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 !