Sylvain Hontans

BTS SIO 2eme Année

Rapport de Stage

Du 12 janvier au 20 février 2026

Maser Engineering
Green Park Bâtiment 1
57 avenue Jean Monnet
31770 Colomiers

Institut Limayrac

Année 2025-2026

INSTITUT LIMAYRAC

Sommaire

1. Présentation de la structure d'accueil

1.1. Présentation générale

Maser Engineering est une société de 561 collaborateurs répartie au sein de plusieurs entités dans toute la France. Son siège social se situe à Paris. Dans la figure 1 nous pouvons voir les différentes implantations de la société en France.

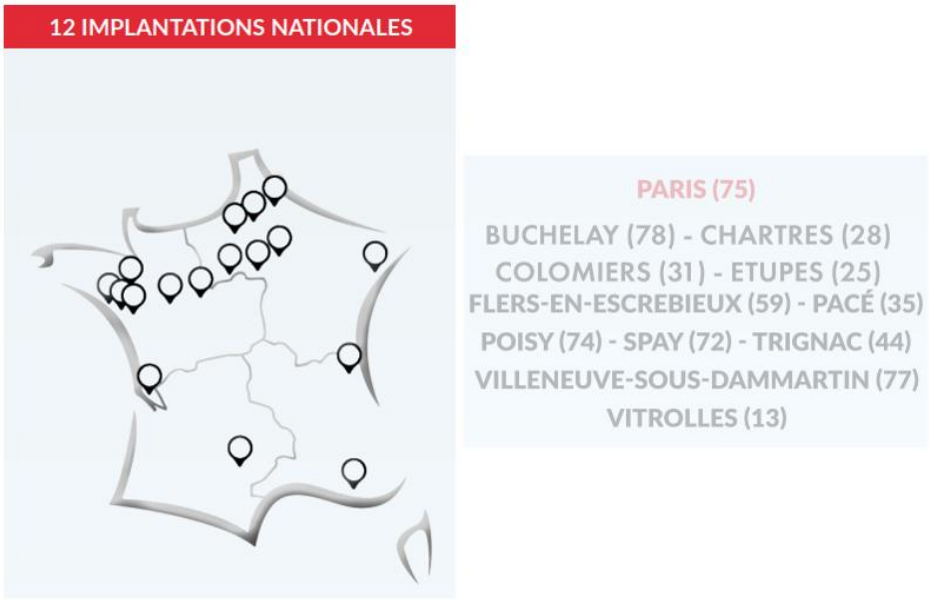


Figure 1 : Entités de Maser Engineering en France

1.2. Localisation

L'entité Maser Engineering ou j'ai été accueilli se situe à Colomiers au 57 avenue Jean Monnet. Dans la figure 2 nous pouvons voir la localisation des locaux par rapport à l'aéroport Toulouse-Blagnac.

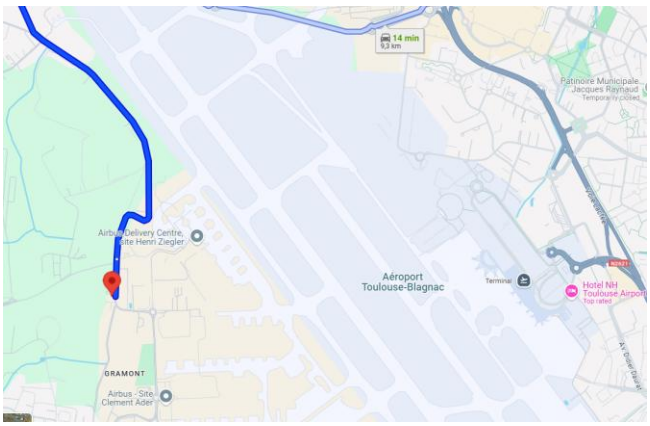


Figure 2 : Localisation de l'entreprise

Ci-dessous dans la figure 3 nous pouvons voir l'entrée du bâtiment de Colomiers.

Figure 3 : Entrée du bâtiment Maser Engineering

1.3. Statut juridique

La société a été créé en 1973. Maser Engineering est une société à responsabilité limité (SARL), immatriculée sous le numéro SIREN 732050026. Maser Engineering est une filiale du Groupe CRIT (voir figure 4), acteur majeur en Ressources Humaines, Assistance Aéroportuaire, Ingénierie et Maintenance Industrielle, côté sur Euronext Paris.

Figure 4 : Maison mère de Maser Engineering

1.4. Cœur de métier

Maser Engineering contribue au développement de la performance de ses clients dans les secteurs Aéronautique, Aéroportuaire, Agroalimentaire, Automobile, Naval, Eolien et Pharmaceutique (Voir figure 5).

Les quatre pôles d'expertise de Maser Engineering se trouvent dans le conseil, la maintenance, les projets et la formation industrielle.

- Le conseil : Concepteur de solutions, dans une démarche d'amélioration continue, pour favoriser la transformation numérique et la productivité des entreprises.
- la maintenance industrielle : La maintenance des moyens de production, d'exploitation et leurs périphériques est l'une des expertises clés et historiques de Maser Engineering
- La gestion de projets industriel : Maser Engineering accompagne les industriels dans leurs projets globalisés, d'implantation, de transfert et de mise au point, de modernisation et d'optimisation de leurs unités automatisées de production et/ou d'exploitation
- la formation industrielle : le pôle formation de Maser Engineering accompagne les entreprises dans la montée en compétences de leurs ressources internes.

Figure 5 : Les différents secteurs d'activités

1.5. Clients

Cette société travaille pour plus de 280 grands groupes industriels tels qu'Airbus, Airbus, Total Energies, etc…Quelques grands clients sont cités dans la figure 6 ci-dessous.

Figure 6 : Quelques gros clients

1.6. Chiffres clés

La société Maser Engineering est composé de 561 collaborateurs répartis dans toute la France. Elle travaille pour plus de 300 clients dans le monde et son chiffre d'affaires en 2024 s'élevait à plus de 57 millions d'euros.

Figure 7 : Chiffres clés

2. Présentation du contexte du stage

2.1. Maître de stage

Mon maitre de stage monsieur Valentin Jauze est chef d'équipe au sein de la société Maser Engineering.
De plus j'ai été formé par Monsieur Robert Hoffmann, développeur full stack, dans le service informatique.

2.2. Positionnement dans l'organisation

L'organisation du pôle conseil dans lequel j'ai effectué mon stage est donné dans la figure 8. J'ai effectué mon stage au sein de l'équipe Business développement Ingénierie de maintenance et méthodes, avec mes maitres de stage.

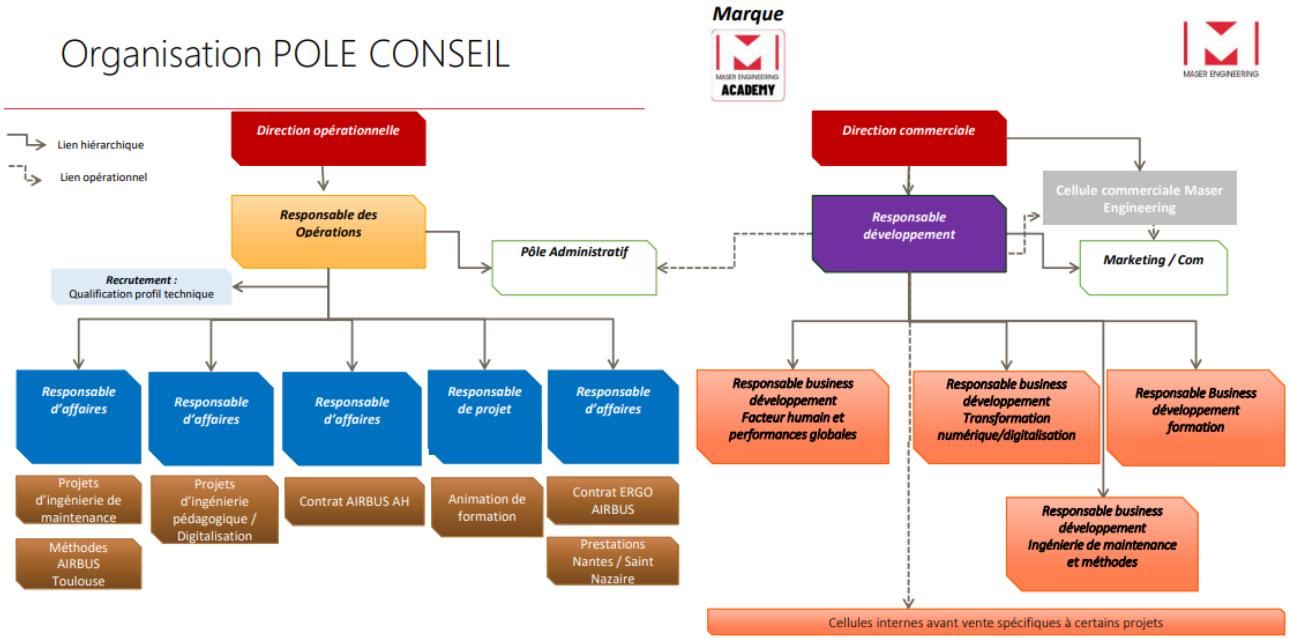


Figure 8 : Organisation du pôle Conseil

2.3. Missions réalisées

Le stage a consisté à créer une application supportant à la fois les systèmes Android et iOS. Nous avons abandonné l'approche native en Java au profit d'une application web hybride, permettant de cibler les deux plateformes avec une seule base de code. Cette application répond à la demande d'un client de Maser, qui est directeur ergonome. Il travaille sur le projet Excalibur, qui consiste à accompagner l'équipe projet pour concevoir les situations futures de la prochaine usine de Rennes en intégrant l'ergonomie. Pour cela, il a confié à Maser la tâche de faciliter et d'automatiser au maximum la saisie d'audits d'ergonomie qui se faisaient initialement dans des fichiers Excel.

3. Environnement technique

3.1. Ressources logicielles

Durant mon stage j'ai utilisé différentes technologies.
Dans un premier temps, pour définir la structure de mon application ainsi que les différentes pages et interfaces, j'ai utilisé l'outil Balsamiq (figure 9). Il s'agit d'un logiciel de conception de maquettes fonctionnelles (wireframes), facile à prendre en main. Les utilisateurs n'ont pas besoin de connaissances particulières en conception webdesign.

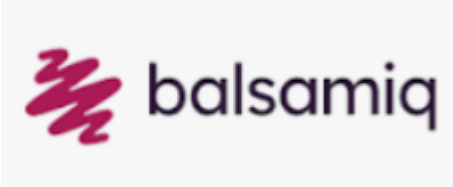


Figure 9 : Outil de maquettage (Balsamiq)

Pour créer et gérer les dépendances du projet, j'ai utilisé le gestionnaire de paquets JavaScript PNPM de Node.js. Avec PNPM (Performant Node Package Manager), on peut installer des packages Node.js dans l'application. Un paquet Node.js est un répertoire contenant un ou plusieurs modules ou bibliothèques JavaScript. Il est utilisé pour ajouter diverses fonctionnalités aux applications ou aux scripts. Sans paquets, un développeur doit réécrire du code pour chaque fonctionnalité dont son projet a besoin.

Figure 10 : Gestionnaire de bibliothèques JavaScript

Pour construire une application web couvrant un large spectre de fonctionnalités, tant pour le front-end que pour le back-end, le choix technologique principal NuxtJS, s'est avéré le plus judicieux. NuxtJS s'appuie sur le framework Vue.js et s'intègre parfaitement avec Tailwind CSS. De plus, nous avons choisi TypeScript (TS), une surcouche de JavaScript qui apporte le typage statique afin de prévenir certaines erreurs dès l'écriture du code.

Figure 11 : Framework utilisé (NuxtJS)

Nous avons choisi d'utiliser Tailwind CSS, un framework CSS qui permet d'écrire du CSS utilisable directement dans les balises HTML.

Figure 12 : Framework utilisé (Tailwind CSS)

Nuxt UI est une librairie qui permet d'intégrer à un projet des composants d'interface web préfabriqués (c'est une surcouche de NuxtJS).

Figure 13 : Librairie utilisé (Nuxt UI)

DexieJS est une librairie qui permet d'intégrer et d'interagir avec une base de données locale dans le navigateur, et plus précisément avec l'instance Chrome de l'application mobile.

Figure 14 : Librairie utilisé (DexieJS)

Pour le stockage des audits et le questionnaire du client, j'ai dû utiliser le drive Box employé par Maser. Grâce à l'espace développeur de Box, il est possible de créer une application utilisant la méthode d'authentification OAuth. Cette authentification permettra à l'application de communiquer avec le drive via les différentes API (Application Programming Interface) de Box.

Figure 15 : Drive Box (Box API)

OAuth est une méthode d'authentification. Pour assurer l'authentification entre l'application et le drive, nous avons choisi d'utiliser la librairie NuxtAuth, qui propose un provider Box.

Figure 16 : Librairie utilisé (NuxtAuth)

Pour assurer la transition entre l'application web et l'application mobile, le framework Capacitor est parfaitement adapté à ce besoin. Il fonctionne en créant un exécutable pour les téléphones qui, une fois lancé, ouvre une instance de Chrome à travers l'application mobile dans notre cas pour un projet Nuxt.

Figure 17 : Framework utilisé (Capacitor)

Pour éditer le code, j'ai utilisé Visual Studio Code, ainsi que différentes extensions comme l'extension officielle Vue.js, essentielles pour gérer l'autocomplétion et analyser le code.

Figure 18 : Editeur de code

Pour gérer les différentes versions du code en local, j'ai utilisé Git, qui m'a permis de suivre les modifications et de revenir à des versions précédentes en cas d'erreur.

Figure 19 : Gestionnaire de versions

J'ai intégré GitHub pour sauvegarder à distance les différentes versions du projet.

Figure 20 : Sauvegarder du projet à distance

4. Création d'une application web hybride

4.1. Compétences mises en œuvre

Compétences issues du BLOC 1 du référentiel - Support et mise à disposition de services informatiques :

1) Gérer le patrimoine informatique :
Comme précisé plus tôt, dans le cadre de la compétence « gérer le patrimoine informatique », j'ai utilisé la librairie DexieJS, qui m'a permis d'intégrer une base de données dans l'application elle-même et d'interagir selon les besoins des utilisateurs.

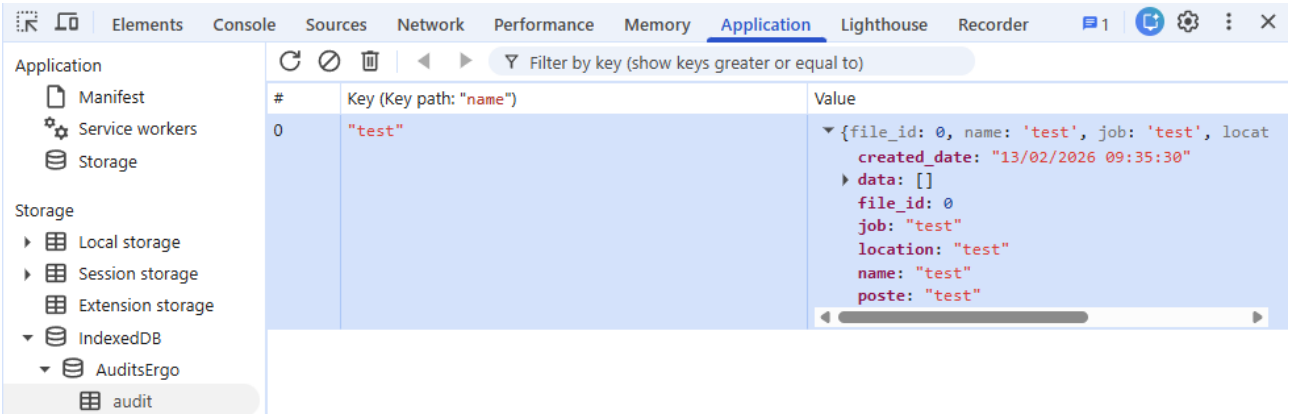


Figure 21 : Base de données du navigateur (DexieJS)

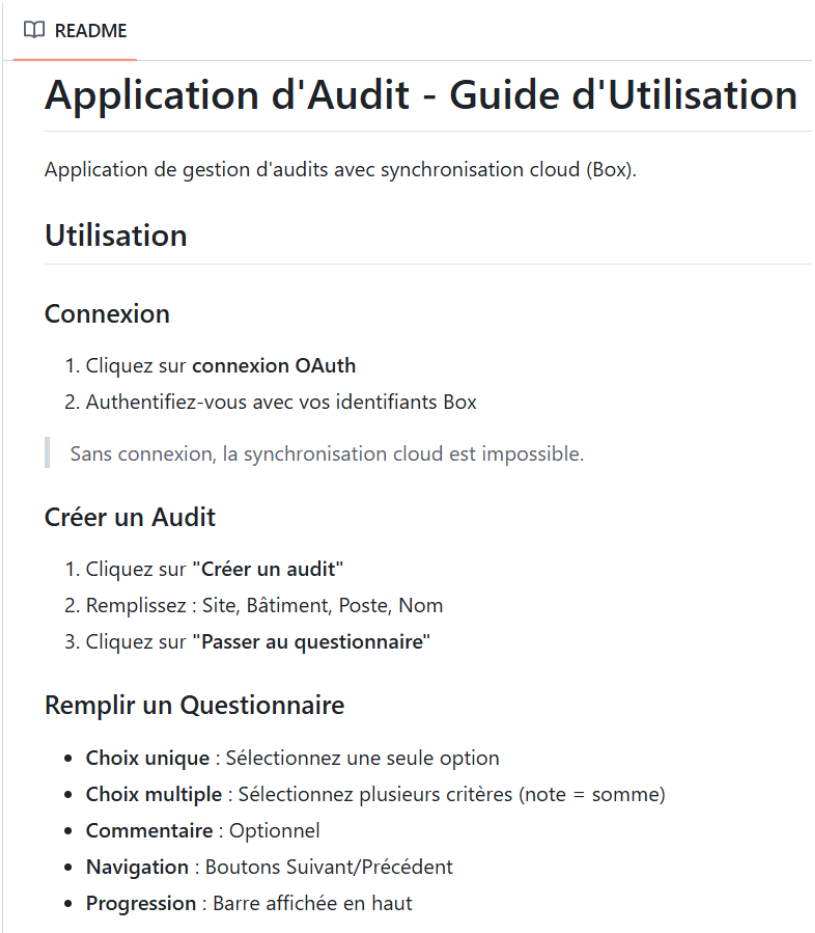
De plus, l'un des points importants de ce projet, en rapport avec la compétence « Gérer le patrimoine informatique », est la gestion des versions du projet avec GitHub (versioning).

Figure 22 : Deux versions du projet (Commits)

2) Répondre aux incidents et aux demandes d'assistance et d'évolution :
Pour pouvoir retracer et signaler les différents problèmes et bugs de mon application, j'ai utilisé les issues sur GitHub. L'approche est très simple : lorsque je constate un bug ou un problème, j'ouvre une issue correspondante que je clôture ensuite une fois le bug résolu.

Figure 23 : Bug clôturé sur GitHub (Issue)

3) Mettre à disposition des utilisateurs un service informatique :
Dans le cadre de cette compétence, j'ai pu établir une documentation sur l'utilisation détaillée et le fonctionnement de l'application, que l'on peut retrouver au sein du dépôt GitHub. Elle servira ensuite de guide d'utilisation et de maintenance pour le client.


README

Application d'Audit - Guide d'Utilisation

Application de gestion d'audits avec synchronisation cloud (Box).

Utilisation

Connexion

1. Cliquez sur **connexion OAuth**
2. Authentifiez-vous avec vos identifiants Box

> Sans connexion, la synchronisation cloud est impossible.

Créer un Audit

1. Cliquez sur "**Créer un audit**"
2. Remplissez : Site, Bâtiment, Poste, Nom
3. Cliquez sur "**Passer au questionnaire**"

Remplir un Questionnaire

- **Choix unique** : Sélectionnez une seule option
- **Choix multiple** : Sélectionnez plusieurs critères (note = somme)
- **Commentaire** : Optionnel
- **Navigation** : Boutons Suivant/Précédent
- **Progression** : Barre affichée en haut

Figure 24 : Extrait de la documentation

4) Travailler en mode projet :
Dans le cadre de cette compétence, j'ai pu mener une démarche professionnelle en participant à des réunions régulières avec le client et en établissant un cahier des charges que nous allons aborder maintenant.

4.2. Cahier des charges

L'objectif principal du stage est de développer une application web.

Description générale :

Tout d'abord, il est important de comprendre ce qu'est l'ergonomie dans les entreprises. Il s'agit de l'ensemble des méthodes visant à adapter le travail, les outils et l'environnement aux capacités et aux besoins des employés, afin d'améliorer leur confort, leur sécurité et leur efficacité. Dans notre contexte, les ergonomes effectuent des inspections dans des lieux tels que des bureaux, des bâtiments et des sites industriels. Cela s'appelle des audits : ils sont réalisés au moyen d'un questionnaire universel, lui-même composé de différents critères correspondant à des questions comme « Le niveau sonore permet-il de travailler sans gêne ? », suivies de plusieurs propositions de réponses.

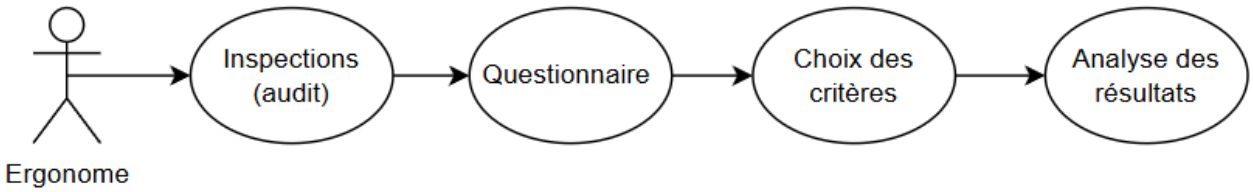


Figure 25 : Processus des ergonomes

Le besoin du client est d'avoir une application mobile (Android et IOS). Elle doit comporter les fonctionnalités suivantes :

- Accéder librement à l'application sans être authentifié auprès du drive Box.

- Créer des audits avec les informations : site concerné, bâtiment, poste et nom.

- Lister les audits, les rechercher, les modifier (changer les réponses du questionnaire) et les supprimer.

- Compléter le questionnaire avec les critères issus des tableaux Excel (choix uniques et multiples).

- Afficher un récapitulatif des critères sélectionnés avec calcul d'une note finale de l'audit (méthode utilisée par les ergonomes).

- Fonctionner hors connexion pour la gestion des audits (tous les points ci-dessus).

- Envoyer individuellement un audit sur le drive Box.

- Synchroniser globalement tous les audits et le questionnaire.

Le but est de faciliter le travail chronophage de réalisation des audits et d'automatiser la mise à jour du questionnaire si le client décide de le faire évoluer. Pour cela, le questionnaire sera standardisé et stocké au format JSON sur le drive Box.

Description détaillée :
Ci-dessous, je vous présente la description détaillée de l'application, subdivisée en trois grandes parties, ainsi que chaque action et information disponible.

L'application :

- **Page d'accueil**
  - Avatar (Drop down menu)
    - Se connecter (si non authentifié à Box)
    - Bouton synchroniser avec Box (si authentifié à Box)
    - Se déconnecter (si authentifié à Box)
  - Recherche d'audit
  - Liste des audits
    - Bouton modifier audit
    - Bouton envoyer audit (si authentifié a Box)
    - Bouton supprimer audit
  - Bouton créer audit
    - Modal de création d'audit
      - Nom de l'audit
      - Site concerné
      - Bâtiment
      - Poste
      - Bouton passer au questionnaire

- **Page Questionnaire**
  - Barre de progression du questionnaire
  - Nombre de critère effectué sur nombre total
  - Critère courant
    - Bouton plus de détail sur le critère
    - Liste des réponses (choix multiple ou unique)
    - Texte area ajoute un commentaire (non obligatoire)
  - Bouton critère précédent
  - Bouton critère suivant (un choix minimum)

- **Page récapitulatif**
  - Nombre total de critère effectués
  - Liste des critères effectués
    - Nom de la question
    - Catégorie du critère
    - Note du critère
  - Note de l'audit (calcul spécifique et automatique)

Figure 26 : Spécifications fonctionnelles

4.3. Démarche / Mode opératoire

A partir du cahier des charges et en utilisant le logiciel Balsamiq, j'ai maquetté chaque page de l'application. Grâce à cette approche, j'ai obtenu une vue sommaire de l'ensemble, permettant de visualiser la position des boutons, les icônes, etc… Cela permet également de définir les actions possibles pour l'utilisateur. Dans ce rapport, nous verrons toutes les pages afin de les détailler le plus précisément possible.

La figure 27 ci-dessous représente une vue schématique de la page d'accueil. Sa mise en forme et les composants qu'elle contient respectent le cahier des charges. Les fonctionnalités proposées à l'utilisateur sont : le listing des audits ; la recherche d'un audit par nom ; la création d'un nouvel audit via un bouton ouvrant une modale ; la modification d'un audit, qui renvoie à la page du questionnaire ; l'envoi d'un audit et la suppression d'un audit.

Les autres fonctionnalités incluses sur cette page sont : un clic sur l'icône « Avatar » ouvrant un sous-menu (drop-down) permettant de se connecter si nécessaire, de synchroniser les audits avec Box et de se déconnecter via des boutons.

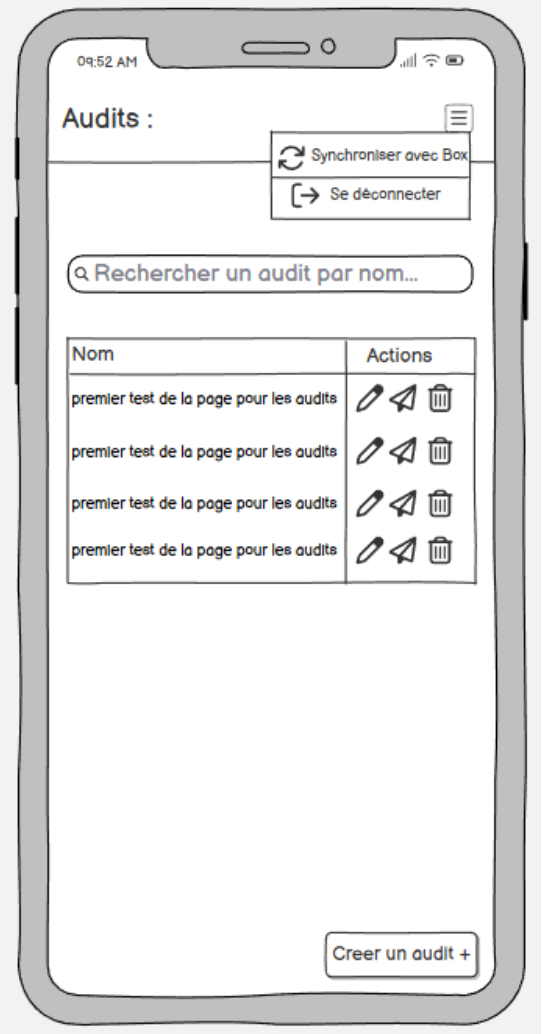


Figure 27 : Page d'accueil (Balsamiq)

De la même manière, via l'outil Balsamiq, on voit dans la figure ci-dessous la modale qui s'affiche une fois le bouton « Créer un audit » activé. Sa mise en forme et les composants qu'elle contient respectent le cahier des charges : l'utilisateur peut créer un audit en renseignant un nom, un site, un bâtiment et un poste.

Les autres fonctionnalités présentes sur cette page sont : un clic sur l'icône « croix » pour fermer la modale et un bouton « Passer au questionnaire » qui mène à la page du questionnaire et enregistre l'audit.

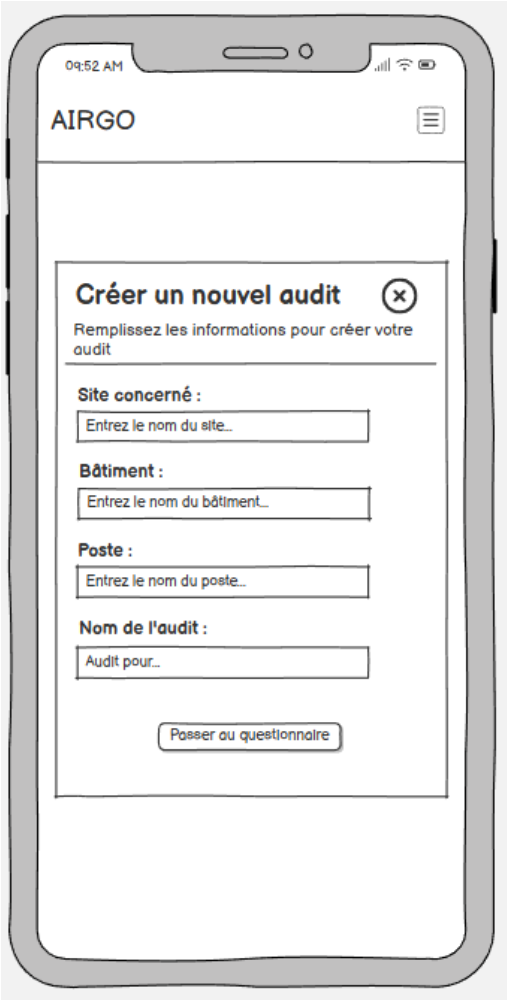


Figure 28 : Modal de création d'audit (Basalmiq)

Toujours via l'outil Balsamiq, on peut voir dans la figure ci-dessous la page questionnaire. Une fois l'audit créé, ou en cliquant sur l'icône « modifier » d'un audit dans la liste de la page d'accueil, la page questionnaire s'ouvre. Sa mise en forme et les composants qu'elle contient respectent le cahier des charges : une barre de progression qui avance au fur et à mesure que le questionnaire est complété ; l'affichage du nombre de critères complétés sur le nombre total ; le nom du critère courant (position de l'utilisateur dans le questionnaire) ; et une icône « ? » permettant d'obtenir plus de détails sur le critère courant.

Les autres fonctionnalités de cette page sont : la liste des réponses au critère, qui peut être multiple (cases à cocher) ou unique (boutons radio) ; la possibilité d'ajouter un commentaire. En bas de la page, un bouton « Précédent » : si le critère courant est le premier, il ramène l'utilisateur à la page d'accueil, sinon il passe au critère précédent. Enfin, le bouton « Suivant » passe au critère suivant ; si c'est le dernier critère, il ramène l'utilisateur à la page résumé.

Figure 29 : Page questionnaire (Basalmiq)

Enfin dans la figure 30 ci-dessous, on peut voir la page résumée. Une fois le questionnaire terminé, la page résumée s'affiche pour récapituler les notes attribuées aux différents critères. On peut y voir le nombre de questions traitées sur le nombre total de questions.

Les autres fonctionnalités présentes sur cette page comprennent le calcul du total des notes, effectué selon la méthode utilisée par les ergonomes.

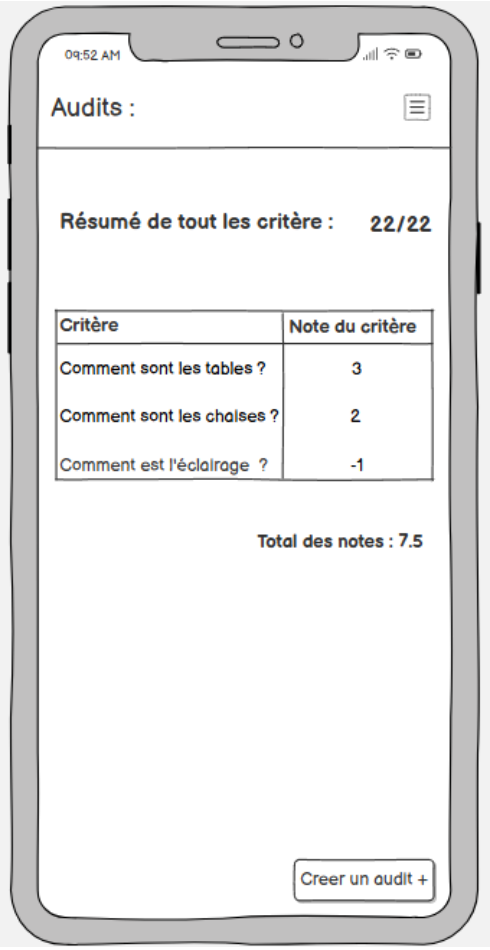


Figure 30 : Page résumé (Basalmiq)

4.3.1. Contexte technique du projet

Dans cette partie du rapport, nous présenterons toutes les pages de l'application, à l'exception de la page de résumé qui n'apporte pas d'élément notable. Nous en profiterons pour détailler au maximum chaque fonctionnalité ainsi que les différentes possibilités d'utilisation qu'offre l'application. Dans un souci de clarté, j'expliquerai ce qui a été nouveau pour moi et ce qui diffère par rapport à la formation SIO SLAM. Dans la suite, afin de faciliter la compréhension, du texte en couleur a été introduit et est associé aux parties encadrées de la même couleur dans les figures.

4.3.2. Base de données

À partir des différentes pages créées avec le logiciel Balsamiq et du cahier des charges, j'ai défini la démarche pour mener à bien ce projet. Tout d'abord, j'ai créé une base de données en suivant la très bonne documentation de DexieJS. Par la suite, j'ai testé l'ajout, la modification et la suppression d'éléments dans ma base afin de mieux évaluer l'étendue des possibilités qui s'offraient à moi.

La création a été réalisée dans un simple fichier bd.ts, où j'ai défini la structure de ma base de données nommée AuditsErgo. Elle est constituée d'une table audit dont la clé primaire est name, et d'une table questions dont la clé primaire est title. La structure des tables est définie à l'aide d'interfaces. On peut faire un parallèle avec la programmation orientée objet : les objets sont ici les tables et les interfaces représentent la structure de ces objets. Pour les audits, par exemple, j'ai défini l'interface AuditSchema, elle-même composée de l'interface AuditDataSchema. Quant à la table questions, elle est définie par l'interface QuestionSchema, elle-même composée de CriteriaSchema.

```ts
interface AuditShema {
    file_id: number,
    name: string,
    batiment: string,
    poste : string,
    site: string,
    created_date: Date,
    update_date: Date,
    data: AuditDataShema[]
}

interface AuditDataShema {
    choices : number[],
    note: number,
    comment : string,
    documents : string[]
}

interface QuestionSchema {
    title: string,
    category: string,
    help: string,
    multiple: boolean,
    criteria: CriteriaSchema[]
}

interface CriteriaSchema {
    title: string,
    notation: number
}

const db = new Dexie('AuditsErgo') as Dexie & {
    audit: EntityTable<AuditShema>;
    questions: EntityTable<QuestionSchema>
};

db.version(1).stores({audit: 'name',questions: 'title'});
```

Figure 31 : Base de données (db.ts)

4.3.3. Page d'accueil

Une fois la base de données établie, j'ai réalisé la page d'accueil, dont la version finale, visible sur téléphone par l'utilisateur, est présentée dans la figure 32 ci-dessous.

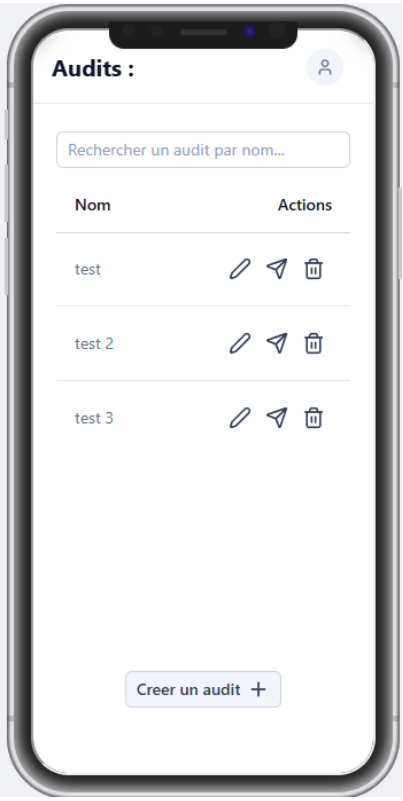


Figure 32 : Page d'accueil

4.3.3.1. Tableau d'audit

Le premier composant que nous allons détailler est la liste des audits présenté sous forme de tableau. Nous y verrons également deux fonctionnalités : modifier et supprimer. La fonctionnalité « envoyer » sera abordée plus tard, par souci de cohérence avec ce qui a été présenté jusqu'à présent. Le tableau est visible ci-dessous.

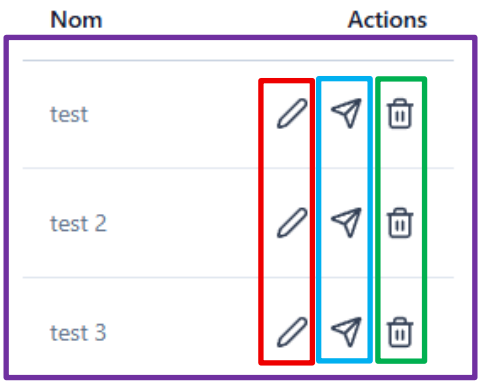


Figure 33 : Composant tableau (Nuxt UI)

Dans la figure 34, on voit la structure du tableau, un composant de Nuxt UI, ainsi que le code TypeScript pour la modification et la suppression des audits. À noter que la recherche d'audits se fait automatiquement à partir d'un nom d'audit.

```vue
<template>
    <UTable ref="table" :data="auditList" :columns="columns" class="flex-1" />
</template>

<script lang="ts" setup>
    import type { TableColumn } from '@nuxt/ui'
    import type { AuditShema } from '~/dataBase/db'
    import { db } from '~/dataBase/db'
    const UButton = resolveComponent('UButton')
    const Audits: AuditShema[] | any = await db.audit.toArray()
    const auditList = ref<AuditShema[]>(Audits)
    const columns: TableColumn<AuditShema>[] = [
        {id: 'name', accessorKey: 'name', header: 'Nom'},
        {
            header: 'Actions', id: 'actions', meta: {
                class: {th: 'text-right', td: 'text-right'}
            },
            cell: ({ row }) => {
                return h
                ('div',
                    [
                        h(UButton, {
                            icon: 'i-lucide-pen',
                            color: 'neutral',
                            variant: 'ghost',
                            to: `/${row.original.name}/questionnaire-0`
                        }),
                        h(UButton, {
                            icon: 'i-lucide-trash-2',
                            color: 'neutral',
                            variant: 'ghost',
                            onclick: async () => {
                                const name = row.original.name
                                console.log(name)
                                await db.audit.where('name').equals(name).delete()
                                window.location.reload();
                            }
                        })
                    ]
                )
            }
        }
    ]
</script>
```

Figure 34 : Structure du « tableau » et code Type Script

Dans cette structure, nous allons détailler les fonctionnalités suivantes :

1. Affichage d'un audit pour chaque enregistrement de la base de données.
2. Possibilité de modifier l'audit de son choix.
3. Possibilité de supprimer l'audit de son choix.

Le tableau est alimenté par « auditList », elle-même remplie en parcourant l'objet « Audits ». Il correspond au contenu de la table audit dans la base de données. À noter que les deux objets sont de type « AuditSchema » : les crochets indiquent qu'il s'agit d'un tableau d'objets.

```
10      const Audits: AuditShema[] | any = await db.audit.toArray()
11      const auditList = ref<AuditShema[]>(Audits)
```

Figure 35 : Objets « Audits » et « auditList »

Le fonctionnement de la modification d'un audit est très simple, car je renvoie l'utilisateur à la page du questionnaire. C'est là que nous verrons comment se font les mises à jour de la base de données. En attendant, voici comment fonctionne le bouton Modifier de la liste d'audits. Tout d'abord, j'ai créé un objet « columns », composant de Nuxt UI. J'ai défini deux colonnes: « Nom » et « Action ».

```
const columns: TableColumn<AuditShema>[] = [
    {id: 'name', accessorKey: 'name', header: 'Nom'},
    {
        header: 'Actions', id: 'actions', meta: {
            class: {th: 'text-right', td: 'text-right'}
        },
        cell: ({ row }) => {
            return h
            ('div',
                [
                    h(UButton, {
                        icon: 'i-lucide-pen',
                        color: 'neutral',
                        variant: 'ghost',
                        to: `/${row.original.name}/questionnaire-0`
                    }),
                    h(UButton, {
                        icon: 'i-lucide-trash-2',
                        color: 'neutral',
                        variant: 'ghost',
                        onclick: async () => {
                            const name = row.original.name
                            console.log(name)
                            await db.audit.where('name').equals(name).delete()
                            window.location.reload();
                        }
                    })
                ]
            )
        }
    }
]
```

Figure 36 : Objet « columns » Nuxt UI

De plus, dans les cellules de la colonne « Action », j'ai défini un bouton de Nuxt UI. Il renvoie vers l'URL correspondant au nom de l'audit sélectionné, suivi du reste du chemin. Cela se fait via un data-binding.

```
22  h(UButton, {
23      icon: 'i-lucide-pen',
24      color: 'neutral',
25      variant: 'ghost',
26      to: `/${row.original.name}/questionnaire-0`
27  }),
```

Figure 37 : Bouton Nuxt UI (modifier)

Le bouton Supprimer, situé dans la colonne « Action », supprime l'enregistrement dans la table audit. On utilise le champ « name » de l'audit sélectionné et on le supprime s'il correspond à la condition passée à la clause WHERE, comme en MySQL.

```
28  h(UButton, {
29      icon: 'i-lucide-trash-2',
30      color: 'neutral',
31      variant: 'ghost',
32      onclick: async () => {
33          const name = row.original.name
34          await db.audit.where('name').equals(name).delete()
35      }
```

Figure 38 : Bouton Nuxt UI (supprimer)

4.3.3.2. Bouton et modal de création d'audit

Enfin, nous allons voir le dernier composant de la page d'accueil : le bouton et la modal « Créer un audit ».

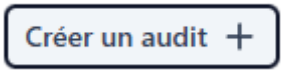


Figure 39 : Bouton Nuxt UI et modal (Créer un audit)

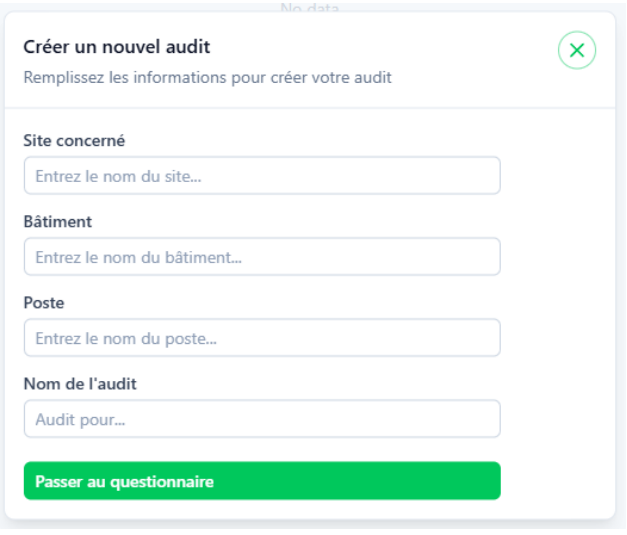


Figure 40 : Modal Nuxt UI (Créer un audit)

Le bouton n'ouvre en réalité qu'une modal Nuxt UI superposée à la page d'accueil. Dans cette modal, on renseigne les informations de l'audit que l'on souhaite créer. C'est lorsque l'on clique sur le bouton « Passer au questionnaire » que le nouvel audit est enregistré dans la base de données dans le code TypeScript.

```vue
<template>
    <UModal title="Créer un nouvel audit" description="Remplissez les informations po
        <template #body>
            <UFormField label="Site concerné" class="mb-3"/>
            <UInput v-model="site" placeholder="Entrez le nom du site..."/>
            <UFormField label="Bâtiment" class="mb-3"/>
            <UInput v-model="batiment" placeholder="Entrez le nom du bâtiment..."/>
            <UFormField label="Poste" class="mb-3"/>
            <UInput v-model="poste" placeholder="Entrez le nom du poste..."/>
            <UFormField label="Nom de l'audit"/>
            <UInput size="md" placeholder="Audit pour..." v-model="nameOfNewAudit"/>
            <UButton  label="Passer au questionnaire" @click="addAudit()"/>
        </template>
        <UFooter class="fixed bottom-px right-px left-px">
            <template #right>
                <UButton label="Créer un audit" trailing-icon="i-lucide-plus"/>
            </template>
        </UFooter>
    </UModal>
</template>
<script lang="ts" setup>
    import type { AuditShema } from '~/dataBase/db'
    import { db } from '~/dataBase/db'
    const site = ref('')
    const batiment = ref('')
    const poste = ref('')
    const nameOfNewAudit = ref('')
    const router = useRouter()
    async function addAudit() {
        const audit: AuditShema | any = {}
        audit.file_id = 0
        audit.name = nameOfNewAudit.value
        audit.job = site.value
        audit.location = batiment.value
        audit.poste = poste.value
        audit.created_date = new Date().toLocaleString()
        audit.data = []
        await db.audit.put(audit)
        router.push(`/${audit.name}/questionnaire-0`)
    }
</script>
```

Figure 41 : Modal Nuxt UI (Créer un audit)

Dans cette structure, nous allons détailler les fonctionnalités suivantes :

1. L’affichage de la modal.
2. L’enregistrement de l’audit dans la base de données.

Le bouton « Créer un audit » ouvre automatiquement la modal, car il est le seul bouton situé en dehors du <template #body>. Ci-dessous, on peut voir un exemple plus simple, mais identique dans le principe.

```
<UModal title="Créer un nouvel audit">
<UButton label="Créer un audit" trailing-icon="i-lucide-plus"/>
<template #body>
    <UFormField label="Site concerné" class="mb-3"/>
    <UInput v-model="site" placeholder="Entrez le nom du site..."/>
</template>
```

Figure 42 : Fonctionnement de la structure d'une modale Nuxt UI

Pour la partie enregistrement, je récupère les données des formulaires avec un v-model pour chaque input de Nuxt UI. Par la suite dans une fonction, ces v-model sont passés aux propriétés correspondantes d'un objet audit de type AuditSchema. J'ajoute ce nouvel audit à la table « audit » de la base de données, puis j'envoie l'utilisateur vers la page du questionnaire.

```
const site = ref('')
const batiment = ref('')
const poste = ref('')
const nameOfNewAudit = ref('')
const router = useRouter()
async function addAudit() {
    const audit: AuditShema | any = {}
    audit.file_id = 0
    audit.name = nameOfNewAudit.value
    audit.job = site.value
    audit.location = batiment.value
    audit.poste = poste.value
    audit.created_date = new Date().toLocaleString()
    audit.data = []
    await db.audit.put(audit)
    router.push(`/${audit.name}/questionnaire-0`)
}
```

Figure 43 : Code pour la création d'un l'audit

4.3.4. Page questionnaire

Une fois la page d'accueil réalisée, j'ai conçu celle du questionnaire. Il existe deux manières d'accéder à la page questionnaire : en créant un audit ou en modifiant un audit. Dans les deux cas, la page fonctionne de la même manière, excepté que lorsqu'on se trouve dans le cas de la modification d'un audit, on retrouve les réponses saisies précédemment. Vous pouvez voir la vue utilisateur sur téléphone dans la figure ci-dessous.

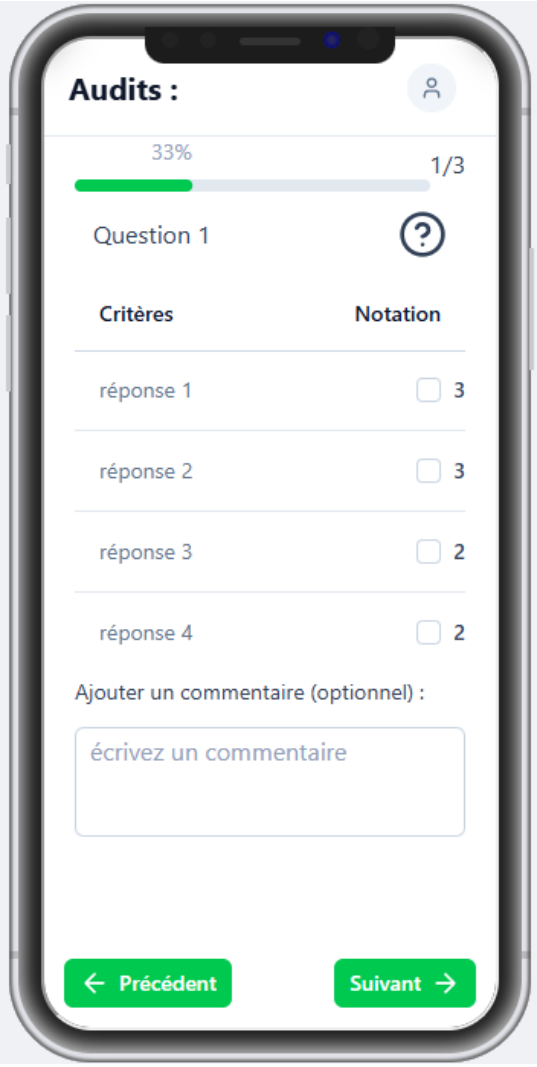


Figure 44 : Page questionnaire

Afin de pouvoir montrer le fonctionnement de la page, j'ai établi un questionnaire de test dans un fichier JSON qui respect la structure de la base de données que l'on peut voir dans la figure ci-dessous.

```json
[
    {
        "title": "Question 1",
        "category": "ergonomie",
        "help": "",
        "multiple": true,
        "criteria": [
            { "title": "réponse 1", "notation": 3 },
            { "title": "réponse 2", "notation": 3 },
            { "title": "réponse 3", "notation": 2 },
            { "title": "réponse 4", "notation": 2 }
        ]
    },
    {
        "title": "Question 2",
        "category": "ergonomie",
        "help": "",
        "multiple": false,
        "criteria": [
            { "title": "réponse 1", "notation": 1 },
            { "title": "réponse 2", "notation": 3 },
            { "title": "réponse 3", "notation": 5 }
        ]
    },
    {
        "title": "Question 3",
        "category": "ergonomie",
        "help": "",
        "multiple": true,
        "criteria": [
            { "title": "réponse 1", "notation": 2 },
            { "title": "réponse 2", "notation": 3 },
            { "title": "réponse 3", "notation": 2 },
            { "title": "réponse 4", "notation": 2 }
        ]
    }
]
```

Figure 45 : Questionnaire test (JSON)

Par la suite, j'ai simplement ajouté que, lors du premier lancement de l'application, ce fichier est ajouté à la base de données s'il n'y est pas déjà.

```
if(jsonQuestionListe.value.length === 0){
    await db.questions.bulkPut(QuestionListeJsonFile)
}
```

Figure 46 : Code d'ajout à la base de données du ficher JSON

4.3.4.1. Barre de pourcentage

Le premier composant que nous allons détailler est la barre de pourcentage qui, ici, sert à permettre à l'utilisateur de savoir où il en est dans le questionnaire. On peut la retrouver ci-dessous.

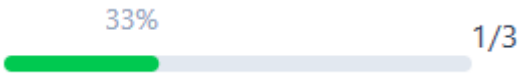


Figure 47 : Barre de pourcentage

La barre de pourcentage est un composant Nuxt UI, UProgress, alimenté par une v-model que je détermine ainsi :
- Je récupère la totalité des questions dans la base de données et les transforme en tableau, que je place dans une ref typée en tableau de QuestionSchema.
- Ensuite, je récupère l'id de la question dans l'URL avec useRoute.
- Puis je calcule la position actuelle de l'utilisateur dans le questionnaire avec un computed.
- Enfin, je calcule le pourcentage à fournir à UProgress.

```vue
<template>
  <div class="flex justify-between items-center">
    <UProgress v-model="value" status class="flex-1" />
    <h1>{{ curentePage+1 }}/{{ totalQuestion }}</h1>
  </div>
</template>

<script lang="ts" setup>
  import { db } from '~/dataBase/db'
  import type { QuestionSchema } from '~/dataBase/db'

  const questionBdd = await db.questions.toArray()
  const jsonQuestionListe = ref<QuestionSchema[]>(questionBdd)
  const route  = useRoute()
  const urlId = ref(route.params.id)

  const curentePage = computed(() => {
    if (typeof(urlId.value) == "string") {
      return parseInt(urlId.value)
    }
    return 0
  })

  const totalQuestion : any = jsonQuestionListe.value.length
  const value = ref((curentePage.value+1)*100/(totalQuestion))
</script>
```

Figure 48 : Code de la barre de pourcentage

4.3.4.2. Liste des critères

Le composant liste des critères est un tableau qui énumère les critères et permet de choisir une ou plusieurs notes. Vous pouvez le voir dans la figure ci-dessous.

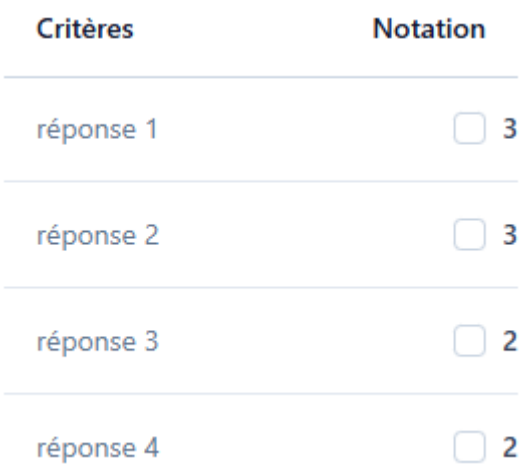


Figure 49 : Liste des critères

La liste des critères est un tableau Nuxt UI. La première colonne est alimentée par le critère courant et affiche les réponses qui lui sont associées. Ce tableau fonctionne avec le même code que la barre de pourcentage.

```vue
<template>
    <UTable :data="jsonQuestionListe[curentePage]?.criteria"/>
</template>

<script lang="ts" setup>
  import { db } from '~/dataBase/db'
  import type { QuestionSchema } from '~/dataBase/db'

  const questionBdd = await db.questions.toArray()
  const jsonQuestionListe = ref<QuestionSchema[]>(questionBdd)
  const route  = useRoute()
  const urlId = ref(route.params.id)

  const curentePage = computed(() => {
    if (typeof(urlId.value) == "string") {
      return parseInt(urlId.value)
    }
    return 0
  })
</script>
```

Figure 50 : Liste des critères

La deuxième colonne liste les notes possibles pour chaque réponse et permet de sélectionner une ou plusieurs notes selon le type de critère. Cela se fait de la manière suivante :
- Dans chaque cellule de ma colonne, il doit y avoir une case à cocher si le choix est multiple, sinon un bouton radio.
- Dans tous les cas, je stocke la réponse sélectionnée dans une variable via une fonction onChange.

```
const columns: TableColumn<any>[] = [
    {accessorKey: 'title', header: 'Critere'},
    {
      header: 'Notation', accessorKey: 'notation',
      meta: {class: {th: 'text-right',td: 'text-right flex flex-row-reverse',}},
      cell: ({ row }) => {
        if (multiple == true){
          return h('div',
              h(UCheckbox , {
                label : String(row.getValue('notation')),
                value: row.getValue('title'),
                description: jsonQuestionListe.value[curentePage.value]?.help,
                modelValue: listCritereSeletioner.includes(row.index) ? true : false,
                onChange: () => {
                  let index = row.index;
                  if (listCritereSeletioner.includes(index)) {
                    let position = listCritereSeletioner.indexOf(index);
                    listCritereSeletioner.splice(position, 1)
                  }
                  else {
                    listCritereSeletioner.push(index);
                  }
                }
              })
          )
        } else {
          return h('label',
            { class: 'flex items-center gap-2 cursor-pointer' },
            [
              h('input', {
                type     : 'radio',
                name     : jsonQuestionListe.value[curentePage.value]?.title,
                value    : row.getValue('notation'),
                checked  : singleChoice === row.index,
                onChange : (e: Event) => {
                  const target = e.target as HTMLInputElement
                  if (target.checked) {
                    singleChoice = row.index
                  }
                },
                class: 'h-4 w-4 text-primary-600 focus:ring-primary-500'
              }),
              h('span', { class: 'text-sm' }, String(row.getValue('notation')))
            ]
          )
        }
      }
    }
  ]
```

Figure 51 : Code de la colonne des notes

4.3.4.3. Composant ajouter un commentaire

Le composant ajouter un commentaire est présent si l'utilisateur souhaite ajouter des détails supplémentaires à sa réponse.

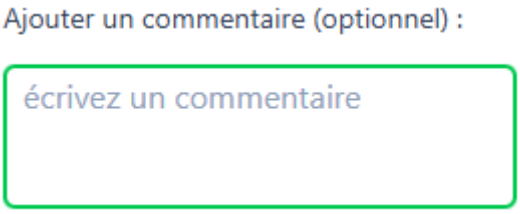


Figure 52 : Composant ajouter un commentaire

Ce composant fonctionne de manière très simple. Il s'agit d'un composant Nuxt UI, UTextarea. L'utilisateur saisit le texte souhaité, que je stocke via un v-model nommé commentaire.

```
1  <template>
2      <label for="Utexte" class="text-sm">Ajouter un commentaire (optionnel) :</label>
3      <div class="flex justify-center mt-3">
4        <UTextarea color="primary" placeholder="écriver un commentaire"v-model="commentaire"/>
5      </div>
6  </template>
7
8  <script lang="ts" setup>
9    const commentaire = ref("")
10 </script>
```

Figure 53 : Code pour ajouter un commentaire

4.3.4.4. Boutons suivant et précédent

Enfin, le dernier composant que nous allons voir est constitué des boutons « Suivant » et « Précédent » : ils permettent de naviguer entre les différentes questions et d'enregistrer les réponses.

Figure 54 : Boutons suivant et précédent

Ces deux boutons sont des composants Nuxt UI (UButton) qui appellent chacun une fonction différente.

```
<template>
    <UButton label="Suivant" trailing-icon="i-lucide-arrow-right"@click="nextPage()"/>
    <UButton label="Précédent" icon="i-lucide-arrow-left"@click="previousPage()"/>
</template>
```

Figure 55 : Code des boutons suivant et précédent

Tout d'abord, le bouton « Suivant » appelle la fonction nextPage que vous pouvez voir ci-dessous. Elle fonctionne de la manière suivante :
- Appelle une fonction pour sauvegarder les réponses de la page courante en base de données avant de naviguer.
- Récupère les choix (choices) sélectionnés sur la page courante de l'audit.
- Filtre les valeurs vides ou nulles du tableau choices pour ne conserver que les choix réellement effectués par l'utilisateur.
- Vérifie qu'au moins un choix a été sélectionné pour permettre l'avancement ; si aucun choix n'est fait, la navigation est bloquée.

```
function nextPage() {
  registerCritere()
  const auditCheck = audit.data[curentePage.value]
  const choices = auditCheck?.choices
  const check = choices.filter((e : any ) => e !== undefined && e!= null)
  if(check.length > 0){
    if (curentePage.value < jsonQuestionListe.value.length -1 ){
      router.push(`./questionnaire-${curentePage.value + 1}`)
    } else {
      router.push(`./resumai`)
    }
  }
}
```

Figure 56 : Fonction nextPage

La fonction registerCritere qui est appelée par la fonction nextPage fonctionne de la manière suivante :
- Initialise la note à 0 et l'index du critère sélectionné.
- Si le type de question est multiple, alors je parcours tous les critères sélectionnés et additionne leurs notations une par une.
- Sinon, je récupère directement la notation du critère unique sélectionné.
- Met à jour l'objet audit et l'ajoute à la base de données.

```
async function registerCritere() {
  let noteTotal = 0
  let index : any = []
  if (multiple) {
    for(let i = 0; i < nbQuestion; i++) {
      index = listCritereSeletioner[i]
      noteTotal += jsonQuestionListe.value[curentePage.value]?.criteria[index]?.notation ?? 0
    }
  } else {
    index = singleChoice
    noteTotal += jsonQuestionListe.value[curentePage.value]?.criteria[index]?.notation ?? 0
  }
  audit.data[curentePage.value] = {
    choices : multiple ? listCritereSeletioner : [singleChoice],
    note : noteTotal,
    comment : commentaire.value,
    documents : [""]
  }
  await db.audit.put(audit)
  noteTotal = 0
}
```

Figure 57 : Fonction registerCritere

4.3.5. Méthode d'authentification OAuth2

Pour cette application, j'ai dû utiliser la méthode d'authentification OAuth2, qui permet à une application d'accéder via des ressources au nom d'un utilisateur, sans jamais connaître son mot de passe. Le principe est le suivant :

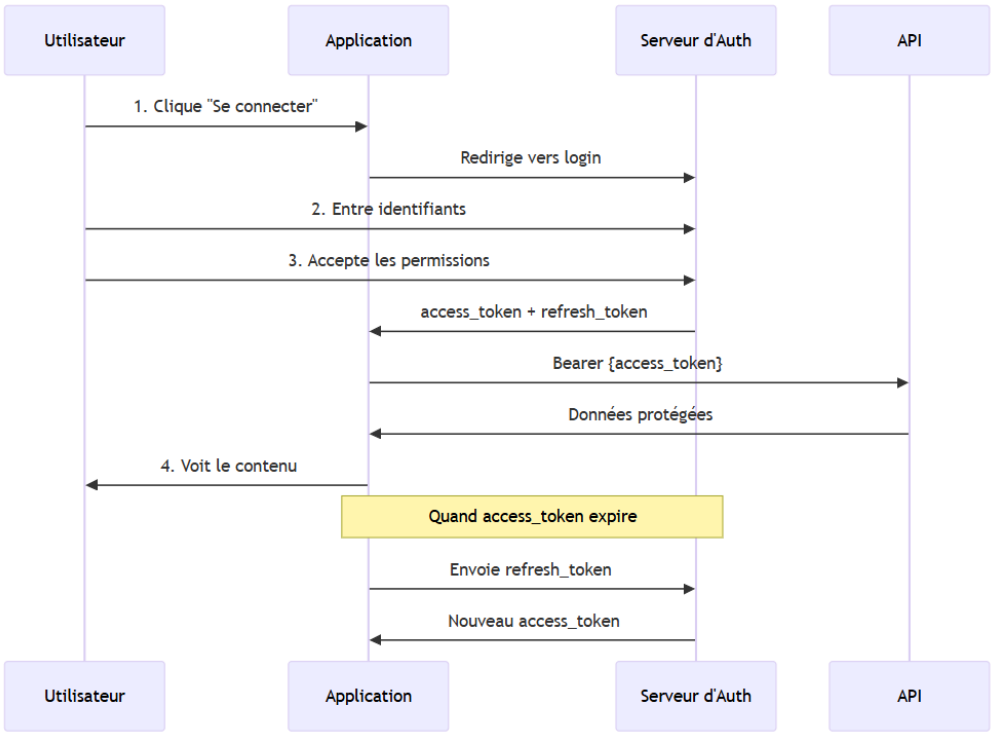


Figure 58 : Fonctionnement de l'OAuth2

Dans un premier temps j'ai dû créer une application de test sur dans mon espace de développement box.

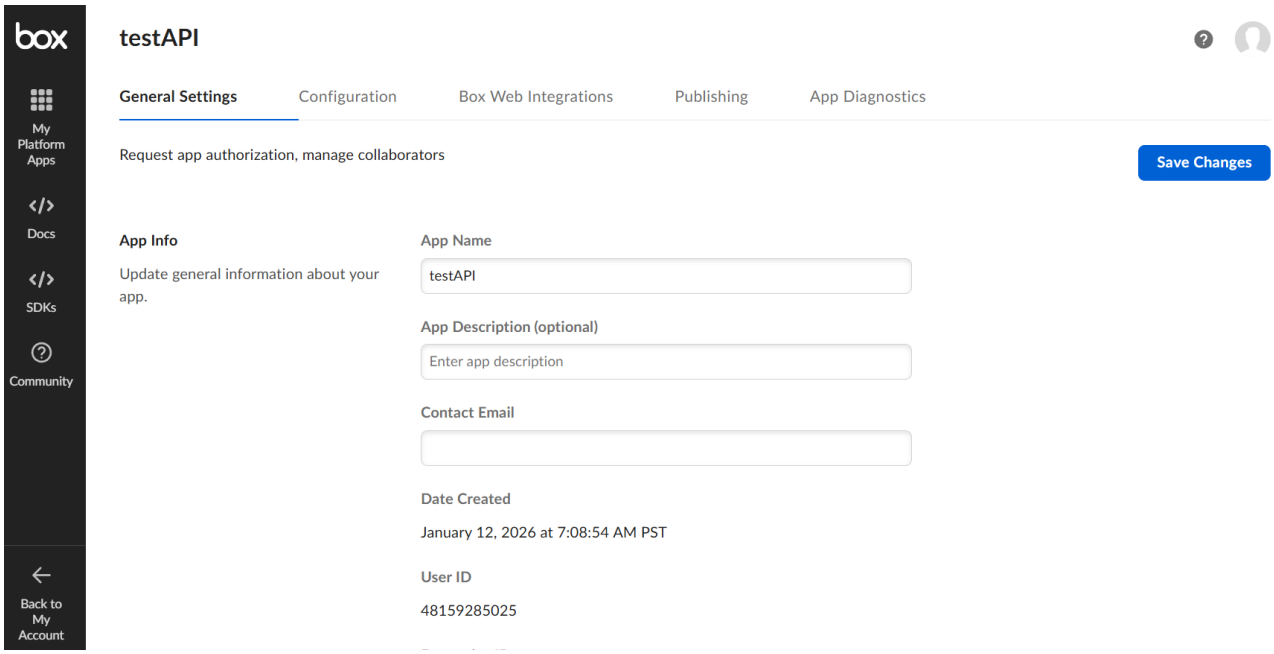


Figure 59 : Espace développeur Box

Par la suite, j'ai récupéré un token client ID et un client secret, que j'ai ajouté dans un fichier d'environnement de mon application, comme dans l'exemple ci-dessous.

```
.env
1   BOX_CLIENT_ID=abc123xyz456def789
2   BOX_CLIENT_SECRET=abc123xyz456def789
```

Figure 60 : Exemple de contenu du fichier environnement

De plus, j'ai ajouté deux URL de callback pour rediriger l'utilisateur une fois qu'il est connecté à Box. Comme dans l'exemple ci-dessous.

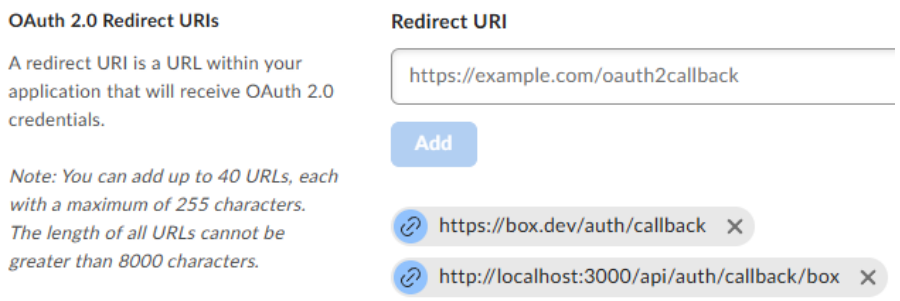


Figure 61 : URL de callback Box

Une fois les tokens dans un fichier d'environnement, il ne me reste plus qu'à les ajouter au provider Box de NuxtAuth, ce qui se fait très simplement car le provider Box existe déjà dans la librairie de NuxtAuth, comme dans l'extrait ci-dessous.

```ts
async function refreshBoxAccessToken(refreshToken: string): Promise<{
    access_token  : string
    refresh_token : string
    expires_at    : number
}> {
    console.log('[Box Auth] Rafraîchissement du token...')

    const response = await fetch('https://api.box.com/oauth2/token', {
        method  : 'POST',
        headers : {
            'Content-Type' : 'application/x-www-form-urlencoded'
        },
        body : new URLSearchParams({
            grant_type    : 'refresh_token',
            refresh_token : refreshToken,
            client_id     : process.env.BOX_CLIENT_ID     ?? '',
            client_secret : process.env.BOX_CLIENT_SECRET ?? ''
        })
    })

    if (!response.ok) {
        const error = await response.text()
        console.error('[Box Auth] Erreur refresh token:', error)
        throw new Error(`Refresh token failed: ${response.status}`)
    }

    const tokens = await response.json()

    console.log('[Box Auth] Token rafraîchi avec succès')

    return {
        access_token  : tokens.access_token,
        refresh_token : tokens.refresh_token,
        expires_at    : Math.floor(Date.now() / 1000) + tokens.expires_in
    }
}
```

Figure 62 : Extrait du provider box de NuxtAuth

Pour se connecter, il suffit de cliquer sur l'avatar afin de se connecter pour être redirigé vers la page OAuth2 de box.

Figure 63 : Processus de connexion étape 1

Ensuite, il faut rentrer ses identifiants Box pour se connecter et cliquer sur Grant Access to Box.

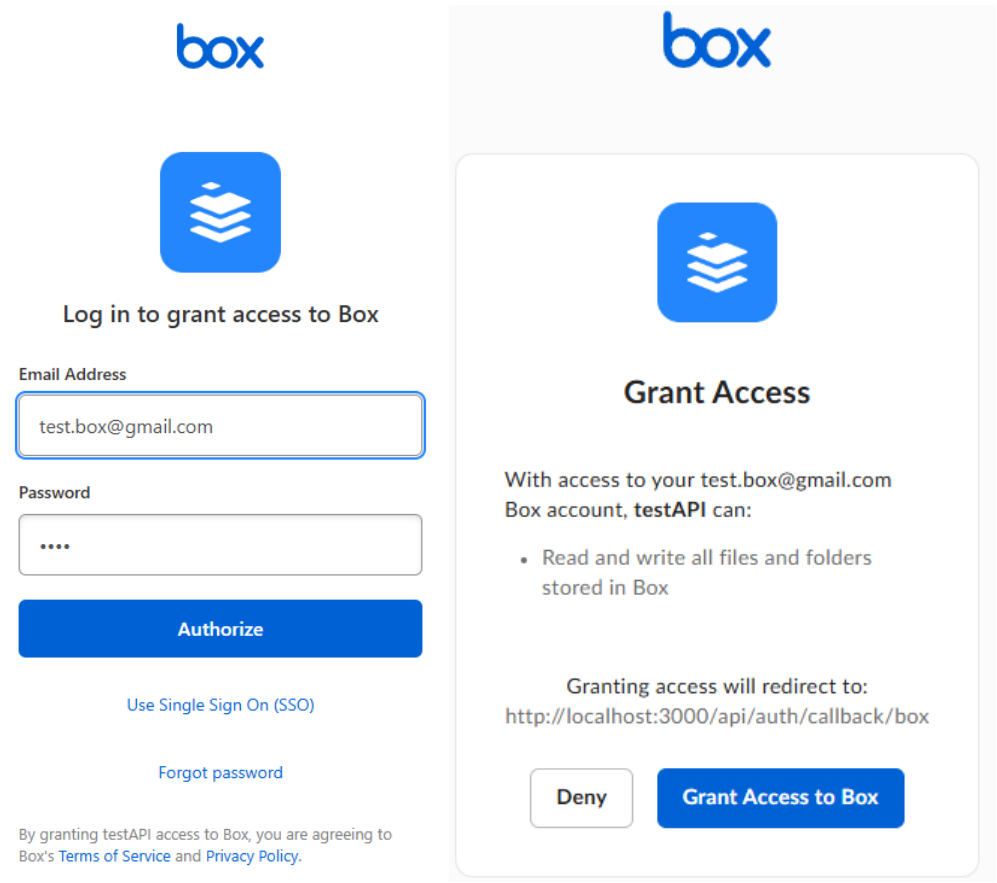


Figure 64 : Processus de connexion étape 2

Une fois que l'utilisateur a appuyé sur le bouton Grant Access to Box, il est redirigé sur la page d'accueil. On voit comme changements qu'il y a les initiales du nom utilisateur dans l'avatar et qu'il y a la possibilité de synchroniser ses audits avec Box et de se déconnecter, comme dans l'exemple ci-dessous.

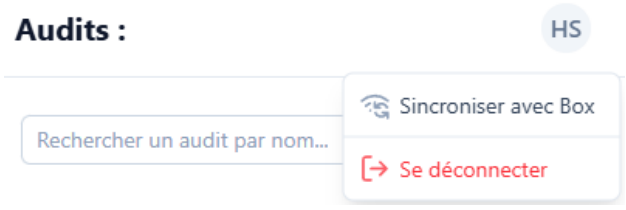


Figure 65 : Fonctionnalités disponibles une fois connecté

L'avatar est un UAvatar dans un UDropdownMenu, tous deux provenant de Nuxt UI. Ils permettent trois choses :
- Si l'utilisateur est connecté, j'affiche le bouton Synchroniser avec Box qui appelle des API Box, dont nous allons parler juste après, ainsi qu'un bouton Se déconnecter qui appelle la fonction signOut du provider.
- Sinon, j'affiche un bouton Se connecter qui appelle la fonction signIn du provider.

```vue
<template>
    <UDropdownMenu :items="items">
        <UButton variant="link">
            <UAvatar size="md" :items="items" :icon="iconTest" :alt="nameUp"/>
        </UButton>
    </UDropdownMenu>
</template>
<script lang="ts" setup>
    import type { DropdownMenuItem } from '@nuxt/ui'
    let items: DropdownMenuItem[][] = []
    const {signOut, signIn, data, status} = useAuth()
    const name = data.value?.user?.name
    const nameUp = name?.toUpperCase()
    let iconTest = ""

    if (nameUp == undefined) {
        iconTest = "i-lucide-user"
    }

    if (status.value == "authenticated") {
        items = [
            [{
                label: 'Sincroniser avec Box',
                icon: 'i-lucide-wifi-sync',
                onSelect: async () => {
                }
            }],
            [{
                color: 'error',
                label: 'Se déconnecter',
                icon: 'i-lucide-log-out',
                onSelect() {
                    signOut()
                }
            }]
        ]
    } else {
        items = [
            [{
                label: 'Se connecter',
                icon: 'i-lucide-log-in',
                onSelect() {
                    signIn('box')
                }
            }]
        ]
    }
</script>
```

Figure 66 : Code du UDropDownMenu

4.3.6. API Box

Une fois la connexion à Box établie, j'ai pu mettre en place les API Box dans un fichier dédié. Ici nous allons nous intéresser aux quatre API principales, que j'ai créées avec des fonctions du nom de getAuditApi, listAuditsApi, addAuditApi et updateAuditApi.

Dans un premier temps, pour récupérer les informations d'un audit individuel dans le drive Box, j'utilise la fonction getAuditApi qui prend en paramètre le file_id. Elle fonctionne de la manière suivante :
- Appel à l'API Box pour récupérer le contenu brut d'un fichier spécifique via son file_id.
- Si l'API répond avec une erreur HTTP (404 par exemple), je retourne false.
- Parse la réponse JSON (le contenu du fichier audit) et le retourne.
- Retourne audit

```
1  async function getAuditApi (file_id: number) {
2    const response = await fetch(`https://api.box.com/2.0/files/${file_id}/content`, {
3      method : 'GET',
4      mode: 'cors',
5      headers : {
6        'Authorization' : `Bearer ${(data.value as any)?.accessToken}`
7      }
8    })
9
10   if (!response.ok) {
11     const error = await response.text()
12     console.log(error)
13     return false;
14   }
15
16   const audit : any = await response.json()
17   return audit
18 }
```

Figure 67 : Fonction getAuditApi

Pour récupérer la liste des audits dans le drive Box, j'utilise la fonction listAuditsApi. Elle fonctionne de la manière suivante :
- Appel à l'API Box pour lister le contenu d'un dossier spécifique (id: 362889302146).
- Si l'API répond avec une erreur HTTP (404 par exemple), une exception est levée avec le détail de l'erreur.
- Parse la réponse JSON.
- Pour chaque élément du dossier, j'appelle getAuditApi pour récupérer le contenu complet du fichier audit. Je stocke le contenu dans audits et l'id de l'audit dans auditsId.
- Retourne audits et auditsId.

```
async function listAuditsApi() {
  const response = await fetch("https://api.box.com/2.0/folders/362889302146/items", {
    method : 'GET',
    mode: 'cors',
    headers : {
      'Authorization' : `Bearer ${(data.value as any)?.accessToken}`
    }
  })

  if (!response.ok) {
    const error = await response.text()
    throw new Error(`Box API error: ${response.status} - ${error}`)
  }

  const response_audits : any = await response.json()
  let audits = []
  let auditsId = []
  for (const audit of response_audits.entries) {
    audits.push(await getAuditApi(audit.id))
    auditsId.push(audit.id)
  }
  return {audits, auditsId}
}
```

Figure 68 : Fonction listAuditsApi

Pour ajouter un audit dans le drive Box, j'utilise la fonction addAuditApi qui prend en paramètre audiName et jsonAuditFile. Elle fonctionne de la manière suivante :
- Convertit l'objet audit en fichier JSON formaté.
- Construit un formulaire multipart/form-data requis par l'API Box Upload avec comme métadonnées le nom du fichier et dossier parent (362889302146) et le contenu JSON du fichier audit.
- Envoie le fichier à l'API d'upload Box.
- Parse la réponse et retourne le premier élément de entries qui contient les métadonnées du fichier créé.

```
async function addAuditApi(auditName : string, jsonAuditFile : string){
  let my_file = new Blob([ JSON.stringify(jsonAuditFile, null, 4) ], { type: "application/json" });

  const formData = new FormData();
  formData.append('attributes', JSON.stringify({
    name: `${auditName}.json`,
    parent: { id: "362889302146" }
  }));
  formData.append('file', my_file, 'fichier.json');

  const response = await fetch(`https://upload.box.com/api/2.0/files/content`, {
    method : 'POST',
    mode: 'cors',
    headers : {
      'Authorization' : `Bearer ${(data.value as any)?.accessToken}`,
    },
    body: formData
  })

  let return_response = JSON.parse(await response.text())
  return return_response.entries[0]
}
```

Figure 69 : Fonction addAuditApi

Pour mettre à jour un audit dans le drive Box, j'utilise la fonction updateAuditApi qui prend en paramètre audiName et auditFileId. Elle fonctionne de la manière suivante :
- Convertit l'objet audit en fichier JSON formaté.
- Construit un formulaire multipart/form-data requis par l'API Box Upload avec comme métadonnées le nom du fichier et dossier parent (362889302146) et le contenu JSON du fichier audit.
- Envoie le fichier à l'API d'upload Box en ciblant le fichier via son auditfileId.

```
async function updateAuditApi(auditName : string, auditfileId : number, jsonAuditFile : string)
{
  let my_file = new Blob([ JSON.stringify(jsonAuditFile, null, 4) ], { type: "application/json" });

  const formData = new FormData();
  formData.append('attributes', JSON.stringify({
    name: `${auditName}.json`,
    parent: { id: "362889302146" }
  }));
  formData.append('file', my_file, 'fichier.json');

  const response = await fetch(`https://upload.box.com/api/2.0/files/${auditfileId}/content`, {
    method : 'POST',
    mode: 'cors',
    headers : {
      'Authorization' : `Bearer ${(data.value as any)?.accessToken}`,
    },
    body: formData
  })
}
```

Figure 70 : Fonction updateAuditApi

J'utilise enfin ces fonctions dans le bouton envoyer audit qui avait été laissé de côté dans la présentation de la page d'accueil.

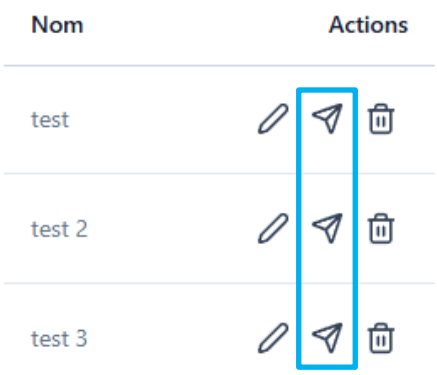


Figure 71 : Bouton envoyer audit

Le bouton fonctionne de la manière suivante :
- Vérifie si l'utilisateur est connecté via OAuth Box.
- Si non déclenche signIn('box') pour lancer l'authentification.
- Récupère depuis la ligne du tableau file_idApi, le nom de l'audit et les données complètes de l'audit depuis la base de données.
- Si le file_id est égale à 0 alors je créer un nouveau fichier avec addAuditApi
- Si non je le mets à jour avec updateAuditApi.

```
onClick: async () => {
    if(status.value == "authenticated") {
        const file_idApi = row.original.file_id
        const name = row.original.name
        const auditLocal : AuditShema | any = await db.audit.get({name})

        if (file_idApi == 0) {
            console.log('click -> add', name)
            let file = await addAuditApi(name, auditLocal)
            auditLocal.file_id = file.id;
            await db.audit.put(auditLocal)
            await updateAuditApi(name, auditLocal.file_id, auditLocal)
        } else {
            console.log('click -> update', file_idApi)
            await updateAuditApi(name, file_idApi, auditLocal)
        }
    } else {
        await signIn('box')
    }
}
```

Figure 72 : Code du bouton envoyer audit

Nous avons fini de voir la page d'accueil, le questionnaire, la méthode d'authentification OAuth2 et les Api Box. Pour rappel, ce n'est qu'une partie de l'application que je vous présente, car le fonctionnement technique des autres pages et fonctionnalités est similaire, il y a uniquement que le contexte d'usage qui change.

5. Retour d'expérience

5.1. Remerciements

Je voudrais ici remercier monsieur Valentin Jauze et monsieur Robert Hoffmann, mes maitres de stage qui m'ont d'abord accepté au sein de leur équipe de la société Maser Engineering pour effectuer mon stage. Leur accueil chaleureux et leur bienveillance m'ont permis de m'épanouir.

Je tiens à remercier monsieur Valentin Jauze de m'avoir donné ma chance et de m'avoir accompagné tout au long de cette expérience professionnelle avec beaucoup de patience et de pédagogie.

Je remercie également monsieur Robert Hoffmann pour ses conseils au quotidien, le partage de son expérience professionnelle et la transmission de ses méthodes de travail.

Enfin, je tiens à remercier toutes les personnes qui m'ont conseillé et relu lors de la rédaction de ce rapport de stage : mes maitres de stage, ma famille et mes amis.

5.2. Points positifs

Parmi les points positifs de ce stage le premier qui me vient à l'esprit est l'accueil et l'environnement de travail qui est vraiment excellent au sein de Maser Engineering. Que ce soit au niveau de l'open-space ou j'ai effectué mon stage, ou au niveau des mes maitres de stage, toutes les personnes ont vraiment été patiente et bienveillante à mon égard. J'en retient un excellent souvenir.
Ensuite le sujet sur lequel j'ai travaillé a été très intéressant et représentatif de la réelle charge de travail pour réaliser une application web. De plus j'ai appliqué certaine partie des cours de base de données et de développement.

5.3. Pistes de progrès

Il aurait été intéressant d'avoir plus de temps afin de mettre en place la mise en production de l'application.
Une autre partie qui aurait été aussi intéressante d'aborder est la possibilité de prendre des photos avec le téléphone de l'utilisateur et de faire des retouches sur l'image dans le cadre d'une visite d'ergonome.