Informatique

iut Nord Franche-Comté

noehmi

TECH4GAIA

DÉVELOPPEMENT D'APPLICATION NOEHMI V2 FONCTIONNALITÉ GÉNÉALOGIE DES REINES

RAPPORT TECHNIQUE

DU 3 AVRIL AU 26 MAI 2023

Raphaël DANY

Jean CHARBONNEAU

Karine DESCHINKEL

TECH4GAIA

2ème année de BUT informatique

DEVELOPPEUR IA

Professeur référent

Sommaire

Introduction

Ce rapport technique documente la réalisation d'une partie spécifique pour l’application NoehmiV2 dédiée à la gestion des reines pour un apiculteur. L'objectif de cette fonctionnalité est de permettre à l'apiculteur d'ajouter, supprimer et modifier des reines, de les organiser dans des ruches et d'accéder à des informations détaillées sur chaque reine.

L'application globale vise à faciliter la gestion des colonies d'abeilles et à optimiser les activités apicoles. La gestion des reines est d’une importance cruciale, car elles influencent la santé et la productivité des colonies. Cette partie du projet se concentre spécifiquement sur la gestion des reines pour offrir une solution pratique et intuitive à l'apiculteur.

Il est important de noter que ce rapport se concentre uniquement sur la partie du projet liée à la gestion des reines. Nous mettrons en évidence les choix techniques effectués et les résultats obtenus, afin de fournir une compréhension claire de cette fonctionnalité spécifique.

Sans plus attendre, explorons les détails du développement de la gestion des reines sur l’application web, en mettant l'accent sur sa mise en œuvre technique et ses avantages pour l'apiculteur.

1. Prérequis

1.1 Structure du projet

La structure du projet en Svelte peut être résumée comme suit :

- Le serveur gère la récupération et l'enregistrement des données dans la base de données.

- Les routes en Svelte, situées dans le front-end, permettent la navigation entre les différentes pages de l'application.

- Les données sont récupérées depuis le serveur et stockées dans des stores, puis utilisées par les composants des routes et les routes pour afficher les données à l'utilisateur.

- La base de données est accessible via un service web pour interagir avec les données stockées.

1.2 Présentation des langages de programmation et des frameworks employés

Pour la réalisation de cette application, plusieurs logiciels, langages de programmation et frameworks ont été utilisés. Voici les principaux éléments techniques requis pour comprendre le rapport :

- Langage de programmation : TypeScript v4.9.3

TypeScript est un langage de programmation basé sur JavaScript qui offre un typage statique et des fonctionnalités supplémentaires pour le développement d'applications web. Il améliore la fiabilité, la maintenabilité et la lisibilité du code.

- Framework : Svelte v3.55.0

Svelte est un framework basé sur Javascript qui permet de créer des applications web interactives. Il se distingue par sa simplicité et sa performance, en générant un code optimisé qui s'exécute efficacement côté client.

- Framework supplémentaire : Svelte Kit v1.0.1

Svelte Kit est une extension de Svelte qui facilite le développement d'applications web. Il fournit des fonctionnalités telles que la gestion des routes, l'intégration avec des services externes et la création de composants réutilisables. Svelte Kit simplifie le processus de construction, de déploiement et de gestion de l'application.

- Base de données : Supabase v2.8.0

Supabase est une plateforme de base de données open source et sans serveur. Elle offre une interface conviviale pour la gestion des données et fournit des fonctionnalités de stockage et de requêtage efficaces pour les besoins de l'application. Supabase utilise PostgreSQL comme moteur de base de données et offre une intégration transparente avec Svelte et Svelte Kit, il faut l’ajouter grâce à npm install @supabase/supabase-js.

- Ruches connectées : Thingsboard

Thingsboard est une plateforme de gestion des objets connectés qui permet de collecter, stocker et analyser les données provenant des ruches connectées. Elle offre des fonctionnalités avancées telles que la surveillance en temps réel, les alertes et les visualisations personnalisées. Thingsboard facilite l'intégration des données des ruches connectées dans notre application.

La compréhension de ces outils et technologies, tels que TypeScript, Svelte, Svelte Kit, Supabase et Thingsboard, est essentielle pour appréhender les choix techniques et les solutions mises en place dans le développement de l'application. Ils contribuent à assurer la robustesse, la performance et la fonctionnalité de la gestion des reines sur le site web.

2. Explications générales

2.1 Côté back-end (serveur)

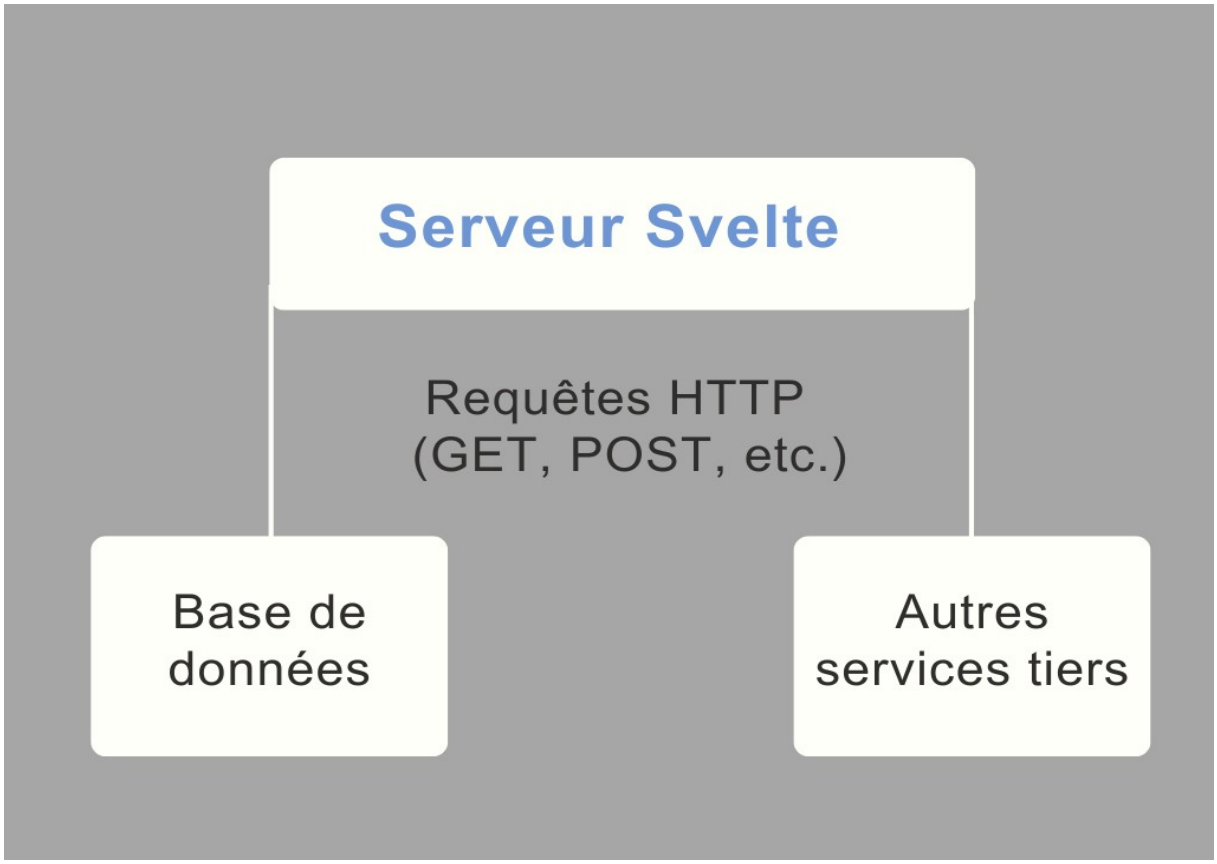

Figure 1: Fonctionnement général back

- Le serveur fournit les fichiers HTML, CSS et TypeScript nécessaires à l'application Svelte. Ces fichiers sont généralement stockés sur le serveur et renvoyés aux navigateurs des utilisateurs lorsqu'ils accèdent à l'application.

- Le serveur peut utiliser n'importe quel langage ou framework pour fournir ces fichiers. Svelte lui-même n'a pas de dépendance spécifique au back-end et fonctionne bien avec diverses technologies.

2.2 Côté front-end (navigateur)

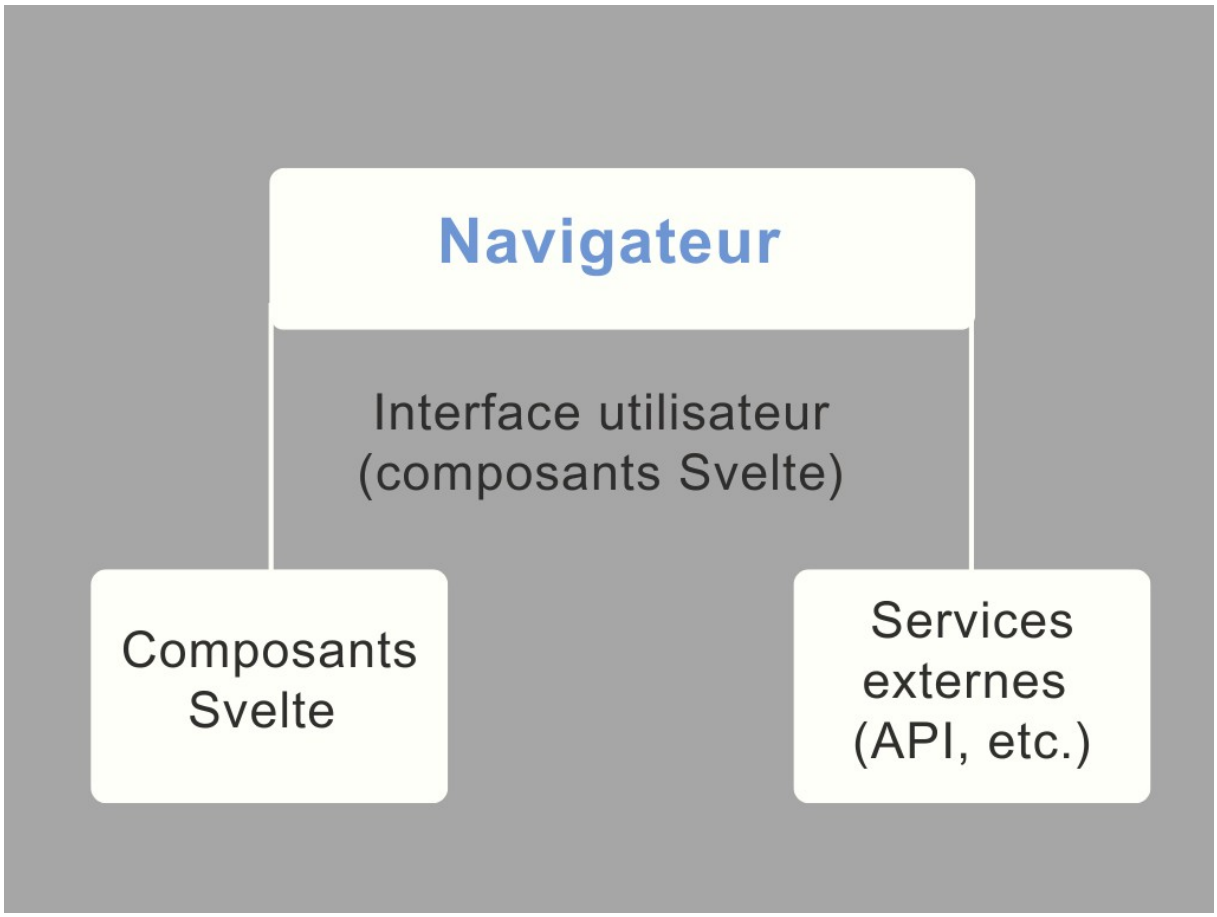

Figure 2: Fonctionnement général du front

- Lorsqu'un utilisateur accède à l'application Svelte via son navigateur, le navigateur télécharge les fichiers HTML, CSS et TypeScript fournis par le serveur.
- Le fichier TypeScript principal de l'application Svelte est généralement appelé "app.ts" ou un nom similaire. Ce fichier contient le code source de l'application Svelte écrit en TypeScript.
- Le navigateur ne comprend pas directement le TypeScript, il a besoin de le traduire en JavaScript pour pouvoir l'exécuter. C'est là que le processus de compilation entre en jeu.

- Pour compiler le code TypeScript en JavaScript, tu utiliseras un outil comme TypeScript Compiler (tsc) ou un outil de build (comme Webpack ou Rollup) avec un plugin TypeScript approprié.

- Le compilateur TypeScript traduit le code TypeScript en code JavaScript standard, générant un ou plusieurs fichiers JavaScript qui seront utilisés par le navigateur.

- Une fois le code JavaScript généré, le navigateur l'exécute, créant et mettant à jour les éléments DOM nécessaires pour afficher l'interface utilisateur de l'application.

- Svelte fonctionne alors de la même manière que précédemment, en gérant les mises à jour efficacement et en mettant à jour uniquement les parties de l'interface utilisateur qui ont changé.

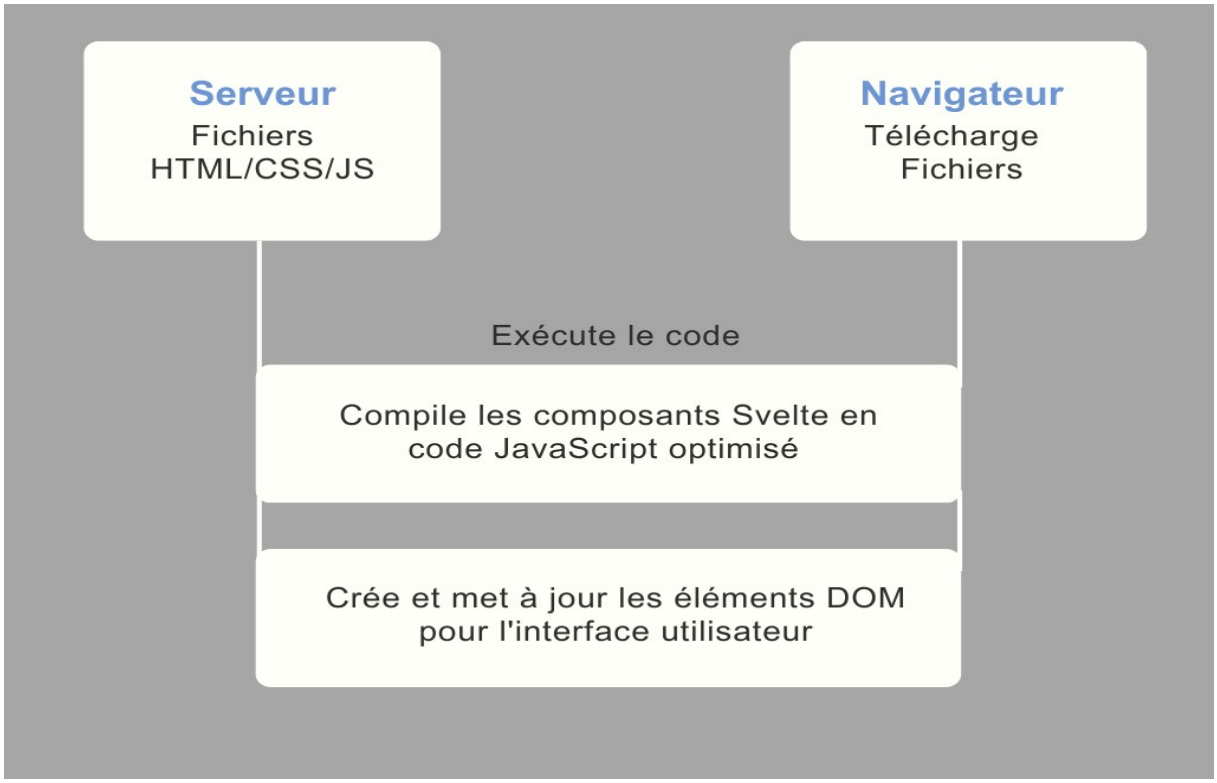

Figure 3: Fonctionnement globale de l'application en Svelte avec la compilation

3. Présentation technique du projet

3.1 Arborescence

3.1.1 Back

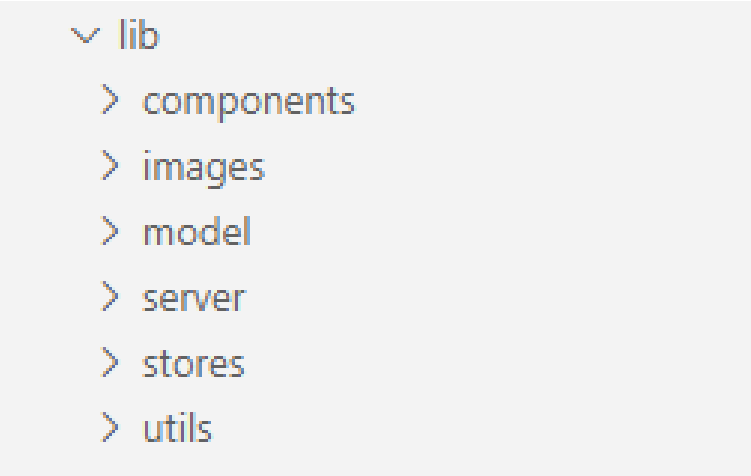

Figure 4: Arborescence Back

- **Composants :** Les composants en Svelte sont des éléments réutilisables qui permettent de créer une interface utilisateur interactive. Ils encapsulent la logique et l'apparence d'une partie spécifique de l'application. Les composants peuvent inclure des éléments de présentation, de logique et de manipulation des données.

- **Server :** Dans un back-end en Svelte, le serveur joue un rôle central en connectant l'application avec la base de données et en gérant les services. Voici une explication de ces éléments :

    - **Les services** sont des modules ou des classes qui fournissent des fonctionnalités spécifiques à l'application. Ils peuvent être utilisés pour effectuer des requêtes HTTP, gérer l'authentification, gérer les interactions avec des API externes, etc. Les services permettent de regrouper la logique liée à une fonctionnalité

spécifique et de la réutiliser facilement dans différentes parties de l'application.

    - **Base de données (BDD)** : La base de données est un système de stockage persistant où les données de l'application sont stockées. Dans le backend en Svelte, on utilise la base de données, Supabase qui utilise PostgreSQL. La base de données permet de stocker, organiser et récupérer les données nécessaires pour l'application.

- **Modèle en TypeScript :** Le modèle en TypeScript fait référence à la couche de logique métier de l'application. TypeScript est un langage de programmation qui ajoute des fonctionnalités de typage statique à JavaScript. En utilisant TypeScript, vous pouvez définir des types stricts pour les données et les fonctions, ce qui améliore la maintenabilité et la fiabilité du code. Le modèle en TypeScript peut inclure des classes, des interfaces et des fonctions qui traitent les données et exécutent des opérations spécifiques.

- **Images :** Les images sont des ressources visuelles utilisées dans l'application. Elles peuvent être des icônes, des illustrations, des photos, etc. Dans un backend en Svelte, on peut charger et afficher des images à partir de différentes sources, telles que des fichiers locaux ou des URL externes. Les images contribuent à l'aspect visuel de l'application et peuvent également être utilisées pour transmettre des informations aux utilisateurs.

- **Stores :** Les stores sont des conteneurs de données réactifs qui permettent de gérer l'état global de l'application. Ils sont utilisés pour stocker et partager des données entre les différents composants de l'application. Les stores en Svelte, tels que le store de l'application ou les stores de domaine, permettent de gérer l'état

de manière centralisée et de le rendre accessible à tous les composants qui en ont besoin.

- **Utils** : Les utils (utilitaires) sont des fonctions ou des classes qui fournissent des fonctionnalités supplémentaires utilisées à travers l'application. Ils peuvent inclure des fonctions utilitaires génériques pour la manipulation de chaînes, le formatage de dates, la validation de données, etc. Les utils ici sont créés pour réutiliser du code commun et simplifier les tâches répétitives.

3.1.2 Front

Figure 5: Arborescence Front

Dans l'architecture Svelte, les dossiers sont généralement nommés d'après les routes de l'application. Chaque dossier représente une page spécifique de l'application. À l'intérieur de ces dossiers, on peut trouver des fichiers tels que +page.server.ts et +page.svelte.

- **Les dossiers de routes :** Les dossiers de routes sont nommés en fonction des chemins de navigation de l'application. Par exemple, si l'application a une route "/queen" et une route "/queen/genealogie", vous pouvez créer deux dossiers nommés "queen" et "genealogie" a l'intérieur du dossier "queen".

- **+page.server.ts** : Ce fichier est généralement utilisé pour gérer la logique côté serveur spécifique à une page. Il peut inclure des opérations telles que la récupération de données depuis le serveur, le traitement des requêtes avec des actions et l'interaction avec la base de données comme ajouter, éditer ou supprimer des reines de l’application.

- **+page.svelte** : Ce fichier contient le composant Svelte qui définit la structure, l'apparence et le comportement de la page spécifique. Il est responsable de l'affichage des données récupérées depuis le serveur et de la gestion des interactions utilisateur.

3.2 Fonctionnement

3.2.1 Model

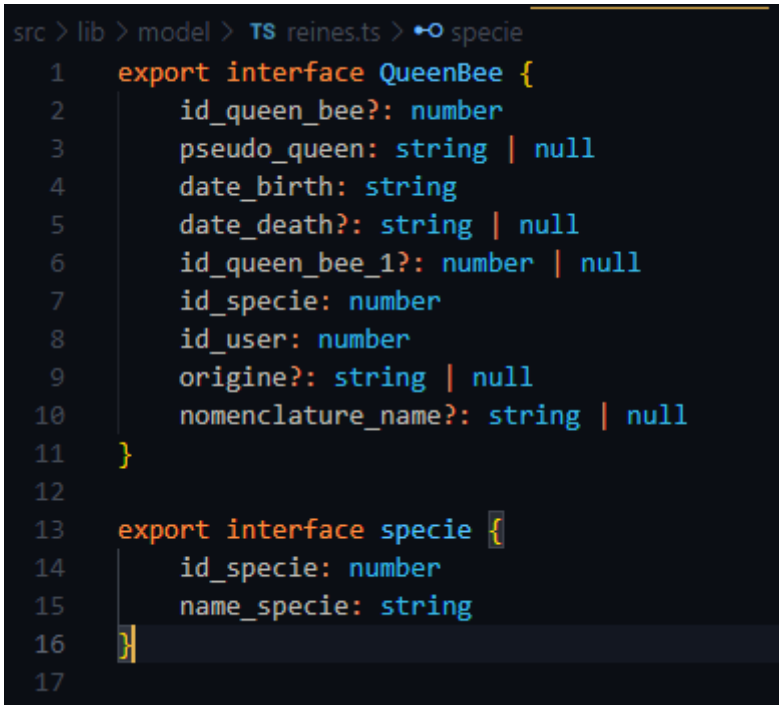

Figure 6: Interfaces utilisées pour définir la structure d'un objet dans le model

En TypeScript, les interfaces sont utilisées pour définir la structure d'un objet. Elles permettent de spécifier les propriétés et les types de valeurs attendues dans un objet.

Les interfaces sont principalement utilisées pour deux raisons principales :

- **Définition de la structure des objets** : Les interfaces permettent de décrire la forme d'un objet en spécifiant les propriétés qu'il doit avoir, ainsi que les types de valeurs attendues pour ces propriétés. Cela permet de s'assurer que les objets respectent une structure spécifique et évite les erreurs de typage lors de la manipulation d'objets.

- **Contrôle du typage** : Les interfaces permettent de définir les types de paramètres et de valeurs de retour des fonctions. Elles garantissent que les fonctions sont appelées avec les bons types de paramètres et renvoient les bons types de valeurs. Cela aide à détecter les erreurs de typage dès la phase de développement et facilite la maintenance du code.

3.2.2 BDD (MCD)

le fichier "db.ts" définit une classe "noehmiDB" avec une variable statique qui représente une instance de client Supabase. Cette instance permettra à l'application d'interagir avec la base de données Supabase.

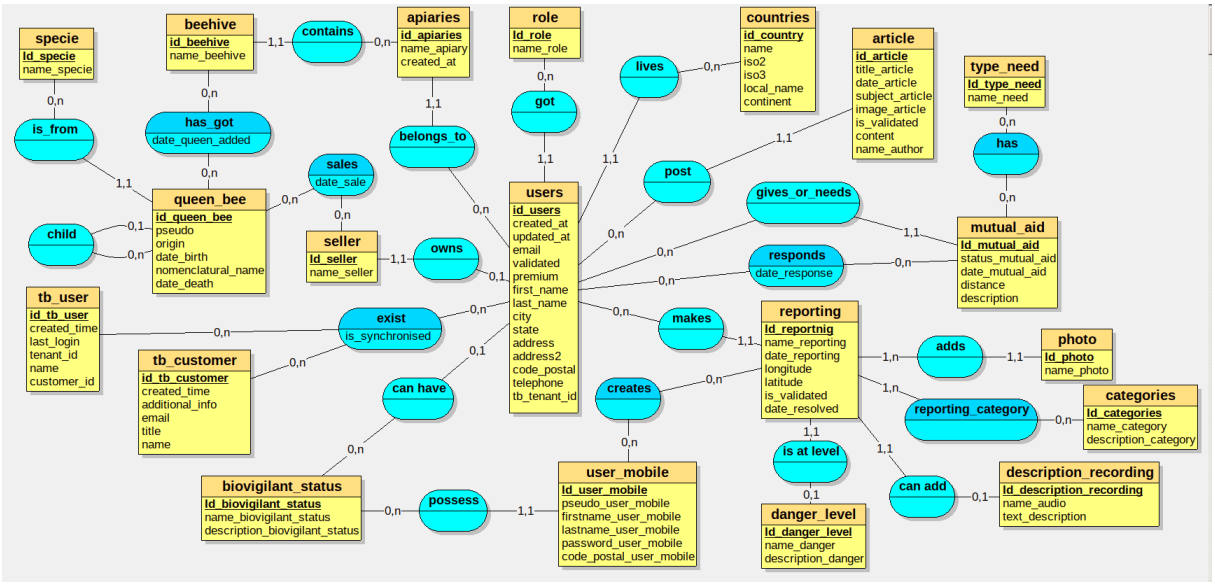

Figure 7: MCD de l'application

Le Modèle Conceptuel de Données (MCD) de l'application, qui a été conçu par moi-même ainsi que deux autres stagiaires, Mélanie et Antoine, comprend plusieurs tables. Cependant, pour la généalogie des reines, les quatres tables principales qui nous intéressent actuellement sont les suivantes :

- **Table "beehive" :** Cette table représente les ruches dans l'application. Elle peut contenir des informations telles que l'identifiant de la ruche et son nom.

- **Table "queen_bee"** : Cette table représente les reines d'abeilles. Elle peut contenir des informations spécifiques sur chaque reine, telles que son identifiant, son pseudo, sa date de naissance, sa date

de décès éventuelle, son origine, nom nomenclaturer. Cette table est probablement utilisée pour enregistrer les détails de chaque reine dans l'application.

- **Table "specie" :** Cette table représente les espèces d'abeilles. Elle peut contenir des informations sur différentes espèces d'abeilles, telles que leur identifiant, leur nom scientifique Cette table est utilisée pour enregistrer les différentes espèces d'abeilles auxquelles les reines appartiennent.

- **Table "has_got" :** Cette table établit une relation entre les reines et les ruches en enregistrant quand une reine a été ajoutée à une ruche spécifique. Elle comporte des colonnes telles que l'identifiant de la reine, l'identifiant de la ruche et la date d'ajout de la reine dans une ruche.

3.2.3 Services

```
async getQueenBees(userId: number): Promise<QueenBee[]> {
    const { data: queenBees, error } = await noehmiDB.supabase.from('queen_bee').select('*').eq('id_user', userId)
    if (error) {
        throw new Error(error.message)
    }
    if (!queenBees || queenBees.length === 0) {
        console.log('No queen bees found')
        return []
    } else {
        return queenBees as QueenBee[]
    }
},
```

Figure 8: Exemple de service pour la généalogie des reines

Le service **queenBeeService** fournit différentes fonctionnalités pour interagir avec la base de données. Voici un aperçu d'une opération possible :

getQueenBees(userId: number): Promise<QueenBee[]>: Cette fonction récupère toutes les reines d'abeilles associées à un utilisateur spécifié par son ID Elle effectue une requête à la base de données pour sélectionner toutes les entrées dans la table "queen_bee" où la colonne "id_user" correspond à l'ID utilisateur spécifié. Si des reines d'abeilles sont trouvées, elles sont renvoyées sous la forme d'un tableau de QueenBee[]. Sinon, un tableau vide est renvoyé.

la fonction getQueenBees(userId: number) permet de comprendre le fonctionnement général des autres fonctions du service queenBeeService il faudra juste remplacer le .select par un insert, update ou delete si on veut ajouter, modifier ou supprimer des données de la BDD et pour si vous rajouter un .select() après avoir ajouter, éditer ou supprimer cela vous renvoie les informations de l'objet ici les reines du coup.

3.2.4 Stores :

```
setQueens: (queens: QueenBee[]) => {
    update((session) => {
        const newSession = { ...session, queens }
        if (typeof window !== 'undefined') {
            localStorage.setItem('userSession', JSON.stringify(newSession))
        }
        return newSession
    })
},
```

Figure 9: Stokage des reines dans le localStorage avec le store

La fonction "**setQueens**" permet de mettre à jour et de stocker les informations des reines dans le localStorage. Chaque fois que nous avons besoin de modifier ou d'accéder à ces informations, la fonction met à jour le localStorage en conséquence des actions de l'utilisateur.

3.2.5 Front

```
export const actions = {
    add: async (event) => {
        const formData = await event.request.formData()

        const queenBee: QueenBee = {
            pseudo_queen: (formData.get('pseudo_queen') as string) || null,
            date_birth: formData.get('date_birth') as string,
            date_death: (formData.get('date_death') as string) || null,
            id_queen_bee_1: parseInt(formData.get('id_queen_bee_1') as string) ||
            id_specie: parseInt(formData.get('id_specie') as string),
            id_user: parseInt(formData.get('id_user') as string),
            origine: (formData.get('origine') as string) || null,
            nomenclature_name: (formData.get('nomenclature_name') as string) || nu
        }
        const id_user = parseInt(formData.get('id_user') as string)

        if (!queenBee.pseudo_queen) {
            return {
                status: 400,
                body: {
                    message: 'No queen bee data provided',
                },
            }
        }

        await queenBeeService.createQueenBee(queenBee)
        const queenBeeAdd = await queenBeeService.getQueenBees(id_user)

        return {
            status: 200,
            body: {
                message: 'Queen bee added',
            },
            addQueens: queenBeeAdd,
        }
    },
```

Figure 10: Exemple d'actions d'un +page.server.ts en Svelte

L'action **"add"** est utilisée pour traiter les données d'un formulaire dans l'application Svelte. Elle récupère les valeurs du formulaire, crée un

objet **"QueenBee"** avec ces valeurs, vérifie si les champs obligatoires sont présents, puis retourne une réponse appropriée.

```
function generateQueenName(Year: number) {
    const birthYear = Year
    const baseYear = 2020 // year based on which the name is generated
    const alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    let letterIndex = birthYear - baseYear // Index de la lettre correspondant à l'année
    // Calculate the queen number
    let queenNumber = 1
    reines.forEach((reine) => {
        const year = new Date(reine.date_birth).getFullYear()
        if (year === birthYear) {
            queenNumber++
        }
    })

    // Calculate the letter based on the birth year and the current year
    let letter = ''
    if (letterIndex < 0) {
        // if the letter index is negative, we take the letter from the end of the alphabet
        letterIndex = Math.abs(letterIndex) - 1
        letter = alphabet.charAt(25 - (letterIndex % 25))
    } else {
        letter = alphabet.charAt(letterIndex % 25)
    }

    // Generate the queen name
    const paddedQueenNumber = queenNumber.toString().padStart(3, '0')
    let name = letter + paddedQueenNumber

    return name
}
```

Figure 11: Algorithme nom nomenclature

L'algorithme **"generateQueenName"** a été développé en collaboration avec un apiculteur nommé Bernard pour générer des noms uniques pour les reines en fonction de leur année de naissance. Il utilise l'année de naissance pour déterminer l'index de la lettre correspondante dans l'alphabet. En comptant le nombre de reines déjà nées la même année, l'algorithme attribue un numéro de séquence à la reine. En

combinant la lettre correspondante à l'année et le numéro de séquence, il génère un nom unique pour chaque reine. Cela permet d'identifier et de suivre facilement les reines d'abeilles dans le cadre de l'élevage apicole.

```
$: if (form?.queenBeeUpdate) {
    console.log('update queens')
    setQueens(form.queenBeeUpdate)
    reines = get(clientUserStore).queens as QueenBee[]
}
```

Figure 12: Flux de mise à jour des reines dans l'application Svelte

Ce code est utilisé pour mettre à jour les données des reines dans l'application Svelte. Si des mises à jour de reines sont détectées, les données sont mises à jour et la variable reines est également actualisée pour refléter les changements.

```
function handleDrop(event: DragEvent, rucheLabel: Ruche)
```

Figure 13: Fonction qui permet de glisser-déposer une reine dans une ruche

La fonction **"handleDrop"** facilite le processus de glisser-déposer d'une reine d'abeille dans une ruche. Elle gère les vérifications de remplacement de reine, met à jour les données des reines, des ruchers et des ruches, puis envoie une requête à une API pour enregistrer l'ajout de la reine dans la ruche dans la table **"has_got"** de la BDD.

```
function isQueenPlaced(reineId: number | null) {
    // Check if the queen bee with reineId is already placed in a hive
    const rucheWithQueen = ruches.find((ruche) => ruche.reinePlaces?.reine?.id_queen_bee === reineId)
    return !!rucheWithQueen
}
```

Figure 14: Fonction qui permet de vérifier si une reine est déjà placée dans une ruche

La fonction **"isQueenPlaced"** permet de vérifier si une reine d'abeille spécifique est déjà placée dans une ruche en parcourant la liste des ruches et en comparant l'identifiant de la reine.

```
<div class="list">
    <h2>List of queen bees</h2>
    <ul>
        {#each reines as reine}
            {#if !isQueenPlaced(reine.id_queen_bee ?? null)}
                <li draggable="true" on:dragstart={(event) => handleDragStart(event, reine)}>
                    <Card>
                        <div class="card-actions">
                            <button class="card" on:click={() => reine.id_queen_bee && toggleDetails(reine.id_queen_bee)}>
                                <div class="queen-info">
                                    <h3>{reine.pseudo_queen}</h3>
                                </div>
                            </button>

                            <div class="queen-buttons">
                                <button on:click={() => handleEditClick(reine)}>
                                    <Icon stroke="none" data={edit} class="text-yellow-300" />
                                </button>
                                <button on:click={() => reine.id_queen_bee && reine.id_user && deleteBee(reine.id_queen_bee, reine.id_user)}>
                                    <Icon stroke="none" data={trash} class="text-red-error" />
                                </button>
                            </div>
                        </div>
```

Figure 15: HTML pour modifier ou supprimer une reine de l'application et voir la liste des reines

le code génère une liste de reines d'abeilles avec des fonctionnalités d'interaction, telles que le glisser-déposer, l'édition et la suppression des reines.

```
let showDetails: any = {}

function toggleDetails(id: number) {
    showDetails = { ...showDetails, [id]: !showDetails[id] }
}
```

Figure 16: Fonction qui permet d'afficher le détails de la reine en fonction de son id

Ainsi, en appelant la fonction toggleDetails avec l'ID d'une reine d'abeille, l'état d'affichage des détails pour cette reine sera inversé. Par exemple, si les détails étaient cachés, ils seront affichés, et s'ils étaient affichés, ils seront cachés. Cela permet de contrôler dynamiquement l'affichage des détails pour chaque reine d'abeille dans l'interface utilisateur.

```
{#if showDetails[reine.id_queen_bee ?? NaN]}
    <div class="details">
        <p>Date of Birth: {reine.date_birth}</p>
        <p>Date of Death: {reine.date_death}</p>
        <p>Species: {especes.find((e) => e.id_specie == reine.id_specie)?.name_specie ?? `pas d'espèces`}</p>
        <p>Parent Queen: {reines.find((r) => r.id_queen_bee == reine.id_queen_bee_1)?.pseudo_queen ?? `lignée inconnue`}</p>
        <p>Origine : {reine.origine ?? `mère inconnue`}</p>
        <p>Queen Name: {reine.nomenclature_name}</p>
    </div>
{/if}
```

Figure 17: Détails d'une reine HTML

Ce code conditionnel permet d'afficher les détails d'une reine d'abeille si son ID correspond à une propriété existante dans l'objet **showDetails** et si cette propriété a une valeur évaluée comme vraie. Les détails incluent la date de naissance, la date de décès, l'espèce, la reine parente, l'origine et le nom de la reine.

```
<div class="ruches-container">
    {#each ruches as ruche}
        {#if !(removeQueens && ruche.reinePlaces)}
            <div on:dragover={(event) => handleDragOver(event, ruche)} on:drop={(event) => handleDrop(event, ruche)} class="card2">
                <Card>
                    <div class="card-content">
                        <h2>{ruche.label} :</h2>
                        {#if ruche.reinePlaces}
                            <div class="icon-container">
                                <Icon stroke="none" data={abeille} />
                            </div>

                            <p>Reine: {ruche.reinePlaces.reine?.nomenclature_name} ({ruche.reinePlaces.reine?.pseudo_queen})</p>
                            <button
                                on:click={() => {
                                    const queenId = ruche.reinePlaces?.reine?.id_queen_bee
                                    if (queenId !== undefined) {
                                        handleRemoveQueen(queenId, ruche.id)
                                    }
                                }}>
                                <Icon stroke="none" data={trash} class="text-red-error" />
                            </button>
                        {/if}
                    </div>
                </Card>
            </div>
        {/if}
    {/each}
</div>
</div>
```

Figure 18: Liste des ruches

La section commence par une case à cocher intitulée "Retirer les ruches avec reine". Lorsque cette case à cocher est cochée, certaines ruches ne seront pas affichées si elles ont une reine associée. La valeur de cette case à cocher est stockée dans la variable **removeQueens**, et tout changement de valeur est géré par la fonction **handleRemoveQueensChange.**

- Ensuite, il y a une boucle qui itère sur la liste des ruches. Chaque itération représente une ruche.

- Avant d'afficher chaque ruche, il y a une condition qui vérifie si l'option "Retirer les ruches avec reine" est cochée (**removeQueens** est vrai) et si la ruche actuelle n'a pas de reine (**ruche.reinePlaces** est null ou non défini). Si la condition est vraie, la ruche est affichée.Chaque ruche est représentée par un élément

HTML avec la classe CSS "card2". Cette div sert de zone de dépôt pour l'opération de glisser-déposer. Deux fonctions, **handleDragOver** et **handleDrop**, sont appelées lors des événements de survol (dragover) et de dépôt (drop) sur cette div.À l'intérieur de chaque ruche, il y a un élément <p> qui affiche le nom et le pseudo de la reine associée à la ruche. L'opérateur de navigation sécurisée (?.) est utilisé pour accéder aux propriétés nomenclature_name et pseudo_queen de l'objet ruche.reinePlaces.reine. Si la ruche n'a pas de reine, ces informations seront affichées comme des valeurs nulles.

• À côté du nom et du pseudo de la reine, il y a un bouton représenté par l'élément <button>. Ce bouton permet de supprimer la reine associée à la ruche. Lorsque le bouton est cliqué, la fonction **handleRemoveQueen** est appelée avec deux paramètres : l'ID de la reine (queenId) et l'ID de la ruche (ruche.id). Avant d'appeler handleRemoveQueen, il est vérifié que l'ID de la reine n'est pas undefined.

```
// Find the queen with the matching ID
queen = queens.find((q: QueenBee) => q.id_queen_bee === Number(queenId)) || null

// Function to check if a queen is alive or deceased
function isQueenAlive(queen: QueenBee) {
    return !queen.date_death
}

// Function to find the daughters of a queen
function findDaughters(queenId: number) {
    return queens.filter((q: QueenBee) => q.id_queen_bee_1 === queenId)
}
```

Figure 19: fonctions de la page genealogie des reines

queen = queens.find((q: QueenBee) => q.id_queen_bee === Number(queenId)) || null :

Cette fonction utilise la méthode **find()** sur un tableau d'objets de type QueenBee appelé queens. L'objectif est de rechercher une reine dont l'ID correspond à une valeur spécifiée (queenId). L'opérateur || null est utilisé pour renvoyer la valeur null si aucune reine n'est trouvée avec cet ID. Sinon, la reine correspondante est stockée dans la variable queen. Cela permet d'obtenir une référence à une reine spécifique dans le tableau.

isQueenAlive(queen: QueenBee) :

Cette fonction prend en paramètre un objet de type QueenBee appelé queen et vérifie si la date de décès (date_death) de la reine est renseignée ou non. Si la date de décès n'est pas définie, la fonction renvoie true, indiquant que la reine est vivante. Sinon, elle renvoie false, indiquant que la reine est décédée. Cette fonction permet de déterminer l'état vital d'une reine spécifique.

findDaughters(queenId: number) :

Cette fonction prend en paramètre un ID de reine (queenId) sous forme de nombre. Elle utilise la méthode **filter()** sur le tableau queens pour trouver toutes les reines dont l'ID de la reine mère (id_queen_bee_1) correspond à l'ID spécifié. Les reines correspondantes sont renvoyées sous forme de tableau. Cette fonction permet de trouver toutes les filles d'une reine spécifique.

```
{#if queen}
    <h2>Genealogie de la reine {queen.pseudo_queen}</h2>
    <p>
        Status: {#if isQueenAlive(queen)}Vivante{:else}Morte{/if}
    </p>

    {#if queen.id_queen_bee_1}
        <h3>Mère: {queens.find((q) => q.id_queen_bee === queen?.id_queen_bee_1)?.pseudo_queen || 'Pas de mère'}</h3>
    {/if}

    {#if queen.id_queen_bee}
        {#if findDaughters(queen.id_queen_bee).length > 0}
            <h3>Filles:</h3>
            <ul>
                {#each findDaughters(queen.id_queen_bee) as daughter}
                    <li>
                        {daughter.pseudo_queen} ({#if isQueenAlive(daughter)}Vivante{:else}Morte{/if})
                    </li>
                {/each}
            </ul>
        {:else}
            <p>Pas de fille trouvée</p>
        {/if}
    {/if}
{:else}
    <p>Reine non trouvée</p>
{/if}
```

Figure 20: HTML de la page genealogie des reines

ce code HTML génère un rapport sur la généalogie d'une reine en utilisant les fonctions pour récupérer des informations sur sa mère et ses filles, ainsi que pour déterminer si la reine est vivante ou décédée. Le code s'adapte en fonction des données disponibles, affichant des messages appropriés si certaines informations sont manquantes.

Conclusion

En réalisant la page "Queen" pour afficher les ruches avec leurs reines associées, j'ai pu mettre en pratique mes compétences en développement web. J'ai identifié des possibilités d'amélioration esthétique pour rendre la page plus attrayante visuellement et offrir une meilleure expérience utilisateur. En ajoutant des éléments de design et en harmonisant l'interface, la page "Queen" pourrait devenir un véritable point fort de l'application.

De plus, concernant la page "**Généalogie**". Il faudrait implémenter un algorithme dédié pour récupérer automatiquement les informations sur les filles et les mères de chaque reine d'abeille. Cette fonctionnalité enrichirait considérablement la généalogie des reines et fournirait aux utilisateurs des informations précieuses sur les relations familiales entre les abeilles.

Dans l'ensemble, ce projet m'a permis de développer mes compétences en programmation et en conception d'interfaces utilisateur. J'ai appris à travailler avec des technologies web modernes et à résoudre des problèmes techniques complexes. Je suis satisfait des résultats obtenus, mais je suis conscient qu'il reste encore des possibilités d'amélioration pour rendre l'application encore plus performante et intuitive.

Je suis reconnaissant d'avoir eu l'opportunité de travailler sur ce projet et je suis convaincu que les améliorations proposées contribueront à créer une expérience utilisateur plus agréable et complète. Ce rapport technique reflète mon implication, mes compétences et mon engagement dans la réalisation de ce projet.

Table des illustrations