Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE CHARGÉ
DE L'EMPLOI

DOSSIER-PROJET

Nom de naissance ‣ MICHALLON
Prénom ‣ Lisa
Adresse ‣ 17 rue du Dauphiné,
38000 GRENOBLE

Titre professionnel visé

Conceptrice développeuse d'applications

Sommaire

REMERCIEMENTS

Tout d'abord, un grand merci à ma chargée de promotion lors de ma formation en développement web, Marie di Tommaso, sans qui je n'aurais pu entrer en contact avec les équipes de Simplon.Prod.

Merci également à Tom Lann, directeur de Simplon.Prod au moment de ma candidature, pour m'avoir donné ma chance.

Je remercie aussi chaleureusement mes collègues, avec une mention spéciale pour les membres de l'antenne lyonnaise de l'entreprise ainsi que l'équipe du projet Simplonline. Pour la place qu'ils et elles ont su me faire, pour leur bienveillance, leurs encouragements... Je n'aurais pu rêver meilleur cadre pour faire mes premiers pas dans le monde parfois intimidant du développement web.

Et enfin un merci tout particulier pour mon tuteur, Clément Causer, dont l'accompagnement et les conseils auront été précieux tout au long de cette année.

LE PROJET

SIMPLONLINE

RÉSUMÉ DU PROJET EN ANGLAIS

Simplonline was born out of a wish from Simplon.co to go digital in order to improve their trainers' and learners' pedagogical experience while in their Factory. The platform aims to provide them with appropriate tools during their training programme. Accordingly, it is designed for encouraging learners‘ upskilling in relation to active learning strategies that are proper to the Simplon experience. All in all it allows keeping track of the progress made by learners and thus ensures a better monitoring of their apprenticeship.

The platform has already been adopted by over 2300 users in 14 countries and is available in 3 languages : French, Spanish and English. It is developed following agile methodologies, which means different trades collaborate closely in order to ensure product quality.

The current version of Simplonline is the third one and was released in March 2020. The application’s user interfaces are developed using the React framework Next.js and the JavaScript superset TypeScript. The REST API on the other hand relies on Symfony and its framework API Platform with a MySQL relational database.

LISTE DES COMPÉTENCES DU RÉFÉRENTIEL

Activité type 1 Concevoir et développer des composants d'interface utilisateur

Compétences :

1. Maquetter une application
2. Développer une interface utilisateur de type desktop
3. Développer des composants d'accès aux données
4. Développer la partie front-end d'une interface utilisateur web
5. Développer la partie back-end d'une interface utilisateur web

Activité type 2 Concevoir et développer la persistance des données

Compétences :

1. Concevoir une base de données
2. Mettre en place une base de données
3. Développer des composants dans le langage d'une base de données

Activité type 3 Concevoir et développer une application multicouche répartie

Compétences :

1. Collaborer à la gestion d'un projet informatique et à l'organisation de l'environnement de développement
2. Concevoir une application
3. Développer des composants métier
4. Construire une application organisée en couches
5. Développer une application mobile
6. Préparer et exécuter les plans de tests d'une application

CONTEXTE DU PROJET

L'agence Simplon.Prod

Simplon.Prod est une agence numérique solidaire créée par Simplon.co en avril 2015. Tout comme Simplon.co, elle parie sur le numérique comme levier d'inclusion et de transformation sociale. C'est pourquoi sa mission prioritaire est d'accompagner la digitalisation des structures de l'Économie Sociale et Solidaire (ESS) ainsi que des petites et moyennes entreprises (PME) en leur fournissant les outils numériques qui leur sont adaptés afin de soutenir leurs missions sociales et/ou environnementales.

L'agence revêt également une mission d'insertion des publics fragiles au sein de sa structure. Dans cet objectif, elle compte dans son équipe une quarantaine d'employé-e-s dont la majorité des développeurs et développeuses sont issu-e-s de formations Simplon.co, destinées aux personnes éloignées de l'emploi. Et si l'agence est née à Montreuil, elle est composée aujourd'hui de 4 autres antennes : en Occitanie, en Auvergne Rhône-Alpes, dans les Hauts-de-France et à La Réunion.

La plateforme Simplonline

C'est à l'issue de 9 mois de formation en développement web et web mobile que j'ai eu la chance de rejoindre l'antenne lyonnaise de Simplon.Prod pour une année en contrat d'apprentissage au sein de leur équipe front-end. J'ai ainsi pu participer, pendant 12 mois, au développement de la plateforme pédagogique Simplonline qui est utilisée dans 14 pays et dont le client n'est autre que Simplon.co.

La plateforme est née il y a 8 ans d'une envie d'utiliser le digital pour améliorer l'expérience pédagogique des apprenant-e-s et formateur-ice-s au sein des Fabriques Simplon. Son objectif est donc de leur faire bénéficier d'outils adaptés au cours de leur formation. En ce sens, elle n'a pas vocation à proposer des modules *e-learning* mais plutôt à favoriser la montée en compétences des apprenant-e-s dans le contexte de pédagogie active propre aux formations Simplon. En somme, cela permet de garder une trace de la progression de chacun-e et donc d'assurer un meilleur suivi des apprentissages.

Les cibles premières de Simplonline sont donc les formateur-ice-s et apprenant-e-s des Fabriques Simplon. Au fil des années cependant, la plateforme a su intégrer de nouveaux profils d'utilisateur-ice-s. Par exemple, les responsables de Fabriques bénéficient aujourd'hui d'un compte spécifique leur permettant notamment d'accéder à une interface d'administration. Depuis celle-ci, il leur est possible de créer de nouvelles promotions ainsi que les comptes des formateur-ice-s et apprenant-e-s de leur Fabrique. Plus récemment par ailleurs, des comptes dédiés aux tuteurs et tutrices ont fait leur apparition dans le cadre de formations en alternance, afin de mieux accompagner ces dernières et de créer des espaces pour les entreprises au sein de la plateforme. Ils et elles ont ainsi la possibilité de visualiser succinctement le parcours de leur tutoré-e lors des semaines de cours, notamment grâce à un accès à leurs rendus et à l'état de validation des compétences de leur référentiel.

Aujourd'hui, Simplonline en est à sa troisième version. Le développement de cette dernière a débuté en septembre 2019 pour une mise en production en mars 2020 - soit 6 mois de refonte

de la plateforme. Celle-ci s'est imposée car la deuxième version comportait des problèmes d'expérience utilisateur-ice et de développement trop importants. Suite à sa mise en production, la plateforme a gagné 80% d'adoption !

Ainsi, le projet tel que je le connais avait près de deux années de développement à son actif lorsque j'ai rejoint son équipe. Elle se composait alors de 9 personnes dispersées entre Paris, Lyon et Arras et formant plusieurs sous-équipes :

- **l'équipe produit** compte deux personnes travaillant au sein de Simplon.co. L'une est membre de leur Studio Pédagogique et revêt sur Simplonline les rôles de *Product Owner* et de chef de projet. L'autre est chargée des relations utilisateur-ice-s de la plateforme et a également pour mission de remonter les bugs constatés aux équipes de développement.
- **l'équipe design** compte deux designers-euse-s membres de l'agence 40/60. L'un est axé *expérience utilisateur-ice* tandis que l'autre est axée *interface utilisateur-ice*.
- **l'équipe de développement back-end** compte deux développeurs qui conçoivent la base de données et gèrent la distribution de ces dernières au client via une API.
- **l'équipe de développement front-end** compte trois développeur-se-s qui conçoivent l'interface de l'application et ce en se basant sur les maquettes fournies par l'équipe design.

La gestion du projet

Depuis près d'un an et demi, la conception de la plateforme s'organise selon les méthodes agiles. Développeur-se-s, designer-euse-s, représentant du produit et utilisateur-ice-s finaux dialoguent donc très régulièrement afin d'assurer la réussite et la qualité du projet.

Pourquoi les méthodes agiles de développement ?

Le *Manifeste pour le développement Agile de logiciels* a été publié en 2001 - soit il y a plus de 20 ans maintenant. Élaboré par un groupe de 17 développeurs ayant pour objectif de "*mieux développer des logiciels par la pratique et en aidant les autres à le faire*", il prend la forme de 4 valeurs et de 12 principes constituant donc l'agilité.

> ***Les individus et leurs interactions*** *plus que les processus et les outils.*
> ***Des logiciels opérationnels*** *plus qu'une documentation exhaustive.*
> ***La collaboration avec les clients*** *plus que la négociation contractuelle.*
> ***L'adaptation au changement*** *plus que le suivi d'un plan.*

Ce sont les valeurs fondamentales de l'agilité. Si les auteurs du manifeste reconnaissent tout de même la valeur des seconds éléments, ils privilégient cependant les premiers.

Je reviendrai par ailleurs sur quelques-uns des 12 principes sous-jacents du manifeste lorsque je détaillerai la forme qu'ils prennent au sein du projet Simplonline puisque de nombreux procédés mis en place pour s'assurer du bon développement de la plateforme s'inscrivent totalement dans leur continuité.

Les cérémonies *ou rituels* de Simplonline

Qui dit méthodes Agiles dit “livrer fréquemment un logiciel opérationnel avec des cycles de quelques semaines à quelques mois et une préférence pour les plus courts” (principe sous-jacent n°3 du manifeste). On parlera donc de *sprint* et non de *deadline* : de ce fait le développement du projet est découpé en périodes de 2 semaines rythmées par des rituels. Sur Simplonline, on en compte neuf :

- la planification du *sprint*
- le découpage en tâches
- la mêlée quotidienne
- l’atelier UI kit
- le *grooming*
- la conception
- le lundi de résolution de bugs
- la revue de *sprint*
- la rétrospective du *sprint*

Les membres de l’équipe étant dispersé-e-s entre Paris, Lyon et Arras, ces rituels se déroulent tous en visioconférence. Par ailleurs, pour certains rituels, nous nous aidons d’outils en ligne comme support afin de faciliter la communication. Je reviendrai plus en détail sur ces derniers dans les parties consacrées à chaque rituel.

Il reste cependant un outil central à toutes les étapes d’un *sprint* : le gestionnaire de projet Jira. Utilisé quotidiennement par tou-te-s les membres de l’équipe, il présente l’avantage de réunir des fonctionnalités à la fois spécifiques aux développeur-euse-s et au produit sur un même logiciel. Cela permet une meilleure communication et coordination entre les différent-e-s acteur-ice-s du projet !

La planification du *sprint*

Commençons donc par le rituel qui entame un *sprint* : sa **planification** ! Sur Simplonline, il prend aussi la forme d'un *poker planning*. Ainsi, chaque premier lundi de *sprint* se réunissent *Product Owner*, développeur-euse-s et *Scrum Master* pour s'engager sur les tâches à réaliser au cours des deux prochaines semaines en gardant toujours en tête que "les processus Agiles encouragent un rythme de développement soutenable" (principe sous-jacent n°8 du manifeste).

L'équipe commence par faire un état des lieux : avons-nous laissé des tâches en cours à l'issue du précédent *sprint* ? ou y a-t-il des bugs urgents à corriger ? Puis le *Product Owner* nous présente un à un les récits utilisateur-ice-s classés par ordre de priorité dans le *backlog* du projet et les développeur-se-s estiment l'effort de développement que représente chaque fonctionnalité.

Pour ce faire, nous utilisons le site *planitpoker* qui nous permet de faire chacun-e nos estimations sans avoir connaissance de celles des collègues afin de ne pas nous influencer. Pour estimer un récit utilisateur, nous lui attribuons un nombre de points selon la suite de Fibonacci : si celui-ci excède 8 nous considérons que le ticket est trop conséquent et qu'il doit être redécoupé. Chacun-e peut donc voter ou bien passer son tour si iel ne se sent pas en capacité d'estimer. Si une fois dévoilées les estimations présentent des écarts trop importants, l'équipe débat pour laisser chacun-e justifier ses choix et arriver à un consensus.

À l'issue de cette cérémonie, le backlog du *sprint* est constitué et son objectif est établi : non seulement en termes de nombres de points mais également en termes d'axes d'améliorations remontés par les membres de l'équipe lors de la rétrospective précédente.

Le découpage en tâches

Vient alors le moment du **découpage en tâches** qui ne réunit cette fois que l'équipe de développement puisque "les meilleures architectures, spécifications et conceptions émergent d'équipes auto-organisées" (principe sous-jacent n°11 du manifeste).

En amont de cette cérémonie, l'équipe chargée du produit définit des chantiers qu'on appelle **épics**. Ce sont des ensembles de récits utilisateur-ice-s réunis de sorte à former des lots de fonctionnalités livrables au bout de plusieurs sprints de développement – dans le but de livrer continuellement de la valeur au client. Ils sont aussi une excellente façon d'organiser le travail en arborescence et de parvenir à le découper tout en travaillant vers un objectif plus grand.

Pour le découpage en tâche, nous prenons pour base les récits utilisateur-ice-s sur lesquels nous nous sommes engagés puis nous commençons par définir ensemble la façon dont on envisage son implémentation. Cela nous permet de déterminer les étapes nécessaires au développement de la fonctionnalité – l'objectif étant de les transformer en petites unités qui deviendront des tâches voire des sous-tâches indépendantes les unes des autres (dans la mesure du possible) et ordonnées chronologiquement. L'idée est que "la simplicité – c'est-à-dire l'art de minimiser la quantité de travail inutile – est essentielle" (principe sous-jacent n°10 du manifeste) et un petit ticket bien défini facilite sa réalisation en évitant les aller-retours entre les équipes et les oublis ou incompréhensions qui mènent parfois à des livrables non conformes.

La mêlée quotidienne

Comme le suggère le principe sous-jacent n°4 du manifeste : "les utilisateurs ou leurs représentants et les développeurs doivent travailler ensemble quotidiennement tout au long du projet." Nous consacrons ainsi 15 minutes chaque matin à un échange qui réunit tou-te-s les membres de l'équipe et pendant lequel chacun-e explique tour à tour ce sur quoi iel a travaillé la veille, les difficultés rencontrées et ce sur quoi iel compte travailler aujourd'hui.

L'atelier UI kit

Le premier mardi d'un *sprint* les designer-euse-s et développeur-se-s se réunissent pour un atelier **UI kit** d'une heure. C'est l'occasion pour ces deux équipes de se synchroniser : on passe en revue les mises à jour du thème de l'application, de ses composants, nouveautés, éléments dépréciés… L'équipe de développement peut également poser des questions ou demander des

précisions quant à ce qu'ils sont en train d'implémenter. L'équipe design étant un prestataire qui travaille sur d'autres projets que *Simplonline*, cette cérémonie est particulièrement utile.

Le *grooming*

Tous les mercredi, le *Product Owner*, l'équipe design et les équipes de développement se réunissent pour le **grooming**. Il s'agit pour les équipes de développement du commencement du cycle de vie d'une fonctionnalité ! Tout le monde participe donc activement à la cérémonie en partageant leurs avis ou recommandations, en soulevant certaines contraintes ou potentiels oublis… C'est un moment d'échange où chacun-e fait en sorte d'anticiper les problèmes et affine ainsi les récits utilisateur-ice-s qui devront être implémentés lors de prochains *sprints*.

La conception

Le point **conception** a lieu un jeudi sur deux : relativement similaire au **grooming** il se situe cependant plus loin dans le cycle de vie d'une fonctionnalité. Lorsque celle-ci est bien définie (ou *groomée*) elle peut ensuite passer au stade de conception où l'objectif est qu'elle en ressorte prête à être développée. L'approche y est donc plus technique et précise car "une attention continue à l'excellence technique et à une bonne conception renforce l'Agilité" (principe sous-jacent n°9 du manifeste). L'équipe design présente ainsi ses premières maquettes aux équipes de développement qui lui font des retours et posent un maximum de questions afin d'assurer le bon développement de la fonctionnalité une fois le moment venu.

Le lundi de résolution de *bugs*

Sur Simplonline, un lundi sur deux est entièrement consacré à la résolution de *bugs* remontés par les utilisateur-ice-s à la personne en charge du support de la plateforme. Celle-ci profite donc de la mêlée quotidienne pour présenter aux développeur-euse-s les tickets de la journée classés par ordre de priorité. C'est une façon pour nous d'accueillir "positivement les changements de besoins, même tard dans le projet. Les processus Agiles exploitent le changement pour donner un avantage compétitif au client" (principe sous-jacent n°2 du manifeste).

La revue de sprint

Le dernier jour d'un *sprint* nous en faisons la **revue** : c'est l'occasion de présenter au client ce qui a été accompli durant les deux semaines écoulées car "notre plus haute priorité est de satisfaire le client en livrant rapidement et régulièrement des fonctionnalités à grande valeur ajoutée" (principe sous-jacent n°1 du manifeste). Le *Product Owner* commence ainsi par rappeler les objectifs du sprint puis les développeur-euse-s font uniquement la démonstration des récits utilisateur-ice-s **complètement** terminés puisque "un logiciel opérationnel est la principale mesure d'avancement" (principe sous-jacent n°7 du manifeste). Une sensibilisation du client en amont est donc nécessaire pour éviter un effet déceptif… Vient ensuite le temps des questions, puis de la conclusion du *Product Owner* qui donne un aperçu de l'objectif ou thème du *sprint* à venir.

La rétrospective du sprint

Enfin, pour clore un *sprint* nous prenons le temps d'en faire la **rétrospective** : "à intervalles réguliers, l'équipe réfléchit aux moyens de devenir plus efficace, puis règle et modifie son comportement en conséquence" (principe sous-jacent n°12 du manifeste). C'est donc un moment d'échange entre les membres de l'équipe et qui est animé par le *Scrum Master* (un rôle tournant sur Simplonline puisque l'équipe est assez mature).

Il commence généralement par une météo qui laisse l'occasion à chacun-e de donner son ressenti sur le *sprint* écoulé. Il existe ensuite de nombreuses méthodes pour recueillir les retours de l'équipe mais en ce qui nous concerne, pour la majorité des *sprints* nous utilisons un tableau fait de 4 colonnes : ce que j'ai aimé / ce qui m'a manqué / ce que j'ai appris / ce qu'il faut continuer. Chacun-e ajoute donc des post-it dans les différentes colonnes, puis nous en discutons ensemble : cela permet de mettre en avant le positif et ce qui a bien fonctionné tout en ayant l'espace de faire remonter d'éventuels problèmes ou obstacles dans un cadre dédié et bienveillant. Cela débouche sur un vote pour déterminer deux axes d'amélioration pour le prochain *sprint* parmi les post-it ajoutés en définissant concrètement les choses à mettre en

place. Le *sprint* est alors terminé !

Les bénéfices de l'Agilité

Les circonstances ont fait que l'équipe du projet s'est peu à peu éloignée de l'Agilité au cours de mes dernières semaines de développement – et cela m'a permis de prendre conscience de tous les bienfaits de celle-ci !

La plus grosse différence s'est fait ressentir au niveau de la communication entre les différents corps de métiers : les incompréhensions se sont tout simplement démultipliées, dans une équipe qui ne rencontre habituellement pas ou peu ce genre de problèmes.

Deux lots complets de fonctionnalités ont été conçus par le *Product Owner* et l'équipe design sans consulter les équipes de développement (ou trop tard dans le processus). Lorsqu'est arrivé le moment de présenter le travail accompli aux développeur-se-s afin de recueillir leurs retours, ces derniers n'ont pratiquement pas été entendus…

Le *Product Owner*, par expérience et souci de bien faire, avait pour autant essayé d'identifier les potentielles sources de complexité technique lors de l'élaboration des maquettes. Cela a d'ailleurs orienté des décisions d'expérience et d'interface utilisateur-ice. Il s'est cependant avéré que les sources de complexité qu'il avait identifiées n'en étaient pas réellement du point de vue des développeur-euse-s, tandis que d'autres obstacles n'avaient pas été considérés !

Par ailleurs, une autre différence majeure marquée par ce détachement de l'Agilité fut la prise de risque que représentait le développement de ces deux lots de fonctionnalités. Plutôt que de raisonner en termes de développement et d'intégration continue, nous nous sommes mis à penser *deadline* : et si tout n'était pas terminé à la date limite, alors rien de ce qui représentait plusieurs mois de développement ne pouvait être mis en production.

En résumé, deux choses primordiales nous étaient permises habituellement par les méthodes

agiles :

- **une meilleure collaboration entre les équipes** car celles-ci ne travaillent pas qu'avec leur pôle métier : des échanges réguliers permettent à chacun-e d'apporter son point de vue et ses connaissances, ce qui évite les incompréhensions.
- **une disponibilité rapide des nouvelles fonctionnalités** et donc une prise de risque moindre pour le client grâce à une logique d'itération plutôt que de *deadline*.

TECHNOLOGIES

La partie front-end de l'application est développée avec Next.js et TypeScript (qui sont respectivement un framework de la librairie React et un sur-ensemble JavaScript permettant son typage). Elle consomme une API REST développée en PHP avec Symfony et son framework API Platform , ainsi qu'une base de données relationnelle MySQL.

L'équipe de développement front-end a fait le choix d'utiliser Next.js au commencement du projet bien que sa fonctionnalité phare ne soit pas exploitée sur Simplonline : le rendu côté serveur requiert la connaissance de l'intégralité des chemins de l'application et se prête donc difficilement à la plateforme qui regorge de chemins spécifiques aux utilisateur-ice-s. De la même façon, devoir récupérer les données en un seul *render* amène de la complexité voire du code dupliqué lorsque l'on utilise des librairies comme Redux ou React Query (que nous utilisons sur la plateforme bien que depuis la dépréciation de Redux nous travaillons au remplacement des dernières briques qui reposent sur la librairie).

Comme mentionné plus haut, nous développons également avec Typescript : un langage de programmation orientée objet *open source* créé et maintenu par les équipes de Microsoft. Il s'agit d'un sur-ensemble de JavaScript : le code TypeScript est transcompilé en JavaScript afin d'être lisible par les navigateurs. Son intégration dans les IDE (et notamment dans Visual Studio Code) permet aux développeur-se-s de profiter d'un environnement sécurisé puisque le typage du code évite de nombreux bugs avant même leur transcompilation vers le JavaScript. Par ailleurs, il permet dans une certaine mesure de documenter son code et de le rendre plus assimilable par une personne qui n'en est pas l'auteur-ice. Je trouve aussi qu'il représente un certain avantage d'un point de vue maintenabilité car il aide grandement au remaniement de code JavaScript grâce aux erreurs que le langage remonte !

La plateforme compte de nombreux formulaires : nous les développons aujourd'hui avec React Hook Form (un dernier formulaire utilise cependant encore Redux Form) couplé avec la librairie Yup afin de gérer la validation des champs. En plus de faciliter grandement la gestion des

événements et états d'un formulaire en ne nécessitant que peu de ligne de code, React Hook Form présente également de meilleures performances que les librairies concurrentes en limitant les *re-renders*.

L'application front-end étant complètement indépendante de sa partie back-end, nous sommes très souvent amené-e-s à faire des requêtes à son interface de programmation. Pour cela, nous utilisons la librairie React Query. Les hooks qu'elle propose réduisent encore une fois considérablement la quantité de code à produire pour pouvoir récupérer des données et permettent une meilleure gestion de la mise en cache et invalidation de celles-ci !

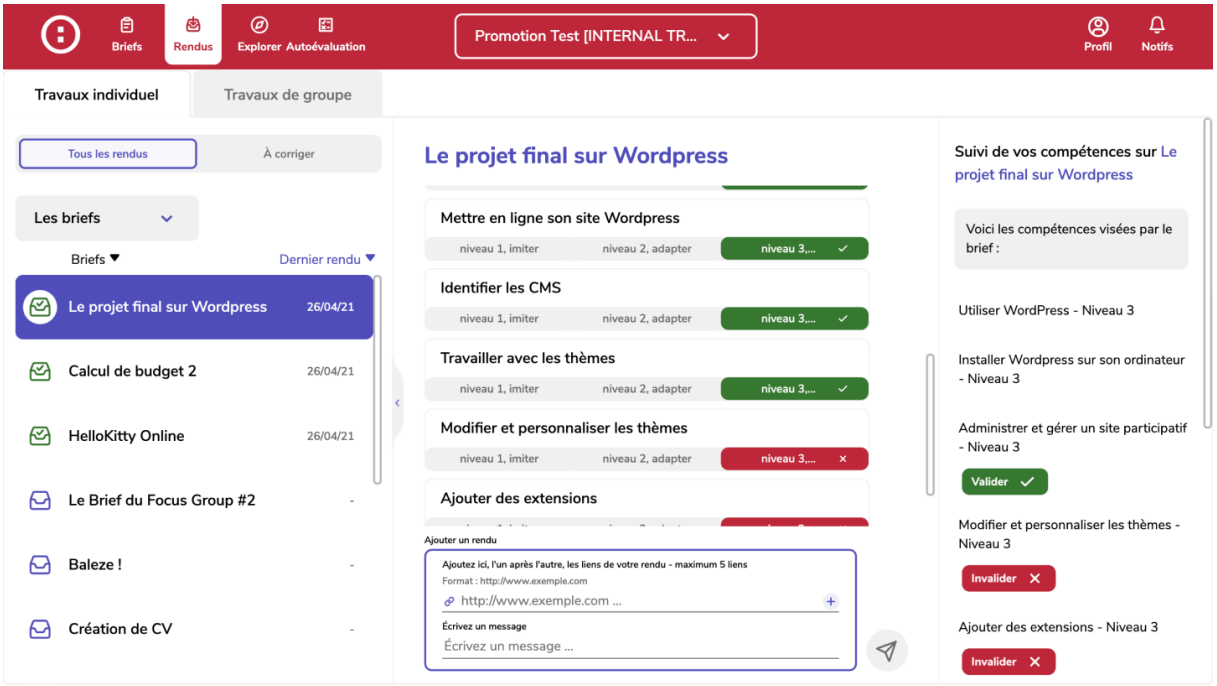
Pour gérer le style de Simplonline nous utilisons styled-components : une framework de style qui utilise les *littéraux de gabarits* (ou *tagged template literals*) JavaScript pour produire du code CSS dans des composants React ! En plus de permettre une gestion du style au niveau du composant (donc dans une feuille .js) le framework présente l'avantage de pouvoir gérer ce style dynamiquement grâce à l'utilisation de props. Par ailleurs, il est aussi possible de l'utiliser pour créer un thème global à l'application.

Enfin, pour écrire et exécuter nos tests automatisés nous utilisons la librairie Cypress : parmi les plus complètes, elle prend en charge les tests de bout en bout mais aussi les tests d'intégration et les tests unitaires !

DÉVELOPPEMENT D'UNE FEATURE

Nouveauté sur la plateforme : les pièces jointes sécurisées

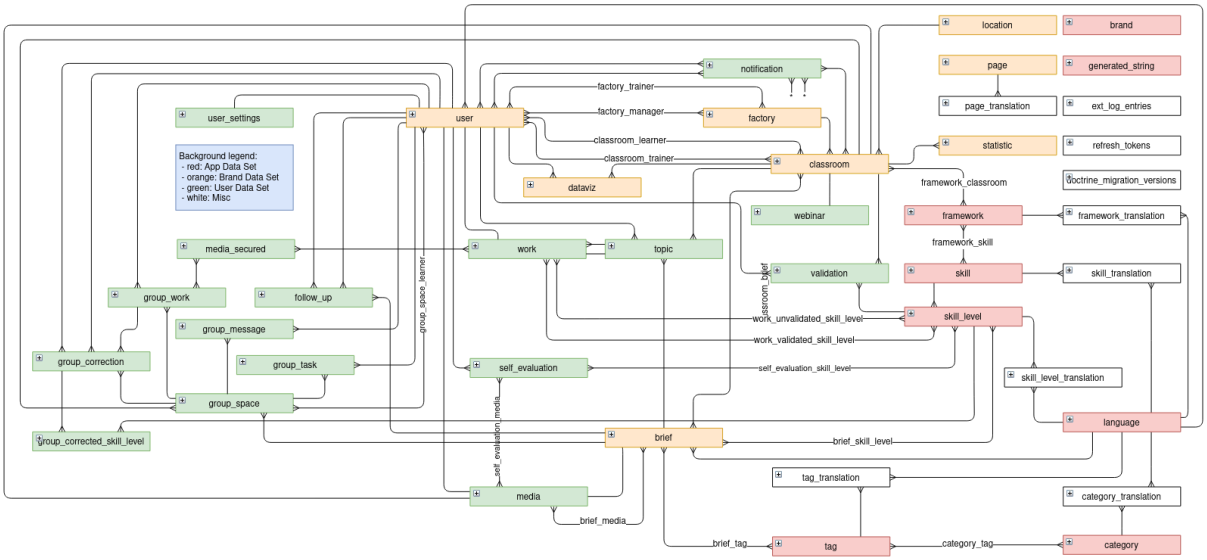
L'équipe de Simplonline a fait le pari d'ouvrir la plateforme à d'autres marques que Simplon quelques mois avant mon arrivée sur le projet ! Dans ce cadre, de nouveaux besoins utilisateur-ice se sont manifestés : bien que la plateforme ne soit utilisée que par des organismes de formation adoptant une approche par validation de compétences, ces dernières ne touchent plus forcément à l'univers du numérique. Un de ces organismes a ainsi proposé au *Product Owner* de permettre aux apprenant-e-s de joindre des documents à leur rendus (et non plus seulement des liens, comme c'était le cas sur l'écran capturé ci-dessous).



Le développement de cette nouveauté nécessitait un travail conséquent pour l'équipe back-end : si la plateforme permettait déjà le téléchargement de pièces jointes pour les formateur-ice-s notamment (dans le cadre de la création d'un brief par exemple) celles-ci n'étaient pas sécurisées. Or la demande du *Product Owner* spécifiait que ces nouvelles pièces-jointes ne

devaient être consultables que par l'apprenant-e et son ou sa formateur-ice.

Il a donc fallu créer une nouvelle table dans la base de données de l'application pour gérer ces pièces jointes *sécurisées* : la table *media_secured* qui est en relation *many-to-many* avec les tables *work* et *group_work*. L'hébergement de ces pièces-jointes a par ailleurs été mis en place avec Amazon Web Services !



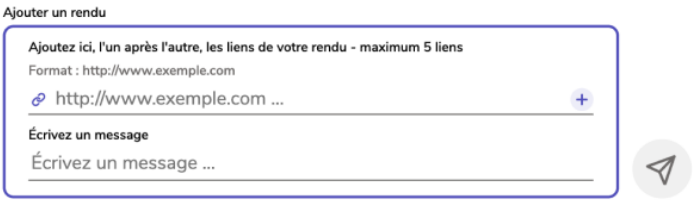
Du côté des interfaces utilisateur-ice, une grosse refonte de l'espace des rendus individuels et en groupe s'est imposée : bien que celle-ci ne soit pas nécessaire au développement de la nouvelle fonctionnalité, ces écrans reposaient sur beaucoup de dette technique dans le code front-end et il était donc temps d'y mettre de l'ordre. Notre fonctionnement Agile nous a donc permis d'ajouter ce *refactoring* à l'ordre du jour.

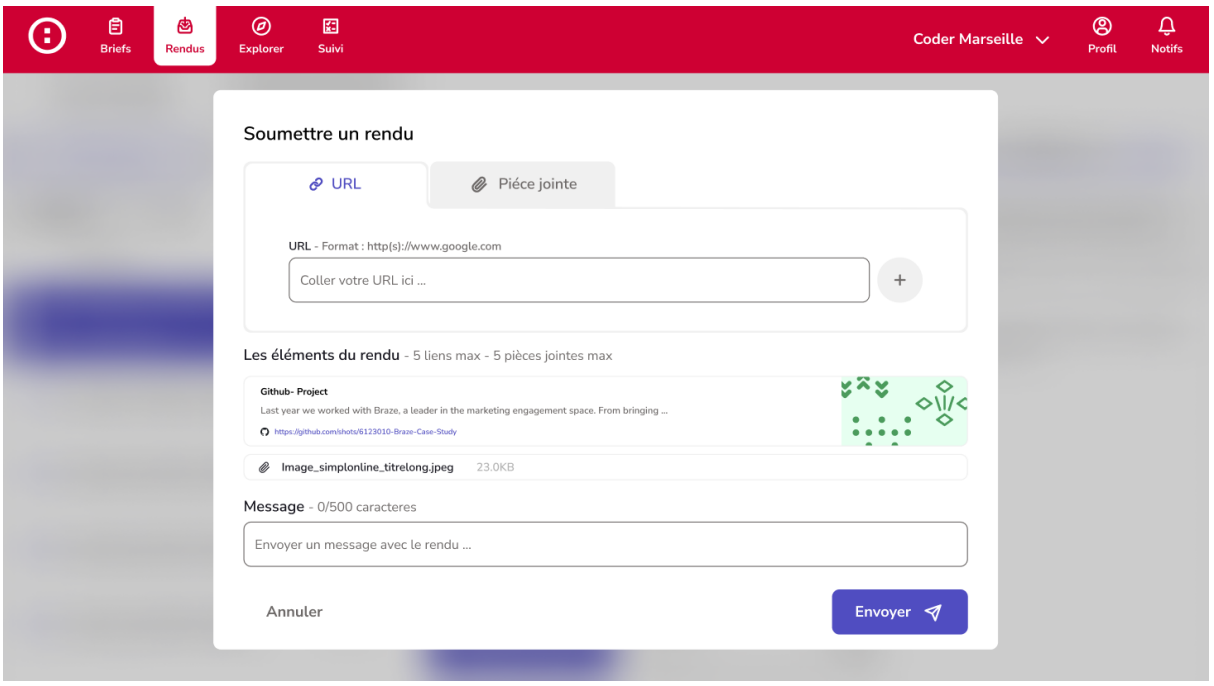
Au vu de la quantité de travail nécessaire autant pour l'API que pour les interfaces utilisateur-ice, nous avons choisi de créer un épic "médias sécurisés" pour rassembler tout le développement relatif à cette fonctionnalité que ce soit sur *Jira* comme sur *git* !

J'ai été la première personne de l'équipe à travailler sur un ticket de cet épic et me suis donc chargée de créer la branche et donc la *merge request* de celui-ci. Le développement de chaque ticket relatif aux médias sécurisés aura donc pour source et destination "epic-media-secured"

afin de faciliter la mise en commun des travaux front-end et back-end, et de ne *merger* sur la branche principale du projet uniquement les fonctionnalités entièrement terminées et testées grâce à un système d'environnements dynamiques mis à disposition de l'équipe produit.

Le ticket sur lequel j'ai travaillé concernait la création d'une nouvelle modale sur la plateforme afin de répondre à la diversification des modalités de rendu pour les apprenant-e-s. Les éléments ci-dessous sont respectivement l'ancien *input* consacré au rendu avec la possibilité de joindre un lien uniquement – le nouvel *input* qui se simplifie afin de créer un nouvel espace consacré aux rendus plus complexes (une modale) et accessible au clic sur le bouton "soumettre un rendu " – et enfin la modale en question / l'objet de mon ticket :





Des nouveaux composants utiles à toute l'application

J'ai entamé le ticket par une phase de conception en me basant sur les maquettes à ma disposition. Cela m'a permis d'identifier "grossièrement" les composants que j'allais devoir utiliser ou développer et les responsabilités de chacun d'entre eux !

Parmi les nouveaux composants nécessaire à l'intégration des maquettes, deux pouvaient avoir être utiles au reste de la plateforme également :

- une nouvelle icône : le trombone qui symbolise les pièces-jointes
- un composant de *textarea* afin d'éditer un message sur plusieurs lignes

J'ai donc commencé par le développement de ces derniers en prenant soin de faire un *commit* correspondant pour chacun d'eux : l'idée étant de garder un historique clair et qui peut grandement nous faciliter la tâche en cas de problème.

Paperclip.tsx

Pour ajouter une nouvelle icône sur la plateforme, nous nous rendons sur le *Figma* du projet où se trouve le *design system*. Une bibliothèque d'icône a été mise en place par l'équipe design : c'est ici que je me rends pour trouver puis exporter en format SVG l'icône dont j'ai besoin.

Je crée ensuite mon composant *Paperclip.tsx* dans le dossier qui répertorie toutes les icônes du projet. En termes d'arborescence, ce dossier d'icônes a pour parent un dossier *modules* dans lequel on retrouve des parties de la logique de l'application ainsi que de son interface. Seuls les composants qui fonctionnent indépendamment du contexte dans lequel ils sont utilisés peuvent être placés dans ce dossier.

Pour revenir à la création de mon icône : j'englobe la balise de mon SVG avec un composant *SvgIcon.tsx* qui contient une configuration minimale que nous utilisons sur toutes nos icônes, et

qui met à disposition des *props* qui facilitent le développement en permettant par exemple d'en gérer la taille et la couleur. En termes d'accessibilité, il évite également sa lecture par les technologies d'assistance !

Textarea.tsx

En ce qui concerne la création du composant de *textarea* j'ai eu recours à la librairie *react-textarea-autosize* qui était déjà utilisée sur l'application. Je n'ai fait qu'augmenter le composant *TextareaAutosize* que la librairie met à disposition en lui ajoutant du style via *styled-components*. Là encore j'ai utilisé des outils déjà mis en place sur le projet : notamment une fonction qui détermine la couleur de l'input selon son état de validation.

Intégration de la modale

Une fois ces composants créés j'ai pu entamer l'intégration de la modale, dans un premier temps sans me soucier des formulaires d'ajout de liens et de pièces jointes car le développement back-end nécessaire n'était pas encore terminé.

```
const MOCK_LEARNER_LINKS = [
    { url: "https://youtube.com" },
    {
        url: "https://gooooooooooooooooooooooooooooooooooooooooooooooooooooooooogle.com",
    },
];

const MOCK_LEARNER_ATTACHMENTS = [
    {
        filename: "mock_image.png",
        size: "23.0KB",
    },
];
```

J'ai donc utilisé de fausses données pour simuler ce que m'aurait renvoyé le formulaire ! J'utilise des URL de tailles différentes afin de m'assurer que les plus longues soient tronquées pour ne pas casser le *layout*.

Je commence par utiliser des composants déjà existant sur la plateforme puisque faisant partie du *design system* :

- la modale *Modal.tsx* à laquelle je passe des composants enfant ainsi qu'un booléen à sa *props* "open" afin de déterminer si elle est doit être visible ou non

- le décorateur d'input *InputDecorator.tsx* qui permet d'avoir des *inputs* uniformes sur la plateforme et d'assurer leur accessibilité
- les boutons *Button.tsx* et *GhostButton.tsx* qui sont des *styled-components* avec différentes variantes et options configurables via les *props*
- un ensemble de composants (*Tabs.tsx*, *Tab.tsx*, *TabButton.tsx* et *TabPanel.tsx*) qui s'utilisent en composition afin de créer les deux onglets de formulaire et leur panneau respectif

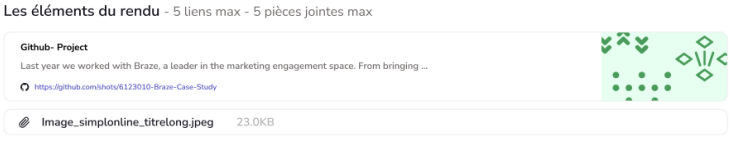
Pour ces derniers, un léger remaniement a cependant était nécessaire pour coller parfaitement aux maquettes : le texte contenu dans les boutons devait être violet et non plus noir comme c'était le cas jusque là.

J'ai donc créé dans le composant *TabButton.tsx* une fonction permettant de déterminer la couleur du texte selon la variante spécifiée.

```tsx
export type TabVariant = "default" | "primary";

type Props = ComponentPropsWithoutRef<typeof TabButton> & {
    tabId: number;
    isSelected?: boolean;
    variant?: TabVariant;
};

export const getVariant = (theme: Theme, variant: TabVariant) => {
    switch (variant) {
        case "primary":
            return {
                textColor: theme.palette.grey.base,
                textColorSelected: theme.palette.purple.base,
            };
        default:
            return {
                textColor: theme.palette.grey.base,
                textColorSelected: theme.palette.black.base,
            };
    }
};
```

Il m'est ensuite possible grâce à *styled-components* de déstructurer les valeurs renvoyées par cette fonction et de les exploiter dans le CSS-in-JS !

En dehors des tous ces composants déjà existant sur l'application et des deux que j'avais déjà développés sur cette *merge request* j'ai développé un composant *WorkElement.tsx* permettant l'affichage d'un élément d'un rendu :



Il contient la logique permettant un affichage différent de l'élément selon que ce soit un lien ou une pièce jointe – il reçoit également des *props* différentes selon ces mêmes modalités.

Développement des formulaires

Lors de la conception de la modale, plusieurs question se sont posées concernant le rendu d'un brief : il me fallait absolument éviter d'avoir un formulaire imbriqué dans un autre formulaire puisque cela aurait posé problème non seulement en termes de sémantique mais encore plus d'un point de vue accessibilité !

Mais les contraintes posées par le *workflow* de validation des différents champs et les librairies que nous utilisons pour développer les formulaires de l'application nous imposaient d'avoir recours à plusieurs balises *form*.

J'ai donc tranché pour la solution suivante : chaque panneau servant à ajouter un élément au rendu sera un formulaire indépendant – le formulaire en charge du véritable rendu ne contiendra que l'*input* de message de l'apprenant-e et la liste des éléments du rendu dont les données seront ajoutées programmatiquement au moment de la soumission.

Le panneau d'ajout de lien

Je commence par créer le composant *AddLinkForm* qui consiste donc en un formulaire d'ajout de lien et que l'on utilisera avec le composant *TabPanel* pour parent :

Pas de complexité sur ce bout de code. Il a simplement été nécessaire de créer une règle de validation *personnalisée* car je souhaitais m'assurer qu'un-e utilisateur-ice ne puisse pas ajouter plusieurs fois le même lien. La librairie yup ne disposait pas d'une fonction permettant cette

vérification : j'ai donc utilisé sa fonction *test()* afin d'appliquer ma propre logique.

Cela m'a poussée à créer une fonction *getSchema()* en dehors de la déclaration du composant afin qu'elle ne soit pas recalculée à chaque *render* – cette dernière me retourne le schema de validation. Je lui passe en argument de quoi traduire le message d'erreur que je souhaite renvoyer (afin que la fonction soit pure) ainsi que la totalité des éléments ajoutés au rendu !

```
const getSchema = (t: TFunction, medias: IMediaResource[]) => {
    return yup.object().shape({
        title: yup.string(),
        url: yup
            .string()
            .url()
            .test("unique", t("component.AddLinkForm.uniqueLinkErrorMessage"), value => {
                return !medias.some(media => media.url === value);
            })
            .required(),
    });
};
```

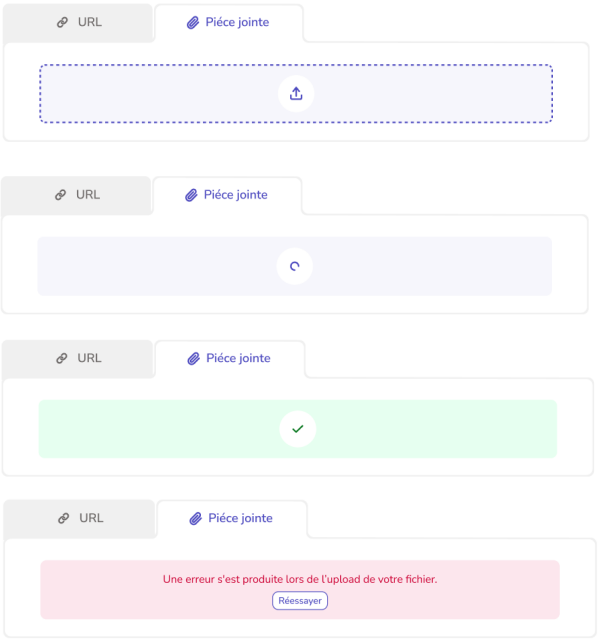
Le panneau d'ajout de pièce jointe

Pour le développement du deuxième panneau (qui concerne l'ajout de pièces jointes) là encore je crée un composant ne contenant qu'un formulaire appelé *AddMediaForm :*

Le dépôt du document pouvant se faire à la fois via un *input* ouvrant l'explorateur de fichier de l'utilisateur-ice ou via glisser-déposer, l'utilisation de la librairie *react-dropzone* s'est imposée. Il s'agit encore une fois d'une librairie déjà utilisée sur la plateforme. Il m'a cependant été nécessaire de la mettre à jour afin de bénéficier d'une nouvelle fonctionnalité dont nous avions besoin : la connaissance de la règle de validation qui n'a pas été remplie en cas d'erreur !

Les données rendues disponibles par la librairie concernant l'état de la zone de glisser-déposer, couplées avec le style dynamique permis par le CSS-in-JS de *styled-components* m'ont par ailleurs permis de gérer leur retranscription sur les interfaces utilisateur-ice sans trop de difficulté :

```
type DropzoneProps = {
    isDragActive?: boolean;
    isLoading?: boolean;
    isSuccess?: boolean;
    isError?: boolean;
};

const Dropzone = styled.div<DropzoneProps>`
    border-radius: ${({ theme }) => theme.shape.borderRadius.medium};
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
    width: 100%;
    height: 105px;

    ${({ theme, isDragActive, isLoading, isSuccess, isError }) => {
        if (isDragActive) {
            return css`
                background-color: ${theme.palette.purple.lightest};
                border: 2px dashed ${theme.palette.purple.base};
            `;
        }

        if (isLoading) {
            return css`
                background-color: ${theme.palette.purple.lightest};
                --icon-bg-color: ${theme.palette.white.base};
                --icon-color: ${theme.palette.purple.base};
            `;
        }

        if (isSuccess) {
            return css`
                background-color: ${theme.palette.green.lightest};
                --icon-bg-color: ${theme.palette.white.base};
                --icon-color: ${theme.palette.green.base};
            `;
        }

        if (isError) {
            return css`
                background-color: ${theme.palette.red.lightest};
            `;
        }
    }};
`;
```



Création du hook useMediaUpload

Le développement back-end de la fonctionnalité étant terminé et *mergé* sur la branche d'épic j'ai pu commencer à mettre en place les outils nécessaires à la création de pièces jointes sécurisées. Il a d'abord fallu créer le hook permettant de requêter l'API pour la création de médias : soit à la fois les liens et les pièces jointes, qu'elles soient sécurisées ou non. Le typage peut par ailleurs paraître assez verbeux car un gros travail a été nécessaire afin d'outrepasser ce qui est habituellement retourné par le hook *useMutation* de *React Query* : j'avais en fait besoin de

retourner l'avancement du téléchargement du média et cela n'était possible qu'en remaniant les données retournées par mon *hook* :

```ts
type UseMediaUploadReturnType<TResult, TError, TVariables, TSnapshot> = [
    MutateFunction<TResult, TError, TVariables, TSnapshot>,
    MutationResult<TResult, TError> & { progress: number },
];

type UnSecuredFileType = { file: File; type: MediaType.FILE };
type UnSecuredUrlType = { url: string; type: MediaType.URL };
type SecuredPayload = { file: File; title?: string; secured: true };
type UnSecuredPayload = Merge<{ title?: string; secured: false }, UnSecuredFileType | UnSecuredUrlType>;

type UploadMediaVariables = {
    payload: SecuredPayload | UnSecuredPayload;
    axiosConfig?: AxiosRequestConfig;
};
```

```ts
const useMediaUpload = <
    TResult = AxiosResponse<IMediaResource | IMediaSecuredResource>,
    TError = any,
    TSnapshot = unknown,
>(
    config: MutateConfig<TResult, TError, UploadMediaVariables>,
): UseMediaUploadReturnType<TResult, TError, UploadMediaVariables, TSnapshot> => {
    const api = useApi();
    const [progress, setProgress] = useState(0);

    const uploadMedia = async ({ payload, axiosConfig }: UploadMediaVariables): Promise<TResult> => {
        const { secured, title } = payload;

        const url = secured ? "/media_secureds" : "/media";

        const media = new FormData();

        media.append("uuid", uuid());
        title && media.append("title", title);

        if (payload.secured) {
            media.append("file", payload.file);
        } else {
            if (payload.type === MediaType.FILE) {
                media.append("file", payload.file);
                media.append("type", MediaType.FILE);
            } else {
                media.append("url", payload.url);
                media.append("type", MediaType.URL);
            }
        }

        return api.post(url, media, {
            ...axiosConfig,
            headers: {
                "Content-Type": "multipart/form-data",
            },
            onUploadProgress: ({ total, loaded }) => setProgress((loaded / total) * 100),
        });
    };

    const mutation = useMutation<TResult, TError, UploadMediaVariables>(uploadMedia, config);

    return [mutation[0], { ...mutation[1], progress }];
};
```

Je l'ai ensuite utilisé dans le composant *AddMediaForm* en l'intégrant au *useDropzone* – le hook que met à disposition la librairie *react-dropzone* :

```
const [uploadAttachmentMutation, { isLoading }] = useMediaUpload({ onSuccess: () => reset() });

const { getRootProps, getInputProps, open, isDragActive } = useDropzone({
    noClick: true,
    maxSize: MAX_FILE_SIZE,
    onDropAccepted: async (acceptedFiles: File[]) => {
        const file = acceptedFiles[0];

        const upload = await uploadAttachmentMutation({
            payload: isMediaSecured
                ? { secured: true, title, file }
                : { type: MediaType.FILE, secured: false, title, file },
        });

        if (upload) {
            onSubmit(upload.data);
            setIsSuccess(true);

            return new Promise(resolve => setTimeout(resolve, 1500)).then(() => setIsSuccess(false));
        } else {
            setErrorMessage(getErrorMessage(t));
        }
    },
    onDrop: (_, rejectedFiles) => {
        if (rejectedFiles.length > 0) {
            // eslint-disable-next-line @typescript-eslint/ban-ts-comment
            /* @ts-ignore */
            const errorMessage = getErrorMessage(t, rejectedFiles[0].errors[0].code);
            setErrorMessage(errorMessage);
        }
    },
});
```

Le composant de modale

Une fois les panneaux fonctionnels il a fallu les intégrer à ma modale de rendu et y mettre en place les dernières requêtes à l'API afin de créer le rendu en base de données.

Le composant *WorkAttachmentModal* contient donc :

- le *mapping* des valeurs du formulaire afin que le *payload* de la requête API contiennent les clés attendues
- les requêtes API qui diffèrent selon que le rendu concerne un brief assigné individuellement ou en groupe : j'utilise ainsi deux *hooks* que j'ai développés dans le cadre de cette nouvelle fonctionnalité – et qui retournent une mutation *React Query*

```tsx
const WorkAttachmentModal = ({ type, workspaceId, onCancel, onSuccess, visible, className }: Props) => {
    const [t] = useTranslation();

    const [createWorkMutation, { error: workError, reset: workReset }] = useCreateWork();
    const [createGroupWorkMutation, { error: groupWorkError, reset: groupWorkReset }] = useCreateGroupWork();

    const onSubmit = async ({ message, workElements }: SubmitWorkFormValues) => {
        const mediaSecured = workElements?.filter(workElement => workElement["@type"] === EntityTypes.MEDIA_SECURED);
        const learnerLinks = workElements.filter(
            workElement => workElement["@type"] === EntityTypes.MEDIA && workElement.type === MediaType.URL,
        );

        let promise = null;

        if (type === "individual") {
            promise = createWorkMutation({
                uuid: uuid(),
                topic: `/topics/${workspaceId}`,
                learnerComment: message,
                learnerLinks: learnerLinks?.map(link => ({ url: link.title ?? link.url })),
                mediaSecured: mediaSecured?.map(media => getResourceUuid(media)),
            });
        }

        if (type === "group") {
            promise = createGroupWorkMutation({
                uuid: uuid(),
                comment: message,
                groupSpace: `/group_spaces/${workspaceId}`,
                links: learnerLinks?.map(link => ({ url: link.title ?? link.url })),
                mediaSecured: mediaSecured?.map(media => getResourceUuid(media)),
            });
        }

        const result = await promise;

        onSuccess();

        return result;
    };

    const dismiss = () => {
        const mapTypeToReset: { [key in Props["type"]]: () => void } = {
            individual: workReset,
            group: groupWorkReset,
        };

        return mapTypeToReset[type]();
    };
```

LE PROJET

BRIZE DE PODCAST

RÉSUMÉ DU PROJET EN ANGLAIS

I have always loved watching TV series and as years passed by, the shows I was following started accumulating to the point I could not keep track of their latest episodes on my own. That is why I became a regular user of platforms that provide you with a TV shows calendar and other tracking tools : the most useful feature being that when I watch an episode, I can mark it as such on the application !

Lately however, I have also been listening to more and more podcasts. And just as with my favourite TV shows I began to lack a management tool to keep track of them, even more so since I listen to them on different platforms that are not primarily conceived for podcasts but for songs…

When looking for that kind of application on the internet I couldn't find any. After a while, I thought I should just code it myself ! And that is how I came up with *Brize de podcast*.

The application's user interfaces are developed using the React framework Next.js and the JavaScript superset TypeScript. The REST API on the other hand relies on Symfony and its framework API Platform with a MySQL relational database.

DÉVELOPPEMENT

Environnement de développement

Pour faciliter le développement du projet ainsi que sa maintenance j'ai fait le choix d'utiliser Docker. En plus de permettre une installation rapide du projet, il solutionne les problématiques de compatibilité entre différentes versions ou systèmes d'exploitation et assure une exécution identique de l'application dans tous les environnements.

Docker est un outil de conteneurisation (une forme plus légère de virtualisation) c'est-à-dire qu'il permet de lancer des applications dans des conteneurs logiciels. Ces derniers sont des environnements complètement autonomes (en dehors du noyau de système d'exploitation qui leur est commun) et embarquent tout ce dont une application a besoin pour fonctionner : fichiers sources, environnement d'exécution, librairies…

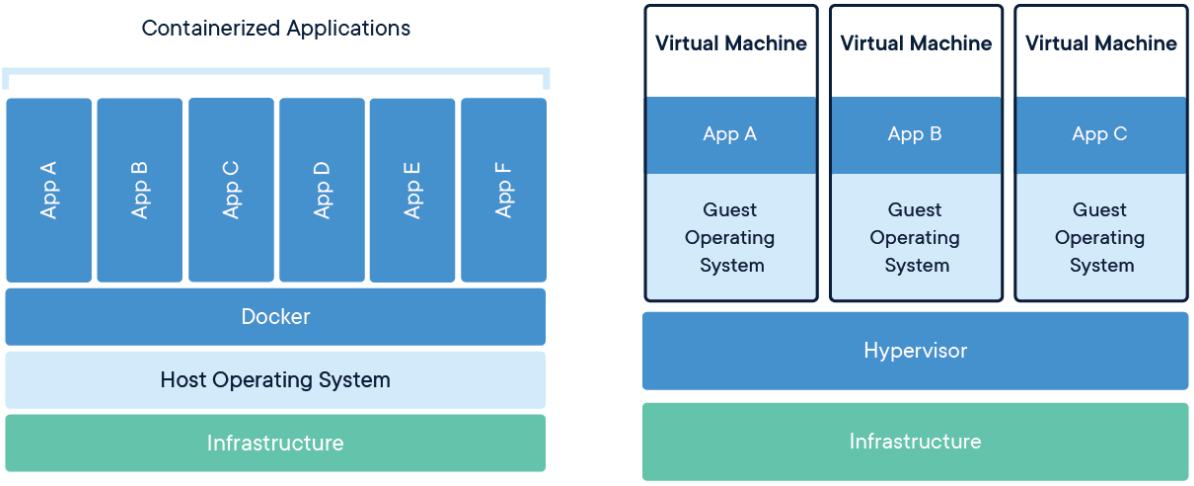
Les conteneurs logiciels existaient bien avant Docker (depuis une dizaine d'années) mais ils étaient complexes à mettre en place puisque très bas niveau : c'était la couche système d'exploitation qui était virtualisée. Ce que propose Docker est donc un outil informatique de haut niveau qui facilite grandement la conteneurisation pour les développeur-euse-s.

Il se distingue véritablement des machines virtuelles par la partie du système d'exploitation qu'il virtualise. Un système d'exploitation se compose de deux couches :

- le *noyau* qui communique avec le matériel (donc le CPU, les mémoires…)
- les *applications* qui se basent et fonctionnent avec le noyau

Une machine virtuelle virtualise ainsi la couche système d'exploitation au complet : en créant donc son propre noyau ! Alors qu'un conteneur Docker ne virtualise que la couche applicative et utilise le noyau de l'hôte (donc l'ordinateur sur lequel il est déployé). Et si cela peut poser des problèmes de compatibilité, ce mode de conteneurisation permet d'obtenir des images bien plus

petites et performantes !



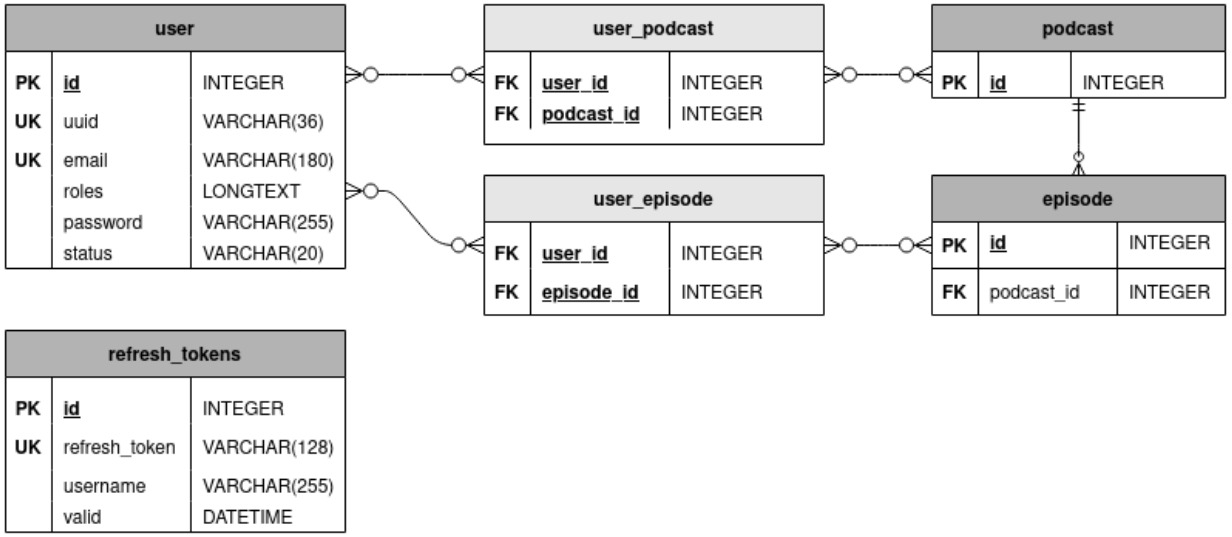
Dans le cadre du projet *Brize de podcast* j'ai créé deux *Dockerfiles* afin de construire mes propres images Node et PHP, respectivement pour les parties front-end et back-end de mon application. Un *Dockerfile* contient les instructions pour créer une image Docker, et les bonnes pratiques invitent à minimiser le poids d'une image car cela facilite leur transfert et leur déploiement lors de la construction des conteneurs.

Pour construire des images plus légères, il est donc essentiel de limiter le nombre de couches requises pour les créer : chaque instruction COPY, ADD ou RUN d'un *Dockerfile* crée une nouvelle couche, et chaque couche garde en mémoire la différence entre la version courante de l'image et la précédente (ce qui peut demander beaucoup de stockage). C'est pour cette raison qu'il est préférable de construire des images "multi-stage" car elles permettent de fusionner plusieurs couches en une !

Toujours dans l'idée de limiter le poids d'une image, il est important de bien choisir celle sur laquelle celle-ci se base. Pour mon image Node j'ai donc fait le choix d'une image basée sur Alpine Linux : une distribution Linux légère et orientée sécurité. Elle est d'ailleurs recommandée dans la documentation Docker pour son poids inférieur à 6 mégabytes et son contrôle strict.

Base de données

Pour concevoir la base de données du projet *Brize de podcast* j'ai utilisé la méthode Merise et notamment un Modèle Conceptuel des Données (MCD). J'ai défini les cardinalités de chacune des tables afin de déterminer le type de relation qui en découlait et donc les tables intermédiaires nécessaires ainsi que les clés étrangères et leur emplacement :



Les données que je peux stocker concernant les `podcasts` et leurs `épisodes` sont limitées par les conditions d'utilisation de `ListenAPI`, une API qui répertorie de nombreux podcasts disponibles sur les plateformes de streaming et que j'utilise sur mon projet ! Je me limite donc à leur identifiant.

Pour assurer la sécurité de ma base de données, j'ai créé un utilisateur *brizedepodcast_admin* spécifique au projet et ne lui ai attribué des droits que sur la base de données *brizedepodcast*.

Toujours pour améliorer la sécurité de mes données, dans chacune des tables j'inclus une colonne `uuid` afin d'identifier les données sans avoir à rendre leur véritable `id` publique : ils ne seront ainsi jamais divulgués en dehors de l'application back-end.

Authentification

J'ai ensuite entamé le développement d'un système d'authentification dans le but d'assurer la sécurité de mon API REST. La gestion de l'authentification dans un projet API Platform est complètement déléguée à Symfony puisque le framework en est une surcouche et repose donc sur ses composants : on utilisera ici Security Core et Security HTTP via le Security Bundle et d'autres bundles communautaires.

J'ai ensuite dû faire plusieurs choix quant aux modalités d'authentification. D'abord en déterminant si je souhaitais avoir recours à des sessions PHP ou bien à des JSON Web Tokens pour identifier les consommateur·trice·s de mon API.

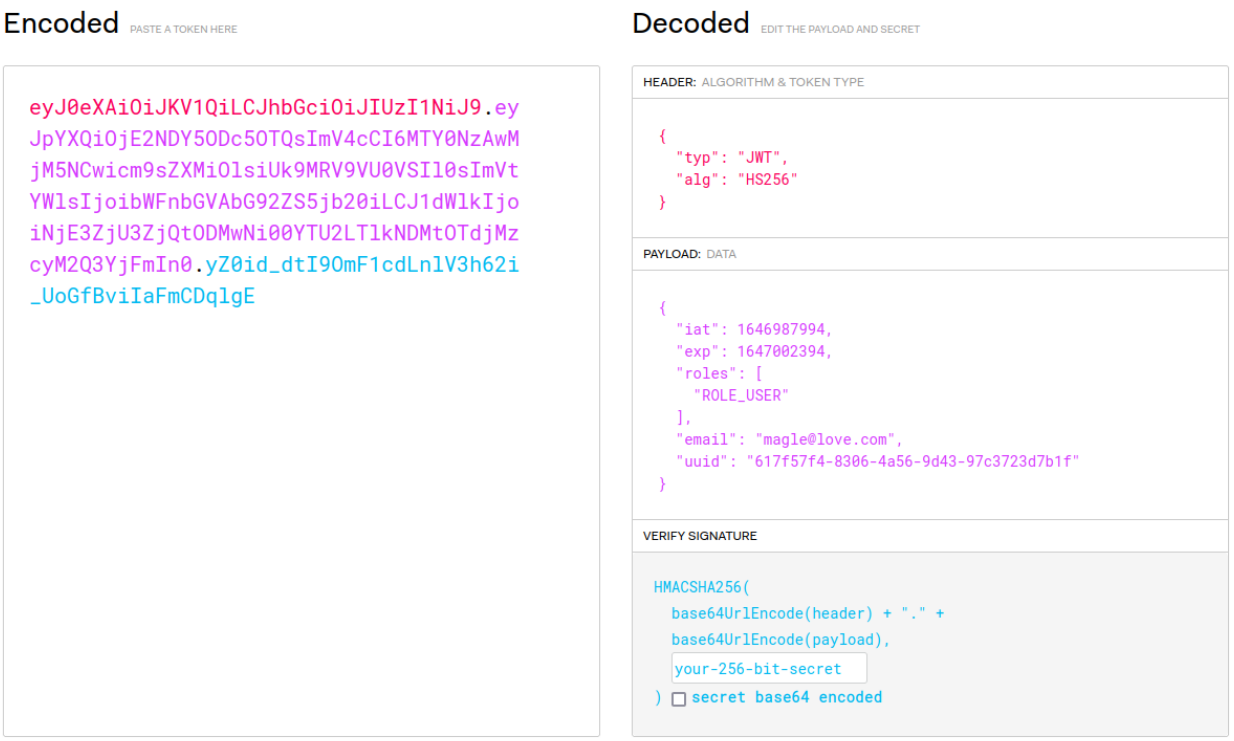
J'ai tranché pour une authentification par JSON Web Tokens car l'utilisation de sessions PHP posait deux problèmes. Le premier était que les sessions ne sont pas RESTful puisqu'elles rendent l'API dépendante d'un contexte qui est stocké sur le serveur. La requête du client au serveur n'est donc plus "auto-suffisante" car elle ne contient pas toutes les informations nécessaires à sa compréhension. Cette nécessité de stocker un état côté serveur pose également une seconde question : elle rend l'authentification difficilement *scalable* puisque la quantité de mémoire requise pour leur stockage grandit en même temps que le nombre d'utilisateur·ice·s.

L'utilisation de JSON Web Tokens présente ainsi plusieurs avantages. Pour pouvoir exploiter un jeton, il est uniquement question de rendre disponible les clés qui permettent de le forger et de le vérifier aux machines impliquées dans la transaction : c'est la clé privée qui est utilisée pour signer le jeton tandis que la clé publique permet de vérifier que sa signature est correcte et peut par conséquent être partagée. Puisque que cela ne nécessite pas le stockage d'un état côté serveur, cette solution est parfaitement *scalable*. De plus, puisqu'on les utilise via un en-tête de requête HTTP Authorization, il est également possible de les exploiter dans un contexte où les cookies sont désactivés. Cela peut s'avérer particulièrement utile dans le cas d'applications mobiles où les cookies sont plus compliqués à gérer.

JSON Web Token

Le standard JSON Web Tokens (JWT) permet l'échange sécurisé de jetons entre plusieurs parties : la sécurité de chaque échange est assurée par la vérification de l'intégrité et de l'authenticité des données grâce aux algorithmes HMAC ou RSA. Un jeton se compose donc de trois parties :

- un en-tête : objet JSON qui décrit le jeton
- une charge utile : object JSON représentant les informations embarquée dans le jeton
- une signature numérique



Aucun des paramètres de la spécification de JWT n'est obligatoire et cette flexibilité lui vaut quelques critiques puisque l'absence de clés pourtant standard peut donner lieu à des trous de sécurité – notamment la clé *exp* qui limite la durée de vie du jeton.

Création des routes d'obtention des tokens

Pour limiter l'usage de mon API aux utilisateur-ices inscrit-es en base de données j'ai besoin de :

- une route publique `/register` pour permettre aux utilisateur-ices de d'inscrire
- une route publique `/login` pour permettre aux utilisateur-ices de se connecter

Tous les autres points de terminaison seront privés : en cas de requête envoyée sans en-tête HTTP Authorization ou contenant un jeton non valide, une erreur HTTP 401 *Unauthorized* sera retournée !

```yaml
# Easy way to control access for large sections of your site
# Note: Only the *first* access control that matches will be used
access_control:
    - { path: ^/$, roles: PUBLIC_ACCESS }
    - { path: ^/login, roles: PUBLIC_ACCESS }
    - { path: ^/users, methods: [POST], roles: PUBLIC_ACCESS }
    - { path: ^/, roles: IS_AUTHENTICATED_FULLY }
```

Pour cela j'ai donc déclaré dans mon fichier `security.yaml` les routes en question ainsi que le niveau d'authentification nécessaire à leur utilisation. On notera que la route `/register` est plutôt une route `/users` en méthode `POST` mais cela sera sûrement amené à changer pour mettre en place une vérification d'email avant l'inscription de l'utilisateur-ice en base de données afin d'éviter les comptes *spam*.

La configuration des modalités d'authentification se fait un peu plus haut dans ce même fichier `security.yaml` – on y déclare :

- l'entité et le champ utilisés pour identifier un-e utilisateur-ice
- la route `/login/refresh_token` permettant le renouvellement d'un JSON Web Token : je spécifie que l'authentification est *stateless* afin d'éviter l'ouverture d'une session PHP dans les *cookies* en cas de succès de la requête
- la route `/login` permettant l'authentification ainsi que les champs utilisés pour identifier mon utilisateur-ice et les services à utiliser en cas de succès ou d'échec de cette dernière

```yaml
# https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider
# used to reload user from session & other features (e.g. switch_user)
providers:
    users:
        entity:
            class: App\Entity\User
            property: email

firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false

    refresh_token:
        pattern: /login/refresh_token
        stateless: true
        refresh_jwt: ~

    main:
        stateless: true
        provider: users

        json_login:
            check_path: /login
            username_path: email
            password_path: password
            success_handler: lexik_jwt_authentication.handler.authentication_success
            failure_handler: lexik_jwt_authentication.handler.authentication_failure

        jwt: ~
        entry_point: jwt
```

La route `/users` en méthode `POST`

Le point de terminaison permettant l'inscription est le premier que j'ai développé : son seul enjeu est le hachage du mot de passe de l'utilisateur-ice avant son ajout en base de données. J'ai pour cela utilisé un `subscriber` que j'ai appelé `UserListener`. Je crée d'abord la fonction `hashPassword` qui permet le hachage du mot de passe la `UserPasswordHasherInterface` de `Symfony` – puis je l'utilise dans une fonction `prePersist` qui sera appelée avant la création d'un-e utilisateur-ice en base de données.

```yaml
App\EventListeners\UserListener:
    class: App\EventListeners\UserListener
    tags:
        - name: doctrine.orm.entity_listener
          entity: App\Entity\User
          event: prePersist
```

Il me faut pour cela déclarer ce service dans le fichier `services.yaml` en indiquant l'entité concernée par le `subscriber` ainsi que l'événement Doctrine du même nom que la fonction créee.

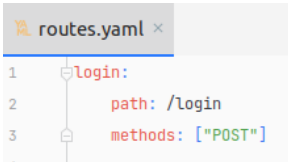
UserListener.php

```php
class UserListener
{
    private UserPasswordHasherInterface $passwordHasher;
    private EntityManagerInterface $entity;

    public function __construct(UserPasswordHasherInterface $passwordHasher, EntityManagerInterface $entity)
    {
        $this->passwordHasher = $passwordHasher;
        $this->entity = $entity;
    }

    public function prePersist(User $user)
    {
        $this->hashPassword($user);

        $user->setToken(Uuid::uuid4()->toString());

        if (!$user->getStatus()) {
            $user->setStatus( status: User::STATUS_UNACTIVATED);
        }
    }

    public function hashPassword(User $entity)
    {
        if (!$entity->getPlainPassword()) {
            return;
        }

        $hashedPassword = $this->passwordHasher->hashPassword(
            $entity,
            $entity->getPlainPassword()
        );

        $entity->setPassword($hashedPassword);
    }
}
```

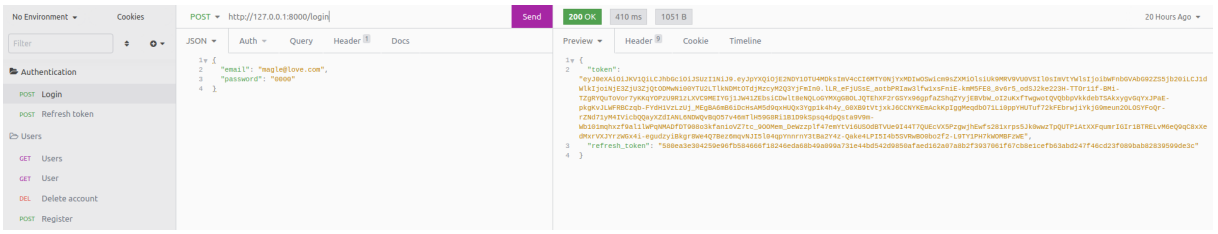
La route `/login`

Je suis ensuite passée au point de terminaison permettant la connexion d'un-e utilisateur-ice et dont l'enjeu est un peu plus complexe – bien que `Symfony` se charge notamment du hachage du mot de passe et de sa comparaison avec celui stocké en base de données grâce aux configurations faites dans le fichier `security.yaml` !

Dans un premier temps j'ai simplement déclaré le point de terminaison dans le fichier `routes.yaml` propre à `Symfony`. Il me restait donc à faire en sorte que :



- les clés `token` et `refresh_token` soient retournées en cas de succès de la requête afin que l'utilisateur puisse les utiliser pour authentifier ses futures requêtes
- retourner une erreur `HTTP 401 Unauthorized` en cas d'échec de la requête

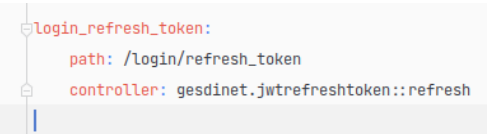


Tout cela a été mis en place dans le fichier de configuration `security.yaml` détaillé plus tôt !

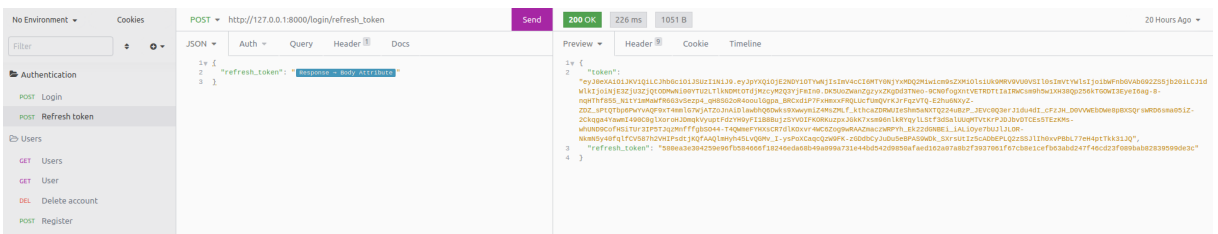
La route `/login/refresh_token`

Une fois encore en ce qui concerne le renouvellement des jetons l'intégralité du code a été écrit dans les fichiers :

- `routes.yaml` pour déclarer le chemin ainsi que le service du bundle communautaire `JWTRefreshTokenBundle` qu'il doit utiliser
- `routes.yaml` pour paramétrer ce même `JWTRefreshTokenBundle`

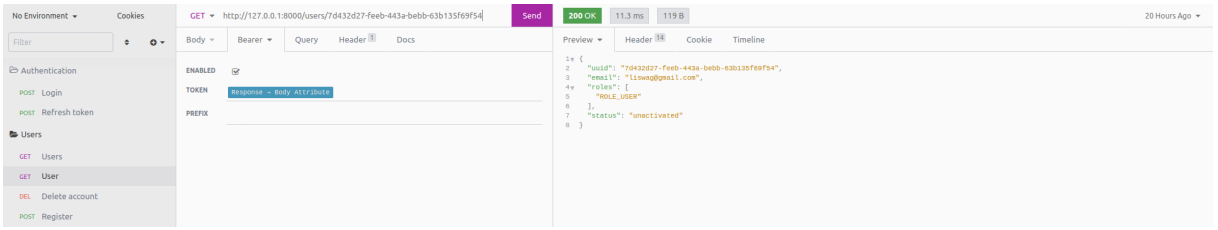


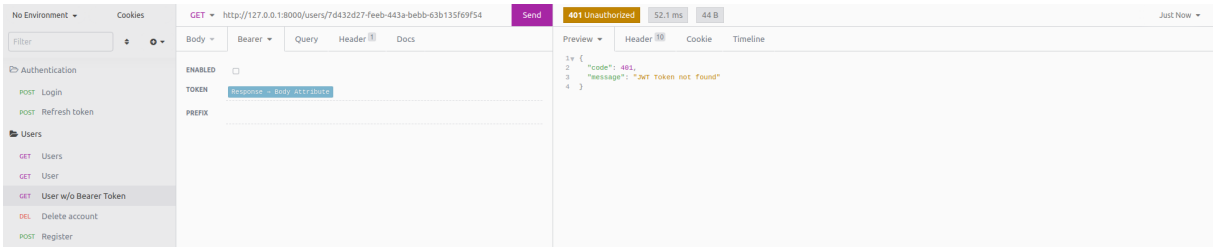
Le point de terminaison renvoie ainsi deux nouvelles clés `token` et `refresh_token` à l'utilisateur-ice (et ce uniquement si iel est authentifié-e).



Les autres points de terminaison

Lorsqu'une requête est faite à mon API – en dehors des routes déclarées publiques dans mon fichier `security.yaml` – les signatures sont vérifiées avant de renvoyer des données. En l'absence d'en-tête `HTTP Authorization` renseignant le `Bearer Token` ou en cas de jeton expiré ou ne correspondant pas : la requête n'aboutit pas et l'API renvoie une erreur `HTTP 401 Unauthorized`.





Hébergement

Je me suis ensuite penchée sur l'hébergement de l'application en louant un serveur privé virtuel (VPS aussi appelé serveur dédié virtualisé) chez OVHcloud. Contrairement à un hébergement mutualisé, un serveur dédié virtualisé permet d'avoir la main sur sa gestion technique – ce qui m'intéressait d'autant plus que je souhaitais acquérir des connaissances de base en administration de serveurs. Le fait de ne pas passer par un serveur dédié m'évite par ailleurs d'avoir à me préoccuper de la gestion du matériel (comme la mémoire vive, le CPU…) qui aurait peu d'intérêt au vu de la simplicité de mon besoin.

Installation du système d'exploitation

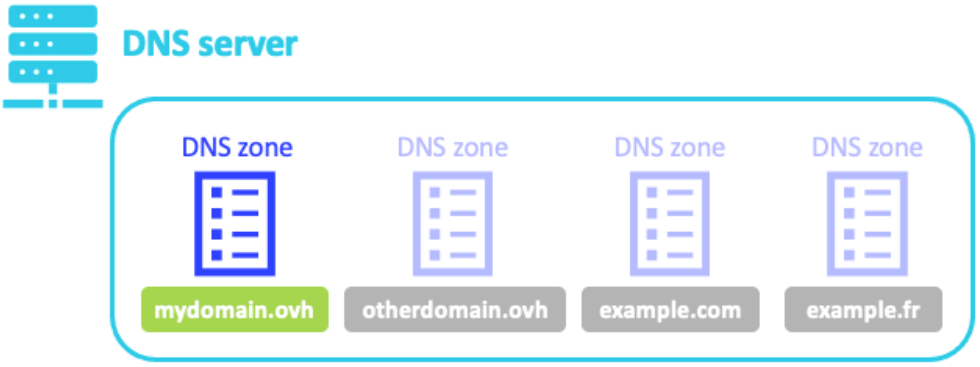
J'ai donc commencé par y installer un système d'exploitation et mon choix s'est porté sur Debian Linux. Habituée à développer sous la distribution Ubuntu, son utilisation dans le cadre d'un serveur privé virtualisé me semblait moins judicieuse dans la mesure où Debian est une distribution d'une grande stabilité (un critère important dans un environnement de production) et qu'Ubuntu se base par ailleurs sur celle-ci.

Debian est l'une des premières distributions de Linux (disponible depuis 1993) et une nouvelle version stable est généralement sortie tous les deux ans – bien qu'elles ne sortent que lorsqu'elles sont prêtes et non selon un calendrier fixe. Cela leur laisse donc une grande marge de manoeuvre pour tester et corriger les bogues, d'où sa stabilité et sa sécurité (qui est prise en charge jusqu'à un an après le lancement de la version stable). Toutes ces raisons en font donc une distribution plus que fiable pour mon serveur privé virtuel.

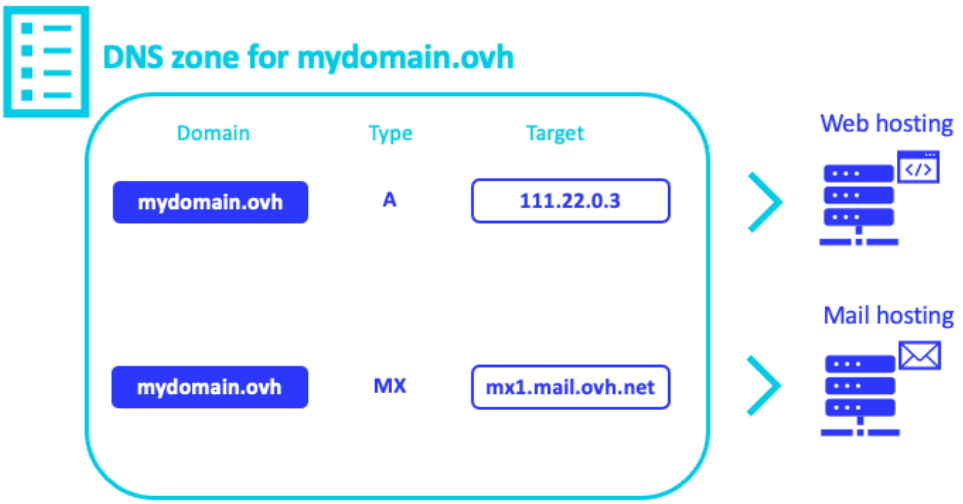
Création du nom de domaine

Lorsque deux ordinateurs échangent sur un réseau IP, un protocole leur attribue une adresse IP

afin de les identifier sur le réseau : en l'état cela implique que chaque utilisateur-ice-s doive retenir l'adresse IP d'un site pour pouvoir y accéder. C'est pourquoi un mécanisme a été mis en place afin de faciliter l'accès aux hôtes en permettant d'associer un nom à une adresse IP : c'est ce qu'on appelle un nom de domaine, et le Domain Name System (DNS) est le service informatique qui se charge de sa conversion.

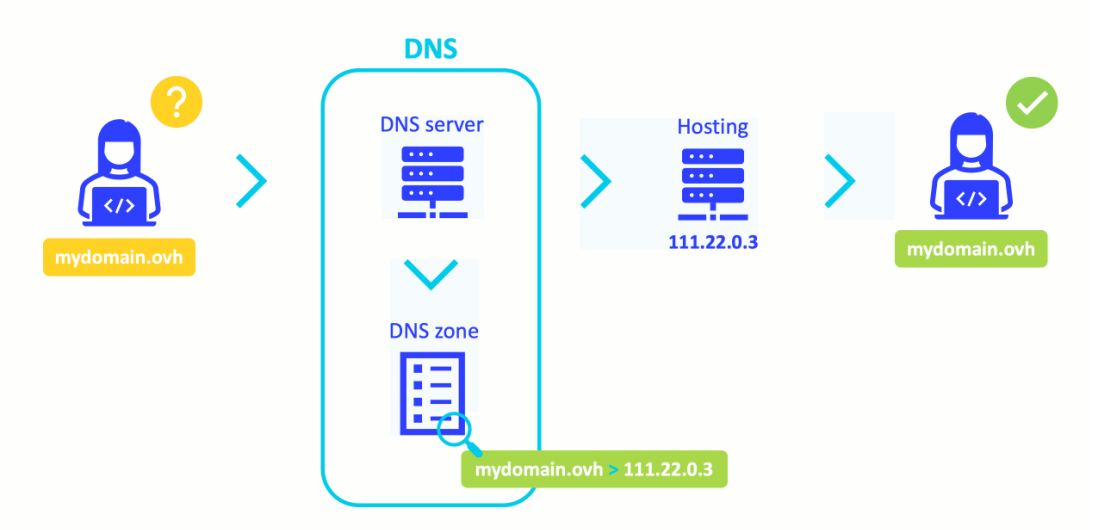


J'ai donc fait l'acquisition de mon propre nom de domaine, toujours chez OVHcloud. Il m'a fallu créer une nouvelle entrée pour celui-ci depuis l'interface de gestion de sa zone DNS. Une zone DNS est un sous-domaine DNS qui prend la forme d'un fichier de configuration : il est composé d'enregistrements reliant un nom de domaine au serveur qui héberge le service en question (dans mon cas via l'enregistrement A consacré aux sites web).



Ainsi lorsque l'URL *mydomain.ovh* sera demandée, la résolution du nom se fera en une recherche

dans la base de données des DNS. Les serveurs DNS contiennent la zone du nom de domaine où est renseignée l'adresse IP de son hébergement :



Puisque le projet *Brize de podcast* est construit de façon à ce que les parties front-end et back-end soient totalement indépendantes, elles ne peuvent être hébergées sur le même nom de domaine :

- le domaine *brizedepodcast.com* renverra ainsi sur l'interface utilisateur-ice
- le sous-domaine *api.brizedepodcast.com* renverra sur l'interface de programmation

De la même façon j'ai créé plusieurs sous-domaines correspondant aux environnements de développement autres que celui de production :

- le sous-domaine *dev.brizedepodcast.com* a son utilité au cours du développement
- le sous-domaine *staging.brizedepodcast.com* fait office de pré-production

Tous renvoient vers la même adresse IP : c'est ensuite mon serveur privé virtuel qui se chargera de renvoyer le site demandé selon le nom de domaine qui sera renseigné.

Configuration du serveur web NGINX

Je commence par placer le code source de chaque site dans un dossier `/var/www/nom-de-domaine`. En ce qui concerne leur configuration, tout se passe dans un fichier `/etc/nginx/site-available/nom-de-domaine` pour lequel je crée un lien symbolique vers un dossier `/etc/nginx/site-enabled/nom-de-domaine` : cela permet de faciliter la gestion de la mise en ligne ou hors ligne de chaque site !

L'hébergement virtuel possible avec `NGINX` me permet ainsi d'héberger plusieurs sites web sur un même serveur HTTP : pour les différencier on utilisera soit le nom de domaine, soit l'adresse IP soit le port ! J'ai personnellement opté pour une différenciation par nom de domaine : pour cela j'utiliserai donc dans un contexte `server` la directive `server_name` suivie du nom de domaine de mon application. Dans un contexte `location` je spécifie par ailleurs l'`adresse IP` ainsi que le `port` associés au nom de domaine grâce à la directive `proxy_pass` – de façon à permettre la résolution du nom de domaine.

Je spécifie également le chemin absolu vers les fichiers d'enregistrement des `logs` grâce aux directives `access_log` et `error_log`. Les `logs NGINX` fournissent des informations détaillées de débogage. Le serveur garde ainsi une trace de ses événements - et ce sous deux formes : les logs d'accès et les logs d'erreurs. Les logs d'accès contiennent des informations sur les requêtes faites par le client, tandis que les logs d'erreur contiennent eux des informations à propos des problèmes rencontrés par l'app et le serveur.

Une partie du fichier a par ailleurs été générée automatiquement par `Let's Encrypt` / `Certbot` afin de sécuriser mon site à l'aide du `protocole HTTPS` : les utilisateur-ices provenant du `port 80` (une connexion non chiffrée) sont ainsi redirigé-es vers le `port 443` ! Le `certificat SSL` expirant au bout de 90 jours, il m'a fallu mettre en place un `cron` afin d'automatiser le processus de renouvellement : la commande nécessaire est ainsi exécutée une fois par mois.

```
server {

    server_name api.brizedepodcast.com www.api.brizedepodcast.com;
    access_log /var/log/nginx/api.brizedepodcast.access.log;
    error_log /var/log/nginx/api.brizedepodcast.error.log;

    location / {
    # reverse proxy for symfony server
    proxy_pass http://51.83.97.44:8000;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection 'upgrade';
    proxy_set_header Host $host;
    proxy_cache_bypass $http_upgrade;
    }

    listen [::]:443 ssl; # managed by Certbot
    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/api.brizedepodcast.com/fullchain.pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/api.brizedepodcast.com/privkey.pem; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot

}

server {

    if ($host = www.api.brizedepodcast.com) {
        return 301 https://$host$request_uri;
    } # managed by Certbot


    if ($host = api.brizedepodcast.com) {
        return 301 https://$host$request_uri;
    } # managed by Certbot

    listen 80;
    listen [::]:80;

    server_name api.brizedepodcast.com www.api.brizedepodcast.com;
    return 404; # managed by Certbot

}
```

```
server {

    server_name brizedepodcast.com www.brizedepodcast.com;
    access_log /var/log/nginx/brizedepodcast.com.access.log;
    error_log /var/log/nginx/brizedepodcast.com.error.log warn;

    location / {
        # reverse proxy for next server
        proxy_pass http://51.83.97.44:3000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }

    listen [::]:443 ssl ipv6only=on; # managed by Certbot
    listen 443 ssl; # managed by Certbot
    ssl_certificate /etc/letsencrypt/live/www.brizedepodcast.com/fullchain.pem; # managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/www.brizedepodcast.com/privkey.pem; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot

}

server {

    if ($host = www.brizedepodcast.com) {
        return 301 https://$host$request_uri;
    } # managed by Certbot


    if ($host = brizedepodcast.com) {
        return 301 https://$host$request_uri;
    } # managed by Certbot


    listen 80;
    listen [::]:80;

    server_name brizedepodcast.com www.brizedepodcast.com;
    return 404; # managed by Certbot

}
```

CONCLUSION

Dans l'équipe de *Simplonline*, mon envie d'apprendre à *bien* coder a été plus que satisfaite : j'ai eu la chance de travailler dans une équipe soucieuse des bonnes pratiques et de la qualité du code. Il était entendu par toutes et tous que nous pouvions passer plus de temps que estimé sur le développement d'une fonctionnalité afin que celle-ci respecte tous les procédés mis en place sur le projet et qu'elle ne devienne pas source de dette technique ! J'ai même pu y aborder des questions d'accessibilité numérique, qui m'intéressent particulièrement. Par ailleurs, l'expérience de la mise en pratique des méthodes Agiles est une chose que je suis heureuse d'emporter pour la suite de mon parcours.

Brize de podcast m'aura permis de découvrir une partie de la vie d'un projet qui était source de curiosité pour moi puisque je ne l'avais encore jamais pratiqué : il s'agit de son déploiement. J'ai beaucoup appris au cours de son développement et il me reste encore de nombreuses choses à découvrir. Ce que j'aime lorsque je travaille sur un tel projet c'est qu'il me pousse à me documenter pour ne rien laisser au hasard et cela me permet de m'approprier des concepts plus complexes – j'ai donc hâte de continuer à apprendre et de voir comment tout cela va évoluer !