**Présentation personnelle, parcours :**

Suite à l’obtention de mon BTS assistant de gestion en 2017, j’ai décidé d’entamer une reconversion dans le domaine du numérique. Pour cela à côté d’un apprentissage constant en autodidacte sur des technologies liées à ce secteur. J’ai commencé par un stage en tant que responsable web marketing, au sein d'une entreprise spécialisée dans la production d’objets publicitaire personnalisés à la destination de professionnels et particuliers.

Ensuite j’ai souhaité entamer une formation de chef de projet digital en alternance, sans trouver d’alternance. C’est lorsque j’avais débuté une formation en ligne de chef de projet digital, via Openclassroom que j’ai alors compris que ces 2 années de recherches étaient centrées sur le numérique et que c’est le développement web qui attisait le plus ma curiosité.

C’est ainsi que j’ai rejoint l’organisme Simplon en intégrant cette formation intensive de développeur web et mobile. Simplon met l’avant sur la pédagogie active, c’est via des projets essentiellement pratique réalisé aussi bien groupe ou qu’en autonomie qu’on développait nos compétences numériques. Un mélange efficace entre autodidacte et partage constant de connaissance, l’entraide au sein de cette promotion fut la pierre angulaire de la formation.

**Présentation de Simplon :**

Créé par Frédéric Bardeau, Andrei Vladescu-Olt et Erwan Kezzar. Simplon est un réseau de fabrique numérique ayant vu le jour et formant depuis 2013 plus de 6 200 apprenants avec un ratio de 40% de femmes et 50% d'individus non ou peu diplômés. C’est une entreprise sociale et solidaire mettant l’accent l'accès de connaissances numérique, via la formation gratuite de métiers dans le secteur informatique.

Accueillant un large portrait varié d’apprenants : Demandeurs d’emplois, personnes réfugiées, personnes en situation d’handicap, allant de moins de 25 ans à plus de 45 ans. Aucun prérequis n'est nécessaire pour les nouveaux apprenants, levant un frein et permettant l’accessibilité au plus grand nombre

Les formations Simplon sont centrées sur la pédagogie active. C’est face à des cas et des projets concrets que l’apprenant développe ses compétences et son appétence pour le secteur numérique. Avec l’appui des formateurs, devenant de plus en plus autonome, il devient maître de sa propre formation. La grande diversité sociale apporte une richesse conséquente dans le partage de connaissances et l’expérience de chacun est l’élément central des formations Simplon.

**Présentation du stage :**

**Résumé du projet :**

Etant une grande passionnée de culture populaire, et plus particulièrement d'arts créatif comme la bande dessinée dans toutes ses formes possibles (bd, comics, manga...). UniArts est un projet ayant vu le jour bien avant le début de cette formation. Son origine provient de ma volonté d'offrir plus de visibilités aux artistes francophones en mettant en place un collectif qu'ils pourraient rejoindre et quitter comme bons leur semblent. Pour cela je présenterais chacun des artistes ayant rejoint le collectif. Le but est aussi de leur offrir une plateforme leur permettant de gérer leur profil, publier et présenter des projets, poster des liens vers les autres plateformes où l'artiste est disponible.

**Etude concurrentiel :**

La naissance de ce projet n’a pas été alimenté à la suite d’une étude concurrentiel approfondie. Néanmoins j’ai beaucoup apprécié la mise en place du réseau social collaborative pour des artistes du mon entier du nom de « Daisie » mis en place par l’actrice Maisie Williams

**Compétences REAC recouvertes par le projet :**

REAC :

**1 – Maquetter une application**

**2 – Réaliser une interface utilisateur web statique et adaptable :**

**3 – Développer une interface utilisateur statique et dynamique :**

**4 – Réaliser une interface utilisateur avec une solution de gestion de contenu ou e-commerce :**

**5 - Créer une base de données :**

**6 - Développer les composants d’accès aux données :**

**7 - Développer la partie back-end d’une application web ou web mobile :**

**Présentation des spécificités techniques du projet :**

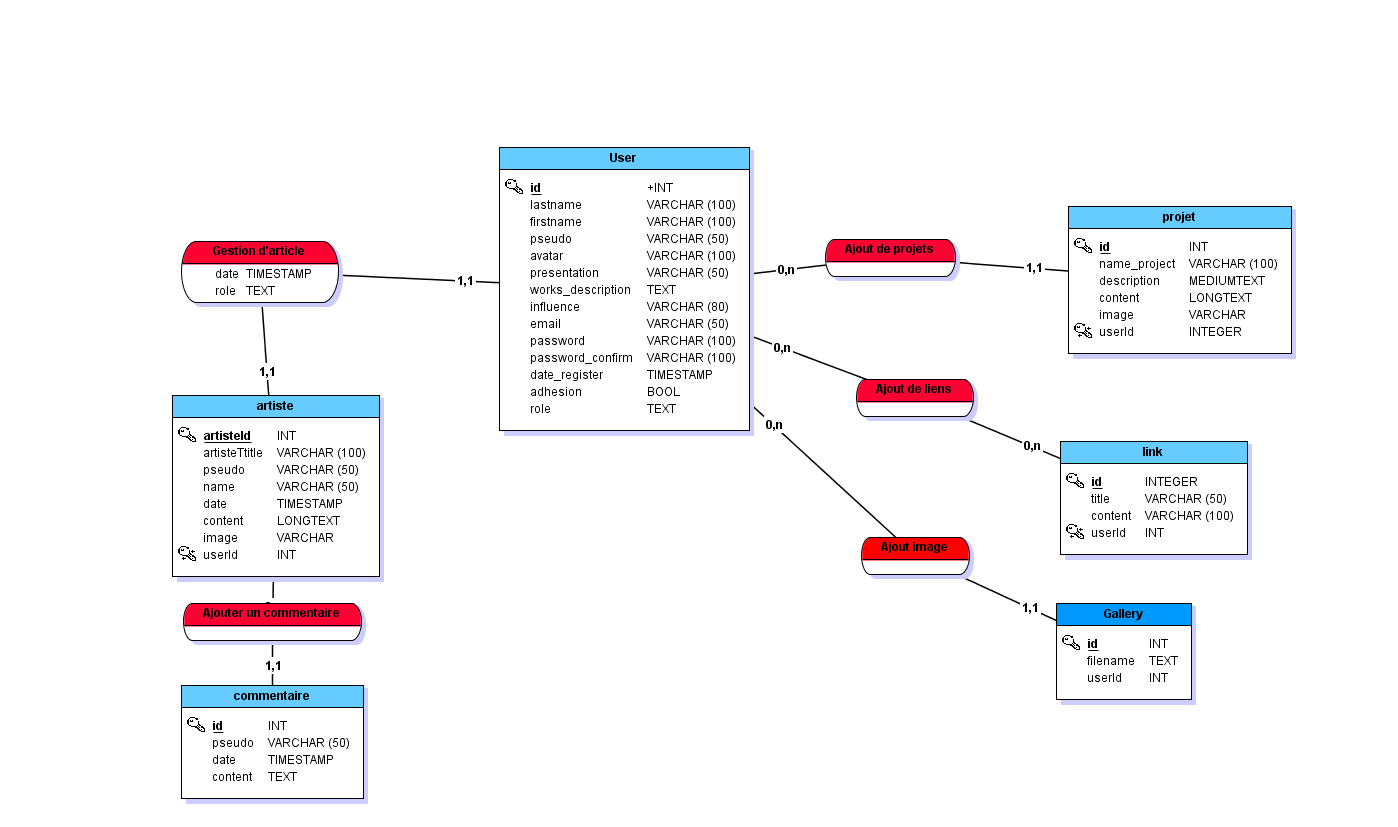
**Conception :**

**Les uses cases** :

* **Les artistes (users)** : Les artistes pourront au sein du système gérer leurs projets (add, update et delete), ajouter des liens externes pour permettre aux visiteurs de savoir où se diriger pour les soutenir ou en découvrir davantage sur ce qu’ils font. Pour cela, ils ont besoin d’adhérer au collectif en s’inscrivant via le formulaire d’inscription, l’inscription sera d’abord vérifiée par l’admin.

* **L’administrateur (users)**: l’admin pourra réaliser les articles de présentations des artistes et les gérer (add/update/delete), sur un back office il gérera également les demandes d’adhésions des artistes au collectif.
* **Les visiteurs** : les visiteurs pourront lire les articles de présentation et également les commenter, il l’en est de même pour les projets des artistes.

**Le mcd :**



Il s’agit d’une représentation d’une base de données facilement compréhensible du fonctionnement du système de base de données.

J’ai défini dans un premier temps une table Users, la différence entre l’administrateur et l’user classique qui représente l’artiste se situe au niveau du champs rôle. Le champs rôle contient deux valeurs : « Admin » et « Artistes », selon le rôle qu’aura l’user la redirection ne serait pas identique.

* Table user : Il m’a paru pertinent lors de l’élaboration de ce projet de procéder à la création d’une table unique d’utilisateurs. Elle contient toutes les informations liées à celui-ci, ce qui me permettra de différencier les différents types d’utilisations des utilisateurs est la définition d’un rôle par le champ « rôle ». Ainsi, au sein de mon serveur selon le rôle qu’obtiendra l’utilisateur il disposera d’un certain nombre d’actions :
* Un user ayant le rôle « Admin » aura la possibilité pour gérer la création/la modification/la suppression des articles de présentation des artistes. Les cardinalités indiquent que l’administrateur pourra réaliser 1 article présentant 1 artiste. Et chaque article pourra disposer d’un commentaire. Il gère également les demande d’adhésions des artistes au collectif.

* L’user qui aura le rôle « artiste » accèdera au back-office lui permettant de gérer la création/la modification/la suppression de projets. Un artiste poura publier autant de projets qu’il le souhaite, tandis que les projets sont liés qu’à un seul artiste
* Il en est de même pour la gestion des liens externes, l’artiste pourra en publier autant qu’ils le souhaitent et chacun des liens sera liés à 1 artiste. Et dispose également d’une galerie d’images

**Les wireframes & les maquettes :**

Pour la réalisation de ce projet j’ai adopté la méthode mobile first en procédant à la réalisation, en priorisant sur l’aspect mobile du projet j’ai conçu les wireframes de la version mobile/tablette et dekstop de la page home, le back-office du user et ainsi que son profil. Pour la réalisation des maquettes j’ai utilisé le logiciel Figma et Justinmind pour leurs grands nombres de fonctionnalités proposées et leur accessibilité très intuitive. Il en est de même pour les maquettes.

**La charte graphique** :

**La typographie**: Comme pour l’ensemble du design du site web, j’ai souhaité rester sur des polices sobres mais permettant tout de même d’attirer l’œil du visiteur.

* **LES TITRES**: les titres de type h1 ou h2 auront la font Teko, une police à la structure sobre se mariant correctement aux titre.
* **LES PARAGRAPHE**: pour garder cette synergie, la police font est utilisé pour les paragraphes.
* **LES MENU & BUTTONS :** les menus avec la police **source sans pro.**

**Les couleurs :**

Les deux couleurs principales ont été choisi pour garder cette cohérence dans la sobriété du site, la dernière couleur a été choisi afin de faire ressortir certains éléments du site pour les rendre plus visibles (tels que le menu, etc)

**Descriptifs techniques du projet :**

**Méthode et organisation de travail :**

Pour la réalisation de ce projet au vu de la charge considérable de la mise en place de celui-ci, l’utilisation de nombreux supports ont été primordial quant à la manière d’appréhender ce projet. Dans un premier temps j’ai procédé à la mise en place d’un Trello dont les colonnes sont organisées de la manière suivante :

* **Backlog** : Elle contient l’ensemble de mes tâches
* **A faire** : Elle contient mes tâches à réaliser
* **En cours** : Elle concerne mes tâches en courts
* **Bug/à revoir** : Elle concerne mes bugs en termes de code ou l’amélioration des spécificités du projet

J’ai défini un code couleur permettant d’établir des ordres de priorités sur les tâches, généralement tâche concerne une fonctionnalité.

En plus de cela, afin de garder un suivi constant de l’évolution du projet, je versionnais régulièrement sur Ghithub.

Au court de la conception du projet, j’éditais en parallèle l’aspect back et front du site web. Lorsqu’une fonctionnalité était opérationnelle côté serveur à la suite de plusieurs tests via potsman, je procédais à la mise en place du front de cette même fonctionnalité.

**Environnement de développement :**

**Technologies utilisées :**

**REACTJS ?**

**ExpressJS ?**

**MongoDB ?**

**Implémentation des fonctionnalités primordiales du site :**

**Le Back :**

**Mise en écoute du serveur :**

**Middleware :**

* **Express :**
* **Bodyparser :**
* **Mongoose :**
* **Cors :**
* **BaererToken :**
* **Dotenv :**
* **Multer :**
* **Path :**

**Type de bdd :**

Pour la réalisation de ce projet j’ai opté pour l’utilisation de base de données noSql de type document. Mon choix s’est porté sur ce type de données pour sa flexibilité, car elle me permet facilement de m’adapter à différents cas possibles d’utilisation de mon site web, ainsi que pour sa performance.

**Les models/schémas :** Grâce à l’utilisation de mongoose, cette dépendence/middleware facilite le traitement des données de type noSQL, notamment avec mongodb. Il permet dans le cas présent la mise en place de schémas/models qui permet de faciliter la gestion de base de données qu’on souhaite recevoir. Ainsi j’ai pu mettre en place différents models (user, projects, links, articles, comments…)

**Les contrôleurs :** on insère dans les fichiers contrôleur les function call-back des routes où on effectue les requêtes. J’ai adopté cette méthode afin d’alléger et découper mon code au niveau du serveur qui commençait à gagner en volume et donc manquer en visibilité. Ces contrôleurs effectuent donc la logique des routes définis en appelant les méthodes définis dans le fichier contrôleur.

**Contrôleurs Auth :**

Le contrôleur Auth contient la logique d’inscription et de connexions de mes routes **register/login.**  Le contrôleur est passé en fonction callback de deux routes respectives afin « d’importer » les méthodes que j’ai conçu pour la connexion et l’inscription. Afin de sécuriser la phase de connexion, j’ai procédé à la méthode la jwt (json web token), pour cela dans un premier temps j’ai créé un fichier caché qui contient mes variables d’environnement, dans le cas présent la clé secrète utile à l’utilisation de token. L’utilisation de package **dotenv** va permettre de trouver ce fichier caché, pour pouvoir ensuite grâce au **process.env** récupérer ma variable d’environnement.

**Contrôleur User :**

Le contrôleur User contient l’ensemble des méthode liés à l’utilisateur(l’artiste), Le contrôleur contient la méthode getById permettant de retourner toutes les informations de l’utilisateur.

**Contrôleur Project :** Le contrôleur Project contient toute la logique de la gestion de projet de l’utilisateur. Passé en fonction callback de l’ensemble des routes spécifiques à la gestion de projet, j’importe dans chacune d’entre elles les différentes méthodes permettant à l’user de créer ses projets. Mettant en place ainsi un CRUD complet

**Contrôleur Link :** Le contrôleur Project contient toute la logique de la gestion de projet des généré par l’utilisateur.

**Contrôleur Profile-User :** Le contrôleur ProfileUser contient les méthodes liées à la gestion du profil, plus particulièrement la gestion d’avatar avec la fonctionnalité d’ulpoad des images et de tous les autres champs liés au profile de l’utilisateur.

**LES ROUTES :**

**user/register :**

* Cette route permet aux utilisateurs (artistes) de procéder à l’adhésion au collectif en remplissant ce formulaire d’inscription. J’ai mis en place la gestion des cas d’erreurs au sein du contrôleur.
* Les cas d’erreurs :

**user/login :**

* Cette route permet à l’utilisation de s’authentifier à la suite de son inscription. Lors de la connexion, un jeton sera généré à l’aide de jwt. Celui-ci contiendra l’id du user, son rôle ainsi que la clé secrète qui est situé dans un fichier caché récupérable grâce à module dotenv.

**BACK-OFFICE(USER):**

**Gestion de projets:**

* **user/addProject:** La route addProject permet à l’utilisateur de créer/poster des projets. Je procède à la mise en place de la méthode **jwt.verify** afin de vérifier l’existence du token, dans le cas où un token n’est pas reconnu le serveur génère **un status 401**. Dans le cas contraire, je procède à l’instance du projet. Lorsqu’un projet est créé grâce à la méthode save de mongodb, si le rôle « Artiste » contenu dans le token est bien reconnu j’update par la même occasion l’User en ajoutant au champs projet l’id du projet nouvellement créé. Ainsi, dès la création d’un projet celui-ci sera automatiquement lié à l’utilisateur qu’il l’aura créé.
* **user/getAllProjects:** getAllProjects a pour but de retourner l’ensemble des projets liés à un utilisateur, l’objectif étant de pouvoir les afficher côté client pour qu’il soit visible pour tout les utilisateur du site.
* **user/updateProject:** Cette route permet à l’utilisateur de modifier son projet grâce à la métode updateOne de mongodb, en spécifiant dans le set les champs que l’utilisateur pourra modifier.
* **user/deleteProject:**  La route deleteProject permet à l’utilisateur de supprimer ses projets.

**Gestion des liens:**

* **user/addLink:** La route addLink permet à l’utilisateur de créer/poster des liens. Tout comme la méthode addProjet, je procède à la mise en place de la méthode **jwt.verify** afin de vérifier l’existence du token, dans le cas où un token n’est pas reconnu le serveur génère **un status 401**. Dans le cas contraire, je procède à l’instance d’un lien Lorsqu’un lien est créé grâce à la méthode save de mongodb, si le rôle « Artiste » contenu dans le token est bien reconnu j’update par la même occasion l’User en ajoutant au champs projet l’id du lien nouvellement créé. Ainsi, dès la création d’un lien celui-ci sera automatiquement lié à l’utilisateur qu’il l’aura créé.
* **user/updateLink:** Cette route permet à l’utilisateur de modifier les liens que l’utilisateur aura créé grâce à la métode updateOne de mongodb, en spécifiant dans le set les champs que l’utilisateur pourra modifier.
* **user/getAllLinks:** getAllLinks a pour but de retourner l’ensemble des liens liés à un utilisateur, l’objectif étant de pouvoir les afficher côté client pour qu’il soit visible pour tous les utilisateur du site.
* **user/deleteLink:** La route deleteLink permet à l’utilisateur de supprimer ses liens.

**Gestion de profil :**

* **user/updateAvatar :** Cette route permet au user de mettre à jour le champs avatar spécifié dans le model. En effet, lors de la phase d’inscription définis dans le contrôleur auth, le champs avatar ne fait pas partie des champs requis pour procéder à l’inscription.

Il faut donc à la suite de l’inscription du user, update celui-ci en mettant à jour le champs avatar. Pour cela j’assigne dans un premier temps à une variable **avatar** « req.file.filname » afin que lorsque la requête effectuer, d’enregistrer le nom du fichier téléchargé. Dans le header je précise **l’autorisation** où sera spécifié le **jeton d’authentification** du user. La méthode « findOneAndUpdate » de mongodb permet de rechercher l’id décodé du user contenu dans le token, et « $set » « modifier » le champs avatar qui contiendra le filename du fichier enregistrer.

* **user/getAvatar :** La route getAvatar permet de retourner l’avatar du user créé grâce à l’id du user.

L’upload des fichiers :

var multer = require('multer');

const storage = multer.diskStorage({

    destination: (req, file, cb) => {

        cb(null, './uploads');

    },

    filename: (req, file, cb) => {

        cb(null, file.originalname);

    } // Si il trouve que le nom d'origine existe déjà, en cas doublons

})

Pour procéder à l’upload de fichier j’ai utilisé le middleware multer qui permet d’upload des fichiers sur le serveur, dans le cas de mon projet les fichiers sont essentiellement des images (jpeg et png). La fonction ci-dessus permet dans un premier temps de vérifier l’existence du dossier upload. Dans un second temps, elle vérifie l’existence du fichier via le nom d’origine du fichier, si le nom du fichier existe déjà celui-ci n’est pas enregistré afin d’éviter les doublons d’images.

**Le Front :**

**Mise en place d’une application react :**

Dans ReactJS il existe deux types de components :

* **Les components fonction** : Ce sont des fonction Javascript représentant un composant renvoyant un élément React. Ce type de fonction accepte généralement un argument props.
* **Les components classe** : Ce sont des composants à état nécessitant obligatoirement une méthode render, permettant l’affichage du rendu des balises JSX. Cette syntaxe permet l’utilisation d’état : les states et les méthodes de cycles de vies.

**Les hooks :**

Pour le côté front de l’application, utilisant habituellement les components class. J’ai souhaité pour la réalisation de ce projet, d’utiliser le principe des hooks permettant de bénéficiers de states (d’état) et de méthodes(fonctionnalités) sans avoir recours aux habituels classes.

Mon choix s’est penché vers ce type d’utilisation de react pour sa simplicité d’écriture, sa lisibilité du code, son accessibilité et la réutilisabilité du code. Au sein de React js il est généralement recommandés de pouvoir découper son application en plusieurs components. Les hooks, permettent en plus de pouvoir découper son application en plusieurs, de découper ses composants en petites fonctions basés sur les fonctionnalités intrinsèque de ce même composant.

Par exemple les états sont contenus dans des fonctions hooks de nom de **« useState »**, les méthodes **componentDidMount**, **componentDidUpdate**, et **componentWillUnmount** retrouvent l’équivalent dans une seule et même fonction : les **« useEffects »**

**Organisation de l’application (components, routage) :**

**L’authentification des utilisateurs :**

1. **Le component Register user, l’inscription de l’utilisateur :**

Afin de récupérer la valeur des inputs du formulaires d’inscription, j’ai mis en place les states suivantes **name, pseudo, email, password, des chaînes de caractères vides qui se rempliront dès qu’un user remplira les champs souhaités.**

Afin de pouvoir récupérer la valeur des inputs du formulaires d’inscription, j’ai défini dans **une constante (values, setValues)** un hooks d’état qui contient les états suivants : **name, pseudo, email, password** définis par défaut en chaîne de caractères vide. Me permettant ainsi d’utiliser facilement ces états dans mes méthodes suivantes :

* **La méthode handleChange() :** Cette méthode me permet de cibler les inputs d’attribut name et leurs valeurs en attribuant aux constante name et value l’évènement de ciblage event.target. Ensuite je mets à jour **l’état « Values »** en créant une nouvelle entrée où j’indique que **l’attribut name est une clé** et qu’il en faut récupérer la valeur par value
* **La méthode handleSubmit() :**

Cette méthode accueille la requête fetch qui permettra de communiquer avec le serveur. Ayant indiqué côté serveur que je souhaitais traiter des donnés **JSON, je précise dans le header le content-type**. Enfin les données récupérées par la méthode **handleChange** contenu dans **le state values**, celles-ci seront changées dans le corps de la requête en chaîne JSON.

Les requêtes Fetch fonctionnent avec **le système de promesses javascript,** au sein de la première promesse j’effectue dans un premier temps **la remontée des cas d’erreurs.** La gestion de cas d’erreurs étant définis côté serveur, je remonte celle-ci côté clients afin de l’afficher aux utilisateurs. Pour cela j’utilise la méthode switch afin de définir des conditions lorsqu’un cas d’erreur est reconnu par le serveur.

* **Status 401** : Ce statut affiche à l’utilisateur que son mot de passe ne respecte pas la regex défini par le serveur.
* **Status 400 :** Ce statut affiche à l’utilisateur l’impossibilité de s’inscrire sans le remplissage de l’ensemble des champs requis.
* **Status 200 :** la requête est correcte, l’utilisateur pourra être rediriger sur le formulaire de connexion.

A chaque cas d’erreurs reconnu par le serveur, je mets à jour le state en définissant un message d’erreur personnalisés correspondant à chaque status. L’erreur s’affiche au client dans une balise p qui se remplira à chaque mise à jour du state

Enfin la seconde promesse permettra de retourner les données au format json.

J’applique l’événement **onChange** sur chacun des champs afin d’effectuer la logique de la méthode **handleChange**. L’événement **onSubmit** sur le formulaire afin de procéder à la soumission de la requête.

1. **Le component LoginUser , l’authentification de l’utilisateur:**

La logique du component LoginUser fonctionnent de la manière que le register. La différence s’effectue au sein de la requête fetch. La promesse gère le cas d’erreur où l’utilisateur se trompe de mot de passe ou d’email.

En cas de requête effectué avec succès, le token généré côté serveurà la connexion de l’utilisateur est transférer dans le localStorage. Afin de l’utiliser à chaque action effectuée par l’utilisateur (ex : la création de projet via son back-office) ou la limiation de l’accès à certains composant à l’utilisateur disposant de cette clé d’authentification.

1. **Mise en place du back office User côté Front :**

**Organisation du back-office user :**

Au sein de la fonction **hooks useEffect,** grâce à l’utilisation du token en le récupérant du localstorage j’ai pu définir que seul un utilisateur disposant d’un token (incluant donc l’id et le rôle du user), à la possibilité d’avoir accès à ce chemin : le back-office correspondant au composant **« OfficeUser »**

Le composant **OfficeUser** contient plusieurs **components enfants :**

* **OfficeUser :** Office user correspond au back office de l’utilisateur, au chargement de ce component je récupère grâce à la propriété getItem du localStorage le token de l’utilisateur. Ainsi les visiteurs ne pourront pas avoir accès à ce component. OfficeUser est découpé en trois autres components
* **OfficeProfile :** OfficeProfile représente la partie gestion de profile du back-office front.

Au chargement du component grâce à la useEffect je mets en place une requête **fetch type GET** qui va dans une première promesse envoyer **la reponse** au format json.

Dans la seconde promesse je vérifie dans une condition si l’utilisateur ne dispose pas d’avatar. Si cela est bien respecté, je lui affecte un avatar par défault en indiquant le chemin de l’image. Pour cela, je **mets à jour** le state avatar avec **setAvatar**. Le srcde l’image correspond au state avatar qui changera selon les conditions. Dans le cas où l’utilisateur dispose déjà d’un avatar, je concatène l’adresse où se situe les images enregistrées avec la réponse qui côté serveur correspond au **filename** actuel de l’image qui représente l’avatar.

La méthode **HandleChangeAvatar** permet via une requête fetch de type post de procéder à l’upload d’image pour l’avatar côté front. Dans un premier temps dans la méthode HandleChangeAvatar j’utilise le state avatar afin de récupérer la vlaleur de tyupe file par la suite.

Etant un upload de fichier, j’assigne à une variable data la méthode javascript **formData**. La propriété append de formData permet d’indiquer avec une clé et une valeur la donnée qu’on souhaite envoyer. Etant un inputfile je précise que je cible la propriété files de l’input en y récupérant qu’un seul fichier ([0]).

Enfin dans la première promesse j’envoyé la réponse au format json, ensuite dans la seconde je récupère la réponse de l’update et comme celle-ci contient le filname de l’image, je setState l’adresse du dossier uploads avec la réponse de l’ulpoad.

Enfin la méthode subFormProfile à soumettre toutes les données liées au profile de l’utilisateur. J’ai procédé à une méthode dynamique de récupération de donnée d’input de type text similaire au Register. Afin de me permettre si besoin de compléter l’aspect profile de l’utilisateur sans avoir à affecter les méthodes du component OfficeProfile.

* **OfficeProject :**

**Les states :**

* **Projects/setProjects :** Le but de cet etat est de définir un tableau vide qui accueillera l’ensemble des projets créés par l’utilisateur
* **Message/setMessages :** ce sont les messages qui seront affiché à l’utilisateur en cas de réussite ou échec d’une requête
* **File /setFile :** Cet état aura pour objectif de récupérer la valeur de l’input de type file
* **Values/setValues :** Cet état aura pour objectif de récupérer les valeurs de types textes

**HandleChangeCover** : La méthode handleChangeCover permet de récupérer la valeur de l’input de type file, en mettant à jour le state file afin de cibler la propriété file de l’input. Cela permet à l’utilisateur de définir une image de couverture illustrant son projet.

**HandleChangeProject** : La méthode handleChangeProject permet de récupérer la valeur des inputs te type texte correspondant à la création de projet.

**SubFormProject**: Contraire aux différentes méthodes que j’ai pu définir où je récupère généralement uniquement des inputs de types textes, là création de projet nécessite d’avoir la possibilité d’intégré un fichier de type file.

J’assigne donc de nouveau à une variable data la méthode **formData, puis** dans la propriété append je précise toutes les données que j’indique dans le body de la requête : la valeur des inputs de type text contenant dans le state values et cella de l’input de type file contenu dans le state File.

Dans la variable newProjects je crée une nouvelle entée dans le tableau du state projects en y insérant les valeurs des inputs. Ainsi à chaque requête effectuée je mets à jour le state Projects en précisant les newProjects.

Enfin j’effectue un map du state projects afin qu’à chaque création de projet, la fonction map générera automatiquement un component **Project en rappelant la props définit dans ce component.**

**La méthode useEffect** récupère les projets créés par l’utilisateur à chaque appel du component OfficeProject.

**Projects :** Grâce à l’utilisation du framework Material Ui, chaque component est contenu dans un modal, lorsque l’utilisateur clique sur « le button modifier » grâce à la méthode handleOpen le projet s’affiche, handleClose permet de fermer ce modal.

**Le component Projects** disposent des même méthodes **handleChangeCover**, **handleProject** du component parent afin de récupérer la valeur des inputs.

La méthode **subUpdateProject** est également similaire au **subFormProjec**t, c’est uniquement le type de requête et la concaténation de l’adresse de la requête et de l’id du projet qui diffère.

La méthode **deleteProject** permet à l’utilisateur la suppression de ses projets

* **OfficeLinks :**

**La logique du component OfficeLink** est identique au component **OfficeProject**

**Links :**

**La logique du component OfficeLink** est identique au component **Links**

* **WhiteBoard :**