu avec `ActiveAdmin`, seul un utilisateur avec l'attribut `admin` à *TRUE* peut accéder à l'administration via `/admin`.

Cet accès est protégé par la méthode d'authentification `authenticate_user_admin!` présente dans l'`application_controller` et spécifié dans l'initialize d'ActiveAdmin.

API

L'API est conçue avec la gemme `api-versions` et possède deux méthodes liées aux deux actions possibles sur un avatar :

- **Show** pour récupérer un avatar sur base d'un id unique propre à la gestion utilisateur du client.
- **Update** pour mettre à jour l'avatar sur base de l'id unique et de données b64 de l'image souhaitée.

L'update sur un profil non-existant conduit à sa création en base de données.

`ActiveModel Serializer` a été inclus dans le projet pour la création de réponses conformes aux spécifications JSON:API.

Les tests API situés dans `spec/requests` contiennent des exemples précis des structures de données utilisées pour la communication avec l'API.