Développement d'une plateforme de gestion/reporting de stock et de trésorerie

TO52

Rapport

Aurélien BROUTIN
Alexis OUKSEL

utbm
université de technologie
Belfort-Montbéliard

Table des matières

Cahier des charges

I) Expression du besoin

A) Présentation du client

Le Bureau des Festivités est une association de l'Université de Technologie de Belfort-Montbéliard aux mains de étudiants bénévoles qui gère les lieux de vie de l'UTBM et les divers événements pour divertir les étudiants. Nourriture, boissons, divertissements et d'autres services sont disponibles et vendu aux étudiants UTBM à toute heure dans le but d'enrichir leur vie étudiante.

B) Besoin du client

a) Besoin principal

Le Foyer à Belfort, la Maison des Étudiants à Sevenans et la Gommette à Montbéliard sont des lieux de vie qui nécessite une organisation et une gestion permanente pour assurer son fonctionnement : réapprovisionnement, gestion des locaux, main d'oeuvre pour les services,... Actuellement, le réapprovisionnement en consommables et la gestion des stocks sont gérés à la main grâce à des fichiers Excel, néanmoins, celle-ci est compliquée, parfois mal organisée et pas complète.
C'est pourquoi, un outil qui automatisent et regroupe tous les services nécessaire au bon fonctionnement de l'association est indispensable au vu de l'augmentation de nombres de produits vendus et pour une expansion assurée.

b) Présentation et fonctionnalités du projet

Ce logiciel sera un ERP (Entreprise Ressource Planning). **L'ERP est un progiciel qui permet de gérer l'ensemble des processus opérationnels d'une entreprise en intégrant plusieurs fonctions de gestion** : solution de gestion des commandes, solution de gestion des stocks, solution de gestion de la paie et de la comptabilité, …

L'outil à développer doit tout d'abord permettre l'accès à certaines personnes de l'association. C'est pourquoi, un système d'authentification est nécessaire.
Ensuite, l'outil doit pouvoir afficher certaines données extraites d'un fichier de données de l'association (fichier de type CSV) qui regroupe la plupart des commandes effectuées par les étudiants dans les différents lieux de vie. En effet, elles sont générées par le site de l'Association des Étudiantes, le but étant de rendre notre plateforme indépendante de celui-ci. Les données de ce fichier doivent être affichées sous forme de diagrammes et d'histogrammes pour une vision globale des analyses demandées par le client :

- Reporting général sur différents critères de tri (ex : produit le plus vendu sur les différents lieux de vie)
- Reporting des ventes et des bénéfices

L'ERP aura un module de gestion de stock permettant de savoir en temps réel les produits à approvisionner ou à vendre en priorité. Il sera couplé d'une fonctionnalité qui fournira une liste des courses générée automatiquement qui suggère les produits à acheter selon un seuil de quantité manquante.

En fournissant ces modules précédents en priorité et amélioration prévisible, un module de gestion de trésorerie sera implémenté pour faciliter au mieux les diverses tâches du trésorier de l'association. Ce module sera décrit dans un autre document et est considéré en tant que amélioration.

c) Sécurité

Le logiciel aura un seul point d'entrée avec une interface d'authentification. On estime que seul le personnel de l'association sont autorisés à accéder à la plateforme.

II) Le logiciel

A) Architecture du logiciel

Software as a Service, du français **logiciel en tant que service** ou ***software as a service*** (SaaS) est un modèle d'exploitation commerciale des logiciels dans lequel ceux-ci sont installés sur des serveurs distants plutôt que sur la machine de l'utilisateur.

Architecture d'un logiciel SaaS

Les avantages de cette architecture dans notre contexte sont :

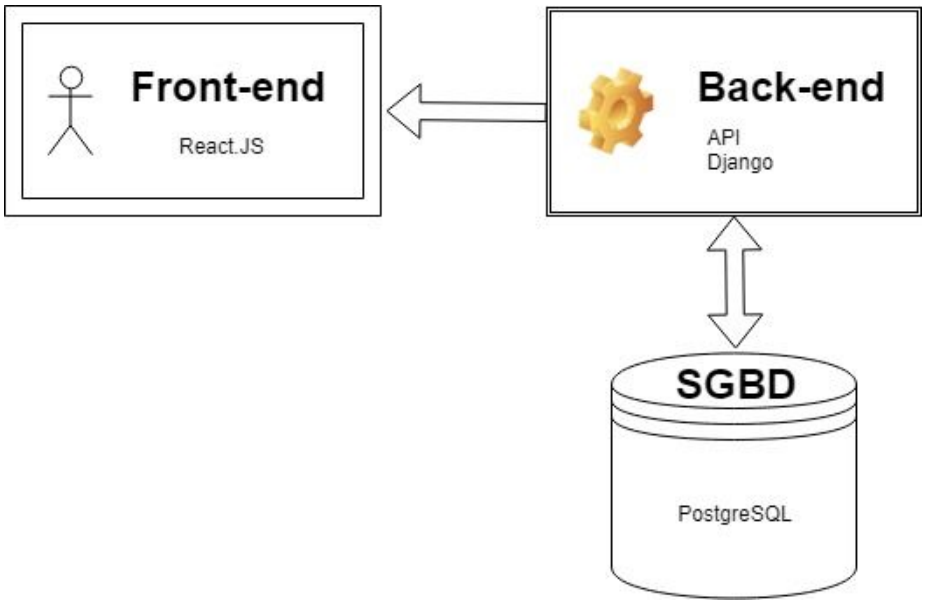
- L'hébergement du logiciel sur un serveur distant permettant l'utilisation de notre logiciel depuis n'importe quel terminal situé dans le monde disposant d'une connexion internet

- L’utilisation de la puissance des serveurs au lieu de celle d’une machine locale
- Une sécurité accrue car la localisation de l’infrastructure informatique réside chez le prestataire de locations de services de cloud computing. Si les serveurs étaient situés dans nos locaux, notre serveur serait plus exposés à des sinistres et problèmes de sécurité.
- Une évolution du logiciel facilité car l’utilisateur aura toujours accès à la version la plus à jour du logiciel
- Une réduction des coûts sur la maintenance de notre infrastructure car elle est gérée par notre prestataire

Les utilisateurs auront accès à cette plateforme depuis les lieux de vie respectifs et n’importe où le besoin d’accès est nécessaire (sur des lieux d'événement....)

B) Technologies utilisées

Les différentes technologies utilisées dans ce projet ont été choisi car elles sont très utilisées en entreprise et sont demandées en tant que compétences dans les offres d’emploi, elles sont tenus à jour régulièrement et sont les plus adaptées pour notre besoin. Connaissant les architectures des API RestFul et les frameworks front-end tel que Angular, notre but est de découvrir de nouveaux framework autant utilisés que ceux que nous connaissons avec des différents langages et types d’implémentations. Après un benchmark des différents frameworks, nous avons décidé d’utiliser Django (back-end), React.JS (front-end) et PostgreSQL (SGBD).



*

1) Back-end de la plateforme

a. Technologie du back-end

Django est un cadre de développement web source ouverte en Python. Il a pour but de rendre le développement web 2.0 simple et rapide.
Il peut être utilisée dans des scénarios d'appareil, de cloud et d'applications IoT.

Nous comptons créer un **webservice** pour gérer toutes les différentes fonctionnalités du logiciel, les interactions de l'utilisateur en front-end et du framework.

Après réunion avec le client, le système d'exploitation des serveurs de leur prestataire est UNIX. Dans ce cadre, le .NET Core ne pourra pas être utilisé pour notre projet car il requiert un environnement de type Windows pour déployer une application .NET Core avec Microsoft IIS. Nous nous sommes rabattus sur **Django, qui est un équivalent à .NET Core**.

b. Base de données

PostgreSQL est un système de gestion de base de données relationnelle et objet (SGBDRO). C'est un outil libre disponible selon les termes d'une licence de type BSD.

Nous avons choisi ce système car c'est un système open-source, gratuit et qu'il est utilisé aussi pour le site de l'Association des Étudiants, comme nous allons utiliser aussi des données provenant de leur base de données.

c. Infrastructure

Ce webservice sera hébergé soit sur un serveur de l'UTBM ou sur un serveur indépendant de l'UTBM. Cette question est toujours en cours car cette plateforme doit être toujours opérationnelle quel que soit le problème rencontré (coupure du serveur UTBM, maintenance, …)

2) Front-end de la plateforme (React.js)

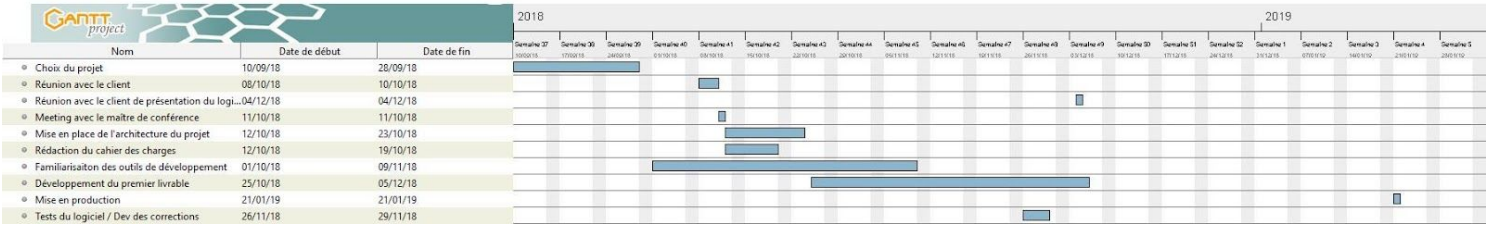
React (aussi appelé React.js ou ReactJS) est une bibliothèque JavaScript libre développée par Facebook depuis 2013. Le but principal de cette bibliothèque est de faciliter la création d'application web monopage, via la création de composants dépendant d'un état et générant une page (ou portion) HTML à chaque changement d'état.

React est une bibliothèque qui ne gère que l'interface de l'application, elle sera considéré comme la vue et comme front-end de notre application.

La version la plus stable actuellement est React 16.5.2

III) Gestion de projet

A) Diagramme de Gantt de la première version



B) Fonctionnalités de la première version

- Mise en place du système d'intégration en continu (Git)
- Mise en place de l'architecture du back-end et front-end (API et front React.JS)
- Déploiement de la version de développement sur un serveur de test
- Mise en place et conception de la base de données
- Développement d'un système de migration de données CSV vers notre base de données
- Développement d'un système d'authentification
- Développement et intégration de l'architecture visuelle de l'interface principale
- Développement du premier module de reporting

Dossier de conception

I) Diagramme Use Case

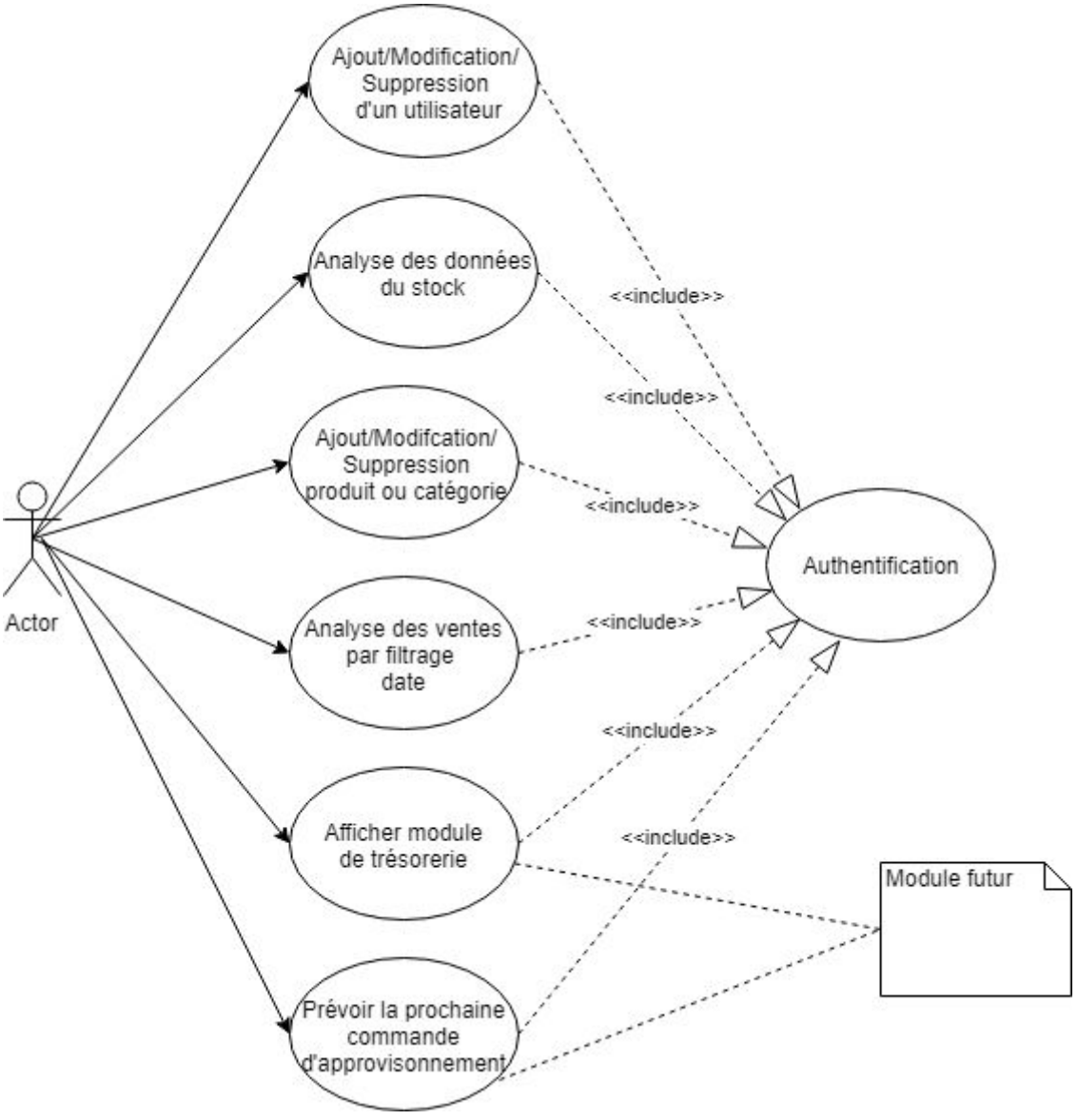


Diagramme use case de l'application

L'utilisateur devra tout d'abord s'authentifier pour pouvoir accéder à l'application. Il pourra effectuer toutes les actions qui sont cités dans le diagramme use case, sachant que celles-ci ne peuvent pas être exécutées sans authentification.

II) Diagramme de séquence

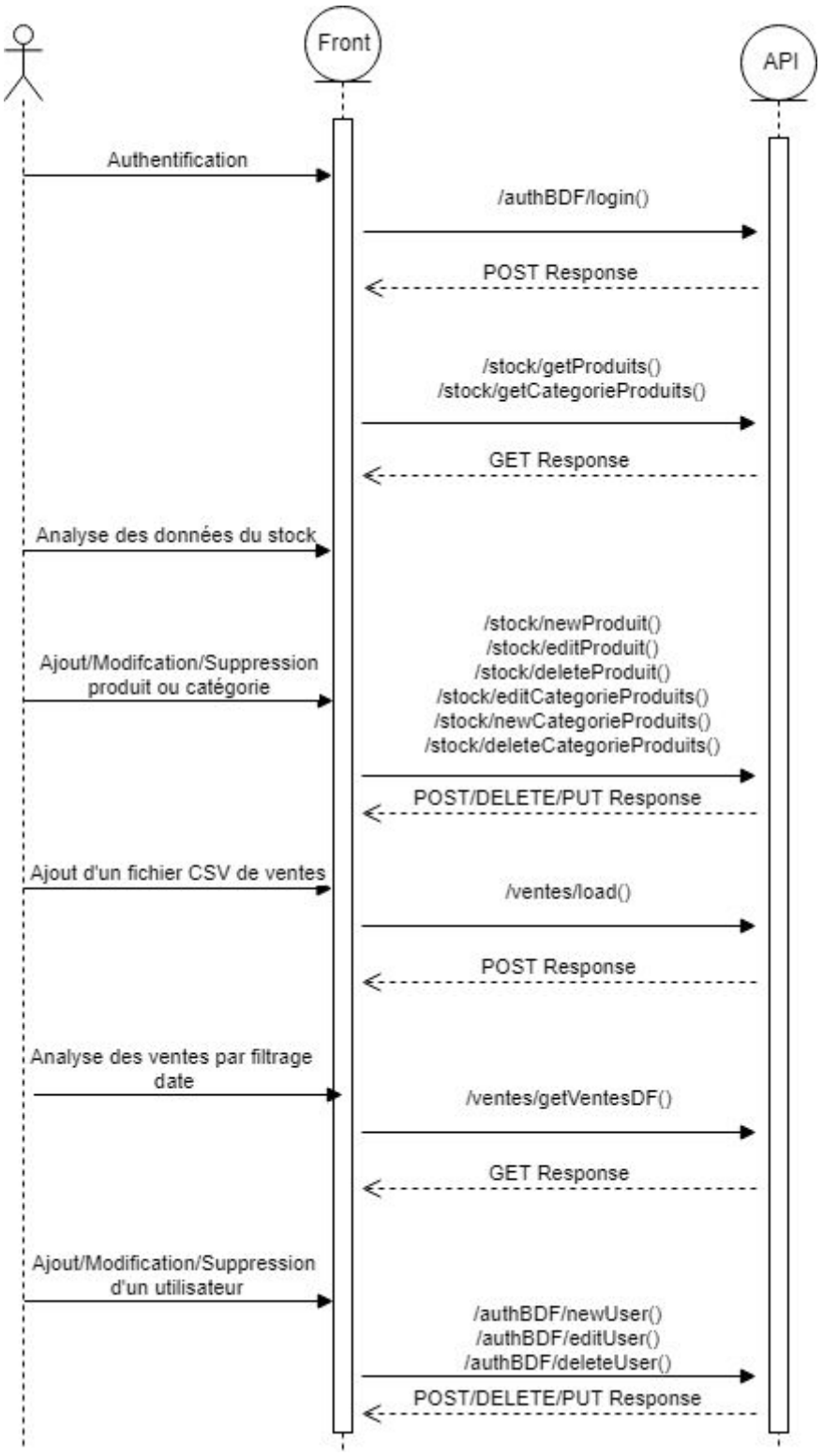


Diagramme séquence des fonctionnalités de l'application

Ce diagramme de séquence détaille tous les messages envoyées entre l'API et l'interface React par le biais des points entrées. Chaque action de l'utilisateur dans le diagramme Use Case est décrit dans ce diagramme de séquence général.

III) Diagramme de classe

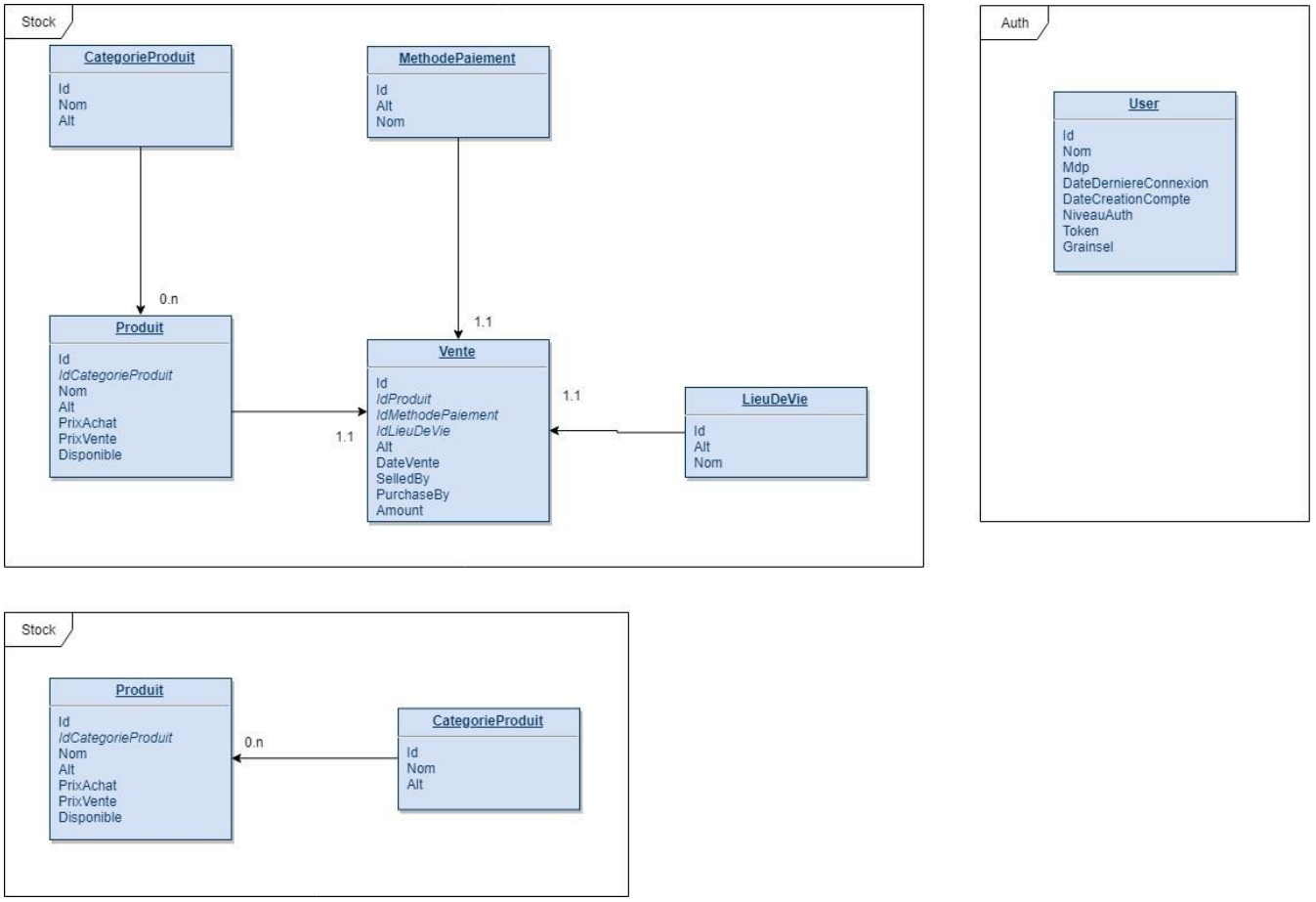


Diagramme de classe des entités

Ce diagramme représente les entités utilisées pour représenter les données en base de données dans l'API et pour pouvoir les traiter et envoyer à l'application React.

IV) Diagramme d’activité

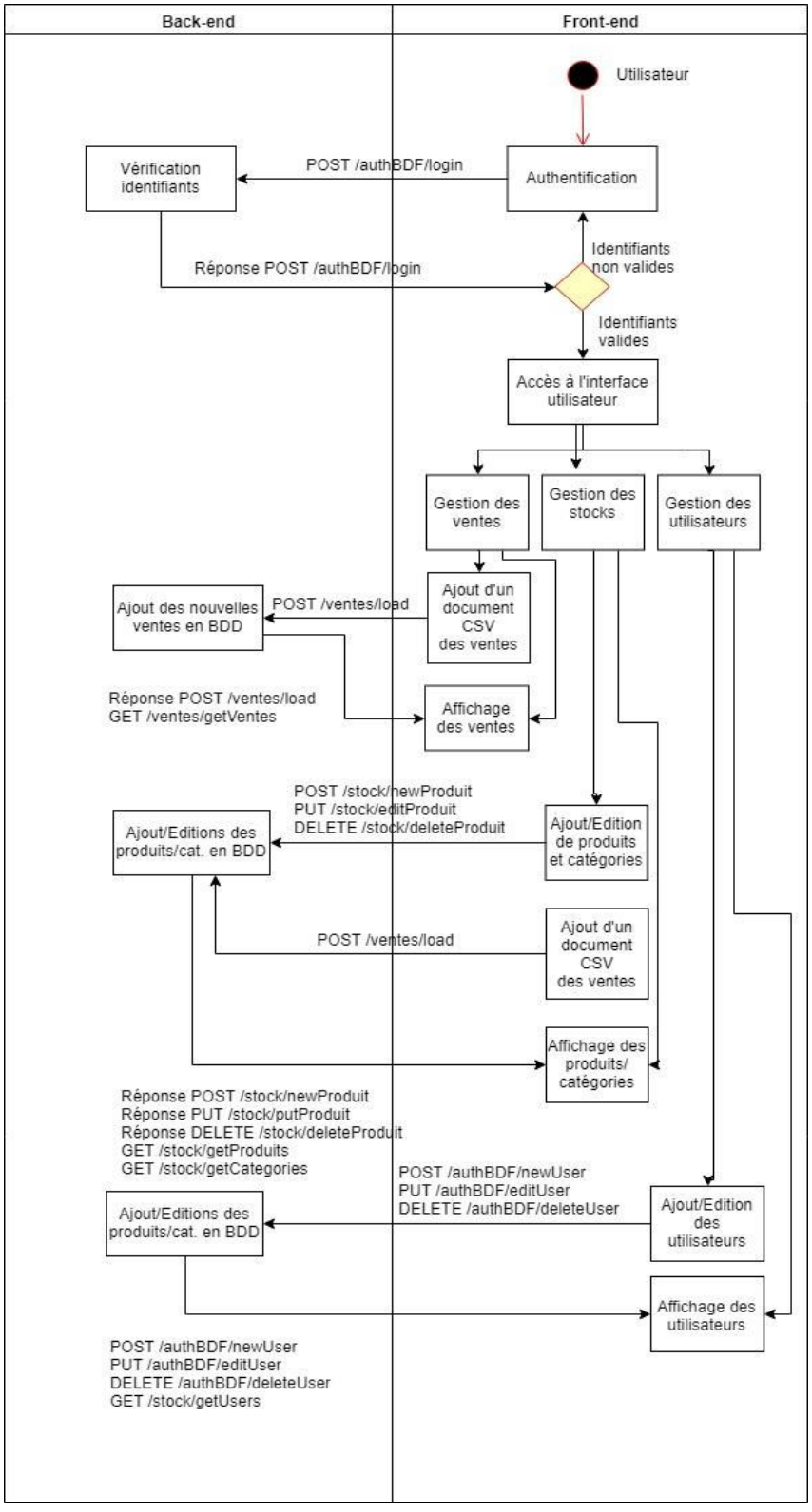


Diagramme d’activité de l’utilisation de l’application

Ce diagramme détaille en précision le fonctionnement de l'application selon les actions effectuées par l'utilisateur

V) Diagramme de déploiement

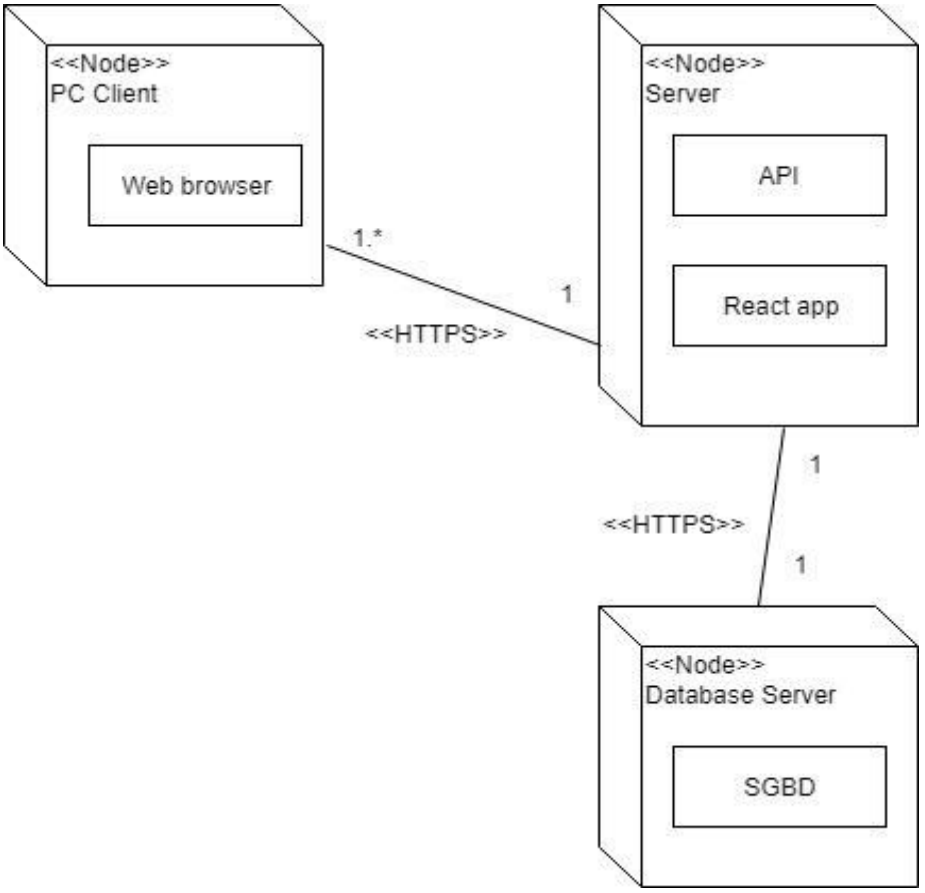


Diagramme de déploiement de l'application

On suppose que notre base de données de l'application est indépendante et est située dans un serveur autre que celui de l'application. L'API et l'application React ont besoin d'être hébergé dans un serveur et déployer sur le même serveur de préférence. Les utilisateurs pourront accéder au serveur de l'application par le biais d'un navigateur internet. Ces composants communiquent par le biais d'Internet et d'un protocole HTTPS.

Dossier technique

I) Présentation des technologies utilisées

1) Python / Django

Le langage principal utilisé pour développer la partie logique du logiciel est Python. C'est un langage de programmation objet interprété offrant des outils de haut niveau et une syntaxe facile à utiliser. Cette technologie est en plein essor dû par son utilisation multiplateforme et dans plusieurs domaines : embarquée, cloud, intelligence artificielle, mathématiques appliqués, ...

Pour des questions d'optimisations et de longévité du logiciel, nous avons utilisé la dernière version de Python (3.7.1). De plus, le déploiement d'un projet Python est facile, il sera détaillé dans la prochaine partie.

Lien de documentation : https://www.python.org/

Pour créer notre API en Python, nous avons utilisé le framework Django qui permet de développer un projet d'API facilement avec des outils déjà conçus pour éviter de "réinventer la roue" :

- S'inspire du modèle MVC
- Des outils sont déjà intégrés pour gérer la sécurité de l'API
- Un routeur est présent pour gérer les différents points d'accès plus facilement
- Propose une architecture qui favorise l'évolution du logiciel

Lien de documentation : https://www.djangoproject.com/

2) React.JS

React (aussi appelé React.js ou ReactJS) est une bibliothèque JavaScript libre développée par Facebook depuis 2013. Le but principal de cette bibliothèque est de faciliter la création d'application web monopage, via la création de composants dépendant d'un état et générant une page (ou portion) HTML à chaque changement d'état.

Elle est utilisé pour la partie front du projet et ne sert qu'à communiquer avec l'API et afficher une interface à l'utilisateur. React propose une architecture basée sur la réutilisation de composants et notre application repose sur ce concept.

Lien de documentation : https://reactjs.org/

3) PostgreSQL

PostgreSQL est un système de gestion de base de données relationnelle et objet (SGBDRO). C'est un outil libre disponible selon les termes d'une licence de type BSD. Étant open-source et gratuit, il correspond à un SGBD adapté pour notre projet.

Lien de documentation : https://www.postgresql.org/about/

Nous avons choisi ce système car c'est un système open-source, gratuit et qu'il est utilisé aussi pour le site de l'Association des Étudiants,

II) Architecture du projet

Voici un schéma de l'architecture principale notre logiciel :

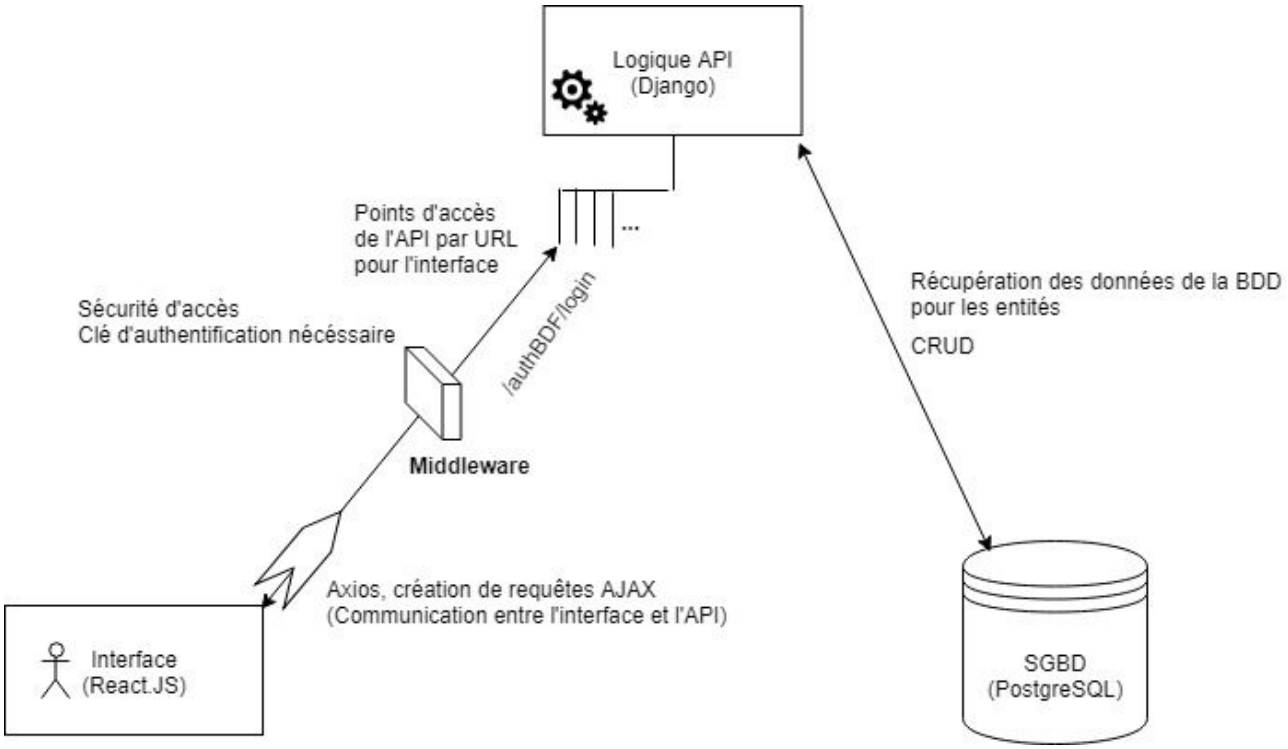


Figure 1 : Architecture principale du projet

De plus voici l'arborescence de notre projet :

Arborescence du projet

1) Architecture du projet BACK-END

Arborescence Back-end

Notre partie Back comporte plusieurs module, stock ou bien même ventes, comme le montre l'image ci dessus.

Le fichier manage.py est le fichier gérant toute la partie du back. On peut grâce à celui-ci, créer des super utilisateur pour l'interface que Django nous fournit pour gérer au mieux notre projet ou bien

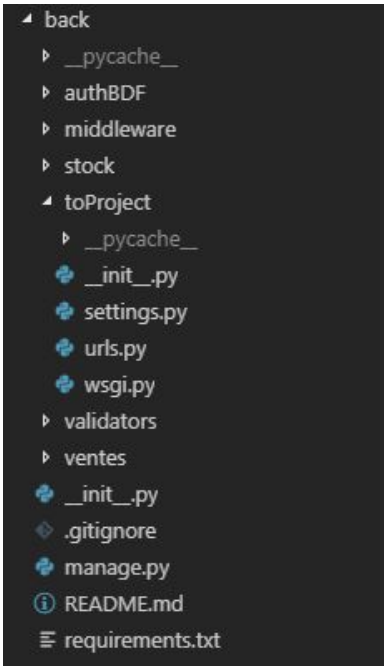
même réaliser des test unitaire sur nos points d'entrée. C'est le fichier qui permet le déploiement de la partie Back.

a) Architecture de l'API

Notre API est une interface de programmation d'application se base sur le concept d'API RestFUL (Representational State Transfer) qui fait appel à des requêtes HTTP pour obtenir (GET), placer (PUT), publier (POST) et supprimer (DELETE) des données. Elle respecte totalement les conventions et les bonnes pratiques qui sont :

- l'URI comme identifiant des ressources
- les verbes HTTP comme identifiant des opérations (POST,GET,PUT,DELETE)
- les réponses HTTP comme représentation des ressources
- Les liens comme relation entre ressources
- Un paramètre comme jeton d'authentification

Lors de la création d'un projet Django, plusieurs fichiers et répertoires sont générés :



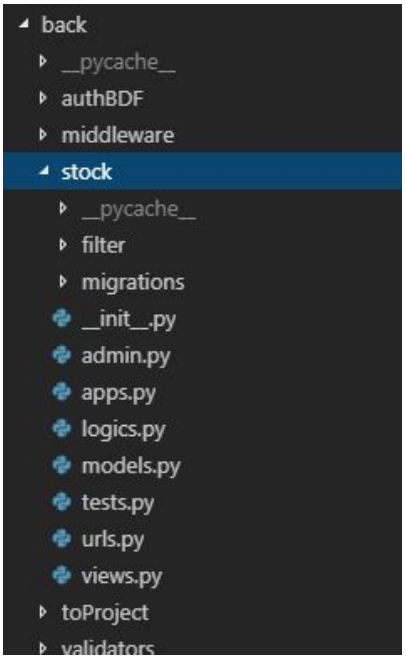
Lien de documentation de l'utilité de chaque fichier :
https://docs.djangoproject.com/en/2.1/intro/tutorial01/

Le répertoire comporte tous les fichiers de configuration du projet :

- Routeur qui permet de lier les points d'accès de l'API avec la logique des modules (urls.py)
- Configuration générale du projet, connection base de données, liaisons des modules au projet, middleware, … (settings.py)

Le fichier manage.py permet de gérer tous les aspects de l'API (déploiement, installation, …)

Afin de faciliter le développement, l'évolutivité et la maintenance, nous avons décidé de décomposer l'API en modules sous la forme de répertoire. Django permet de créer des "sous-projets", ce seront les divers modules, par exemple : la gestion des stocks.
Un module est composé et généré avec Django sous cette forme (ex : stock) :



Admin.py : Permet d'enregistrer ce module dans l'interface d'adminstration de Django
apps.**py** : Permet de configurer le nom du module
migrations/ : Dossier de sauvegarde des migrations du modèle vers la BDD
models.py : Fichier regroupant les modèles (classes) du module
urls.py : Fichier regroupant les divers point d'accès à la logique du module
tests.py : Fichier regroupant les tests unitaires du module
views.py : Non utile à notre projet car la vue est géré par React.JS

Chaque module aura ce type d'architecture de dossiers et de fichiers, ce qui permet de les rendre indépendant entre eux.

Tous les modules ont une logique de base appliqués sur le principe CRUD (Create, Read, Update, Delete), c'est à dire, pour chaque modèle, une requête de création, de modification, de récupération et de suppression est disponible afin d'avoir accès à tout l'ensemble des données si un autre front-end est développé.

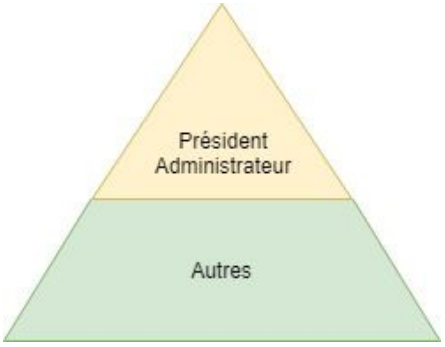
```
 7  def getProduits(request):···
17
18  def newProduit(request):···
83
84  def editProduit(request):···
150
151 def deleteProduit(request):···
```

Requêtes basées sur le concept CRUD

b) Sécurité et authentification

Un module d'authentification et de gestion d'utilisateurs a été développé pour gérer les accès depuis l'API et au front-end. En effet, il est possible de créer, modifier et supprimer un utilisateur et ses droits par le biais des différentes requêtes disponibles à ce module.

Avec le client, nous avons décidé d'établir deux niveaux de sécurité :



Pyramide de droit d'accès à l'API

Le président et l'administrateur de l'application a accès à toute l'application, les autres utilisateurs n'ont pas accès à l'interface de gestion d'utilisateurs.

Lorsqu'un utilisateur se connecte sur le point d'entrée "/authBDF/login", il doit fournir ses identifiants (login et mot de passe). L'API vérifie ces identifiants, elle renvoie une requête 403 (Accès refusé) si ils sont mauvais sinon, elle renvoie une clef d'authentification sur 30 caractères aléatoires qui devra être fourni dans l'entête de chaque requête vers l'API pour y avoir accès.

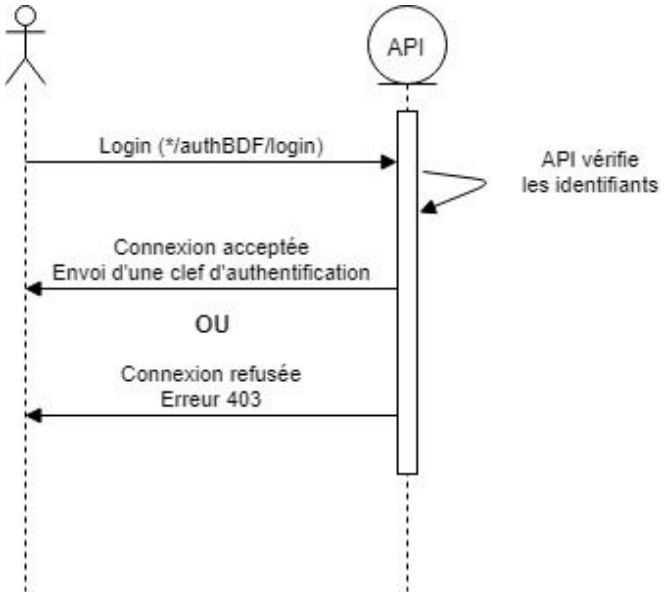


Diagramme séquence du mécanisme d'authentification

Un middleware, appelé aussi script, est un programme qui s'exécute avant la logique qui est liée au point d'entrée. Par exemple, nous avons programmé un middleware qui vérifie avant tout si la clef d'authentification est présente dans l'entête de la requête et si elle est valide.

```python
class EntryMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        # One-time configuration and initialization.

    def __call__(self, request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = self.get_response(request)


        if (request.path != "/authBDF/login" and request.path != "/admin/"):

            fields = ["HTTP_AUTHTOKEN"]
            errors = Validators.keys_are_inside_arrays(request.META, fields)

            if Validators.is_type(errors, list):
                return HttpResponseBadRequest("Vous devez insérer votre token dans le headers de votre requête")

            user = User.objects.filter(token=request.META['HTTP_AUTHTOKEN'])


            if Validators.is_not_empty(user) == False:
                return HttpResponseForbidden("Accès refusé, token invalide")



        return response
```

Exemple d'un middleware implémenté dans l'API

Ces middlewares sont situées dans le répertoire "middleware" du projet API.

De plus, lors de la création d'un compte, un grain de sel est créé. Le **salage**, est une méthode permettant de renforcer la sécurité des informations qui sont destinées à être *hachées* (par exemple des mots de passe) en y ajoutant une donnée supplémentaire afin d'empêcher que deux informations identiques conduisent à la même *empreinte* (la résultante d'une fonction de hachage). Le but du salage est de lutter contre les attaques par analyse fréquentielle, les attaques utilisant des rainbow tables, les attaques par dictionnaire et les attaques par force brute.

Le grain sel et le mot de passe en clair est donc crypté en MD5 dans la base de données pour éviter quiconque lit ces données.
Pour la vérification du bon mot de passe, le mot de passe reçu en clair et le grain de sel du compte associé est crypté, il doit donc correspondre au mot de passe crypté en base de données.

c) Points d'entrée de l'API

Chaque requête nécessite d'avoir comme entête "authtoken" avec en valeur la clef d'authentification, à l'exception du point d'entrée d'authentification, et le corps de la requête doit être sous la forme d'un JSON.

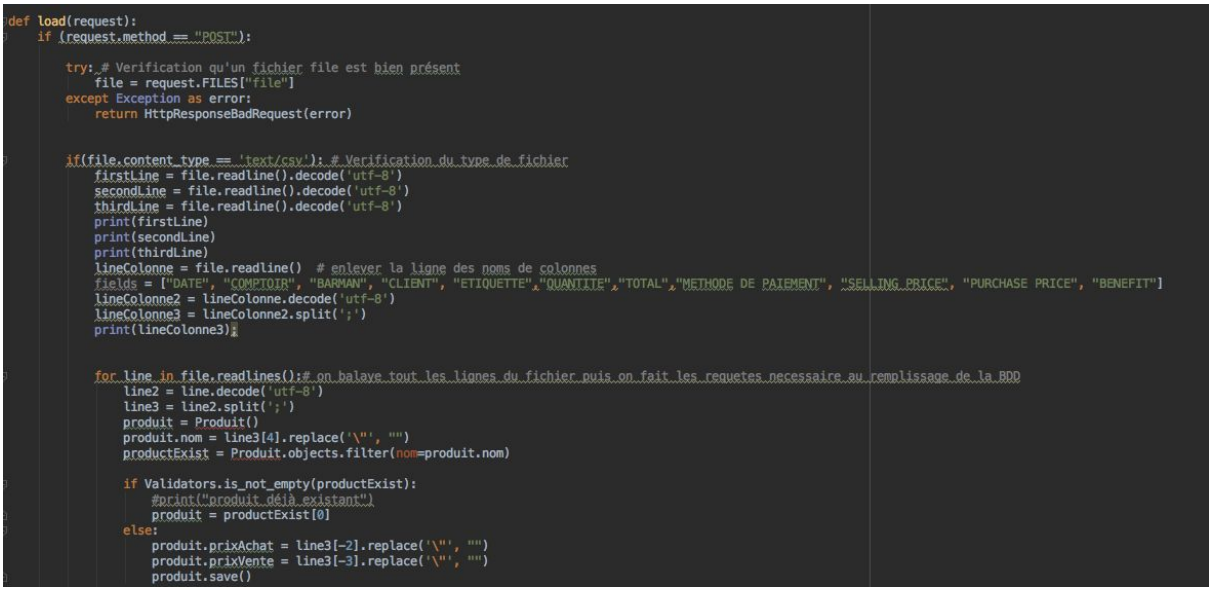
Si un paramètre est manquant à la requête, l'API renverra les champs manquants.
Si le type de requête ne correspond pas au bon type du point d'entrée, l'API renverra une erreur.

Module d'authentification :

Point d'entrée	Type de requête	Paramètres	Description
/authBDF/login	**POST**	**login** : nom de l'identifiant mdp : mot de passe de l'identifiant	Permet l'authentification à l'API
/authBDF/newUser	**POST**	**login** : nom du nouvel utilisateur **mdp** : mot de passe du nouvel utilisateur niveauAuth : niveau d'authentification (1 ou 2)	Permet de créer un nouvel utilisateur
/authBDF/deleteUser	**DELETE**	**login** : nom de l'utilisateur à supprimer	Permet de supprimer un utilisateur
/authBDF/editUser	**PUT**	**login** : nom de l'utilisateur existant **mdp** : mot de passe de l'utilisateur existant **niveauAuth** : niveau d'authentification (1 ou 2)	Permet de modifier un utilisateur existant
/authBDF/getUsers	**GET**	Aucun paramètre	Permet de récupérer la liste des utilisateurs

Module de gestion des ventes :

Point d'entrée	Type de requête	Paramètres	Description
/ventes/load	**POST**	**file: fichier csv**	Permet de lire le fichier csv donnée pour ainsi remplir la base de donnée
/ventes/getVentesDF	**GET**	**dateDébut :** Date de début de filtrage **dateFin :** Date de fin de filtrage	Permet d'extraire les ventes enregistrées selon un intervalle de temps



```python
def load(request):
    if (request.method == "POST"):

        try: # Verification qu'un fichier file est bien présent
            file = request.FILES["file"]
        except Exception as error:
            return HttpResponseBadRequest(error)


        if(file.content_type == 'text/csv'): # Verification du type de fichier
            firstLine = file.readline().decode('utf-8')
            secondLine = file.readline().decode('utf-8')
            thirdLine = file.readline().decode('utf-8')
            print(firstLine)
            print(secondLine)
            print(thirdLine)
            lineColonne = file.readline()  # enlever la ligne des noms de colonnes
            fields = ["DATE", "COMPTOIR", "BARMAN", "CLIENT", "ETIQUETTE","QUANTITE","TOTAL","METHODE DE PAIEMENT", "SELLING PRICE", "PURCHASE PRICE", "BENEFIT"]
            lineColonne2 = lineColonne.decode('utf-8')
            lineColonne3 = lineColonne2.split(';')
            print(lineColonne3)


            for line in file.readlines():# on balaye tout les lignes du fichier puis on fait les requetes necessaire au remplissage de la BDD
                line2 = line.decode('utf-8')
                line3 = line2.split(';')
                produit = Produit()
                produit.nom = line3[4].replace('\"', "")
                productExist = Produit.objects.filter(nom=produit.nom)

                if Validators.is_not_empty(productExist):
                    #print("produit déjà existant")
                    produit = productExist[0]
                else:
                    produit.prixAchat = line3[-2].replace('\"', "")
                    produit.prixVente = line3[-3].replace('\"', "")
                    produit.save()
```

```python
                lieuDeVie = LieuDeVie()
                lieuDeVie.nom = line3[1].replace('\"', "")
                lieuDeVieExist = LieuDeVie.objects.filter(nom=lieuDeVie.nom)

                if Validators.is_not_empty(lieuDeVieExist):
                    #print("lieuDeVie déjà existant")
                    lieuDeVie = lieuDeVieExist[0]
                else:
                    #print("existe pas")
                    lieuDeVie.save()

                vente = Vente()
                vente.dateVente = line3[0].replace('\"', "")
                vente.idProduit = produit
                vente.idLieuDeVie = lieuDeVie
                vente.selledBy = line3[2].replace('\"', "")
                vente.purchaseBy = line3[3].replace('\"', "")
                vente.pruchaseBy = line3[5].replace('\"', "")
                vente.amount = line3[6].replace('\"', "")
                vente.save()

            return HttpResponse(vente)
        return HttpResponseBadRequest("Fichier incompatible ! Veuillez upload un fichier CSV")
    else:
        return HttpResponseForbidden("Demande refusée")
```

Code du parseur de fichier CSV

Explication du code:

Le code doit pouvoir parser notre fichier csv pour remplir notre base de donnée. Pour cela nous devons d'abord vérifier qu'un fichier est bien transmis dans la requête est qu'il est bien de type CSV. Puis nous devons vérifier les trois premières lignes du fichier car celles-ci ne ressemblent pas au reste du fichier..
Ensuite nous devons tester si le fichier respecte le format suivant :
fields = ["DATE", "COMPTOIR", "BARMAN", "CLIENT", "ETIQUETTE","QUANTITE","TOTAL","METHODE DE PAIEMENT", "SELLING PRICE", "PURCHASE PRICE", "BENEFIT"]
Puis nous devons balayer chaque ligne pour remplir notre base en vérifiant de pas créer de doublons.

La récupération des ventes par filtrage par date consiste à récupérer toutes les ventes et récupérer toutes les ventes situées entre les dates envoyées dans les paramètres de la requête

```
def getVenteDF(request):
    reponse = []
    if(request.method == "POST"):


        data = Validators.is_valid_json(request.body)

        if data == False:

            return HttpResponseBadRequest("Probleme data")
        else:
            dateDebut = data["DateDebut"]
            dateFin = data["DateFin"]

            allVente = Vente.objects.all()
            for vente in allVente:
                if dateDebut <= vente.dateVente <= dateFin :
                    reponse.append(Vente_to_json(vente))
```

Module de gestion des stocks :

Point d'entrée	Type de requête	Paramètres	Description
/stock/newProduit	**POST**	**categorieProduit, nom, prixAchat, prixVente, disponible** : données à liées au nouveau produit	Permet de créer un nouvel produit
/stock/deleteProduit	**DELETE**	**login** : nom de l'utilisateur à supprimer	Permet de supprimer un produit existant
/stock/editProduit	**PUT**	**id** : clé primaire du produit existant **categorieProduit, nom, prixAchat, prixVente, disponible** : données à modifier au produit existant	Permet de modifier un produit existant
/stock/getProduits	**GET**	Aucun paramètre	Permet de récupérer la liste des produits

d) Tests unitaires

Chaque module à son fichier de tests unitaires (tests.py) qui permet de tester les différents points d'entrée.

```python
class AuthBDFTests(TestCase):
    def setUp(self):
        # Every test needs access to the request factory.
        self.factory = RequestFactory()

        # Create an instance of a GET, POST, DELETE, PUT request.
        self.requestGet = self.factory.get('/authBDF/getUsers')
        self.requestPost = self.factory.post('/authBDF/getUsers')
        self.requestPut = self.factory.put('/authBDF/getUsers')
        self.requestDelete = self.factory.delete('/authBDF/getUsers')
```

Structure d'un test unitaire avec son constructeur

Nous agençons les fichiers de tests sous cette forme :

- Une classe au nom du module (**NomModuleTests**)
- Une factory qui simule les requêtes vers les points d'entrées
- Une importation du module de test de Django (TestCase)

```python
def test_editUser(self):
    # Request without body
    self.assertEqual(type(logics.editUser(self.requestPut)), type(HttpResponseBadRequest()))


    # Request with wrong json format
    jsonText = str("{kk : pk)")
    self.requestPut._body = jsonText
    self.assertEqual(type(logics.editUser(self.requestPut)), type(HttpResponseBadRequest()))

    # Request with wrong json inputs
    jsonText = str(json.dumps({"badjson" : "badjson"}))
    self.requestPut._body = jsonText
    self.assertEqual(type(logics.editUser(self.requestPut)), type(HttpResponseBadRequest()))

    # Request with unknown user
    jsonText = str(json.dumps({"login" : "whoknowswhoIam", "mdp" : "mdppp", "niveauAuth" : "1"}))
    self.requestPut._body = jsonText
    self.assertEqual(type(logics.editUser(self.requestPut)), type(HttpResponseBadRequest()))

    # Request with str instead of int in niveauAuth
    jsonText = str(json.dumps({"login" : "user", "mdp" : "mdppp", "niveauAuth" : "niveau"}))
    self.requestPut._body = jsonText
    self.assertEqual(type(logics.editUser(self.requestPut)), type(HttpResponseBadRequest()))

    # Others requests than PUT
    self.assertEqual(type(logics.editUser(self.requestPost)),type(HttpResponseForbidden()))
    self.assertEqual(type(logics.editUser(self.requestGet)),type(HttpResponseForbidden()))
    self.assertEqual(type(logics.editUser(self.requestDelete)),type(HttpResponseForbidden()))
```

```python
def test_getUsers(self):
    # Others requests than GET
    self.assertEqual(type(logics.getUsers(self.requestGet)), type(HttpResponse()))
    self.assertEqual(type(logics.getUsers(self.requestPost)),type(HttpResponseForbidden()))
    self.assertEqual(type(logics.getUsers(self.requestPut)),type(HttpResponseForbidden()))
    self.assertEqual(type(logics.getUsers(self.requestDelete)),type(HttpResponseForbidden()))
```

Exemple de tests unitaires

En général, nous testons un point d'entrée en simulant une requête selon plusieurs critères :

- le type de requête
- le format du contenu de la requête qui doit être JSON
- le contenu de la requête

Pour lancer les tests unitaires, il faut exécuter cette commande à la racine du projet API grâce au script python manage.py :

$ python manage.py test

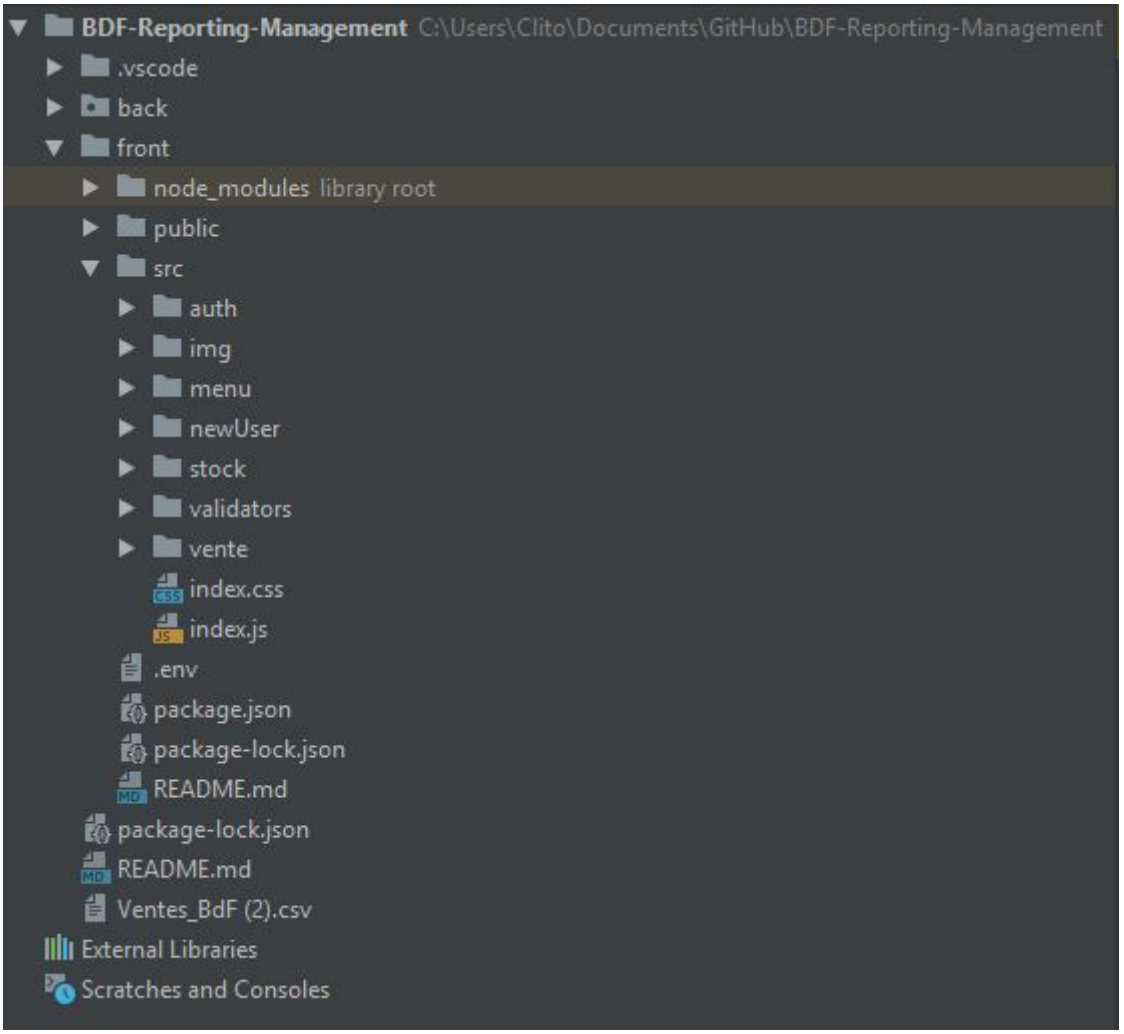
e) Validation des données

Une classe Validators a été mise en place pour pouvoir vérifier l'intégrité des données reçus de la part de l'utilisateur (dans le répertoire /validators). Elle est utilisable dans tout le projet, facilite la lecture du code et évite la répétition du code.

```python
from django.core.exceptions import ValidationError
from django.http import HttpResponseForbidden
import hashlib, json

class Validators:

    @staticmethod
    def is_type(value, value_type):
        """
        Check if value is certain type
        :param value: It can be list or a single value of anything
        :param value_type: For which python type are we testing the type of value (str, int, float, list...)
        :return: True if value is that type, False if not
        """
        if isinstance(value, value_type):
            return True
        else:
            return False

    @staticmethod
    def is_not_empty(value):
        """
        Check if value empty
        :param value: It can be list or a single value of anything
        :return: True if value is not empty, False if empty
        """
        if not value or value == "":
            return False
        else:
            return True
```

Exemple de fonctions de la classe Validators

2) Architecture du projet FRONT-END

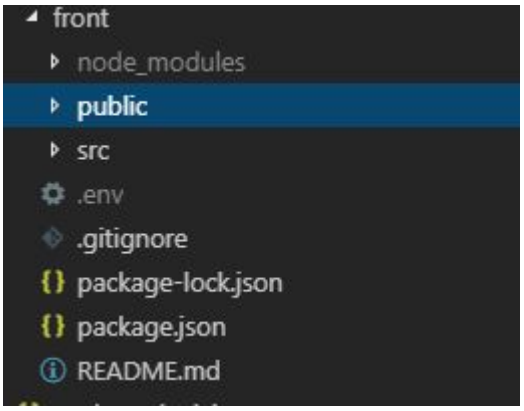


Arborescence du Front-end

Notre partie Front comporte plusieurs modules (menu ou bien même vente) comme le montre l'image

a) Architecture de l'application React

L'architecture de notre application React est aussi basé sur des modules, c'est-à-dire, chaque répertoire dans /src correspond à une fonctionnalité mais aussi à une interface.



public/ : Correspond au répertoire de la page HTML statique de l'application
src/ : Répertoire où est situé tout le code source des modules et du point d'entrée (index.js)
package.json : Fichier de configuration des dépendances
.env : Fichier de variable d'environnement

Un module se compose d'un fichier JS, d'un fichier CSS et un répertoire img. Un composant React est le fichier JS qui comporte la logique du module et le code HTML (situé dans le fichier JS dans la fonction render()). Ce module sera exportable et réutilisable avec toute sa logique en l'invoquant grâce à une balise HTML avec le nom de celui-ci sous-cette forme : **<Menu />**

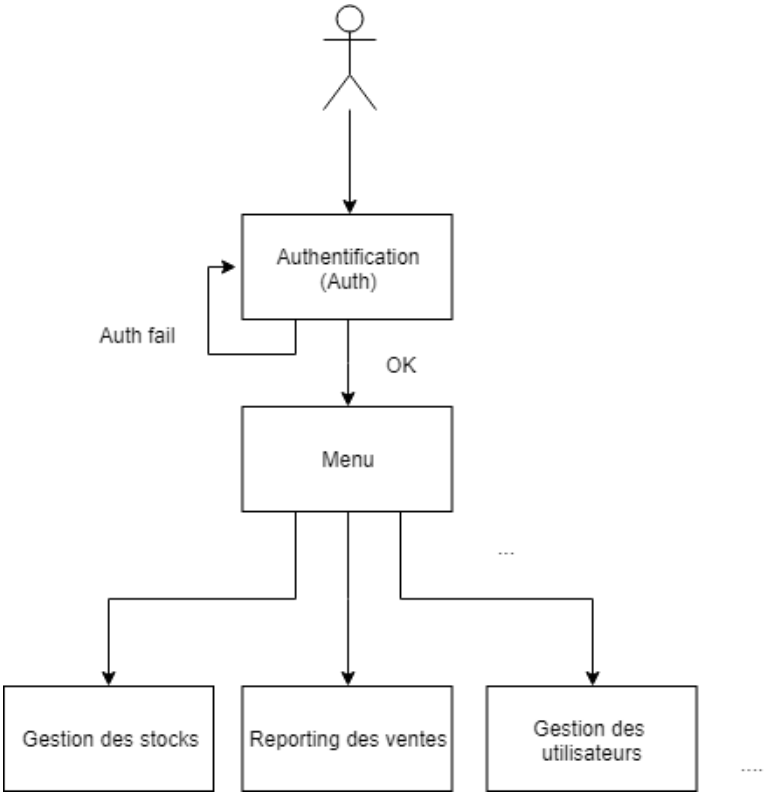
```
import './Menu.css';

class Menu extends Component {

    constructor(props) {···
    }

    display(menu) {···
    }

    // ---------------------- VUE HTML ---------

    render() {
        return (
            <div className="Page"> ···
        )
    }

}

export default Menu;
```

Exemple de composant React (Menu.js)

L'application possède d'un premier point d'accès qui est une interface d'authentification. Ce point d'accès est géré dans le fichier src/index.js. Si l'utilisateur se connecte, il aura accès à l'interface interne sinon il restera sur cette interface. Le menu est statique et génère le contenu des modules sur la page selon le choix de l'utilisateur ci dessus.

```
9
10  if (sessionStorage.getItem('token')) {
11      ReactDOM.render(<Menu />, document.getElementById('root'));
12  } else {
13      ReactDOM.render(<Auth />, document.getElementById('root'));
14  }
15
```

Code du fichier index.js



Architecture de l'accès à l'application

En effet, chaque composant (fichier js qui possède aussi le code html) possède ses propres "variables globales" qu'on appelle "**state**". Chaque fois que l'on veut modifier ces states, il faut utiliser la fonction "**setState**" de React car elle permet de mettre à jour la page HTML avec les nouvelles données si ces states ont été intégrés dans la page HTML (Menu.js) :

```
}

display(menu) {
    switch (menu) {
        case "ventes":
            this.setState({display : <Ventes />});
        break;
        case "stock":
            this.setState({display : <Stock />});
        break;
        case "user":
            this.setState({display : <NewUser />});
        break;

        default:
            break;
    }
}
```

Ici en cliquant sur le menu Ventes, <Ventes /> va être mis dans le state et cette balise contient tout le fichier Ventes.js et l'HTML qui est compris aussi dans ce fichier. C'est manipulant ces states que nous pouvons afficher des données reçus de l'API et construire notre HTML. Un exemple avec l'affichage des produits :

```
showProducts(filter) {


    let table = []

    if (Validators.isDefined(this.state.products.data)) {

        for (let i=0; i < Object.keys(this.state.products.data).length; i++) {
            let product = this.state.products.data[i];
            let dispoimg = (product.disponible) ? <img src={dispo} /> : <img src={notdispo} />;

            if (!Validators.isDefined(filter) || filter == "") {
                table.push(
                    <tr>
                        <td> {product.nom} </td>
                        <td> <span style={{backgroundColor: product.categorieProduit.colorGraph}} className="CategorieProduct">{product.categorieProduit.nom}</span> </td>
                        <td> {product.prixAchat} </td>
                        <td> {product.prixVente} </td>
                        <td> {dispoimg} </td>
                        <td> {product.quantite} </td>
                        <td>
                            <Button color="warning" onClick={this.showModal.bind(this, 'modalEditProduct', product)}><img src={modify} /></Button>{' '}
                            <Button color="danger" onClick={this.showModal.bind(this, 'modalDeleteProduct', product)}><img src={deleteimg} /></Button>{' '}
                        </td>
                    </tr>);
            } else if (product.nom.toLowerCase().indexOf(filter.toLowerCase()) !== -1) {
                table.push(
                    <tr>
                        <td> {product.nom} </td>
                        <td> <span style={{backgroundColor: product.categorieProduit.colorGraph}} className="CategorieProduct">{product.categorieProduit.nom}</span> </td>
                        <td> {product.prixAchat} </td>
                        <td> {product.prixVente} </td>
                        <td> {dispoimg} </td>
                        <td> {product.quantite} </td>
                        <td>
                            <Button color="warning" onClick={this.showModal.bind(this, 'modalEditProduct', product)}><img src={modify} /></Button>{' '}
                            <Button color="danger" onClick={this.showModal.bind(this, 'modalDeleteProduct', product)}><img src={deleteimg} /></Button>{' '}
                        </td>
                    </tr>);

            }


        }
    }
```

Ce code du fichier Stock.js permet d'afficher tous les produits sur la page du module de stock. Après la requête de l'API de récupération de produits, nous vérifions si les données ne sont pas vides afin de pouvoir effectuer du traitement dessus grâce à la fonction Validators.isDefined(). Le traitement consiste à créer une ligne de tableau en HTML en incorporant dans chaque cellule les données du produit et les boutons d'actions sur celui-ci. Cette fonction possède est lié aussi à la recherche d'un produit dans le tableau grâce à un champ texte, c'est pour cela qu'il y a une condition qui cherche dans la liste des produits reçus dans le state par rapport au texte dans le champ de recherche.

Lorsque les produits sont traités dans un tableau, ce tableau est mis dans un nouveau state pour les afficher :

```
this.setState({ productsView : table });
```

Voici la partie concernée de la vue HTML dans Stock.js :

```
<Col lg="8">
    <h5> Liste des produits </h5>
        <Button color="success"  onClick={this.showModal.bind(this, 'modalNewProduct')}><img src={plus} className="NewUser"/> Nouveau produit</Button>{' '}
        <Table className="Table">

            <thead>

            <tr>
                <th>Nom du produit</th>
                <th>Catégorie de produit</th>
                <th>Prix d'achat</th>
                <th>Prix de vente</th>
                <th>Disponible</th>
                <th>Quantité</th>
                <th>Actions</th>
            </tr>
            </thead>
            <tbody>
            <tr>
                <td><Input onChange={this.changeShowProducts} placeholder="Rechercher un produit"/></td>
            </tr>
            {this.state.productsView}

            </tbody>
        </Table>
</Col>
```

b) Communication avec l'API

Les modules qui nécessite de communiquer avec l'API utilise **Axios**, une bibliothèque de création de requête AJAX qui s'installe grâce à Node js.

Lien de documentation : https://github.com/axios/axios

Axios propose de créer facilement des requêtes vers notre API en invoquant le type de requête et en fournissant les divers paramètres nécessaires :

```
login(e) {
  if(e.key === 'Enter' || e.type === 'click'){
    axios.post(process.env.REACT_APP_API_URL+'/authBDF/login',
      {'login' : this.state.login,
       'mdp' : this.state.mdp,
      },
      {
        headers : {
          'Content-Type': 'application/x-www-form-urlencoded'
        }
      }
      )
      .then(r => this.loginOk(r))
      .catch(r => this.loginFail(r));
    }
}
```

Exemple de création de requête POST vers l'API dans Auth.js

Ensuite, Axios gère la réponse de l'API grâce à la fonction then() si il la réponse est un succès ou catch() si elle est un échec. Selon le cas, il est possible de lancer par la suite les fonctions nécessaires au traitement de la réponse de la requête.

c) Validation des données

Une classe Validators a été mise en place pour pouvoir vérifier l'intégrité des données reçus de la part de l'utilisateur (dans le répertoire /validators). Elle est utilisable dans tout le projet, facilite la lecture du code et évite la répétition du code

```
class Validators {

    static isDefined(value) {
        /*
        Check if value is defined
        :param value: your variable
        :return: True if value is defined, False if not
        */

        return (value == "undefined" || value == null) ? false : true;

    }
```

Exemple de fonctions de la classe Validators

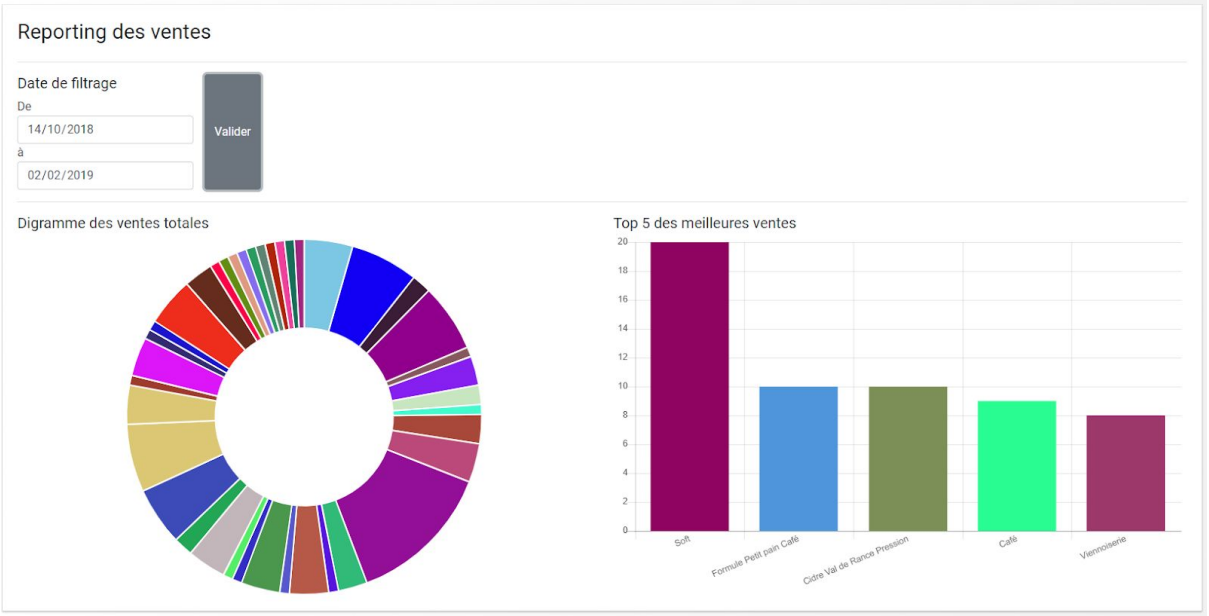
d) Configuration accès à l'API

La configuration de l'accès à l'API depuis l'application React est explicitée dans la partie déploiement du Front par le biais de variable d'environnement.

e) Bibliothèque graphique

Nous avons utiliser la bibliothèque graphique React Chart.js 2 pour représenter les divers graphiques des modules de vente et de reporting.

Lien de documentation : http://jerairrest.github.io/react-chartjs-2/



Capture d'écran de l'interface du reporting des ventes et gestion des stocks

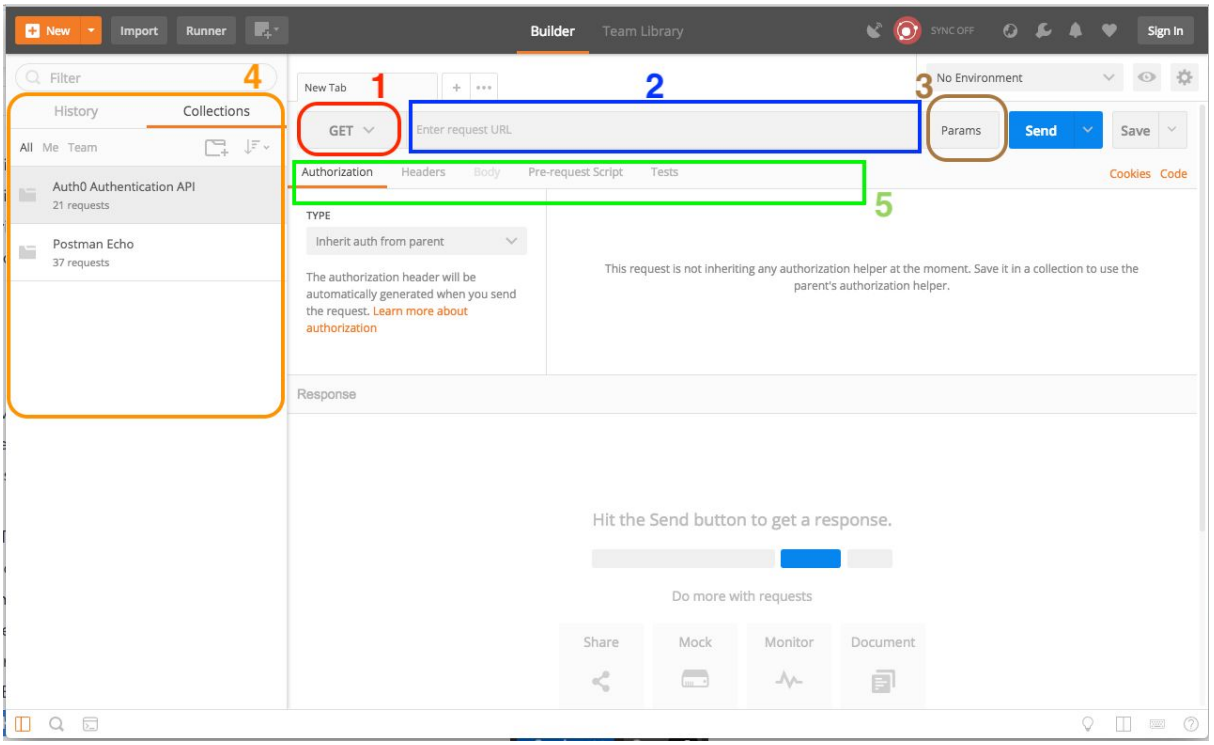
L'avantage de Chart.Js, il suffit juste de formater les données comme la bibliothèque le demande, d'ajouter des options si nécessaire et Chart.Js s'occupe de tout la mise en page et la création du diagramme.

```
<Col lg="6">
    <h5> Digramme des ventes totales</h5>
    < Doughnut
            data={this.state.dataDoughnut}
            options= {this.state.optionDoughnut}
            height={500}
            width={800}
    />
</Col>
```

```
  dataDoughnut: {
      datasets: [{
          backgroundColor: colors,
          data :dataPercentage
      }],
      labels: dataName
  },
  optionDoughnut : {
      animation :{animateRotate : true},
      legend: {
          display: false
      },
  }
});
```

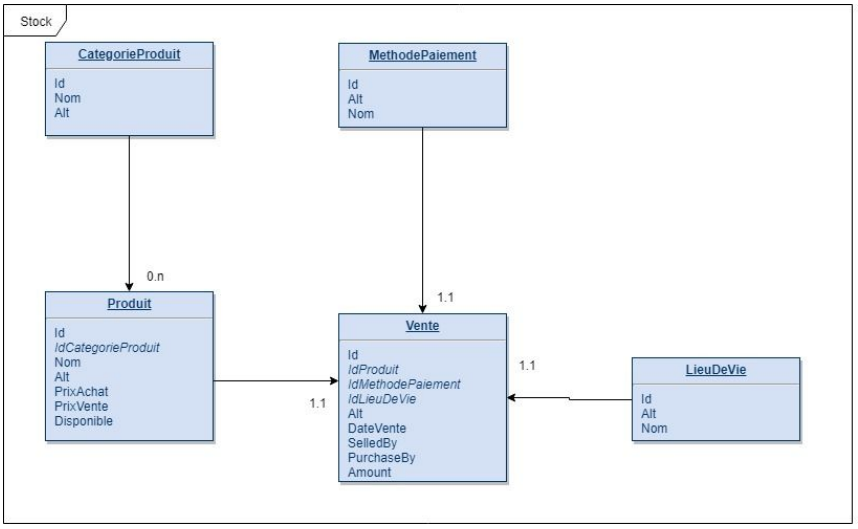
f) Postman

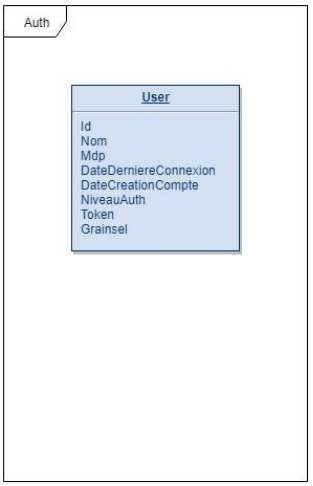
Postman est un logiciel qui se focalise sur les tests des API. Nous avons utilisé ce logiciel pour tester chaque point d'accès durant nos développements.

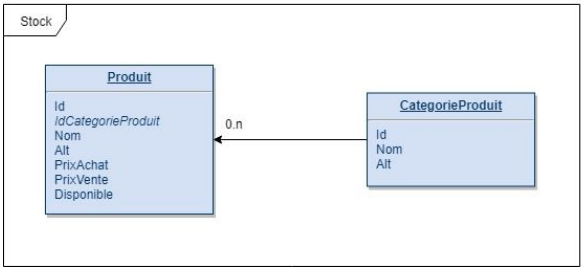


- (1) : Ici vous pouvez choisir le type de requête à envoyer : GET, POST, PUT, etc.
- (2) : L'URL de l'API.
- (3) : Les paramètres à passer avec l'URL. Dans le cas où vous avez des paramètres nommés comme ?prenom=unprenom
- (4) : Ici vous retrouverez les connections que vous avez créées. Nous y reviendrons.
- (5) : Ici vous trouverez tout ce qui constitue une requête Http (header, body, etc.). Vous pouvez à partir de là créer votre requête de façon totalement personnalisée.

III) Architecture de la base de données







La configuration de l'accès à la base de données depuis l'API est explicitée dans la partie déploiement de l'API par le biais de variable d'environnement.

IV) Déploiement

1) Partie BACK-END

A) Déploiement du BACK-END

Pré-requis

Projet

Pour pouvoir déployer ce projet correctement, vous devez au préalable télécharger celui-ci sur GITHUB en suivant ce lien :

https://github.com/AlexOUKS/BDF-Reporting-Management

Python

Vous avez besoin également de Python 3.4 ou d'une version supérieur. Nous vous conseillons de prendre toujours la dernière version disponible.

Si vous possédez un système d'exploitation tel que Ubuntu, Mint ou bien même Debian, vous pouvez installer Python avec le code suivant :

```
$ sudo apt-get install python3 python3-pip
```

Pour tout autre système d'exploitation, vous pouvez l'obtenir grâce au lien suivant :

http://www.python.org/getit/

Django

Install Django depuis "pip" dans votre environnement virtuel (recommandé):

```
sudo pip install django
```

Install Django pour un système d'exploitation Windows :

https://docs.djangoproject.com/en/2.1/howto/windows/

Dépendance

Pour installer tous les paquets nécessaire au lancement de l'API, lancer cette ligne de commande à la racine du dossier BACK:

```
pip install -r requirements.txt
```

PostgreSQL

Nous utilisons PostgreSQL pour le développement du projet. Celui-ci va nous servir de base de donnée local.

Pour installer PostgreSQL, suivez ce lien:

https://www.postgresql.org/download/

Mettez les identifiants dans la base de données::

1. Ouvrez le fichier settings.py dans le repértoire ToProject.
2. Mettez vos identifiants dans les variables environnement.
3. `os.environ["DATABASE_HOST"] = "yourhost"`
   `os.environ["DATABASE_PORT"] = "yourport"`
   `os.environ["DATABASE_NAME"] = "yourname"`
   `os.environ["DATABASE_USER"] = "youruser"`
   `os.environ["DATABASE_PASSWORD"] = "yourpassword"`

Quick start

Pour déployer notre API, aller dans le dossier ou celui-ci est localisé en ligne de commande et exécuter les ligne de commande suivante :

1. Pour créer le mapping entre les modèles créés dans le code et la base de donnée locale :

   ```
   $ python manage.py makemigrations
   ```

2. Mettre à jour la BDD locale :

   ```
   $ python manage.py migrate
   ```

3. Lancer l'API :

   ```
   $ python manage.py runserver
   ```

1. Partie FRONT

A. Déploiement du FRONT

Pré-requis

Node.js

Vous aurez besoin de Node.js pour compiler et lancer notre projet :

Avec Ubuntu, Mint et Debian, vous pouvez installer Node.js comme cela :

```
$ sudo apt-get install nodejs
```

Pour tout autre système d'exploitation, les paquets sont disponible sur :

https://nodejs.org/en/

BDF-Reporting-Management Client

Clonez le projet à l'aide de GIT, utilisez la ligne de commande : git clone.
https://github.com/AlexOUKS/BDF-Reporting-Management

Ou

Télécharger le directement sur la page internet GIT

Dépendance

Pour installer tout les paquets nécessaire pour lancer notre client, exécuter cette ligne de commande dans la racine du répertoire du projet :

```
npm install
```

Variables environnement

Vous devez créer un fichier .F appelé ".env" puis le mettre dans la racine de votre projet avec le code suivant :

```
REACT_APP_API_URL=URL_OF_YOUR_API
```

Cela va paramétrer le chemin de notre API pour toutes les requêtes étant créées par l'application.

Quick start

Pour déployer notre client, allez dans le fichier ou le client se situe et exécuter le code suivant en ligne de commande :

```
$ npm start
```