



UNIVERSITÉ  
DE LORRAINE

MASTER INFORMATIQUE

Projet de Design Pattern  
Tableur collaboratif en ligne : BigSheet  
Rapport

M1 Informatique  
Année 2023-2024

MARCELIN Maxime  
QUEIGNEC Nicolas  
JOLY Clément  
BORGONDO David

## **Table des matières**

<b>1. Introduction</b>	<b>3</b>
a. Backend	4
b. Frontend	4
<b>2. Fonctionnalités</b>	<b>5</b>
a. Création de compte et connexion	5
b. Modification d'informations du compte	8
c. Création et modification d'un document	9
d. Partage d'un document avec droits d'accès	10
e. Connexion simultanée sur un document et synchronisation	10
<b>3. Parties à développer (code, architecture...)</b>	<b>11</b>
a. Modèle	11
b. Authentification	13
c. Synchronisation	15
<b>4. Conclusion</b>	<b>18</b>
<b>Guide de démarrage</b>	<b>19</b>

# 1.Introduction

Dans le cadre de l'UE Design Pattern, il nous a été confié comme projet de faire un tableur collaboratif accessible par navigateur comme Google Sheets. Un utilisateur doit pouvoir créer une feuille, la partager avec d'autres utilisateurs. Les utilisateurs doivent pouvoir travailler simultanément sur une même feuille et voir les modifications en temps réel.

Afin de mener à bien ce projet, il a été décidé d'une architecture client-serveur afin d'avoir un rendu dynamique. Le backend (le serveur) utilise NodeJS dans sa version 18.18.2. Le frontend (le client) utilise VueJS dans sa version 3.3.4.

L'utilisation d'une base de données côté serveur a été préférée pour pouvoir enregistrer les comptes des utilisateurs et sauvegarder les feuilles créées par les utilisateurs. Elle utilise le système de gestion de base de données PostgreSQL. Il y a aussi un autre système, Redis, pour l'authentification des utilisateurs afin de pouvoir stocker les jetons d'authentification avec une durée limitée. Voici un schéma du fonctionnement de l'application avec les échanges entre les différents logiciels et l'utilisateur.

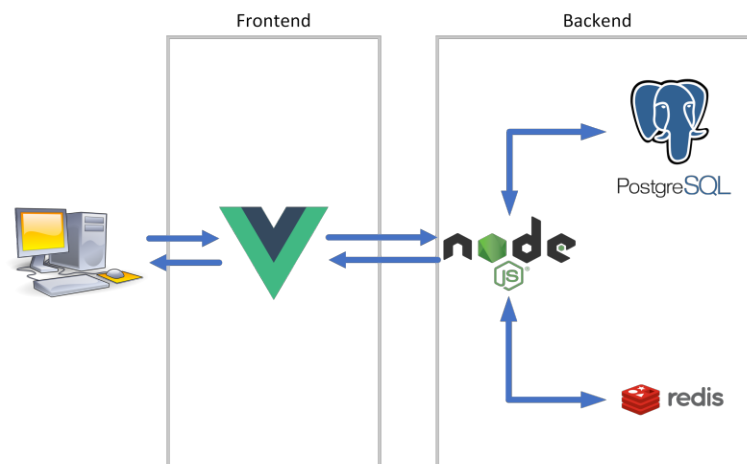


Figure 1 : Architecture de l'application

Docker permet de démarrer facilement les logiciels tiers que nous devons utiliser. En une seule commande, il est possible de démarrer Redis, PostgreSQL et Adminer qui permet de visualiser les tables, et leurs contenus, de notre base de données PostgreSQL.

Le client VueJS sert d'interface Homme-Machine et permet d'interagir avec le serveur NodeJS en masquant la complexité sous-jacente de l'API. Le serveur Node, masque encore davantage les différents fonctionnement de l'application en proposant des routes permettant de modifier les bases de données. Il propose aussi une gestion simplifiée des modifications en temps réel des différentes feuilles de calcul.

Docker permet également d'orchestrer les différents logiciels afin de proposer un déploiement facilité de l'application sur n'importe quel environnement.

Voici les différentes librairies utilisées dans le backend puis dans le frontend.

## a. Backend

- Express 4.18.12 pour créer le serveur HTTP et pouvoir répondre aux requêtes faites par un client.
- Sequelize 6.33.0 pour interagir avec la base de données PostgreSQL. C'est une couche se plaçant entre le développeur et la base de données. Cela permet de traiter chaque table de la base de données comme un objet et où chaque ligne d'une table est une instance de cet objet. Une librairie de ce type s'appelle un ORM (Object Relational Mapping) et permet au programmeur de s'abstraire du langage SQL.
- Socket.io 4.7.2 pour pouvoir synchroniser les utilisateurs lorsqu'ils sont plusieurs à écrire sur la même feuille.

## b. Frontend

- Vue 3.3.4 pour créer une interface dynamique et pour pouvoir utiliser différents composants sur plusieurs pages.
- Vue-Router 4.2.5 pour pouvoir faire différentes routes et associer une route à une vue.
- SASS 1.69.4 pour pouvoir utiliser du SCSS. Le SCSS est un langage de script qui est par la suite compilé en fichier CSS afin de faire la mise en forme.
- Socket.io-client 4.7.2 pour pouvoir voir en temps réel les modifications des autres utilisateurs lorsqu'ils écrivent sur la même feuille.

## 2. Fonctionnalités

Nous allons tout d'abord vous présenter le fonctionnement général du projet ainsi que les différentes fonctionnalités du point de vue de l'utilisateur.

### a. Création de compte et connexion

Lors d'une connexion au site Bigsheet un utilisateur non-connecté est amené sur la page d'accueil sur laquelle il est invité à se connecter ou à créer un compte :

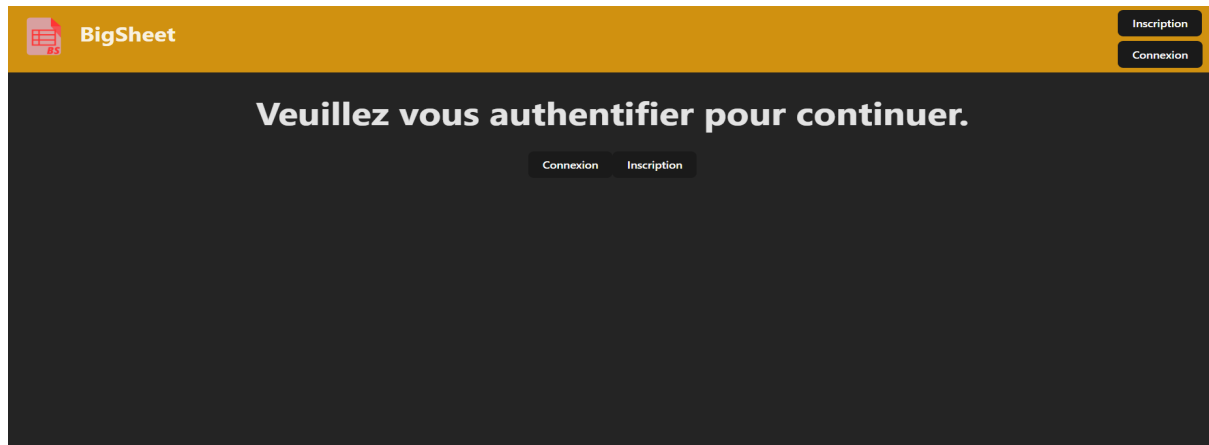


Figure 2 - Page d'accueil

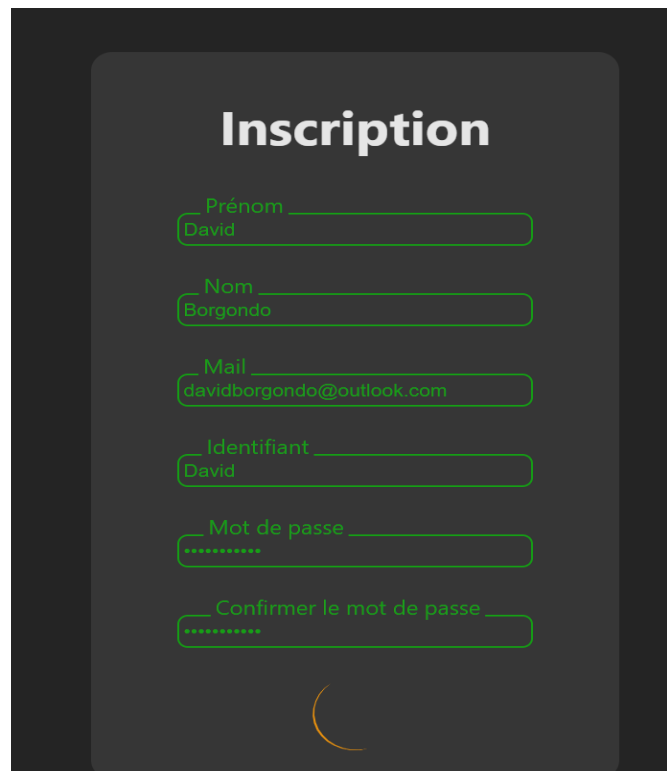
On peut aussi observer que l'en-tête de la page est lui aussi automatiquement modifié pour proposer uniquement les options disponibles pour un utilisateur non connecté.

Si l'utilisateur clique sur le bouton **"Inscription"** il est amené à une page d'inscription sur laquelle il est invité à remplir les informations nécessaires à son inscription :

The image shows the registration page of the 'BigSheet' website. The page has a dark grey background. In the center, there is a white rounded rectangle containing the title 'Inscription' in bold. Below the title, there are six input fields stacked vertically, each with a label: 'Prénom', 'Nom', 'Mail', 'Identifiant', 'Mot de passe', and 'Confirmer le mot de passe'. At the bottom of the white rectangle, there are two buttons: 'Inscription' and 'Déjà un compte ?'.

Figure 3 - Page d'Inscription

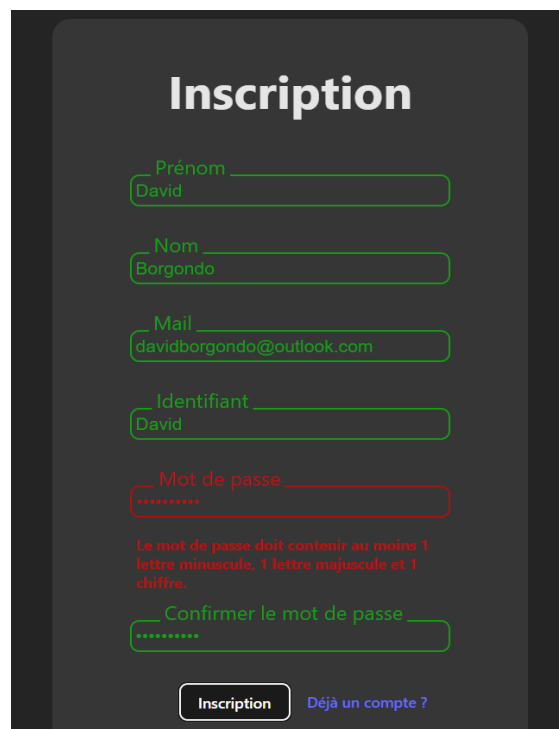
Lorsque l'utilisateur clique sur le bouton "**Inscription**", la validité de ses informations est vérifiée puis si tous les champs sont remplis et corrects, la procédure de création du compte est lancée et un icône de chargement s'affiche:



The screenshot shows a registration form titled "Inscription" on a dark background. The form contains six input fields, all of which are filled with green text and have green borders. The fields are: "Prénom" (David), "Nom" (Borgondo), "Mail" (davidborgondo@outlook.com), "Identifiant" (David), "Mot de passe" (represented by dots), and "Confirmer le mot de passe" (represented by dots). At the bottom center of the form, there is a yellow circular loading icon consisting of two curved lines.

*Figure 4 - Inscription en cours*

Dans l'éventualité où un champ est vide ou incorrect celui-ci est coloré en rouge et il est indiqué la raison du refus de l'information fournie :



The screenshot shows the same registration form as in Figure 4, but with a red border around the "Mot de passe" field. Below this field, there is a red error message: "Le mot de passe doit contenir au moins 1 lettre minuscule, 1 lettre majuscule et 1 chiffre." The other fields remain green. At the bottom of the form, there are two buttons: "Inscription" and "Déjà un compte ?".

*Figure 5 - Exemple d'informations fournit incorrect*

Après une connexion réussie l'utilisateur est automatiquement redirigé vers une page listant les feuilles associé à son compte :



The screenshot shows the BigSheet user interface. At the top, there is a navigation bar with the BigSheet logo on the left and two buttons, 'Mon compte' and 'Déconnexion', on the right. Below the navigation bar, the main content area has a dark background. It starts with a welcome message 'Bienvenue Nicolas !' followed by the heading 'Vos feuilles de calcul :'. Below this heading is a table listing four spreadsheets. Each row in the table contains the spreadsheet name, its details, the owner's name, the creation date and time, and a vertical ellipsis menu icon on the right. At the bottom right of the main content area, there is a red plus sign icon.

Vos feuilles de calcul :				
Sans-Nom2	Détails : Aucun détails	Propriétaire : zobal	Feuille créée le : 27-11-2023 à 17:55	⋮
55	Détails : bonsoir	Propriétaire : fulash	Feuille créée le : 27-11-2023 à 22:11	⋮
New Sheet	Détails : Some details	Propriétaire : fulash	Feuille créée le : 31-12-2023 à 18:46	⋮
yooo	Détails : ttt	Propriétaire : fulash	Feuille créée le : 31-12-2023 à 18:47	⋮

Figure 6 - Page listant les feuilles compte

Si l'utilisateur est propriétaire d'une feuille, il peut la partager ou la supprimer avec le bouton à droite de la feuille. On peut observer aussi que l'entête de la page s'actualise pour permettre l'accès à de nouvelles fonctionnalités.




This screenshot shows the header of the BigSheet user interface. It features a navigation bar with the BigSheet logo on the left and two buttons, 'Mon compte' and 'Déconnexion', on the right.

Figure xx - Exemple d'entête du site après une connexion

## b. Modification d'informations du compte

Une des nouvelles fonctionnalité disponible est la modification des données relatives au compte actif, lors de son arrivée sur la page les données actuelles sont pré-remplies au sein des champs et l'utilisateur peut choisir de les modifier puis de valider ses modifications en appuyant sur le bouton "Enregistrer" :



The screenshot shows a dark-themed form titled "Mon compte". It contains several input fields with placeholder text: "Prénom" (TestPrénom), "Nom" (TestNom), "Mail" (test@outlook.com), "Identifiant" (Test), "Mot de passe", and "Confirmer le mot de passe". At the bottom, there are three buttons: "Enregistrer" (dark grey), "Annuler" (dark grey), and "Supprimer mon compte" (red).

Figure 7 - Page de modification des informations

Cette page offre aussi la possibilité à l'utilisateur de supprimer son compte :



The screenshot shows the same "Mon compte" form, but with a modal dialog box overlaid in the center. The dialog has a red header "Suppression du compte." and the text "Cette action est irréversible. Souhaitez vous continuer ?". Below the text are two buttons: "Oui" (dark grey) and "Non" (dark grey). The background form is dimmed.

Figure 7 - Prompt de suppression de compte



### c. Création et modification d'un document

Une fois la feuille créée, l'utilisateur peut y accéder par le biais de la liste de feuilles :

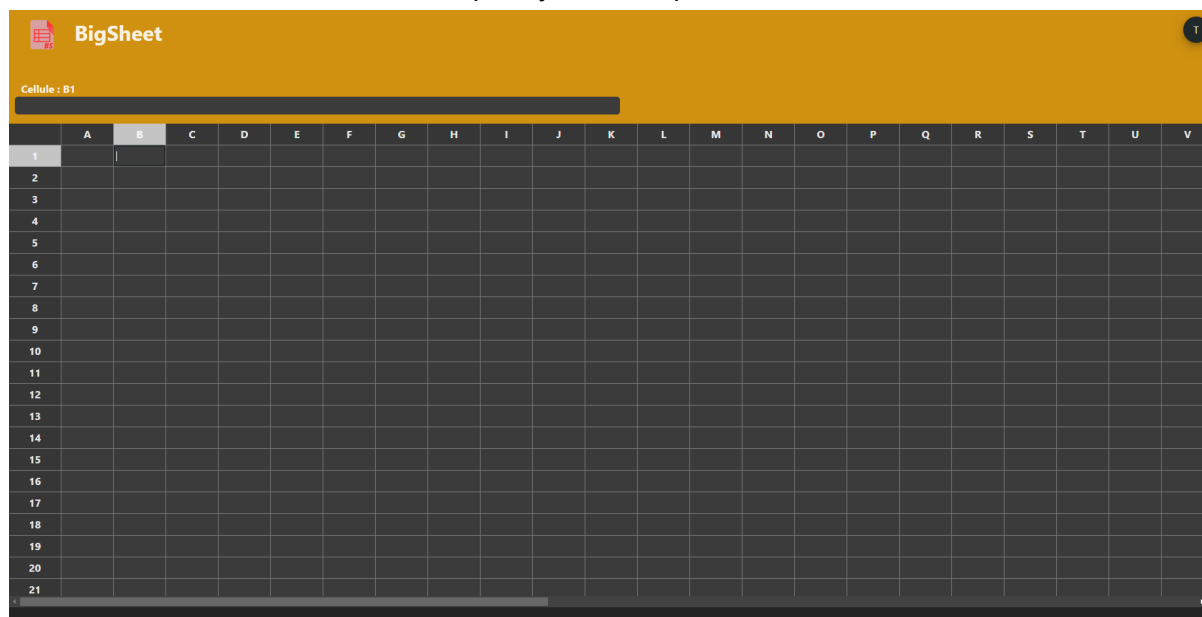


Figure 8 - Exemple de feuille vide

On peut voir ci-dessus un exemple de feuille nouvellement créée avec la cellule B1 étant sélectionnée. L'utilisateur peut alors saisir une chaîne de caractère qui est alors ajoutée à la cellule :

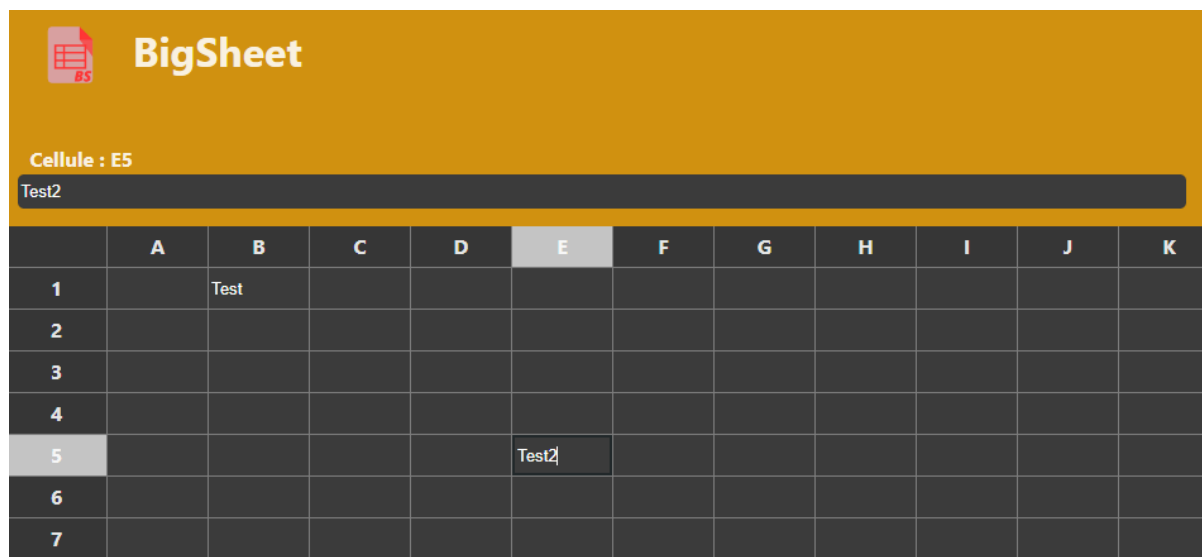


Figure 9 - Exemple de feuille avec des cellules remplies

On peut aussi observer qu'un affichage des coordonnées ainsi que du contenu de la cellule sélectionnée est affiché dynamiquement dans l'entête de la page.

## d. Partage d'un document avec droits d'accès

Il est possible de partager un document avec d'autres utilisateurs en ayant leur login. Le propriétaire d'une feuille peut donner le droit d'écrire ou juste de lire une feuille. On peut écrire le début d'un login puis il faut cliquer sur l'utilisateur voulu avant de valider.

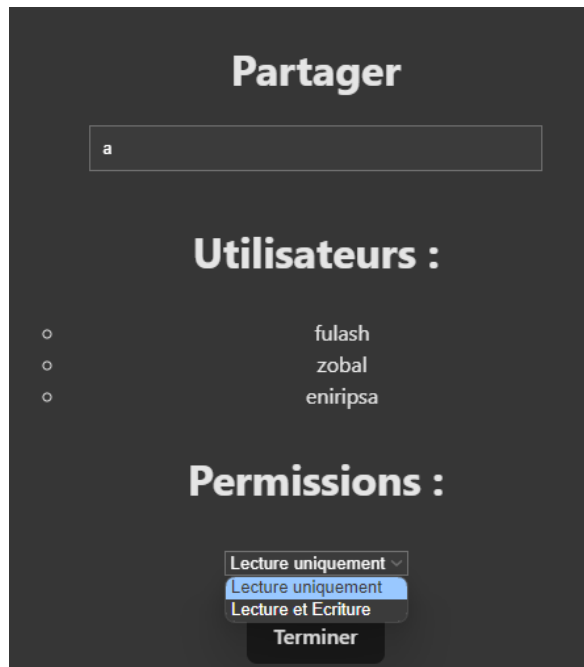


Figure 10 : Partage d'une feuille

## e. Connexion simultanée sur un document et synchronisation

Plusieurs utilisateurs peuvent se connecter simultanément.

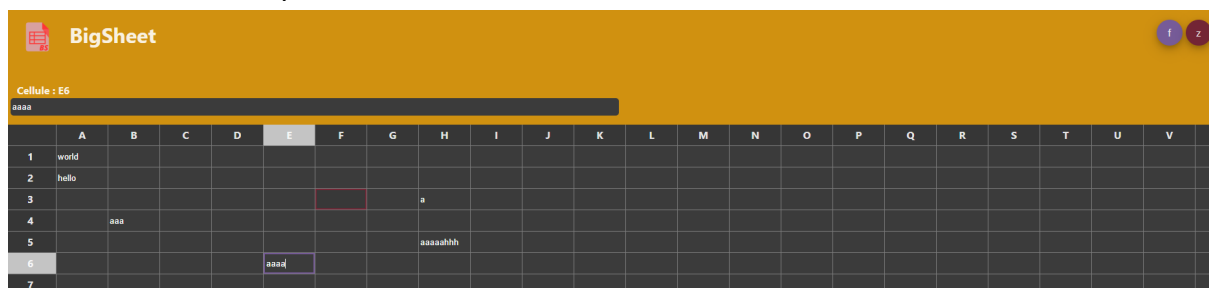


Figure 11 : Feuille avec deux utilisateurs connectés

Comme on peut voir sur la figure ci-dessus, un autre utilisateur est connecté sur la feuille et affiché en haut à droite. On peut voir les logins complets en passant le pointeur de la souris dessus. L'utilisateur violet clair a sélectionné la cellule E6. L'autre utilisateur, la cellule F3. Deux utilisateurs ne peuvent pas être sur la même cellule en même temps. Chacun voit en temps réel les modifications des autres.

Les modifications sont sauvegardées dans la base de données lorsque quelqu'un se connecte ou se déconnecte de la feuille mais aussi une fois toutes les 50 modifications.

### 3. Parties à développer (code, architecture...)

Certaines parties du projet ont une architecture précise à laquelle il a été consacré beaucoup de temps pour la conception et l'implémentation. Voici les détails de ces différentes parties.

#### a. Modèle

Comme on peut le voir sur le diagramme suivant, le modèle stocké dans la base de données est assez simple. Un utilisateur a un certain type d'accès sur des feuilles. Les feuilles contiennent des cellules. Afin de ne pas surcharger la base de données, les cellules vides ne sont pas sauvegardées.

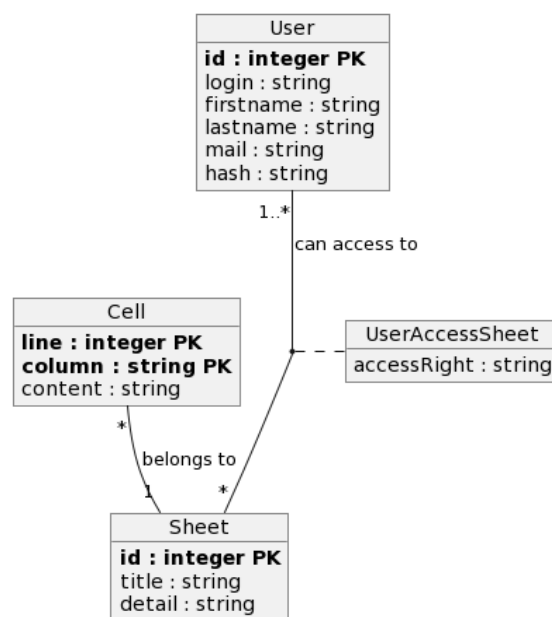


Figure 12 : Diagramme d'entité-association

Pour créer une table avec Sequelize, il faut créer une classe qui hérite de `sequelize.Model` puis initialiser chaque colonne comme suit.

```
SheetModel.init({
  {
    id: {
      type: DataTypes.INTEGER,
      autoIncrement: true,
      primaryKey: true
    },
    title: {
      type: DataTypes.STRING,
      allowNull: false,
      defaultValue: 'Sans-Nom'
    },
    detail: {
      type: DataTypes.STRING,
      allowNull: true,
    }
  },
  {
    sequelize,
    tableName: 'Sheet'
  }
});
```

Figure 13 : Création de la table Sheet

La complication arrive avec les associations. Si une table référence une autre table grâce à une clé étrangère, il faut absolument que l'autre table soit initialisée avant celle qui la référence. Il y a donc un ordre d'initialisation à respecter. Tous les modèles de l'application sont initialisés dans l'index.js qui est le main du serveur. A la base, les modèles étaient dans le répertoire "model" et étaient initialisés dans l'ordre alphabétique. Nous avons donc décidé de créer un répertoire "association" dans lequel il y a les modèles contenant une association et qui sont initialisés après les modèles classiques. Cette méthode pose problèmes s'il y a des associations à la chaîne mais est suffisante pour nos besoins.

Il y aussi certaines requêtes qui ont été compliquées. Par exemple, lorsqu'il faut récupérer une liste de feuilles dans la base de données, il faut aussi avoir le nom du propriétaire de la feuille sans avoir les autres utilisateurs qui ont accès à la feuille. Il faut donc faire une sélection parmi les utilisateurs qui ont un accès à la feuille.

```
static async getAccessibleByUser(userId) {  
  const sheets = await SheetModel.findAll({  
    attributes: ['id'], // get id  
    include: { // we include UserModel to do inner join  
      model: UserModel,  
      as: 'users',  
      attributes: [],  
      where: {  
        id: userId,  
      }  
    },  
  });  
  const sheetsId = sheets.map(sheet => sheet.id);  
  return await SheetModel.findAll({  
    attributes: ['id', 'title', 'detail', 'createdAt'], // get title and creation date  
    where: {  
      id: {  
        [Op.in]: sheetsId  
      }  
    },  
    include: { // we include UserModel to do inner join  
      model: UserModel,  
      as: 'users',  
      attributes: ['id', 'login'],  
      through: {  
        attributes: [], // we don't want attributes in UserAccessSheet  
        where: {  
          accessRight: Data.SERVER_COMPARISON_DATA.PERMISSIONS.OWNER  
        }  
      }  
    },  
  });  
}
```

Figure 14 : Code pour récupérer la liste des feuilles accessibles par un utilisateur

Comme on peut le constater sur la figure précédente, il faut sélectionner les feuilles appartenant à l'utilisateur voulu avec une première requête puis faire une autre requête pour récupérer ces feuilles en sélectionnant l'utilisateur ayant un accès "owner". Deux requêtes ont été nécessaires. On peut normalement faire des sous-requêtes en SQL afin d'inclure la première requête dans la deuxième et ainsi faire une seule grosse requête mais ce n'est pas possible avec Sequelize.

Nous aurions pu aussi modifier le modèle afin d'avoir un attribut "ownerId" dans la table Sheet mais cette méthode oblige une feuille à n'avoir qu'un seul propriétaire. Si, par la suite, il faut transformer le droit de propriétaire en droit administrateur avec plusieurs administrateurs pour une feuille, il faudrait apporter beaucoup de changements.

## b. Authentification

L'authentification est un point majeur du fonctionnement de l'application. Elle permet de distinguer les permissions et les données de deux utilisateurs connectés en simultané.

Néanmoins, il s'agit d'un épineux problème : "Comment peut-on authentifier un client de façon simplifiée et sécurisée ?". Il a été décidé d'opter pour une authentification par utilisation de JWT (Json Web Token). Il s'agit d'une clé qui peut être déchiffrée par le serveur et qui permet de stocker des données pouvant être utilisées plus tard.

Pour gérer l'authentification sans demander à l'utilisateur de se re-connecter assez fréquemment, nous avons opté pour l'utilisation d'une paire de tokens : Un est utilisé pour l'accès classique aux données, l'autre est utilisé pour obtenir une nouvelle paire de tokens lorsque le premier a expiré.

Dans le client VueJS, le fonctionnement de l'authentification est masqué par la classe API. Celle-ci permet de simplifier grandement les requêtes (avec ou sans authentification) en ne demandant qu'un ensemble très réduit de paramètres et en proposant des objets statiques permettant de simplifier la création de requêtes auprès d'autres classes du dossier DAO.

La classe API a été conçue pour être un singleton, permettant ainsi une utilisation centralisée des méthodes d'appel d'API, indépendamment de l'environnement d'exécution (dev ou prod).

La méthode request est la principale méthode d'appel d'API de la classe. Elle permet de réaliser des requêtes HTTP en utilisant différentes méthodes (GET, POST, PUT, DELETE, PATCH). Voici le fonctionnement détaillé :

1. **Construction de l'en-tête (header)** : La méthode construit un en-tête comprenant des éléments tels que l'origine, le type de contenu accepté, et le type de contenu fourni.
2. **Gestion du timeout** : Un mécanisme de timeout a été mis en place pour éviter des temps d'attente excessifs. Si la réponse de l'API n'est pas reçue dans le délai spécifié, une erreur de type `ErrorForDisplay` est renvoyée avec un code 504.
3. **Appel à l'API** : La méthode utilise l'API Fetch pour effectuer la requête vers le serveur distant. Elle gère les réponses réussies (code 200) et les réponses non réussies en renvoyant une instance d'`ErrorForDisplay` avec le code de statut et le message associé.

La méthode `request_logged` étend la fonctionnalité de la méthode `request` en gérant spécifiquement les aspects liés à l'authentification. Voici une analyse approfondie :

1. **Vérification de l'actualisation des tokens** : Avant d'effectuer une requête, la méthode vérifie si une actualisation des tokens est en cours. Si non, elle utilise la méthode `request` pour effectuer la requête en incluant le token d'accès dans l'en-tête.
2. **Gestion des réponses non autorisées** : En cas de réponse non autorisée (code 401), la méthode tente de rafraîchir automatiquement les tokens en appelant la méthode `User.refreshTokens()`. Si une actualisation est en cours, elle attend la fin de celle-ci avant de réessayer la requête.

3. **Utilisation d'un verrou pour le rafraîchissement** : Un mécanisme de verrou (`this.refreshLock`) est employé pour éviter que plusieurs tentatives simultanées de rafraîchissement des tokens ne se produisent. Cela assure une gestion cohérente des actualisations.

```
async request_logged(method, url, body, content_type) :Promise<any> {  
    let response;  
    if (!this.isRefreshing){  
        response = await this.request(method, url, body, content_type, {headers: {'Authorization': 'Bearer ${Storage.getAccessToken()}'}});  
    }  
  
    // If we get an Unauthorized response  
    // It would be of ErrorForDisplay type, so we can use default fields from this class  
    if ((response.error_code !== undefined && response.error_code === 401) || this.isRefreshing) {  
  
        // If we are not refreshing the token, we try refreshing it.  
        if (!this.isRefreshing) {  
            this.isRefreshing = true;  
            // If we were not able to refresh the tokens :  
            // Logout the user, redirect to login page, etc... are handled by the refresh tokens method  
            this.refreshLock = User.refreshTokens()  
                .then(() :void => { this.isRefreshing = false });  
        }  
  
        // Wait for the current refresh to complete before continuing  
        await this.refreshLock;  
        response = await this.request(method, url, body, content_type, {headers: {'Authorization': 'Bearer ${Storage.getAccessToken()}'}});  
    }  
    return response;  
}
```

Figure 15 : Méthode `request_logged` de la classe API.

La classe API offre une approche robuste et bien structurée pour la communication avec une API distante. Les mécanismes de timeout et de rafraîchissement des tokens garantissent une robustesse de la gestion des requêtes serveur. Cette classe constitue un élément central dans la gestion des requêtes d'API au sein de l'application.

Avec cette classe, tous les objets du dossier DAO proposent des méthodes de fetch extrêmement simplifiées. En voici un exemple :

```
/**  
 * This method fetches the data concerning the connected user.  
 *  
 * @returns {Promise<*>}  
 */  
1+ usages  👤 Maxime Marcellin  
static fetchUserData() :Promise<any> {  
    return api.request_logged(  
        api.METHODS.GET,  
        {url: User.BASE_PATH + "/me"  
    });  
}
```

Figure 16 : Exemple de Méthode de DAO faisant appel à la classe API.

Côté serveur, la gestion des Tokens se fait par un routeur proposant 3 routes :

- **/auth** : La route demande login et mot de passe et, s'ils sont valides, retourne une paire de tokens.
- **/refresh** : La route demande le token de refresh de l'utilisateur et, s'il est valide, renvoie une nouvelle paire de tokens. Elle place également le token de refresh utilisé en liste noire.
- **/logout** : La route demande la paire de tokens et les place dans une liste noire jusqu'à ce qu'ils expirent.

Ces 3 routes manient une base de données REDIS permettant de créer la liste noire des tokens. Il y a plusieurs avantages à cette approche :

- **Rapidité** : REDIS est une base de données chargée en mémoire RAM. Elle a donc une latence très faible. De plus, son implémentation en C la rend bien plus rapide que des systèmes comme PostgreSQL. C'est primordial pour des opérations très fréquentes d'authentification.
- **Expiration automatique des clés** : La principale justification à l'utilisation de Redis reste néanmoins dans sa méthode **setExp()**, permettant de définir des paires clés-valeurs avec une suppression automatique à l'expiration du token. Cela permet de ne pas stocker des tokens ayant expiré.

### c. Synchronisation

Afin de pouvoir afficher les modifications en temps réel pour tous les utilisateurs travaillant sur une même feuille, il y a 2 solutions. La première est le long-polling qui consiste à ce que le client envoie une requête HTTP au serveur à intervalle régulier afin qu'il puisse avoir toutes les informations nécessaires pour mettre à jour l'affichage. La deuxième est l'utilisation des sockets qui permettent à deux machines de se connecter entre elles et de s'envoyer des messages. Nous avons opté pour les sockets car le long-polling est assez lourd et le serveur aurait dû envoyer au client l'état complet de la feuille et les utilisateurs connectées à chaque fois. Au contraire, les sockets permettent d'envoyer la seule information nécessaire grâce à l'échange de messages. Par exemple : un utilisateur peut prévenir qu'il a modifié la cellule A1.

Avec les sockets, on peut connecter les clients entre eux directement ou les connecter au serveur et tous les messages qu'ils envoient passent par le serveur.

La première méthode peut poser des problèmes de sécurité car le serveur doit envoyer les adresses IP de chaque utilisateur pour qu'ils puissent se connecter entre eux. De plus, les utilisateurs peuvent s'envoyer n'importe quel message. La deuxième solution était donc la seule envisageable car le serveur peut filtrer les messages et une connexion à celui-ci est obligatoire pour sauvegarder les modifications d'une feuille dans la base de données. Voici un schéma de la méthode de connexion utilisée.

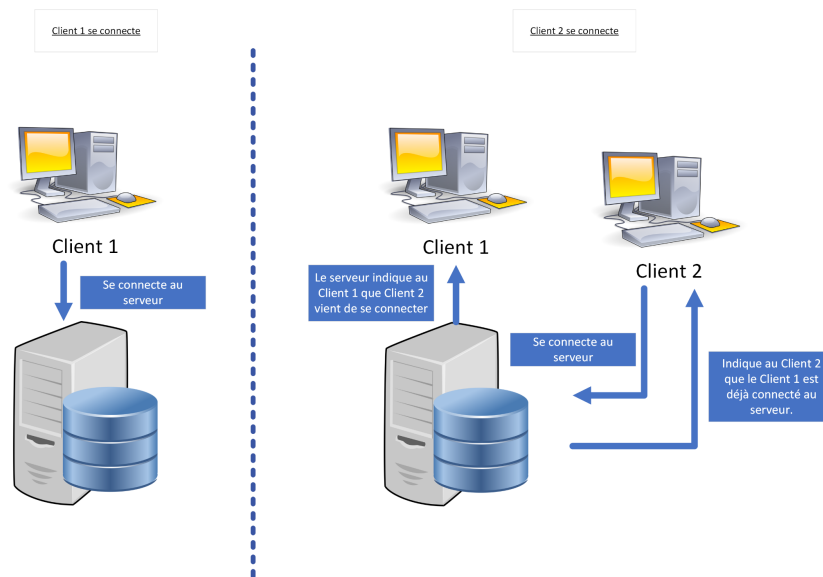


Figure 17 : La méthode de connexion client-serveur

La communication avec les sockets est séparée en deux parties. Le protocole qui dicte quels messages peuvent être envoyés ou reçus, et le gestionnaire qui gère toutes les connexions et effectue les modifications.

Dans un premier temps, le protocole de communication a été établi afin de savoir quel message le serveur doit traiter et quel message il peut envoyer. Afin de garantir l'intégrité du serveur, le serveur doit vérifier les messages reçus avant de le traiter.

```
FROM_CLIENT: {
  WRITE_CELL: {
    name: 'writeCell',
    checkerArg: writeCellChecker,
    event: writeCellEvent
  },
  SELECT_CELL: {
    name: 'selectCell',
    checkerArg: verifyCellCoord,
    event: selectCellEvent
  }
}
```

Figure 18 : Protocole pour les messages reçus par le serveur

Comme on peut voir sur la figure ci-dessus, les clients ont le droit d'envoyer 2 types de messages. Pour chaque type de message, le serveur vérifie les données que le client à envoyer grâce à la fonction "checkerArg" qui est liée. Si cette fonction dit que les données sont valides alors la fonction "event" est exécutée. Cette dernière traite les données et renvoie généralement un message aux autres clients.

Puis le gestionnaire des sockets a été créé. Ce gestionnaire attend les connexions des clients, gère la feuille, retransmet les messages... Il a fallu sécuriser la connexion afin que



n'importe qui ne puisse pas se connecter au serveur et faire n'importe quoi. Pour cela, lorsqu'un client tente de se connecter, le serveur renvoie une demande d'authentification. Le client a 5 secondes pour renvoyer son jeton d'authentification (celui qu'il reçoit du serveur lorsqu'il se connecte au site) et l'identifiant de la feuille à laquelle il veut accéder. Le serveur peut rejeter l'authentification et déconnecter le client si le jeton n'est pas valide ou que l'utilisateur n'est pas censé avoir accès à la feuille demandée.

Les clients connectés sur une feuille ne doivent pas voir les messages des autres clients qui sont sur d'autres feuilles. Pour cela, chaque socket client est lié à la feuille et à son compte utilisateur lors de l'authentification. L'utilisation du broadcasting était nécessaire. Cela permet de créer des salons de discussions et de transmettre les messages d'un client aux autres clients du même salon. Il y a donc un salon par feuille.

Afin d'éviter les conflits lors de l'écriture d'une cellule, nous avons décidé d'autoriser un seul utilisateur par cellule. Lorsqu'un client sélectionne une cellule, il avertit le serveur. Le serveur bloque alors la cellule en la liant à l'utilisateur. Les autres utilisateurs sont au courant des cellules sur lesquels sont chacun. Si quelqu'un tente d'accéder à une cellule déjà occupée, le serveur lui renvoie un message d'erreur. Lorsque que quelqu'un écrit, tout les utilisateurs reçoivent la modification mais la cellule n'est pas sauvegardée dans la base de données.

Il a donc fallu sauvegarder les cellules dans la base de données, à certains moments clés, afin de garantir la persistance du document dans le temps. Pour cela, lors de l'écriture d'une cellule, celle-ci est ajoutée dans une liste qui est liée à l'identifiant de la feuille. La fonction suivante permet de sauvegarder une feuille désignée par son identifiant.

```
async save(sheetId) {
  if (this.cellsNotSavedPerSheet[sheetId] !== undefined) {
    let iterator = this.cellsNotSavedPerSheet[sheetId].values();
    let item = iterator.next();
    while (!item.done) {
      const cell = item.value;
      if (cell.content === '' ) {
        await cell.destroy();
      } else {
        await cell.save();
      }
      item = iterator.next();
    }
    this.saveInNbModif[sheetId] = Data.SAVE_AFTER_MODIFICATION_COUNT;
    delete this.cellsNotSavedPerSheet[sheetId];
  }
}
```

*Figure 19 : Fonction pour la sauvegarde d'une feuille*

Comme on peut le voir, on sauvegarde les cellules s'il y en a qui doivent être sauvegardées. Si leur contenu est vide, elles sont supprimées de la base de données. De plus, on réinitialise le compteur pour la sauvegarde automatique qui a lieu toutes les 500 modifications. Il y a aussi une sauvegarde automatique lorsqu'un utilisateur se connecte ou se déconnecte. La sauvegarde lors de la connexion est obligatoire afin que l'utilisateur puisse récupérer toutes les cellules contenant du texte et être à jour.

## 4. Conclusion

Ce projet nous a permis d'apprendre le fonctionnement de JavaScript pour une utilisation côté serveur. Nous avons aussi conçu une architecture flexible qui permet d'ajouter facilement de nouvelles fonctionnalités. L'utilisation de VueJS est un plus nous ayant permis de séparer le backend du frontend.

# Guide de démarrage

Afin de pouvoir installer et utiliser BigSheet, il vous faut obligatoirement :

- NodeJS avec une version supérieure ou égale à 18.18.2
- Docker-compose avec une version supérieure ou égale à 2.15.1

Si vous n'êtes pas sûr des versions installées, tapez dans un terminal :

- `node -v`
- `docker-compose -v`

Pour installer le projet, ouvrez un terminal à la racine du projet puis tapez dans l'ordre :

- `cd frontend`
- `npm install`
- `cd ../backend`
- `npm install`

Si vous êtes sous Windows, tapez :

- `npm run config_win`

Si vous êtes sous Linux ou MacOS, tapez :

- `npm run config_unix`

Pour démarrer le projet, ouvrez un terminal à la racine du projet puis tapez dans l'ordre :

- `cd backend`
- `docker-compose up -d --force-recreate --no-deps --build`
- `cd ../frontend`
- `npm run dev`

Vous pouvez maintenant accéder à l'application avec l'URL suivante dans un navigateur :

- `http://localhost:5173`