Rapport de stage EE3b

TELECOM LILLE

Réseaux informatiques

Développement d'une API REST pour une plateforme d'impression 3D

DELANGLE Flavien	Tuteur entreprise :	Tuteur école :
INGE1	Romain KIDD, CEO	Pierre BOUGEARD

REMERCIEMENTS :

Je tiens tout d'abord à remercier l'équipe WeDesign avec laquelle j'ai eu la chance de réaliser ce stage. Leur bonne humeur et leur motivation ont permis de rendre cette première expérience en entreprise particulièrement agréable et formatrice. Je souhaite tout particulièrement remercier Manon Abiteboul qui a été la responsable de mon équipe pendant toute la durée de mon stage.

Je remercie plus globalement l'ensemble du personnel de *MyMiniFactory* pour m'avoir permis de participer à ce projet au cours de ces derniers mois.

Merci également à mon tuteur entreprise Romain Kidd ainsi que mon tuteur école Pierre Bougeard pour le temps consacré à s'assurer que mon stage se déroulait correctement.

TABLE DES MATIERES :

INTRODUCTION :

La troisième année de formation initiale à Telecom Lille nous donne l'opportunité d'effectuer un stage de quatorze semaines en entreprise. La majeure partie des étudiants effectuent ce dernier dans des domaines tels que le Marketing ou la Fonction commerciale. La possibilité est cependant laissée aux élèves venant de classes préparatoires d'effectuer un stage plus technique ayant pour thème « *Réseaux Informatiques & Réseaux de Télécommunications* ». Ce stage a pour but de nous apporter une première expérience en entreprise ainsi que des compétences techniques plus avancées dans certains domaines.

J'ai souhaité effectuer ce stage dans un domaine proche de l'informatique avec une préférence pour le développement de sites internet. J'ai donc pris contact avec la société *MyMiniFactory* située à Londres. Cette entreprise travaille principalement dans la création de plateformes en ligne en lien avec l'univers de l'impression 3D.

L'équipe que j'ai intégrée travaille sur une plateforme d'édition d'objets 3D en ligne nommée *WeDesign.live*. Ce projet est accompagné d'un réseau social permettant de partager les créations des autres utilisateurs. J'ai pour ma part dans un premier temps travaillé sur la création d'une *API* (*Application Programming Interface*) et sur son intégration dans l'éditeur d'objets 3D. J'ai ensuite cherché à améliorer les fonctionnalités existantes de la plateforme afin de les adapter à la nouvelle structure du projet.

Dans un premier temps nous proposeront une présentation de l'entreprise *MyMiniFactory* ainsi que de son réseau interne. Nous nous appliquerons à comprendre l'évolution de ce réseau durant ces derniers mois. Nous étudierons ensuite la création de l'API en expliquant comment nous avons cherché à répondre au mieux aux besoins de la plateforme. Dans un troisième temps nous nous pencherons sur le fonctionnement de la collaboration sur la plateforme *WeDesign.live* ainsi que sur les limites du modèle actuel et les améliorations pouvant y être apportées.

1. MyMiniFactory, leader dans un domaine porteur

1.1 Activités et projets de MyMiniFactory

MyMiniFactory est une entreprise développant un ensemble de projets en rapport avec l'univers de l'impression 3D. Le projet principal de cette société est le site internet *MyMiniFactory.com* qui est une plateforme de partage d'objets imprimables en 3D lancée en juin 2013.

L'entreprise ne se limite cependant pas à ce site internet, elle a en effet su se diversifier au travers de plusieurs projets cherchant à tirer parti du potentiel immense que propose l'impression 3D. L'un d'entre eux est le projet *Scan the World* qui cherche à reproduire en 3D des monuments ou des œuvres d'arts. Ce projet est particulièrement important car il permettra dans le futur de garder une trace d'œuvres d'art risquant d'être détruites comme c'est le cas actuellement dans des villes telle que Palmyre en Syrie.

Un autre projet récent de *MyMiniFactory* est la plateforme *WeDesign.live* sur laquelle j'ai travaillé pendant ces derniers mois. Cette dernière vise à rassembler l'ensemble des étapes de la création d'un objet imprimable en 3D au sein d'une seule plateforme collaborative.

L'entreprise tire la majeure partie de son financement de campagne de levée de fonds. En effet la force de la société réside dans sa capacité à proposer des projets visant la rentabilité sur le long terme et non pas juste un gain à court terme. *MyMiniFactory* a cependant plusieurs autres sources de revenu telle que la vente d'imprimantes 3D ou la mise en place prochaine de balises publicitaires sur le site internet.

1.2 Rendre l'impression 3D accessible aux particuliers

On peut aujourd'hui séparer l'impression 3D en deux domaines distincts que sont l'impression 3D industrielle et l'impression 3D grand publique. Si l'impression 3D est actuellement devenue un incontournable dans de nombreuses industries, elle reste un marché de niche chez les particuliers. C'est dans ce contexte que *MyMiniFactory* essaie de mettre en avant le potentiel immense de l'impression 3D grand publique.

1.3 Une stratégie basée sur les partenariats

Depuis plusieurs années, *MyMiniFactory* effectue des partenariats avec des entreprises telles que *Parrot* ou *Pebble*. Ces partenariats prennent la forme de concours où les designers de la plateforme sont invités à créer des accessoires pour un produit de la marque partenaire.

Les partenariats sont un élément central dans la communication de *MyMiniFactory*. En effet ces derniers permettent à l'entreprise de mieux cibler les designers en leur montrant concrètement les possibilités offertes par la plateforme. L'univers de l'impression 3D est un domaine particulièrement propice à de telles collaborations. En effet de nombreuses produits peuvent bénéficier d'éléments imprimés en 3D et il est possible de faire appel à la communauté du site pour imaginer de nouvelles utilisations ou de nouveaux designs.

L'un des partenariats les plus important pour l'entreprise a été celui avec l'entreprise française *Parrot*. Cette dernière a pour cœur de métier la création de drones à destination du grand public. Les deux entreprises ont ainsi proposé à leurs communautés de créer des accessoires imprimables en 3D pour les nouvelles gammes *Minidrone* et *Bebop Parrot*.

pebble Parrot

1.4 Un réseau pour le développement en équipe

Le réseau de *MyMiniFactory* était historiquement entièrement consacré au développement et à la mise à disposition de la plateforme *MyMiniFactory.com*. La création du projet *WeDesign.live* a cependant rendu nécessaire la scission de l'équipement informatique en deux entités indépendantes. En effet notre projet avait la chance de posséder son propre équipement sur lequel nous avions un contrôle absolu.

Nous allons donc essayez de comprendre en quoi l'organisation de cet équipement (présenté sur la figure n°1) répondait avant tout à des besoins de développement. La plateforme étant en cours de création, peu de personnes peuvent accéder à son contenu. Un serveur public existe mais il est réservé aux démonstrations faites avec des sociétés comme la *BBC*. Ce serveur est situé dans un datacenter appartenant à *Amazon* et n'a pas été mis à jour depuis la publication de la première version alpha du projet au début du mois de janvier. L'équipe de développement de *WeDesign.live* utilise à la place un serveur de test situé dans les locaux de l'entreprise et sur lequel il est possible d'effectuer régulièrement des modifications pour tester les changements effectués par l'équipe. Chaque développeur possède également une version du projet sur son ordinateur sur laquelle il peut travailler. Le but de cette organisation est d'éviter au maximum la mise en place de nouveautés non fonctionnelles et d'avoir à tout instant un serveur fonctionnant parfaitement.

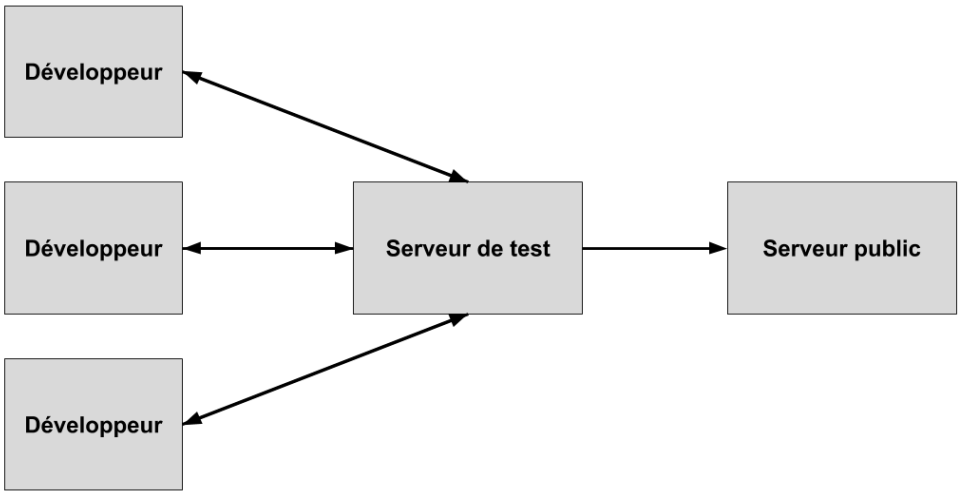


Figure 1. Organisation du réseau utilisé pour le projet *WeDesign.live*

Peu de temps avant la fin de mon stage, le serveur public de notre projet a été remplacé par un nouveau, plus puissant, pouvant supporter le trafic généré par la mise en ligne de la plateforme. Le choix de ce dernier a nécessité une réflexion sur les besoins techniques du site. Nous avons par exemple fait le choix de privilégier une forte quantité de *RAM* afin de rendre la base de données *Redis* plus performante.

Afin de faire le lien entre toutes ces machines, l'équipe de *WeDesign.live* utilise le logiciel de gestion de version *Git*. Ce dernier nous permet de facilement appliquer les modifications d'un développeur sur l'ensemble. Il permet également de facilement surveiller les modifications apportées au projet et ainsi de comprendre plus aisément les erreurs présentes dans le programme.

2. LA CREATION D'UNE API POUR WEDESIGN.LIVE

2.1 Présentation du projet

L'impression 3D, ou impression tridimensionnelle, est un procédé permettant de produire un objet réel à partir d'un objet réalisé en utilisant un outil de *Conception assistée par ordinateur* (*CAO*).

L'objectif de la plateforme *WeDesign.live* est de regrouper l'ensemble des outils nécessaires de la conception de l'objet jusqu'au partage de ce dernier. Le projet est constitué de trois aspects principaux développés en parallèle :

1) Un éditeur de *CAO* permettant la création d'un objet virtuel en 3D.
2) Un *slicer* permettant de transformer l'objet virtuel en une suite d'instructions compréhensibles par une imprimante 3D.
3) Un réseau social permettant de partager ces créations, de collaborer avec d'autres créateurs ou encore de regarder un designer concevoir un objet en direct.

L'équipe en charge de *WeDesign.live* était donc répartie en trois équipes plus restreintes, chacune responsable d'un aspect du projet. J'ai pour ma part intégré l'équipe en charge de toutes les fonctionnalités sociales de la plateforme. Au cours de mon stage, j'ai cependant eu l'occasion de travailler en collaboration avec les autres équipes, ce qui m'a permis d'avoir une vue d'ensemble du projet.

Les fonctionnalités présentées dans les prochaines pages ont toutes été développées par l'équipe en charge des fonctionnalités sociales de la plateforme. Ce sont celles ayant eu à mes yeux le plus d'importance dans mon travail tout au long du stage. Cependant la plupart ont été développées en collaboration avec d'autres membres de l'équipe qui m'ont été d'une grande aide.

2.2 Les technologies utilisées :

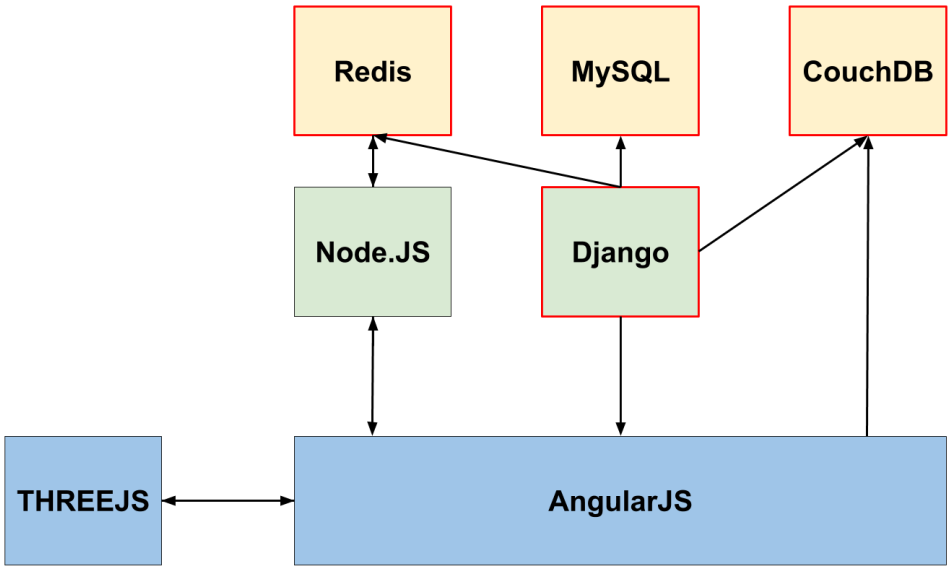


Figure 2. Technologies utilisées par le projet *WeDesign.live*

Architecture frontend :

L'ensemble des scripts exécutés sur l'ordinateur de l'utilisateur ont été développés en *JavaScript*. L'affichage des objets 3D est réalisé grâce à la librairie *THREEJS* qui permet de faciliter l'utilisation des *canvas*. Nous utilisons également le *framework AngularJS* afin de créer une *Single Page Application*.

Architecture backend :

Le serveur physique de *WeDesign.live* compte deux serveurs et trois bases de données distinctes. Nous utilisons tout d'abord *Node.JS* pour toutes les fonctionnalités liées à la collaboration en temps réel. En parallèle nous utilisons *Django* pour la gestion de l'*API* et pour la récupération des fichiers statiques.

En termes de bases de données nous utilisons *MySQL* pour le stockage des données liées à l'API, *CouchDB* pour le stockage des données plus volumineuses et *Redis* pour les données devant être rapidement propagées telles que les notifications ou les messages.

2.3 Pourquoi Django ?

La décision a très vite été prise d'utiliser un *framework* afin de développer notre API. En effet de tels outils permettent d'augmenter fortement la vitesse de création des fonctionnalités. Ils permettent également d'éviter un grand nombre d'erreurs étant donné que de nombreuses fonctionnalités sont incluses et ont pu être testées par un grand nombre d'utilisateurs. Enfin, l'utilisation d'un *framework* force les développeurs à adopter une certaine structure dans leur code ce qui permet la création de programmes plus compréhensibles et plus respectueux des normes.

Le nombre de *framework* de qualité étant important, nous avons dû affiner notre recherche pour trouver l'outil le plus adapté à notre projet et aux connaissances des développeurs. Quand je suis arrivé à *MyMiniFactory*, le seul serveur utilisé était *NodeJS* qui sert principalement à la communication entre les navigateurs des différents utilisateurs travaillant ensemble sur un projet ou regardant un autre utilisateur créer un objet. Une solution aurait donc été d'utiliser *NodeJS* associé au *framework Express.js* pour créer l'API.

La décision a cependant été prise d'utiliser *Django*, un framework *MVC* utilisant *Python*. L'une des forces de *Django* était le projet *Django Rest Framework* qui offrait de nombreux outils facilitant la création d'une *API REST*. Un autre avantage significatif était qu'une bonne partie du projet était déjà réalisé en *Python*. Il était donc simple de migrer certaines fonctionnalités vers *Django* et l'équipe connaissait déjà ce langage de programmation.

De plus, *Django* propose plusieurs fonctionnalités particulièrement puissantes. L'une des plus notables étant la présence d'un *ORM* (ou Object-Relational Mapping) qui nous permet de communiquer avec la base de données *MySQL* sans avoir à écrire explicitement les requêtes *SQL*.

2.4 Pourquoi créer une API ?

Le projet se présentait au départ sous la forme d'une *Single Page Application* étant donné que toutes les actions étaient réalisées à partir de l'éditeur d'objets 3D. Nous avons souhaité continuer dans cette voie en ne modifiant que les éléments nécessaires de chaque page. Ainsi quand le navigateur passe du profil d'un utilisateur à celui d'un autre, l'application ne fait que charger les données personnelles du nouvel utilisateur et ne touche que très peu à l'interface. Nous avions donc besoin de récupérer des données brutes sur le serveur (liste d'objets, photo de profile, notifications …) sans alourdir les requêtes en renvoyant à chaque fois l'interface. Or c'est exactement ce que propose une API : des requêtes renvoyant des données brutes pouvant être utilisées indépendamment du support.

> L'objectif d'une API est de fournir une porte d'accès à une fonctionnalité en cachant les détails de la mise en œuvre.
>
> Wikipédia France

L'utilisation d'une API offre également l'avantage de faciliter grandement la multiplication des supports sur lesquels *WeDesign.live* est disponible. Si l'équipe décide un jour de créer une application *Android* ou *iOS*, il lui suffira de créer une nouvelle interface adaptée à ce type de plateforme et de réutiliser les requêtes déjà existantes. Au-delà de la simple multiplication des supports, une API permet également de facilement proposer des fonctionnalités à des services tiers. Comme nous le verrons dans la présentation de l'authentification de *WeDesign.live*, une *API* peut ainsi permettre de donner accès à certaines données de la plateforme à d'autres sites ou applications. Dans le cadre de notre projet, il pourrait par exemple être intéressant de proposer une requête renvoyant la totalité des données nécessaires pour afficher un objet dans un autre site.

Nous avons plus précisément décidé de suivre la convention *REST* qui permet de créer des API avec des requêtes particulièrement compréhensibles. Ce standard utilise les méthodes HTTP (GET, POST, PUT, PATCH, DELETE) pour indiquer la nature de la requête. Une requête POST visera par exemple toujours à créer un élément alors qu'une requête PATCH ne fera que modifier un élément déjà existant. La décision d'utiliser une *API REST* s'inscrit dans une série de mesures visant à rendre le projet respectueux des standards ce qui est indispensable pour permettre une prise en main rapide pour les nouveaux développeurs.

2.5 Le fonctionnement de Django Rest Framework

L'une des forces des *frameworks* est d'imposer une structure claire à chaque projet. *Django Rest Framework* ne dérogeant pas à la règle, il organise le code en quatre sections principales :

- Le *routeur* qui est exécuté à la réception de chaque requête et envoie celle-ci vers le *Viewset* responsable de cette requête.
- Le *viewset* qui regroupe toutes les vues liées à une fonctionnalité. Pour chaque requête envoyée par le routeur, le *viewset* exécute automatiquement une fonction dépendant de la structure de l'URL et de la méthode HTTP utilisée.
- Le *serializer* qui permet de transformer les données reçues dans la requête en données compréhensibles par les modèles et inversement. Il permet aussi de n'afficher qu'une partie des données récupérées et de vérifier que les données envoyées sont bien valides.
- Le *modèle* qui représente la structure de chaque objet et qui fait le lien entre l'application et la base de données.

On peut illustrer cette organisation avec la requête suivante : *GET /api/users/1/* qui a pour objectif de renvoyer les informations relatives à l'utilisateur n°1.

1) Le routeur reçoit une requête portant l'URL */api/users/1/*. Il sait que toutes les requêtes de la forme */api/users/** doivent être redirigées vers le viewset *UserAccountViewSet.*

2) Le viewset *UserAccountViewSet* reçoit la requête et remarque que cette dernière utilise la méthode HTTP GET. Il remarque également que l'URL comporte un identifiant. Ces deux informations lui permettent de déduire qu'il doit exécuter la méthode *retrieve* du viewset *UserAccountViewSet*.

3) La méthode *retrieve* crée ensuite une instance du serializer qui lui est lié (ici *UserAccountSerializerFullData*). Elle va ensuite demander au modèle *UserAccount* de lui renvoyer le profil de l'utilisateur portant l'identifiant n°1. Enfin elle donne ce profil au serializer qui va le transformer en un objet lisible par l'application recevant la réponse de cette requête.

Django REST Framework proposait également de nombreuses fonctionnalités particulièrement utiles telles qu'une gestion améliorée des permissions. Pour enrichir notre application, nous avons tiré parti de modules tiers permettant d'ajouter des fonctionnalités telles que la pagination de certaines requêtes ou l'authentification via le protocole *OAuth2*.

2.6 La création d'une Single Page Application

Les sites internet développés durant les années suivant la création du *World Wide Web* étaient pour la plupart de simples pages statiques affichant des documents. Les fonctionnalités de ces sites se sont cependant rapidement complexifiées et les développeurs n'ont eu de cesse de trouver des solutions pour créer des plateformes toujours plus ambitieuses et agréables à utiliser.

Au début du XXIème siècle, des technologies comme *AJAX* ont permis de rendre les pages internet dynamiques en récupérant des données sur le serveur sans avoir besoin de recharger la totalité de la page. La plupart des applications que nous utilisons aujourd'hui reposent sur ce principe. En effet des actions comme envoyer un mail ou charger une vidéo n'entrainent pas une recharge de la totalité de la page. Ces données sont envoyées au serveur grâce à l'exécution d'un code *JavaScript* qui va envoyer la requête au serveur sans pour autant arrêter d'afficher la page en cours. Il en résulte une navigation beaucoup plus fluide et agréable pour l'utilisateur.

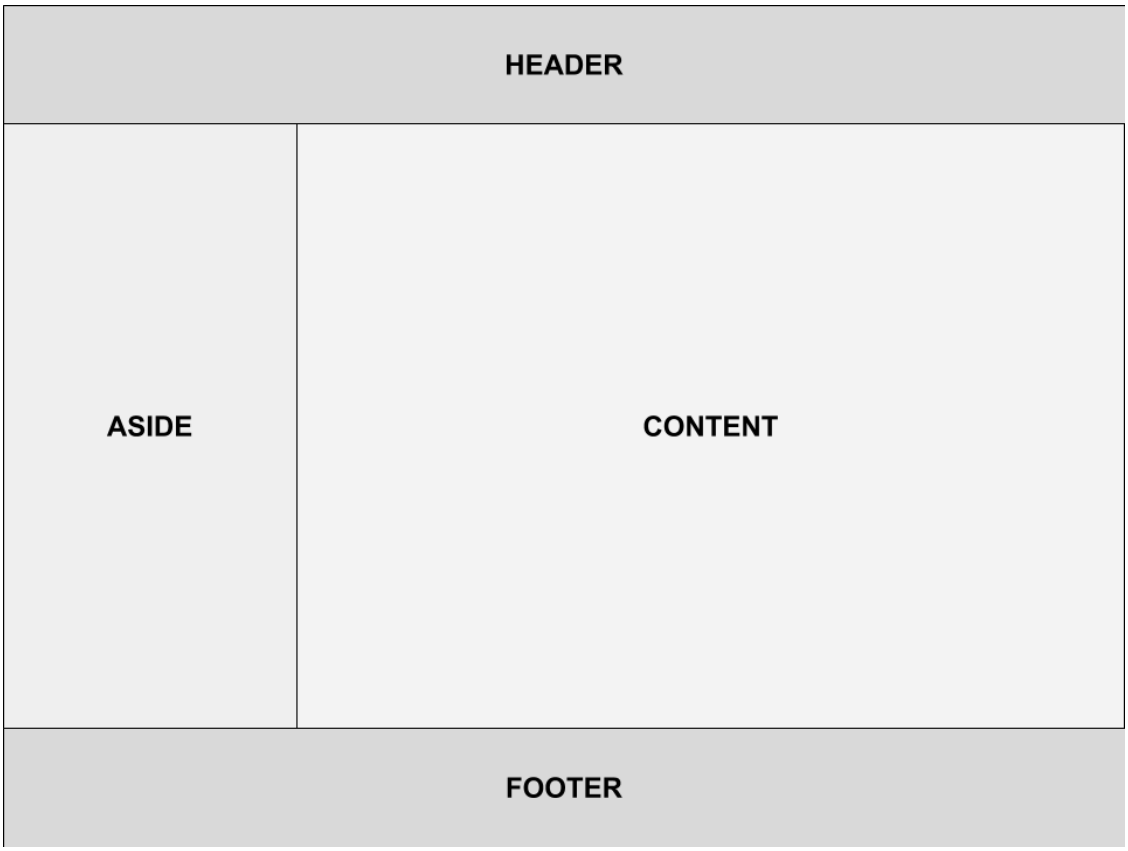


Figure 3. Structure d'une page internet

Ce raisonnement peut également être appliquée lorsque l'utilisateur change de page. En effet une bonne partie des informations, telles que les menus, sont conservés. Il serait donc plus efficace de ne recharger que le corps de la page et de ne toucher aux parties statiques qu'en cas de besoin. Si on suit la structure de page internet sur la Figure 3, il est possible de ne recharger que la section *Content* de la page.

Une autre façon de limiter les transferts de données entre le client et le serveur est de confier au client la génération de la page internet. La manière la plus répandue de créer une page web est de générer le code HTML sur le serveur et de l'envoyer au client pour qu'il l'affiche. Dans certaines situations, il peut cependant être intéressant de n'envoyer qu'un squelette HTML à l'utilisateur et de laisser des scripts exécutés chez le client modifier la page quand cela est nécessaire. Le serveur est alors constitué d'une unique page qui fait office de base commune pour toutes les sections du site. Des programmes *JavaScript* vont ensuite permettre de remplir cette page au gré des actions de l'utilisateur : on a alors une *Single Page Application*.

De nombreux *framework* existent afin de faciliter la création de ce type de site. Le plus connu d'entre eux est *AngularJS* développé par *Google* depuis 2009. L'équipe de *WeDesign* utilisait cet outil depuis plusieurs mois quand je suis arrivé à *MyMiniFactory*. Cependant la plupart des fonctionnalités étaient encore indépendantes d'*AngularJS* et nous avons donc peu à peu migré la majeure partie du projet vers ce *framework* afin d'obtenir un code plus lisible et plus facile à maintenir.

L'utilisation d'une *framework* tel qu'*AngularJS* pour notre plateforme a permis de rendre cette dernière bien plus fluide et agréable à utiliser. En effet l'ensemble du code nécessaire au client dépassait les 30 000 lignes lors de mon arrivé et le charger entièrement prenait plusieurs secondes. Il était donc particulièrement pénible de naviguer de page en page en rechargeant à chaque fois la totalité du contenu. De plus *AngularJS* nous a permis de facilement réutiliser certains pans de notre programme. L'exemple le plus intéressant est sûrement celui de l'éditeur 3D que nous avons pu intégrer dans différentes sections du site telles que le profil des utilisateurs.

2.7 L'authentification de l'API

Comment s'authentifier sur la plateforme ?

La plupart des fonctionnalités de notre plateforme sont accessibles à tous et ne nécessitent aucune authentification. Il est ainsi possible de consulter des objets, de regarder un utilisateur créer un objet en direct ou encore d'imprimer un objet sans avoir préalablement créé de compte. Nous avons cependant inclus un système d'authentification pour les actions telles que les commentaires, les votes ou tout simplement la publication d'objets. Dans notre optique d'accessibilité, nous avons souhaité proposer plusieurs méthodes d'authentification aux utilisateurs. La première d'entre-elles est de créer un compte en renseignant son adresse mail, un nom d'utilisateur et un mot de passe. Il existe néanmoins d'autres méthodes d'authentification telles que la connexion via *Facebook* ou *Twitter*.

Se connecter avec *Facebook* ou *Twitter* :

Les connexions dites « *sociales* » se sont fortement développées ces dernières années au point que la plupart des sites dotés d'une authentification par un compte proposent aujourd'hui ce type de connexion. L'avantage principale pour l'utilisateur est la rapidité d'accès au contenu ainsi que la non-multiplication des données de connexions. Un seul identifiant permet de se connecter simplement à une multitude de sites. Une autre force de ces systèmes d'authentification est de permettre aux utilisateurs et aux développeurs de facilement tirer parti de fonctionnalités de ces réseaux sociaux. Une fois un utilisateur connecté avec son compte *Twitter*, il est particulièrement aisé de lui permettre de poster un message sur ce réseau social.

Pour les développeurs, ces connexions demandent l'utilisation d'*API* permettant de vérifier l'identité des utilisateurs mais également de récupérer les données nécessaires. Arriver à utiliser ces *API* nous a pris un temps important et nous ne sommes pas certains d'avoir trouvé la solution optimale. Cependant la connexion via les réseaux sociaux fonctionne actuellement parfaitement.

Le protocole OAuth2 :

Une fois l'utilisateur connecté, le processus d'authentification est le même quelle que soit la méthode de connexion. Nous avons opté pour le protocole de délégation d'authentification *OAuth2* dont le fonctionnement tel que nous l'avons implémenté est décrit dans les Figures 4 et 5. *OAuth2* est un protocole libre permettant d'authentifier un utilisateur auprès d'une application. La force de ce protocole est de dissocier entièrement l'authentification de la gestion des ressources. L'application peut alors proposer plusieurs authentifications indépendantes et un serveur d'authentification peut être utilisé par plusieurs applications différentes.

Pour mettre en place une authentification respectant le protocole OAuth2, nous avons besoin de quatre éléments :

- Un **client** qui va demander des ressources au serveur pour les afficher à l'utilisateur. Ce client peut prendre la forme d'une application pour téléphone ou bien dans notre cas d'un navigateur internet.

- Un **propriétaire** dont le rôle est de fournir une preuve d'authenticité du client. Dans notre cas se sera l'utilisateur qui jouera ce rôle en renseignant son mot de passe sur la page de connexion.

- Un **serveur d'authentification** qui reçoit la preuve d'authentification et fournit en échange un *token* d'accès ainsi qu'un *token* d'actualisation. Ces deux *tokens* prennent la forme d'une chaine de 30 caractères générée aléatoirement.

- Un **détenteur de la ressource** qui ne renverra les informations demandées que si on lui présente un token d'authentification valide.

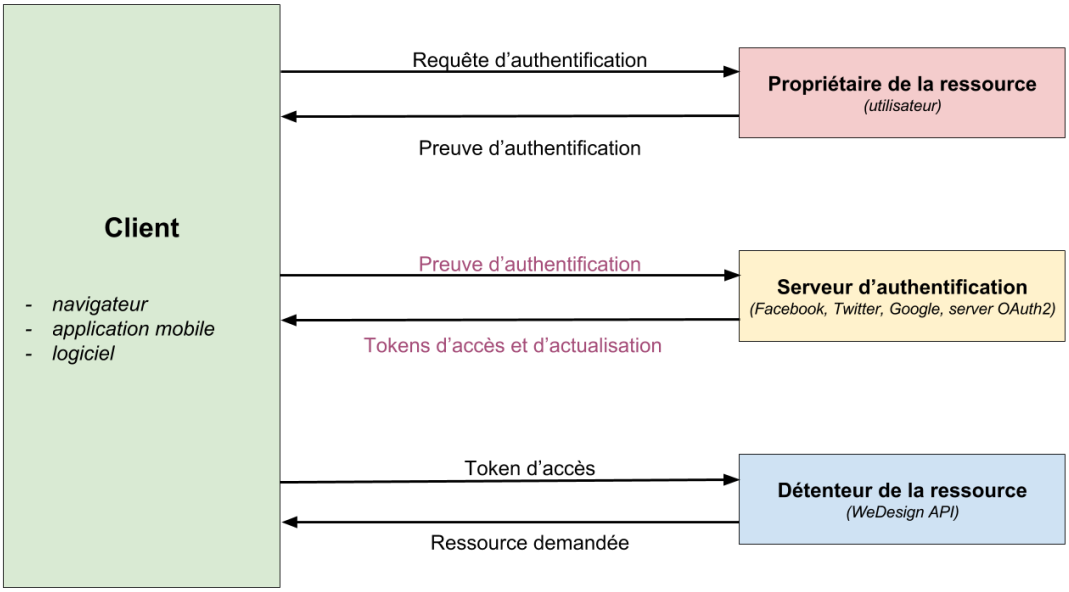


Figure 4. Initialisation d'une authenfication avec le protocole OAuth2

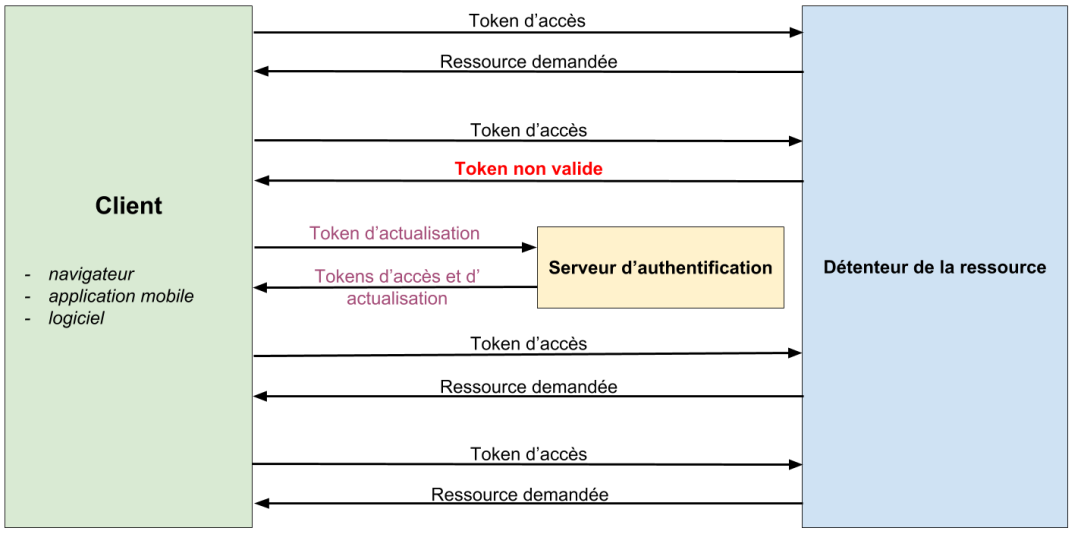


Figure 5. Renouvellement d'une authentification avec le protocole OAuth2

2.8 L'accès au contenu

Une des problématiques majeures quand on souhaite mettre au point une plateforme de création et de partage de contenu est la mise à disposition de ce contenu aux utilisateurs. En effet il faut qu'une personne arrivant pour la première fois sur le site ait accès à un contenu intéressant malgré le manque d'informations et que dans le même temps un utilisateur récurrent ait accès à un contenu reflétant ses goûts. Le développement d'algorithmes de prédiction de contenu est au cœur de la stratégie de grandes entreprises telles que *Facebook*, *Amazon* ou encore *Netflix*. *WeDesign.live* étant une petite structure, nous n'avions nullement la prétention de chercher à égaler ces projets vieux de plusieurs années et forts de plusieurs centaines de développeurs. Nous avons cependant cherché à créer un système simple et efficace permettant d'afficher un contenu de qualité à nos utilisateurs. Notre stratégie de découverte de contenue est découpée en trois grandes sections :

Les *Tendances :*

Afin de proposer un contenu intéressant aux utilisateurs découvrant la plateforme, nous avons mis en place une page de *Tendances* affichant un contenu plaisant au plus grand nombre. Actuellement cette page met à disposition des utilisateurs certains des objets les plus populaires ainsi que certains des utilisateurs ayant le plus de personnes les suivant. Afin de permettre aux utilisateurs de découvrir un maximum de contenu, nous avons cherché à introduire une composante aléatoire dans la récupération des données. Ainsi l'API renverra trois utilisateurs et trois objets choisis aléatoirement parmi les dix utilisateurs et les dix objets répondant le mieux aux critères de sélection.

La sélection est pour l'instant sommaire et nécessitera une refonte avant la publication de la plateforme. Nous avons cependant essayé de préparer les améliorations futures en commençant à enregistrer dès maintenant les informations pouvant être nécessaires à une meilleure hiérarchisation des projets. En effet la plateforme permet aux utilisateurs d'aimer des objets, de les partager ou encore de les commenter. Nous enregistrons également le nombre de vue effectuées par chaque objet.

Le Fil d'actualité :

Un utilisateur se connectant régulièrement sur la plateforme souhaitera avoir accès à un contenu ciblé et plus uniquement à des objets populaires. Nous avons donc permis aux utilisateurs de suivre certains designers afin de pouvoir être avertis en cas de nouvelle création. Ces données sont mises à disposition de l'utilisateur sur son *Fil d'actualité* par ordre chronologique.

Pour l'instant nous affichons la totalité des objets créés par les personnes que notre utilisateur suit. Une amélioration pour le futur sera d'essayer d'évaluer la pertinence de chaque objet afin de n'afficher que les plus susceptibles de lui plaire. Il pourra également être intéressant de chercher à afficher des objets aimés ou commentés par ces amis afin de lui faire découvrir un contenu qu'il ne serait pas forcément allé chercher lui-même.

Les collections :

Pour l'instant, tout le contenu dont nous avons parlé était créé et partagé uniquement par des designers. Nous souhaitions cependant qu'un utilisateur ne voulant pas créer lui-même un objet puisse mettre en avant ces goûts et les partager. Pour cela nous avons décidé d'intégrer à notre plateforme un système de collections. Ces dernières se présentent sous la forme d'une liste d'objets à laquelle un utilisateur peut s'abonner et ainsi être tenu au courant des nouveautés leur étant apportées.

Cette fonctionnalité a été imaginée dès le premier jour de mon stage mais son intégration n'a débuté que très tardivement. Il est donc possible que la direction prise par l'équipe s'éloigne du modèle théorique que nous avons établi. Nous sommes par exemple encore indécis sur le rôle de la page dédiée aux collections et sur l'affichage ou non des modifications de collections sur le fil d'actualité. Cette fonctionnalité est particulièrement centrale dans notre projet et elle permettra donc de faire le lien entre de nombreux programmes que nous avons développés.

2.9 Le stockage des informations

L'une des évolutions majeures causée par la dimension sociale du projet a été la multiplication des technologies utilisées, particulièrement en ce qui concerne les bases de données. Avant mon arrivée, les objets étaient stockés dans de simples fichiers sur le serveur. Cette solution était en bien des points perfectible mais elle fonctionnait et permettait d'éviter l'apprentissage de nouvelles technologies. Cependant l'introduction d'un système de compte a rendu la création d'une première base de données obligatoire. Rapidement nous sommes arrivés à la conclusion qu'aucune technologie de stockage n'était satisfaisante pour tous nos besoins. A la fin de mon stage, notre serveur comptait trois bases de données différentes, chacune spécialisée dans un domaine particulier.

MySQL - Django :

La première base de données que nous avons créée appartient à la famille des bases de données *MySQL*. Cette dernière a pour but de stocker les données de l'API telles que les comptes des utilisateurs ainsi que toutes les informations relatives à ces derniers. Ces données sont souvent très légères mais le serveur peut avoir à en stocker un très grand nombre. On peut par exemple citer la possibilité pour un utilisateur d'en suivre un autre. Chaque lien entre deux utilisateurs étant représenté par une ligne en base de données, on peut rapidement se retrouver avec des milliers de ligne même pour quelques centaines d'utilisateurs.

Dans cette base de données nous avons également enregistré les métadonnées des objets créés sur l'éditeur. En effet une base de données *MySQL* permet de facilement récupérer des sous-groupes de données complexes telles que l'ensemble des objets les plus populaires ou bien l'ensemble des objets d'un utilisateur en particulier.

De plus, *Django* propose un *ORM* (*Object-Relational Mapping*) nous permettant de ne pas écrire explicitement les requêtes *SQL* pour communiquer avec la base de données. Le principal bénéfice est la création d'une application plus cohérente car les accès à la base de données sont effectués avec des requêtes orientés objet ce qui est plus proche de la philosophie de programmation adoptée par l'ensemble du projet. Cette solution permet également d'éviter l'apprentissage du langage *SQL* puisqu'une simple connaissance de la syntaxe *Python* permet de comprendre le fonctionnement des requêtes simples.

Voici deux exemples de requêtes permettant d'accéder à la photo de profil d'un utilisateur ayant pour identifiant 3 :

SQL : SELECT picture FROM API_useraccounts WHERE ID = 3

Django : UserAccounts.objects.get(pk=3).picture

Cette solution fonctionne également pour des requêtes plus complexes. Il est cependant nécessaire d'avoir un minimum de connaissances en *SQL* pour comprendre le fonctionnement des requêtes plus avancées malgré la couche d'abstraction. Voici par exemple une requête permettant de récupérer les identifiants des trois objets ayant le plus de vues :

SQL : SELECT * FROM API_objects3d ORDER BY views DESC LIMIT 3

Django : Objects3D.objets.all().order_by('-views')[:3]

CouchDB - Stockage des objets :

Cette base de données appartient à la famille *NoSQL* qui regroupent toutes les bases de données n'utilisant pas le paradigme classique des bases de données relationnelles. Ces dernières ne forment en aucun cas un tout homogène et peuvent être classées en plusieurs familles. CouchDB appartient à la famille des *Bases de Données Orientées Document* dont elle est le principal représentant au côté de *MongoDB*.

CouchDB est particulièrement simple à prendre en main. En effet la base de données est livrée avec un serveur web complet utilisant une API REST. Nous avons donc pu nous inspirer de nos travaux sur l'API pour réaliser le stockage des objets. Chaque objet est représenté par une clé avec laquelle on peut récupérer son contenu, le modifier ou encore le supprimer. Cette base de données contient également un système de gestion des versions qui nous a permis d'éviter un grand nombre de conflits en cas de projets utilisés sur plusieurs ordinateurs.

Redis – Stockage des notifications

Redis est également une base de données *NoSQL* appartenant à la famille des bases de données dites « Clé / Valeur » et a pour avantage d'être particulièrement légère. C'est cette légèreté qui fait qu'aujourd'hui de nombreuses entreprises l'utilisent pour gérer d'énormes quantités de données devant être accessibles très rapidement. On peut par exemple citer l'exemple de *Twitter* dont les messages sont stockés avec *Redis*, évitant ainsi la lourdeur d'une base de données *MySQL*. *Redis* est particulièrement conseillé pour le stockage de données très légères et devant être accédées un grand nombre de fois. C'est donc tout naturellement que nous avons décidé d'utiliser cette technologie pour le stockage de nos notifications.

Cette base de données est également utilisée pour quelques fonctionnalités demandant la propagation de données légères à un grand nombre d'utilisateurs. C'est le cas par exemple pour la fenêtre de discussion disponible pendant les séances de création en public.

Le stockage des objets :

Comme nous avons pu le voir ci-dessus, les objets ne sont pas intégralement stockés dans une seule base de données. Nous avons en effet décidé de séparer ce que l'on peut appeler les métadonnées (collaborateurs, titre, date de création, nombre de vues …) et le contenu de l'objet. Ces deux entités sont en effet rarement utilisées simultanément et doivent répondre à des critères très différents. Les métadonnées sont par exemple souvent utilisées pour afficher un grand nombre d'objets sur le profil d'un utilisateur ou bien dans son fil d'actualités. A l'inverse le contenu d'un objet n'est nécessaire que si un utilisateur veut afficher cet objet en particulier. Nous avons donc cherché à optimiser l'organisation de notre projet en stockant les métadonnées dans une base de données *MySQL* qui permet des requêtes complexes et le contenu dans une base de données *orientée objet* qui est plus efficace quand on lui demande de renvoyer la totalité d'une entrée.

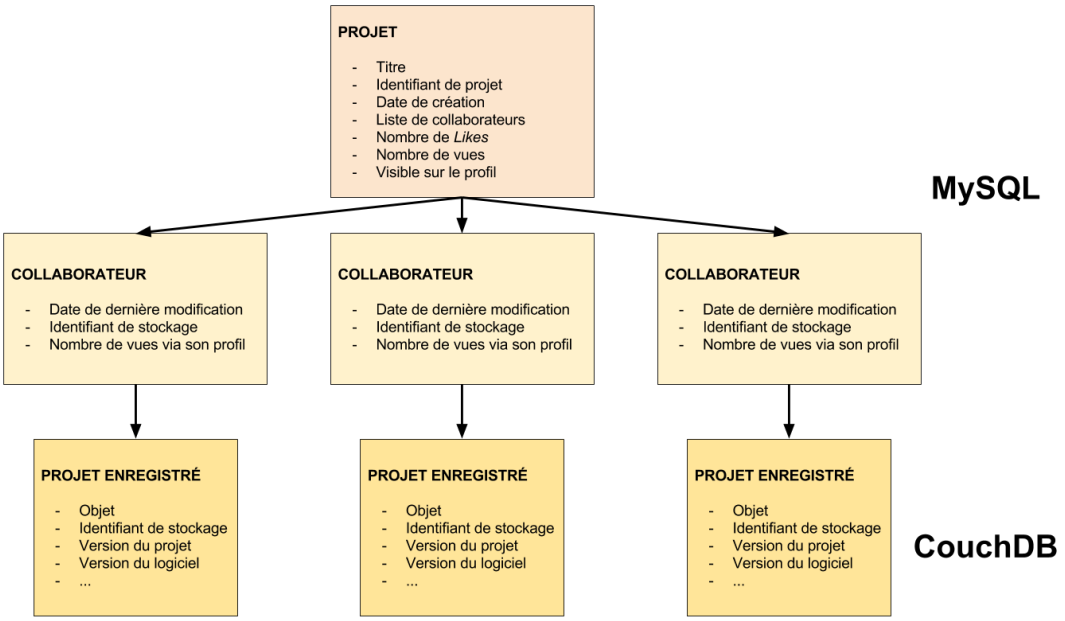


Figure 6. Organisation du stockage des objets

3. LA COLLABORATION EN DIRECT

Lors de mon arrivée, la plateforme permettait déjà de collaborer en direct sur un projet. De nombreux ajustements et améliorations ont cependant été rendus nécessaires avec l'évolution du site.

3.1 Des communications unidirectionnelles

Historiquement, les communications entre un client et un serveur sont dites *unidirectionnelles*. En effet seul le client a la possibilité d'initier une communication, le serveur étant totalement incapable d'envoyer des données au client sans avoir reçu de requête les demandant. Un des problèmes majeurs causée par cette limitation est l'incapacité pour deux clients de communiquer entre eux pas le biais d'un serveur. Imaginons que l'on souhaite créer une application de messagerie et que l'on souhaite faire transiter les messages par le serveur. La solution la plus simple serait alors de suivre le schéma de la Figure 7 :

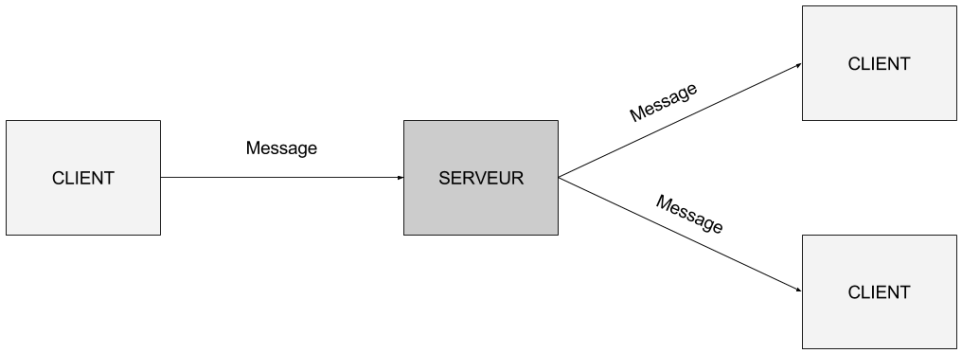


Figure 7. Modèle théorique d'envoi de message

Cette solution, bien qu'intuitive, nécessite que le serveur soit capable d'initier une communication avec le client. Un problème similaire se pose si l'on souhaite créer un système de collaboration entre deux utilisateurs. Il faut en effet que notre programme soit capable de communiquer rapidement les actions d'un utilisateur aux autres collaborateurs afin d'avoir à chaque instant des projets identiques sur chaque éditeur. L'équipe a donc dû trouver une solution pour pallier à ce problème.

3.2 La librairie Socket.io

Depuis quelques années, un nouveau protocole nommé *WebSocket* a été mis en place afin de passer outre cette limitation. Ce dernier permet d'effectuer des communications bidirectionnelles entre le client et le serveur de façon native.

Les communications bidirectionnelles étant nécessaires pour un grand nombre d'applications, des alternatives existaient déjà. La plus répandue d'entre-elles consistait à simplement envoyer une requête *Ajax* au serveur à intervalle régulier pour vérifier qu'aucune information ne devait être récupérée. Cette solution fonctionne parfaitement mais s'avère être particulièrement lourde étant donné qu'un grand nombre de requêtes envoyées sont inutiles comme le montre la Figure 8.

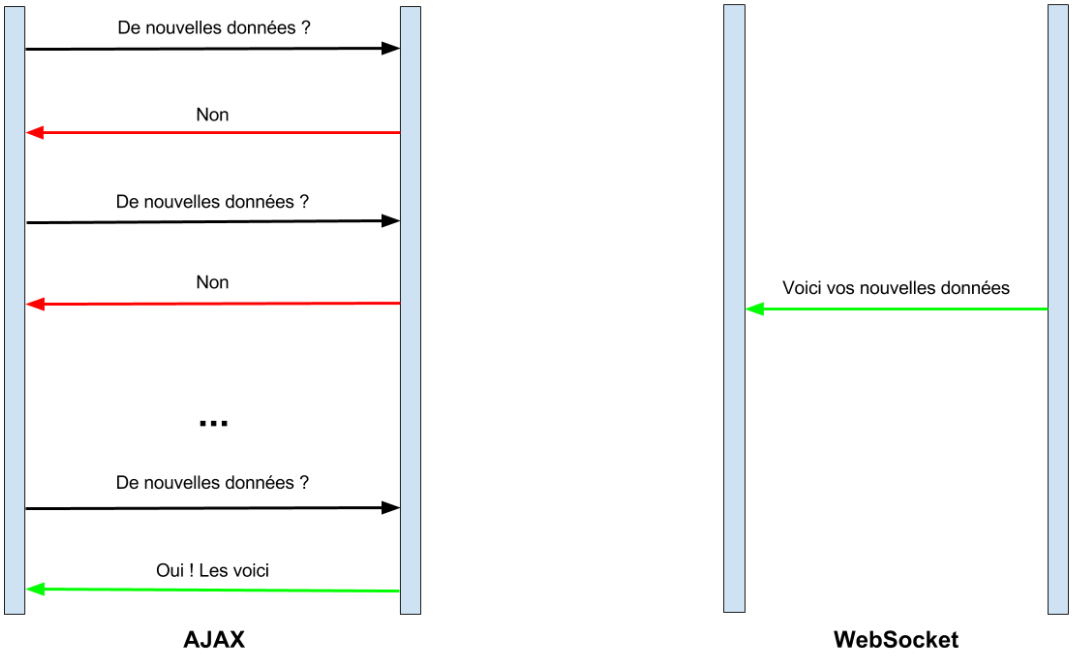


Figure 8. Fonctionnement de Ajax et du protocole WebSocket

Malheureusement, de nombreux navigateurs ne sont pas encore compatibles avec ce protocole et nous avons donc dû utiliser la librairie tierce *Socket.io*. Cette dernière facilite l'utilisation des *WebSockets* et permet d'utiliser d'autres technologies telles que *AJAX* si le navigateur du client n'est pas encore compatible avec le protocole *WebSocket*. Grâce à cette flexibilité dans l'utilisation des technologies, *Socket.io* est compatible avec des navigateurs particulièrement anciens tel que Internet Explorer 5.5.

Le fonctionnement de cette librairie est particulièrement simple et nous avons donc rapidement été capable de tirer parti de son potentiel. En effet les commandes à utiliser sont en grande partie symétriques entre le code exécuté sur le serveur et chez le client. De plus il suffit de deux commandes pour pouvoir utiliser cette librairie dans la majeure partie des cas. La première est le commande *emit* qui permet d'envoyer une requête. La seconde est la commande *on* qui permet d'exécuter une fonction à la réception d'une requête. Par exemple si l'on souhaite envoyer un message d'un utilisateur A vers un utilisateur B, il suffit d'utiliser le code suivant et d'appeler la fonction *send* chez l'utilisateur A, comme illustré sur la Figure 8 :

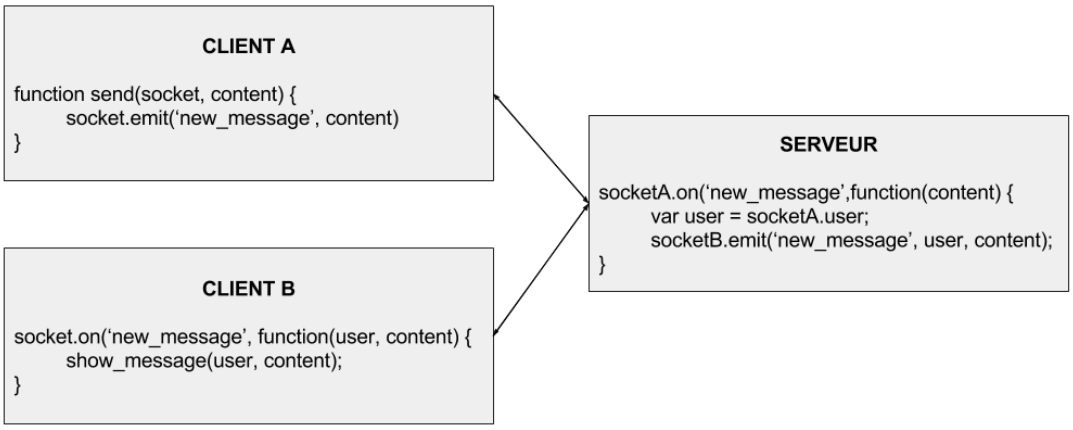


Figure 9. Envoi d'un message avec Socket.io

3.3 La transmission des actions

Un bon système de collaboration se doit de proposer un système de synchronisation automatique fiable et efficace entre les différents utilisateurs. Si un des collaborateurs effectue une action, il faut que tous les autres voient rapidement cette action répercutée sur leur propre éditeur. En analysant les différentes méthodes utilisables pour transmettre ces actions, nous avons relevé deux techniques principales :

1) Dès qu'un utilisateur effectue une modification, il envoie la totalité de son projet aux autres collaborateurs qui le rechargent sur leur éditeur.
2) Dès qu'un utilisateur effectue une modification, il envoie un message permettant d'effectuer cette même action sur les éditeurs des autres collaborateurs sans recharger la totalité du projet.

A première vue, la seconde solution est bien plus viable car plus légère en termes de bande passante et de puissance de calcul. Malheureusement il est très compliqué de construire un système de collaboration reposant uniquement sur cette méthode. En effet cette dernière ne propose aucune solution de secours en cas d'erreur lors de la transmission de l'action ou bien en cas d'actions simultanées.

La solution retenue lors de mon arrivée consistait à envoyer à la fois l'action et le projet complet à chaque modification. Dans un premier temps les collaborateurs tentaient de reproduire l'action et en cas d'erreur le projet était entièrement rechargé. Cette solution a pour principal avantage d'être optimale en terme de puissance de calcul. Elle demande cependant une forte consommation de bande passante étant donné que la moindre modification entraine l'envoi de la totalité du projet. Nous avons donc cherché à améliorer le fonctionnement de notre plateforme afin de garder une fiabilité suffisante toute en allégeant la quantité de données transitant. Plusieurs solutions sont en ce moment même envisagées et l'une d'entre elles devrait être implémentée dans les prochains mois.

Pour savoir si une action était répercutée correctement, l'utilisateur effectuant une action envoyait un *hash* de son projet aux autres collaborateurs. Ces derniers n'avaient alors plus qu'à comparer cet *hash* à un *hash* de leur propre projet une fois la modification appliquée. Si les *hashs* étaient identiques, alors l'action avait été correctement répercutée.

Ne pas envoyer la totalité du projet :

La première solution est de séparer le projet en plusieurs sections indépendantes. Si l'on prend l'exemple d'une équipe travaillant à la création d'un train en 3D, il est inutile d'envoyer les données de la locomotive pour une action concernant uniquement les wagons.

Cette solution n'est néanmoins pas parfaite. En effet elle nécessite la création de *hashs* distincts pour chaque section du projet ce qui demande une gestion plus complexe des informations sur le serveur *Node.JS*. De plus il peut être compliqué de découper un projet en plusieurs sections. En effet si un utilisateur fusionne deux objets venant de deux sections différentes, que ce passe-t-il ? Les sections sont fusionnées ? Si oui on se retrouverait rapidement avec une unique section.

N'envoyer le projet qu'en cas d'erreur :

Une autre solution serait de n'envoyer le projet qu'en cas de nécessité. Par défaut l'utilisateur enverrait uniquement l'action qu'il vient d'effectuer. Les autres collaborateurs essaieraient d'appliquer cette modification à leur projet et en cas d'échec enverraient une requête au serveur demandant la dernière version du projet pour se mettre à jour.

Cette solution est particulièrement attractive car elle permet de n'envoyer en théorie que les informations nécessaires. Il est également possible d'optimiser la récupération du projet complet en utilisant le serveur *Node.JS* pour stocker temporairement les versions complètes du projet si un premier utilisateur échoue à appliquer une action. Cependant ce n'est pas une solution parfaite et la fluidité de la plateforme risque d'être fortement affectée si une partie non négligeable des actions échouent, demandant alors une multiplication des requêtes et donc une plus forte latence.

La solution la plus aboutie est probablement une combinaison de ces modèles théoriques. Certaines actions sont en effet plus sûres que d'autres et il serait possible d'évaluer le risque d'erreur et de choisir la méthode de transmission la plus adéquate pour chaque action. Il faut également prendre en considération la charge de travail demandée au serveur, dans certains cas une solution plus lourde pour le client mais plus légère pour le serveur peut être souhaitable.

3.4 La résolution des conflits

La plateforme permet à plusieurs collaborateurs de travailler simultanément sur un projet mais également de travailler tour à tour. Il est donc nécessaire de toujours s'assurer qu'un utilisateur commençant à travailler sur un projet possède la dernière version de celui-ci.

Pour cela nous avons imaginé un arbre de possibilités essayant de couvrir le maximum d'éventualités. En cas de conflit, le site propose à l'utilisateur de continuer le projet seul ou bien d'être automatiquement synchronisé avec la version la plus avancée du projet. Une version simplifiée de cet arbre est présentée dans les annexes, cette dernière comporte l'ensemble des vérifications à effectuer lors du chargement de l'éditeur. Il ne comporte cependant pas la marche à suivre si l'utilisateur se connecte ou se déconnecte en étant sur l'éditeur. La création de cet arbre de possibilités a donc été un travail conséquent qui a demandé un grand nombre de retouches tout au long de la création de la plateforme.

CONCLUSION

Ce stage au sein du projet WeDesign a été pour moi une expérience particulièrement enrichissante. J'ai eu la chance d'effectuer mon premier séjour en entreprise au sein d'une équipe passionnée qui m'a permis d'acquérir un grand nombre de nouvelles connaissances techniques dans de nombreux domaines.

Les missions qui m'ont été confiées par l'entreprise ont été d'une grande richesse. J'ai pu en quelques mois développer des fonctionnalités allant de la création d'une API à l'amélioration de l'interface. J'ai ainsi peu me familiariser avec des technologies telles que *Node.JS, AngularJS* ou encore *Django* que j'avais déjà utilisé mais uniquement en surface. Cette diversité m'a également permis de mieux appréhender un projet dans sa globalité. Ce stage a également été pour moi l'occasion de me former au développement en équipe et aux outils tels que GIT.

L'année prochaine je souhaiterais effectuer mon stage de 4ème année en informatique théorique. J'ai eu la chance de passer ces derniers mois près de l'équipe en charge de la génération des objets 3D et j'ai été particulièrement charmé par leur travail. Etant attiré par l'informatique mais également par les mathématiques appliquées, la recherche informatique est un domaine qui m'attire particulièrement et que j'aimerai prendre le temps de découvrir.

BIBLIOGRAPHIE / REFERENCES :

Documentations :

Documentation du projet Django : https://docs.djangoproject.com/en/1.9/

Documentation de Django Rest Framework : http://www.django-rest-framework.org/

Documentation de la librairie Socket.io : http://socket.io/docs/

Documentation du langage JavaScript par le W3C : http://www.w3schools.com/jsref/

Documentation de l'API WeDesign.live

RFCs :

RFC 5849 – The OAuth 1.0 Protocol

RFC 6749 – The OAuth 2.0 Authorization Framework

Divers :

Encyclopédie en ligne *Wikipédia* – Article *Interface de programmation*

GLOSSAIRE :

AJAX (Asynchronous JavaScript and XML) : Architecture information permettant d'effectuer des appels à un serveur sans recharger la page en cours d'utilisation.

API (Application Programming Interface) : Ensemble normalisé de classes, de méthodes ou de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels.

Backend : Ensemble des programmes exécutés sur le serveur et non sur l'ordinateur du client.

CAO (Conception Assistée par Ordinateur) : Ensemble des logiciels et des techniques de modélisation géométrique permettant de concevoir, de tester et de réaliser de outils manufacturés.

Canvas : Balise HTML permettant la création de dessins en 2D et 3D dans le navigateur.

Django : Framework open-source de développement web en Python créé en 2003.

Express.js : Framework de développement de serveur en Node.js

Framework : Ensemble d'outils et de composants logiciels conçu en vue d'aider les programmeurs dans leur travail.

Frontend : Ensemble des programmes exécutés sur l'ordinateur du client et non sur le serveur.

JavaScript : Langage de programmation principalement employé dans les pages web interactives mais également pour les serveurs à l'aide entre autres de *NodeJS*.

Hash : Empreinte servant à identifier rapidement une donnée. Celle-ci est générée à partir d'une *fonction de hashage* qui a pour caractéristique principale d'être impossible à inverser. Exemples de fonctions de hashage : MP5, SHA1, SHA256

MVC (Modèle – Vue – Contrôleur) : Modèle destiné à répondre aux besoins des applications interactives en séparant les problématiques liées aux différents composants d'une application au sein de leurs architectures respectives.

MySQL : Système de gestion de bases de données relationnelles développé sous licence publique générale GNU (*GPL*).

Node.js : Plateforme logicielle libre créée en 2009 permettant la création de serveur web en JavaScript.

NoSQL : Famille de système de gestion de base de données qui s'écarte du paradigme classique des bases relationnelles. Son nom vient de l'acronyme *Not only SQL*.

OAuth : Protocole libre de délégation d'authentification permettant la sécurisation d'API

ORM (Objet-Relational Mapping) : Technique de programmation information qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle.

Requête : Message envoyé par un client vers un serveur, qui émet une réponse.

REST (Representational State Transfer) : Architecture reposant sur le protocole http et tirant parti des diverses opérations supportées nativement par http pour accéder à un ensemble de ressources.

Serveur : Dispositif informatique offrant des services à différents clients.

SPA (Single Page Application) : Application web accessible via une page web unique. Le but est d'éviter le chargement d'une nouvelle page à chaque action demandée et de fluidifier ainsi l'expérience utilisateur.

Token : Anglicisme venant de l'anglais *token* signifiant *jeton*, il désigne une solution d'authentification visant à prouver l'identité d'un utilisateur ou d'un programme grâce à la génération d'une clé cryptographique.

WebSocket : Protocole réseau visant à créer des canaux de communication bidirectionnels par-dessus une connexion TCP.

ANNEXES :

Annexe n°1 : Comparaison entre Express et Django

Annexe n°2 : Caractéristiques techniques du serveur public

Annexe n°3 : Documentation de l'API pour la section /users

Annexe n°4 : Documentation de l'API pour la section /objects

Annexe n°5 : Fonctionnement du modèle MVC

Annexe n°6 : Récupération d'un objet au chargement de l'éditeur en ligne

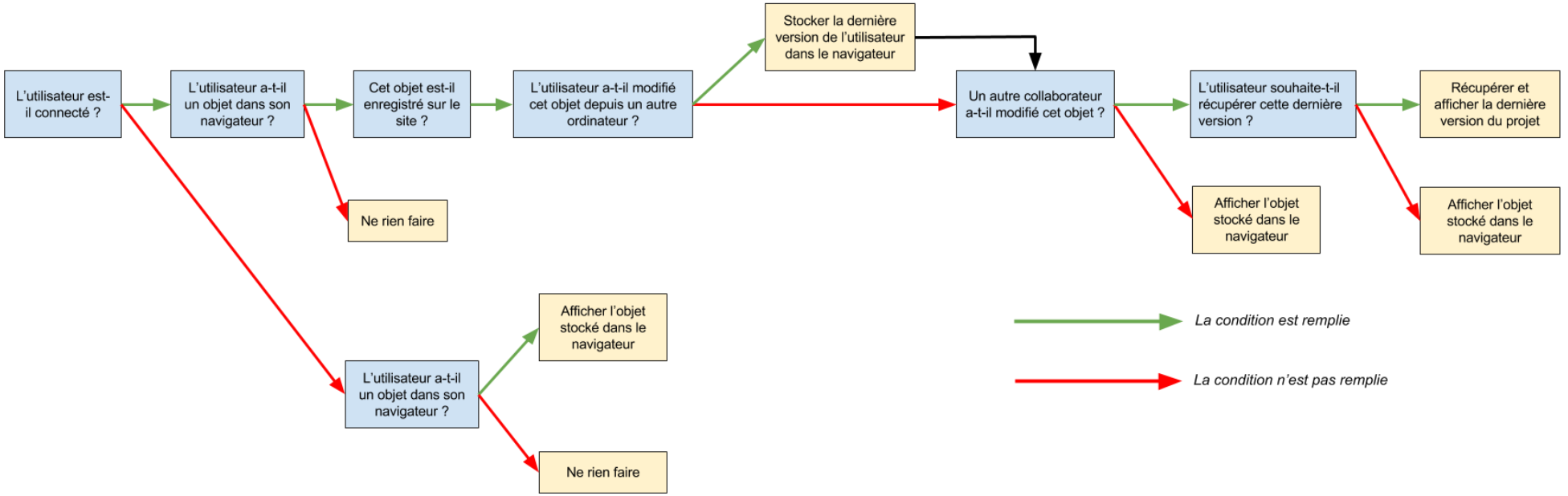
Annexe n°1 : Comparaison entre Express et Django :

	NodeJS + Express	Python + Django
License :	MIT License	BSD License
Langage de programmation :	JavaScript	Python
Support de Socket.io	OUI	NON
Bases de données :	MySQL Redis	MySQL Redis CouchDB
Paradigme de programmation :	Orienté objet Fonctionnel Evènementiel	Orienté Aspect Fonctionnel

Annexe n°2 : Caractéristiques techniques du serveur public :

	Caractéristiques
CPU	Intel Xeon E5-1650v3 6c/12t 3.5 GHz/3.8 GHz
RAM	128 GB DDR4 ECC 2133 MHz
Stockage	2 x 480GB SSD SOFT RAID
Adresses IP comprises	256 IPs
Carte réseau	1 x 1 Gbps

Annexe n°6 : Récupération d'un objet au chargement de l'éditeur en ligne



Annexe n°3 : Documentation de l'API pour la section /users

Method	PATH	Action	Sent data example	Received data example
GET	/users/	List all the users		[{ id: 1, username : "john" }, …]

GET	/users/:ID	Return the profile of a particular user		{
				id : 1,
				username : "john",
				picture_link: null,
				designer: false,
				maker: true,
				…
				}

GET	/users/:ID/profile	Return all the following request at once: - /follow/:ID - /skills/:ID/choosed/ - /machines/:ID/choosed - /users/:ID - /objects/:ID/profile/ - /collections/:ID/profile/		

GET	/users/:username/ID/	Return the ID of a particular user		{
				id : 7
				}

GET	/users/:ID/subscriptions	List the subscription of a user		[{ id : 1, name : "My Collection",

				user: 1 }, …]

POST	/users/	Create a new user	{
			username : “john”,
			password : “azerty”,
			email : john@gmail.com
			}
				{
				username : “john”,
				email : john@gmail.com,
				pk : 13
				}

POST	/users/image/	Update the profile image of a user	{
			user : 2,
			picture : <Picture>
			}	

POST	/users/tweet/	Post a message on Tweeter	{
			tweet : “Tweet example”
			}
				{
				tweet : “Tweet example”
				}

POST	/users/tweetGif	Post a gif of the current project on Tweeter	{
			message : “This is a gif”,
			gif : <Gif Content>
			}
				{
				tweet : “This is a gif”
				}

Annexe n°4 : Documentation de l'API pour la section /objects

Method	PATH	Action	Sent data example	Received data example
GET	/objects/	List every object created		[{ id : 1, title : "Castle", views : 353 }, …]

GET	/objets/:ID/	Get the information about a specific objet		{
				id : 3,
				title : "Castle",
				thumbnail : <Thumbnail in base 64>,
				parent_object_id : 2,
				comment : [<Comment>, <Comment>],
				collaborators : [<User>, <User>],
				moderators : [<User>, <User>],
				likes : 0,
				…
				}

GET	/objects/:ID/user/	List every object created by a user		[{ id : 1, title : "Castle", views : 1 }, …]
GET	/objects/:ID/update/	Get the information about the modifications of a specific object		{ last_version_user : 42,

				is_last_user : false, public : false }

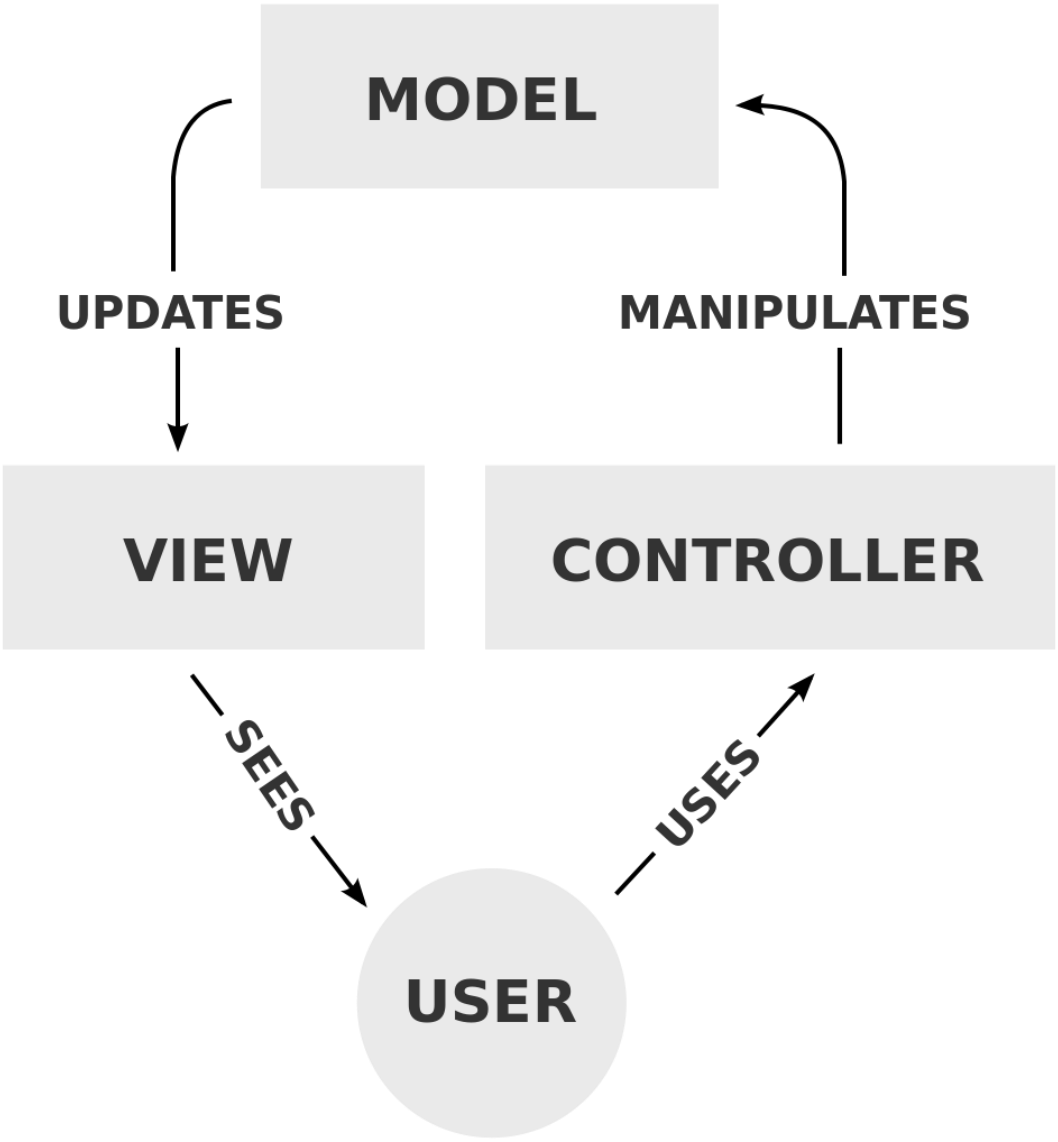
GET	/objects/:ID/content/	Get the content of a specific object		{
				object : <Object content>,
				last_update : 1460979940478,
				metadata : {
				id : 9,
				likes : 0,
				…
				}
				}

POST	/objects/vote/	Vote for an object	{
			object : 1,
			}	

POST	/objects/comment/		{
			object : 1,
			content : “My comment”
			}	

POST	/objects/collaborator/	Add a collaborator	{
			user : 2
			}	

Annexe n°5 : Fonctionnement du modèle MVC



DELANGLE Flavien – INGE1

Stage EE3b – Télécom Lille

Fiche Synoptique de synthèse

Janvier – Mai 2016

Pendant mon stage de première année de cycle ingénieur, j'ai eu la chance de travailler au sein de l'équipe en charge du projet de création et de partage d'objets imprimables en 3D *WeDesign.live* développé par la société *MyMiniFactory*. *MyMiniFactory* est une plateforme d'échange en ligne de modèles d'objets imprimables en 3D. Mon rôle était de développer et d'intégrer une API REST pour doter le projet *WeDesign.live* d'une dimension sociale. Ce stage m'a permis de suivre la création d'un projet ambitieux et d'apprendre à travailler avec une équipe de développeurs.

Une plateforme collaborative

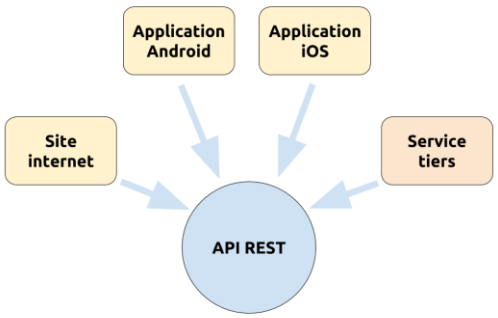
WeDesign.live cherche à rendre la collaboration entre les designers la plus fluide et agréable possible. Ces fonctionnalités ont nécessité l'utilisation de technologies innovantes telles que Socket.io et Node.JS permettant la communication bidirectionnelle entre un client et un serveur.

L'impression 3D

Le projet *WeDesign.live* s'inscrit dans une volonté de rendre l'impression 3D accessible au plus grand nombre et leur proposant une plateforme facilement utilisable et puissante. En effet, si aujourd'hui l'impression 3D souffre de nombreuses limitations techniques, il ne fait aucun doute que les prochaines années verront émerger des machines toujours plus précises et performantes.

La création d'une API REST

Les sites internet sont de plus en plus complexes et nécessite donc des architectures optimisées. Le choix d'une API permet de créer un service plus souple et plus facilement compatible avec un grand nombre de plateformes. Mon projet consistait à développer les fonctionnalités principales d'une API pour le projet WeDesign.live et à l'intégrer sur la plateforme.



Mon expérience au sein du projet *WeDesign.live* a été très enrichissant. J'ai en effet pu confirmer mon envie de travailler plus tard dans le développement web et découvrir les avantages et les contraintes d'un projet professionnel. Ce stage m'a permis de découvrir de nouvelles technologiques qui seront utiles dans ma carrière. J'ai également découvert le monde de l'impression 3D qui est passionnant et dans lequel j'espère pouvoir découvrir plus en profondeur dans les prochaines années.

Mots clés : Impression 3D – API – Développement Web – Django – NodeJS