le cnam
Nouvelle-Aquitaine

# Document de conception

## Projet Bank

-

Arthur Baribeaud et Etienne Monthieux

USSI5D : IHM

Titre : Développement d'une application bancaire de monitoring des opérations

# Sommaire

# 1. Contexte [Etienne]

L'idée du projet date de 2020. Ma banque ne proposait pas de visualisation des transactions. Je ne pouvais voir que mon solde courant et mon en-cours. L'alternative pour visualiser mes dépenses et mes entrées d'argent a été d'utiliser bankin.com. Cependant, je n'étais pas satisfait de cette solution, car je donnais mes identifiants bancaires à une application tierce. L'idée de refaire des visuels pour décrire et analyser mes transactions bancaires m'est alors venue à l'idée. Fin 2020, j'ai donc commencé un projet tout en Python pour répondre à mon besoin.

## 1.1 Besoin

Le besoin est de pouvoir visualiser mes transactions bancaires (entrée et sorties), de les catégoriser et de voir l'évolution de mon solde. Nous souhaitons également ajouter des nouvelles opérations.

## 1.2 Existant développé en Python et problématique rencontré

L'application Python que j'avais créé fonctionnais avec plotly pour le frontend et flask pour la partie backend. J'ai dû résoudre des problèmes de mise à jour d'affichage, de gestion des données (import, mise à jour, sauvegarde). Je n'ai pas maintenu l'application car Python n'est pas fait pour être utilisé de cette manière et mon application était une "usine à gaz". Aussi, ma banque permet désormais de visualiser mes transactions comme sur bankin', je n'ai donc plus besoin de mon application.

## 1.3 Fonctionnalité prévue

Le but de ce projet est de faire une page pour voir l'historique des transactions et une page avec des graphiques résumant les transactions et l'évolution du solde.

# 2. Démarche et choix de la technologie

La nature de l'application, traitant des opérations bancaires, est ainsi rapidement apparu comme sujet naturel pour le projet d'IHM. Il restait alors à choisir quelles technos utiliser pour mener à bien ce projet. Il a nous a ainsi été possible d'étudier les différents choix qui s'offraient à nous afin de choisir les technos les plus pertinentes. Une chose importante à noter est que cette démarche de benchmark n'avait pas spécialement vocation à choisir la "meilleure" technologie, mais plutôt une technologie que nous ne maîtrisions pas dans le but de monter en compétence dessus.

## 2.1 Client léger ou lourd

Deux types d'application était ainsi possible : sur client lourd ou sur client léger. Notre choix s'est rapidement porté sur les clients légers puisque étant particulièrement utilisé aujourd'hui en profitant d'architecture en micro-service communicant via des API par exemple. De plus, il nous avait déjà été donné de travailler sur des projets de ce type dans le passé, que ce soit au CNAM ou en IUT sur des technologies telles que Qt et JavaFX.



Un projet sur client léger web nous a ainsi paru plus pertinent afin de monter en compétence sur des technologies que nous sommes amenés à côtoyer plus fréquemment que les clients lourds.

Dans un second temps, il nous a donc été possible d'observer les différentes solutions existantes pour ce type de projet, notamment du côté des frameworks web. Parmi les outils disponibles, nous avons rapidement écarté les outils full python que nous maîtrisions déjà ou bien que notre niveau en python nous aurait permis d'appréhender trop facilement tel que Plotly ou Dash.
Nos recherches se sont ainsi rapidement portées sur des outils tels que Flask, Django, Angular, VueJS ou encore Laravel. Chacune de ces technos répond ainsi à un besoin particulier (Front-end, Back-end, API, …).

## 2.2 Framework Back-end

Pour la partie Back-end, il nous était ainsi possible de choisir entre deux frameworks python (Flask et Django) et un framework PHP (Laravel). Bien que décidant de sortir de notre zone de confort, l'idée était principalement de mettre en place des choses côté IHM et donc Front-end. Le choix d'un framework que nous connaissions déjà tel que Flask, écris en python, nous permettait ainsi de rapidement mettre en place une application Back-end pour discuter avec notre front.



Pour finir, la motivation derrière le choix de Flask plutôt que Django réside là encore dans la montée en compétence vis-à-vis de Flask par rapport à Django, déjà maîtrisé par le binôme.

## 2.3 Framework Front-end

Pour le framework front-end, deux possibilités nous étaient offertes : Angular ou VueJS. Ces deux technologies sont relativement répandues sur le marché et utilisé par de nombreuses entreprises. Le choix d'utiliser une technologie plutôt qu'une autre était ainsi relativement arbitraire. Notre choix s'est porté sur VueJS, notamment en raison de l'utilisation de cette technologie dans l'entreprise Saas de l'un des membres du binôme, et ce, toujours dans une démarche de pouvoir mieux comprendre l'environnement professionnel dans lesquels nous serons amenés à évoluer.


## 2.4 Base de données

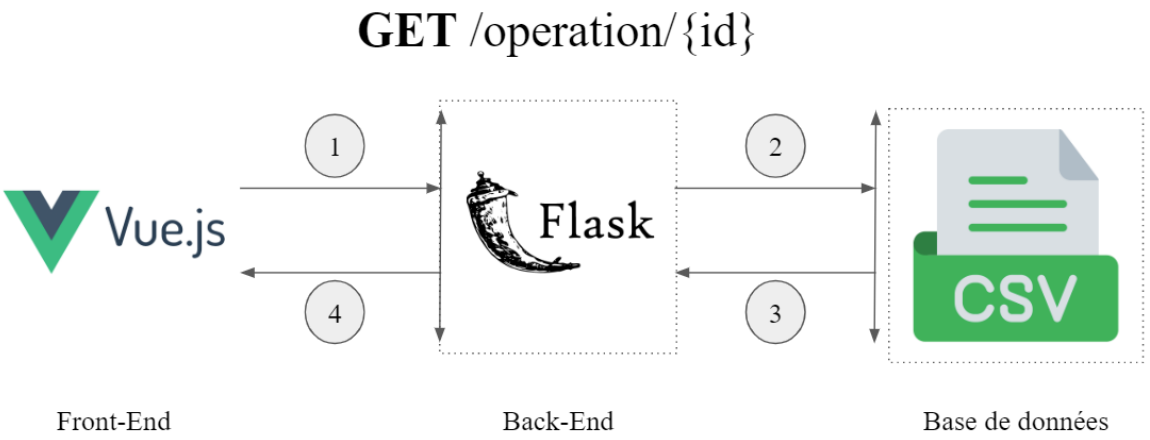
Le déploiement d'une base de données avait également initialement été prévu avec MySql afin de stocker les opérations bancaires. Cependant, la mise en place de cette dernière, branché à notre API Flask, s'est révélé comme une charge de travail supplémentaire conséquente et donc diminuant le temps attribué à la découverte de VueJs. L'équipe a ainsi préféré abandonner l'utilisation d'une BDD pour se concentrer sur la partie Front-end. Ainsi, les données utilisées et stockées dans notre application sont directement issues de fichier csv accessible à notre API Flask via la librairie Pandas.

Bien que les donner soit stockées dans des fichiers csv, l'objectif du projet serait bien entendu à terme de pouvoir tout stocker via un SBGDB.



# 3. Architecture générale : back, front et bdd

L'architecture du projet repose sur 3 blocs communicants les un avec les autres : front, back et BDD. Cette architecture peu être visualisée comme un ensemble de 3 services indépendant, mais communicant. Cette architecture permet d'éviter la mise en place de bloc monolithique qui peuvent être complexe à gérer en cas de monter de version par exemple. Dans notre cas par exemple, les modifications apportés à l'api Flask (monté de version, nouveau point d'entrée, …) sont complétement transparents pour notre application Front-end.




Notre application a ainsi globalement deux objectifs : afficher des statistiques et traiter de nouvelles opérations (ajout ou suppression). Pour ce faire, il a été nécessaire de développer du côté Back les différents points d'entrée API permettant de traiter les différents cas. Une fois fait, il ne reste plus qu'à mettre en place les composants front qui feront appel en tant voulu à notre API. À noter que deux librairies ont été utilisé pour la partie API du côté VueJS : fetch qui est une builtin de Javascript et axios qui est une librairie externe. En effet, la documentation JavaScript recommandait globalement l'utilisation de l'une ou de l'autre avec leurs avantages et inconvénient. Dans une démarche d'apprentissage, les deux librairies ont donc été utilisé dans la vue Opération (Fetch) et dans la vue Statistique (Axios)

## 3.1 Point d'entrées de l'API Back

GET /categories : Récupère la liste des catégories disponibles

GET /sous_categories : Récupère la liste des sous-catégories disponibles

GET /operations : Récupère la liste des opérations

GET /last_10_operations : Récupère la liste des 10 dernières opérations

GET /operations_by_category?start_date[start]&end_date[end] : Récupère la somme des opérations par catégorie sur la période sélectionnée

GET /amount_by_month : Récupère la somme des opérations par mois

GET /balance_evolution : Récupère les sommes cummulées croissantes des opérations par mois

POST /add_operation : Ajoute une opération passée en json dans le corp de la requête

DELETE /delete_operations/{id} : Supprime une opération basée sur un id d'opération

## 3.2 Architecture Vue JS

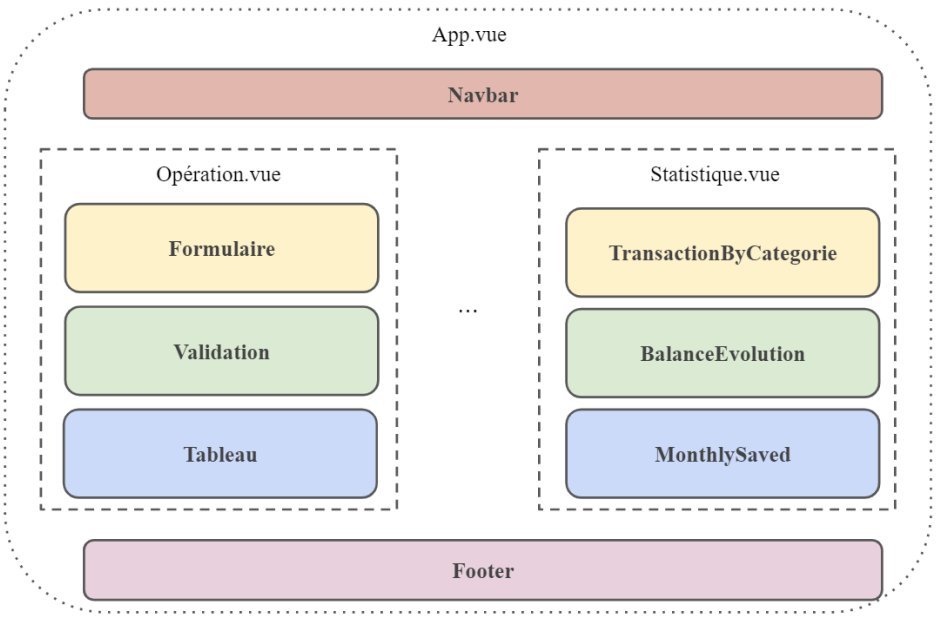
L'architecture de l'application Front reprend les bonnes pratiques conseillées pour les applications Vue JS simple, c'est-à-dire proposant peu de service, et sur des données proches.

Pour ce faire, notre application est donc composée de 3 éléments centraux :

- App : Il s'agit de l'entité de base de notre application, elle permet notamment de fixer certains composants essentiels comme la barre de navigation ou le pied de page.
- Vue : Il s'agit des différentes "pages" auxquels nos utilisateurs auront accès
- Composant : Il s'agit des différents composants qui constitueront les vues.

Bien que compliqué au premier abord, l'architecture proposée par VueJS pour la construction d'application est plutôt intuitive une fois maîtrisé. Chaque "vue" répond à un besoin métier, par exemple affiché des statistiques ou effectué des actions d'opérations. Puis dans chaque "Vue", on vient ajouter des "Composant" dédié au traitement d'une tâche simple. C'est au final l'ajout de plusieurs composants simple (exécutant une tâche précise) dans une même vue qui permet de créer des applications complexes.

Par la suite, VueJS permet aux composants de communiquer entre eux afin de déclencher des actions synchronisées au besoin.



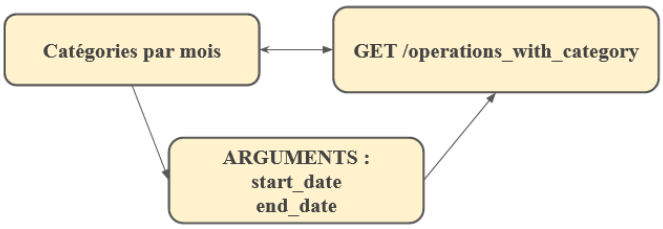

# 4. Front : Statistique

La vue des Statistiques est générée grâce à 3 composants indépendants : *TransactionByCategorie, MonthlySaved, BalanceEvolution.* Chacun de ses composants correspond à un graphique. La page statistique permet de sélectionner le mois et l'année pour le graphique "Transactions par categorie". Cette page permet donc de visualiser :

- La répartition des dépenses par mois et année
- Les économies réalisées chaque mois
- L'évolution du solde

## 4.1. Composant des dépenses par catégories

Ce composant affiche la répartition des dépenses par catégories pour un mois et une année données. Il reçoit depuis la page "Statistiques" le mois et l'année à afficher, le transmet à l'API qui retourne le jeu de données. Avec les données récupérées, le composant crée une image SVG qui résume la répartition des dépenses par catégories. Le graphe doit être supprimé à chaque changement de mois ou d'années sinon il se duplique.



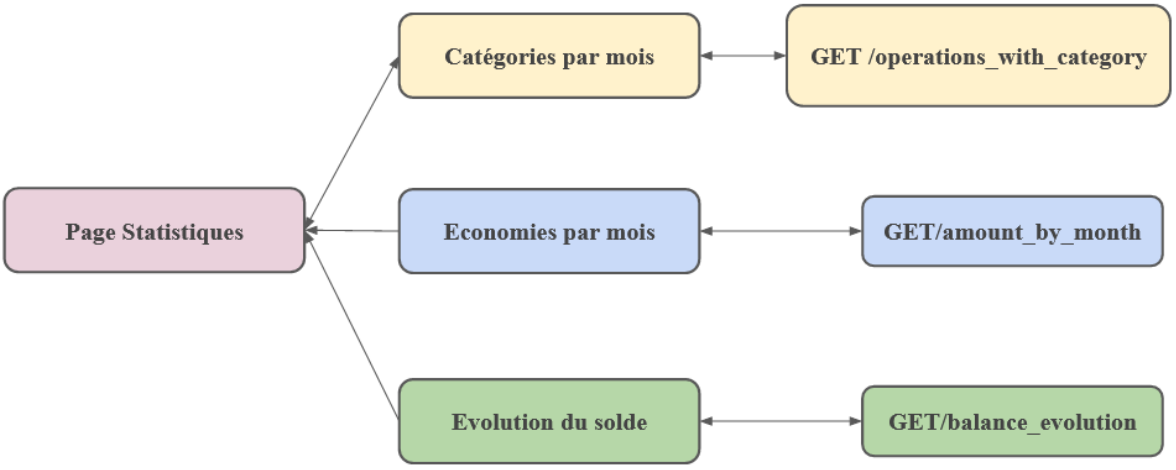

Pour détecter un changement de mois ou d'année, la méthode "watch", native de VueJS, est utilisé sur les changements dans la liste déroulante. L'envoie d'arguments est par la suite forcé à l'aide de la méthode "props" native de VueJS.

## 4.2. Les autres composants

Les composants d'économies mensuelles et d'évolution du solde émettent des appels API et récupèrent le jeu de données. Puis, ils affichent un graphe et sont importés depuis la page "Statistiques".

## 4.3. Le fonctionnement global

Le fonctionnement global des composants pour la page "Statistiques" peut être résumé par le graphe suivant.




La page appelle les composants pour afficher les graphes et envoie le mois et la date à sélectionner pour l'affichage des catégories. Les composants appellent l'API pour obtenir les données et génèrent des graphes. Une stratégie asynchrone est utilisée avec la méthode "async" pour récupérer les données uniquement lorsque cela est nécessaire et "await" pour attendre que la donnée soit chargée avant de dessiner le graphique.

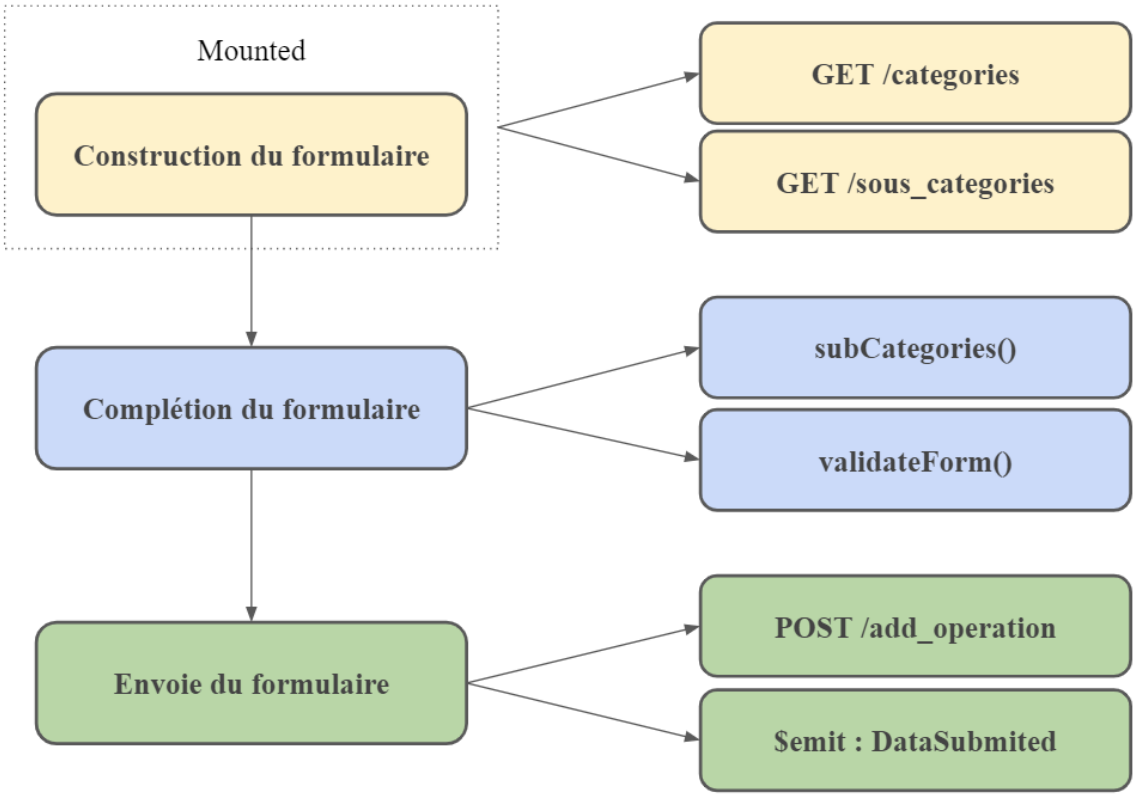
# 5. Front : Opérations

La vue des Opérations est constituée de 3 composants inter-dépendant et communicant entre eux : *FormulaireOpéraiton*, *ResumeOperation*, *TableauOperation*. Chacun de ces composants répond à une tâche précise qui, combiné avec le reste de la Vue, permet 3 actions :

- Ajouter une nouvelle opération
- Lister l'historique des opérations
- Supprimer une opération de l'historique

## 5.1. Composant du formulaire

Le composant du formulaire répond au besoin de "En tant qu'utilisateur de la plateforme, je veux pouvoir ajouter une nouvelle opération". Pour ce faire, un formulaire de 6 champs est construit dans le composant à travers un cycle de vie précis et défini (Lifecyle hook) dans lequel vont s'exécuter différentes actions de notre composant.




La première partie du cycle de vie de notre composant est le "mounted" qui désigne les actions de construction du composant, avant même l'affichage de notre page. Ainsi, la construction de notre formulaire requiert la liste des catégories et sous-catégories disponibles afin de les afficher dans la liste déroulante de notre formulaire. Ces données donc récupérées via deux appels à notre API Flask.

Une fois construit, notre formulaire ce mets en attente jusqu'à ce qu'un utilisateur commence à remplir le formulaire. À cet instant, deux mécaniques se déclenchent, la première validateForm() s'assure que le formulaire est entièrement complété, sans quoi le bouton d'envoi reste bloqué. La deuxième mécanique vient, elle, mettre à jour dynamiquement les données de la liste déroulante des sous-catégories afin de s'assurer que les sous-catégories possibles correspondent à la catégorie sélectionnés.

Une fois le formulaire prêt et le bouton "Envoyé" déclenché, notre composant entre dans la dernière phase de son cycle de vie : l'envoie des données. Il récupère ainsi les données du formulaire et les envoie sous la forme d'un dictionnaire JSON pour notre API Flask. Cette action d'envoi est effectué pour enregistrer l'opération en base. En cas de retour "ok" (réponse 200) signifiant que l'opération a bien été envoyé en base, alors notre composant émet le dictionnaire de données au reste de notre vue Opération pour les autres composants qui voudrait récupérer les informations.

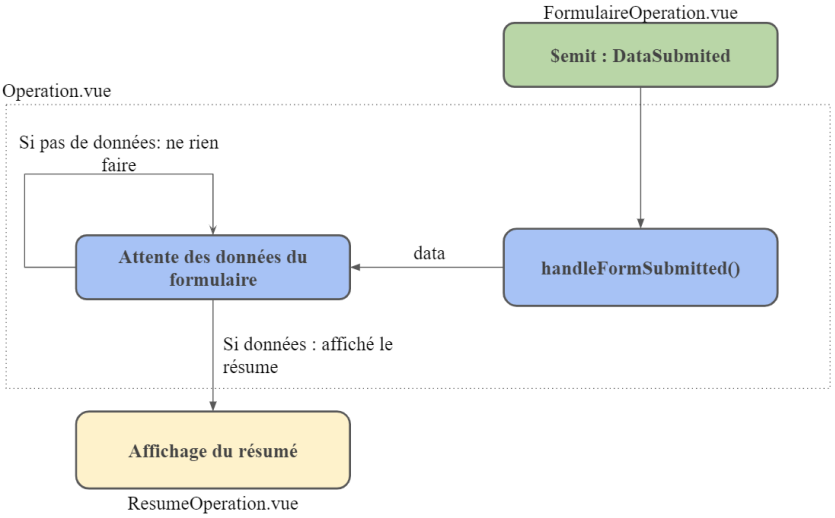
L'émission de ce dictionnaire est catch au niveau de la vue Opération par une fonction "computed" handleFormSubimitted()

```
<FormulaireOperation @form-submitted="handleFormSubmitted" />
```

## 5.2 Composant du résumé

Le composant de résumé est le plus simple à traiter et réppond au besoin "En tant qu'utilisateur, lorsque j'envoie un formulaire, alors, je veux être en mesure de récupérer un récapitulatif de l'opération créé". Il attend ainsi simplement l'émission d'un dictionnaire par le composant du formulaire via la fonction "computed" handleFormSubmitted(). Une fois les données émise et

récupérée (et donc bien ajouté en BDD), il affiche un résumé des données ajouté en base de données.



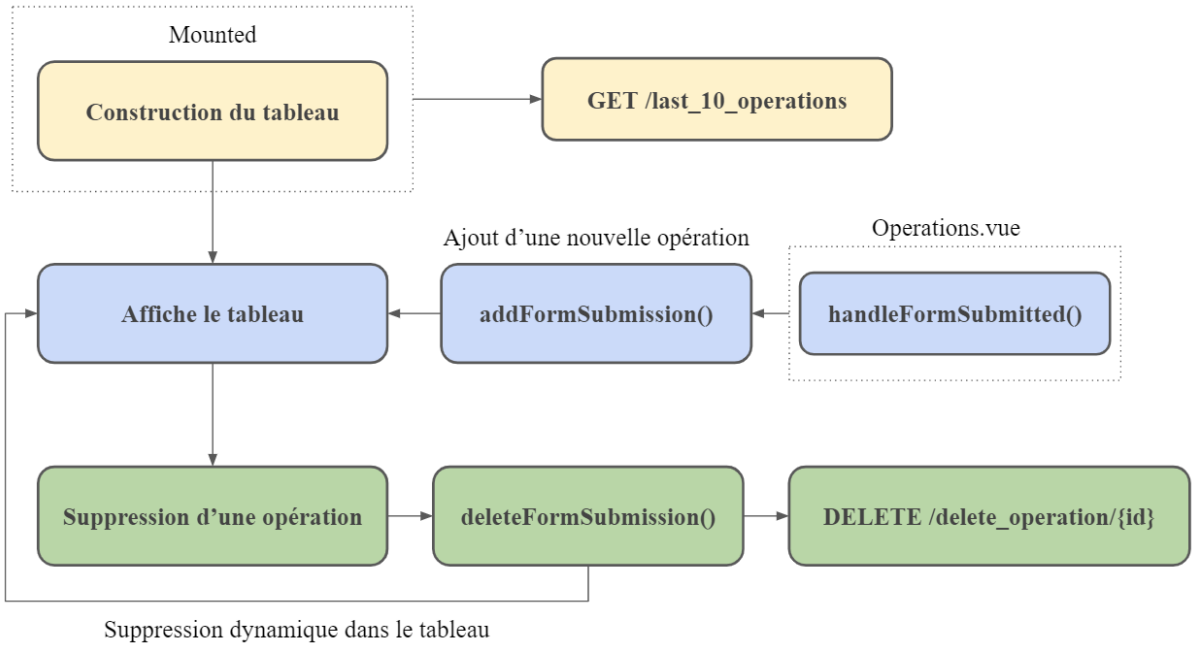

## 5.3 Composant du tableau

Le dernier composant de la vue Opération supervise l'affichage dynamique de l'historique des opérations ainsi que de la suppression d'opérations de l'historique. Il répond au besoin "En tant qu'utilisateur, je veux avoir accès à un historique dynamique des opérations me concernant ainsi que la possibilité de supprimer une opération". Trois mécaniques sont ainsi utilisées pour mettre en place ce composant.

La première mécanique est la construction du tableau ("mounted") avec la récupération des opérations à afficher via l'api Flask.

La deuxième mécanique est l'ajout dynamique des opérations ajouté lorsque le formulaire est posté. En effet, lors de l'ajout d'une opération, cette dernière est ajoutée en base de données. Cependant, à cet instant, le tableau est déjà construit et ne sait pas que l'état de la base de données à changé. Ainsi, pour éviter de devoir faire un nouvel appelle en base lors de chaque changement, on met à jour le tableau dynamiquement et localement via la fonction handleFormSubmission() de la vue Opération. On vient donc ajouter à la première place de notre tableau la nouvelle opération.

La troisième mécanique est celle de suppressions d'une opération. Ici, on déclenche un événement lorsque l'icône poubelle est cliqué en face d'une opération. Cet événement fourni à une méthode l'id de la ligne cliqué. Ainsi, il devient possible de faire un appel à notre API Flask pour supprimer notre élément. Une fois supprimé en base, on supprime la ligne localement (afin là aussi d'éviter de solliciter la base à chaque suppression).




## 6. Planning et gestion de projet

La gestion et le planning de projet a été divisé en trois étapes : choix du projet et des technos, découverte de VueJS et pour finir développement de l'application.

Le choix du projet a été relativement rapide étant donné l'existant déjà développé en Python. La réflexion a ainsi plutôt été faite sur ce qui n'allait pas dans l'existant et ce qu'il fallait donc améliorer ou ajouter.

Une fois le projet définis, il a été nécessaire de découvrir les technos pouvant être utilisé puis d'essayer de les appréhender. Pour cela, différentes ressources d'apprentissages ont étés utilisés notamment autour d'Angular. En parallèle, un schéma de BDD a été réfléchi et déployé sur une base de données MySQL dans l'optique de gérer les opérations bancaires. Malheureusement, la mise en place d'une telle base de données branchée à notre API implique de nombreuses contraintes à gérer qui prennent du temps. De ce fait, il a été décidé d'abandonner l'utilisation d'une vraie base de données et de la remplacer par des fichiers csv.

La deuxième étape a pu démarrer une fois la techno front-end arrêté sur le framework VueJS. Le binôme a ainsi pu prendre en main l'outil, notamment en suivant les tutoriels proposés par la communauté VueJS. Cette étape indispensable a pris un peu plus de 10h pour chaque membre de l'équipe.

Enfin, la troisième étape a été le développement de l'application. Le binôme s'est ainsi répartie les tâches afin d'avancer en parallèles sur les deux vues *Opérations* et *Statistiques*. Cette étape a été la plus longue, mais la plus intéressante puisque permettant de réellement mettre en place une application VueJS en comprenant les subtilités et l'intérêt de cette technologie.

| **Action** | **Temps** |
|---|---|
| Choix de projet | 2h |
| Choix de la techno | 4h |
| Prise en main VueJS | 10h |
| Base de données | 4h |
| Développement API Flask | 2h |
| Développement VueJS | 14h |
| Documentation technique | 6h |
| **Total** | **42h** |

## 7. Déploiement du projet

Les instructions de déploiement sont indiqué dans le fichier README.md à la racine du projet

## 8. Conclusion et poursuite

Le projet final est ainsi constitué de 2 vues : Opération et Statistique, répondant toutes les deux à un besoin particulier. Bien qu'à première vue assez simple comme application, il a été très intéressant de comprendre comment fonctionnait une architecture d'application VueJs. Ainsi, la majorité de nos efforts ont été porté sur la compréhension à la fois du langage JavaScript que nous ne connaissions pas ainsi que la découverte du framework VueJS.

Les différents composants utilisés dans l'application nous ont ainsi poussé à comprendre certaines subtilités de VueJs comme le Lifcycle Hook ("mounted", "computed", …) qui définit l'ordre de création des vues, ou encore le principe des exécutions asynchrones avec "await". De plus, de nombreux cas d'usage ont pu être traité simplement grâce à l'utilisation "d'outil" mis à disposition par

VueJs. On notera l'émission de données dans un périmètre plus grand que les composants eux même (avec $emit par exemple), le parcours de liste ou les assertions directement dans le code HTML grâce au builtin *v-for* ou *v-if* .

Pour finir, l'application livrée n'est pas utilisable dans un environnement de production puisque de nombreux éléments sont encore manquant comme la connexion à une vraie base de données. De plus, la gestion d'application bancaire requiert de  quant à la sécurité des données, ce qui n'est pas le cas de notre application. Pour poursuivre ce projet, différents axes d'amélioration serait donc possible : utilisation d'une base de données, possibilité d'édité une opération, ajout de plus de graphes, ajouter le guide utilisateur dans le site (FAQs par exemple).