

TPI

Gestion des activités d'un apiculteur

Kevin Avdylaj – CID4B ETML

Chef de projet : M. Gaël Sonney

Experts : Nicolas Borboën et Pascal Benzonana

Lieu : ETML Vennes, Av. de Valmont 30, 1014 Lausanne

Date : Du jeudi 02 mai au lundi 03 juin 2024

Durée : 88 heures

1	Analyse préliminaire.....	6
1.1	Introduction.....	6
1.2	Objectifs.....	6
1.3	Planification initiale.....	6
2	Analyse / Conception	7
2.1	Concept	7
2.2	Méthode de projet.....	8
2.3	Technologies du projet	9
2.3.1	TypeScript	9
2.3.2	Node.js	9
2.3.3	Express.js.....	9
2.3.4	MySQL	9
2.3.5	Prisma (ORM)	9
2.3.6	Vue.js	9
2.3.7	Figma	10
2.3.8	Draw.io.....	10
2.3.9	Visual Studio Code	10
2.3.10	Docker	10
2.4	Fonctionnalités.....	10
2.4.1	Authentification.....	10
2.4.2	CRUD.....	10
2.4.3	Consulter les activités par année	10
2.4.4	Consulter les activités d'une ruche ou d'un rucher	10
2.4.5	Ergonomie des interfaces.....	10

2.5 Maquette	11
2.5.1 Login	11
2.5.2 Page principale	12
2.5.3 Activités	12
2.5.4 Détails ruche	13
2.5.5 Détails rucher	14
2.6 Authentification	15
2.7 Base de données	16
2.7.1 MCD	18
2.7.2 MLD	18
2.7.3 MPD (Modèle Prisma)	19
2.8 API	20
2.8.1 Architecture	20
2.9 Frontend	23
2.9.1 Architecture	23
2.10 Stratégie de test	24
3 Réalisation	24
3.1 Dépôt GIT	24
3.2 Base de données	25
3.2.1 MCD	25
3.2.2 MLD	26
3.2.3 MPD (Modèle Prisma)	27
3.3 API	28
3.3.1 Routeurs	28

3.3.2	Contrôleurs	29
3.3.3	Authentification.....	34
3.3.4	VerifyToken	34
3.4	Frontend.....	35
3.4.1	Requête API.....	35
3.4.2	Authentification.....	36
3.4.3	Consultation des ruches/rucher.....	38
3.4.4	Consultations des activités.....	41
3.4.5	Modals	43
3.5	Tests.....	47
3.5.1	API	47
3.5.2	Interface.....	56
4	Conclusion	61
4.1	Objectifs atteints	61
4.2	Objectifs non-atteints	61
4.3	Problèmes rencontrés.....	62
4.4	Améliorations possibles	62
4.5	Bilan de la planification	63
4.6	Bilan personnel.....	63
5	Sources – Bibliographie	64
6	Glossaire	65
7	Annexes.....	67

1 Analyse préliminaire

1.1 Introduction

Ce rapport contient la réalisation du début à la fin de mon travail pratique individuel (TPI) effectué à l'ETML.

Le sujet de TPI est un site web qui servira à la gestion de ruches et rucher pour un apiculteur.

Le choix du sujet du TPI n'a peu d'importance et j'ai laissé celui-ci à mon chef de projet, ce qui va nous intéresser c'est la conception d'un site web full-stack.

1.2 Objectifs

Le but est de fournir une application web permettant la journalisation des activités pour un apiculteur.

L'application doit pouvoir :

- Gérer l'authentification d'un apiculteur
- Fournir les opérations CRUD sur un rucher, une ruche et les activités qui y sont liées.
- Consulter la liste de toutes les activités par années
- Consulter la liste des ruchers et ruche d'un apiculteur

L'application doit également avoir une base de données conçue selon ce qui a été vu lors des modules ICT (104, 105, 1153).

Le code source doit être lisible et respecte les conventions de nommage standards pour le langage de programmation utilisé.

1.3 Planification initiale

La planification initiale est fournie en annexe.

2 Analyse / Conception

2.1 Concept

Lors de la saison des abeilles, un apiculteur doit régulièrement réaliser des inspections et relever les détails propres à chaque ruche. Il doit aussi exécuter des travaux ou des activités spécifiques.

Cette application est destinée à un apiculteur qui s'occupe de plusieurs ruches et réalise les travaux nécessaires à la bonne conduite de son rucher. Un rucher est composé de plusieurs ruches. Il possède un numéro de rucher, un nom et une localisation.

Une ruche possède un numéro, une description, une couleur, l'année de naissance de la reine associée à une couleur. Un apiculteur peut posséder plusieurs ruchers.

Les activités possèdent une catégorie, une description, une durée et une date. Les catégories d'activités sont inspection, mise des hausse, extraction, traitement et nourrissement. Une activité peut être réalisée sur une ruche ou un rucher (toutes les ruches du rucher).

2.2 Méthode de projet

Pour ce projet je vais utiliser la méthode des 6 pas :

❖ **INFORMER**

Ici il va falloir s'informer sur le projet, prendre connaissance des objectifs, des outils à utiliser, etc.

Pour cette étape j'ai pris connaissance du cahier des charges et eu une discussion avec le premier expert sur le déroulement du TPI. J'ai aussi obtenu des clarifications sur le cahier des charges après avoir fait part de mes questionnements à mon chef de projet.

❖ **PLANIFIER**

La phase de planification consiste simplement en la réalisation de ma planification initiale.

❖ **DÉCIDER**

Pour cette phase, je dois choisir la façon avec laquelle je vais réaliser ce projet. Les technologies utilisées pour ce projet ont déjà été décidées au préalable lors du P_APPRO 1 et 2. Il ne manque plus qu'à établir le modèle de base de données, la maquette du site ainsi que la stratégie de test. Une fois tout cela fait, on pourra passer à la phase de réalisation.

❖ **RÉALISER**

C'est ici qu'on commence à coder ! Il faut implémenter le backend (base de données, API, CRUD) et le frontend (intégration de la maquette).

❖ **CONTRÔLER**

Pour le contrôle, je vais effectuer les tests prévus sur l'application, relire et finaliser le rapport.

❖ **ÉVALUER**

L'évaluation consiste à la rédaction de la conclusion de ce rapport. Elle contient tous les bilans du projet, les problèmes rencontrés, l'état final de l'application, etc.

2.3 Technologies du projet

2.3.1 Typescript

TypeScript est le langage de programmation utilisé dans ce projet. Il permet de réaliser du code en frontend ainsi qu'en backend, me permettant d'utiliser qu'un seul langage pour tout le projet. TypeScript apporte des éléments supplémentaires à JavaScript, notamment les types. Cela me permet de typer mes variables et de débugger plus simplement.

2.3.2 Node.js

Node.js est un environnement d'exécution pour JavaScript qui permet d'utiliser du code JavaScript en dehors du navigateur, nécessaire pour la réalisation du backend.

2.3.3 Express.js

Express.js est un framework JavaScript qui permet de réaliser une API en fournissant les éléments de base pour leur création (routes, middlewares, etc.).

2.3.4 MySQL

MySQL est le SGBD que je vais utiliser dans ce projet. Il permet de gérer des bases de données relationnelles.

2.3.5 Prisma (ORM)

Prisma est un ORM qui supporte le TypeScript. Celui-ci va me permettre de communiquer avec la base de données en ayant les types des données directement traduits en TypeScript et ainsi me donner plusieurs avantages comme l'auto-complétion dans mon IDE, requêtes simplifiées, migrations, etc.

2.3.6 Vue.js

Vue.js est le framework frontend que j'ai choisi. Il va me simplifier l'intégration du site en me permettant de créer des composants qui contiennent leur propres templates (HTML), styles (CSS) et scripts (TypeScript), facilitant la création d'interfaces réactives.

2.3.7 *Figma*

Figma est un éditeur graphique, permettant la réalisation de maquettes pour des sites web principalement.

2.3.8 *Draw.io*

Draw.io est un site qui permet de créer plusieurs types de diagrammes. Je l'ai utilisé pour schématiser ma base de données (MCD, MLD, MPD).

2.3.9 *Visual Studio Code*

VSCode est l'IDE que j'ai utilisé pour ce projet.

2.3.10 *Docker*

Docker sera utilisé pour faire tourner la base de données ainsi que phpMyAdmin.

2.4 Fonctionnalités

2.4.1 *Authentification*

L'authentification s'effectue avec un nom d'utilisateur et un mot de passe. Une fois authentifié, l'utilisateur possède tous les droits sur l'application. Un utilisateur non-authentifié n'a accès à aucune fonctionnalité.

2.4.2 *CRUD*

Les opérations CRUD devront être établies pour les ruchers, les ruches et les activités.

2.4.3 *Consulter les activités par année*

L'application doit permettre de pouvoir consulter toutes les activités, en filtrant par année.

2.4.4 *Consulter les activités d'une ruche ou d'un rucher*

Les activités liées à une ruche ou un rucher doivent pouvoir être consultées depuis la page « Détails » de celui-ci.

2.4.5 *Ergonomie des interfaces*

Une maquette du site doit être réalisée dans le respect des critères UX (simplicité, cohérence, interaction, crédibilité, etc.).

2.5 Maquette

Pour la conception de la maquette avec Figma, j'ai décidé de me baser sur le concept du design atomique.

Le design atomique (atomic design) décompose une interface en atome, molécule, organisme, puis finalement la page entière. Un atome serait par exemple une typographie ou une couleur, puis une molécule serait une combinaison d'atomes, comme un bouton (texte avec une couleur en background).

Un organisme est une combinaison de molécules et la page finale une combinaison d'organismes. Pour mon TPI, j'ai simplifié ce processus en sélectionnant une typographie et une palette de couleurs, puis j'ai directement créé les organismes pour produire la page. C'est suffisant pour mon utilisation.

Voici un exemple :



Dans mon projet, j'ai choisi la police « Inter » et un jaune comme couleur principale et c'est à partir de ces éléments que je vais créer les pages du site.

2.5.1 Login



2.5.2 Page principale

API-culture

[Rucher](#) [Activités](#) [Logout](#)

Rucher(s) :

[Ajouter +](#)

#<Nb rucher>	Nom:<Nom>	Localisation:<localisation>	Détails				
 #<Nb ruche>	Détails	 #<Nb ruche>	Détails	 #<Nb ruche>	Détails	 #<Nb ruche>	Détails
 #<Nb ruche>	Détails	 #<Nb ruche>	Détails	 #<Nb ruche>	Détails	 #<Nb ruche>	Détails
Ajouter +							

#<Nb rucher>	Nom:<Nom>	Localisation:<localisation>	Détails			
#<Nb rucher>	Nom:<Nom>	Localisation:<localisation>	Détails			
#<Nb rucher>	Nom:<Nom>	Localisation:<localisation>	Détails			
#<Nb rucher>	Nom:<Nom>	Localisation:<localisation>	Détails			



2.5.3 Activités

API-culture

[Rucher](#) [Activités](#) [Logout](#)

Activité(s):

[< 2024 >](#)

<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier			
Description: <small>Lorum ipsum dolor sit amet, consectetur adipiscing elit. Nullam et auctor lectus. In eleifend ornare tincidunt. Suspendisse sollicitudin arcu leo, sed sodales dolor egestas sed. Curabitur sed turpis a mauris auctor blandit. Sed ac euismod purus.</small>							
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier			
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier			
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier			
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier			
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier			



2.5.4 Détails ruche

API-culture

[Rucher](#) [Activités](#) [Logout](#)**Rucher: <NomRucher>****Numéro: <Nb ruche>****Couleur: Jaune****Reine:**

- Couleur: Bleu
- Naissance: 2024

Description:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam et auctor lectus. In eleifend ornare tincidunt. Suspendisse sollicitudin arcu leo, sed sodales dolor egestas sed. Curabitur sed turpis a mauris auctor blandit. Sed ac euismod purus.

[Modifier](#)

Activité(s)

[Ajouter +](#)

<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier	Supprimer	⋮
Description: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam et auctor lectus. In eleifend ornare tincidunt. Suspendisse sollicitudin arcu leo, sed sodales dolor egestas sed. Curabitur sed turpis a mauris auctor blandit. Sed ac euismod purus.						
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier	Supprimer	⋮
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier	Supprimer	⋮
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier	Supprimer	⋮
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier	Supprimer	⋮
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier	Supprimer	⋮

2.5.5 Détails rucher

API-culture

Rucher Activités Logout

Nom: <nom du rucher>

Numéro: <Nb>

Localisation: <localisation>

Modifier

Activité(s)

Ajouter +

<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier		
Description: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam et auctor lectus. In eleifend ornare tincidunt. Suspenisse sollicitudin arcu leo, sed sodales dolor egestas sed. Curabitur sed turpis a mauris auctor blandit. Sed ac euismod purus.						
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier		
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier		
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier		
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier		
<Catégorie>	<DD:MM:YYYY>	<HH:MM>	Ruche:<Nb ruche>	Modifier		

TPI - 2024
Gestion des activités d'un apiculteur

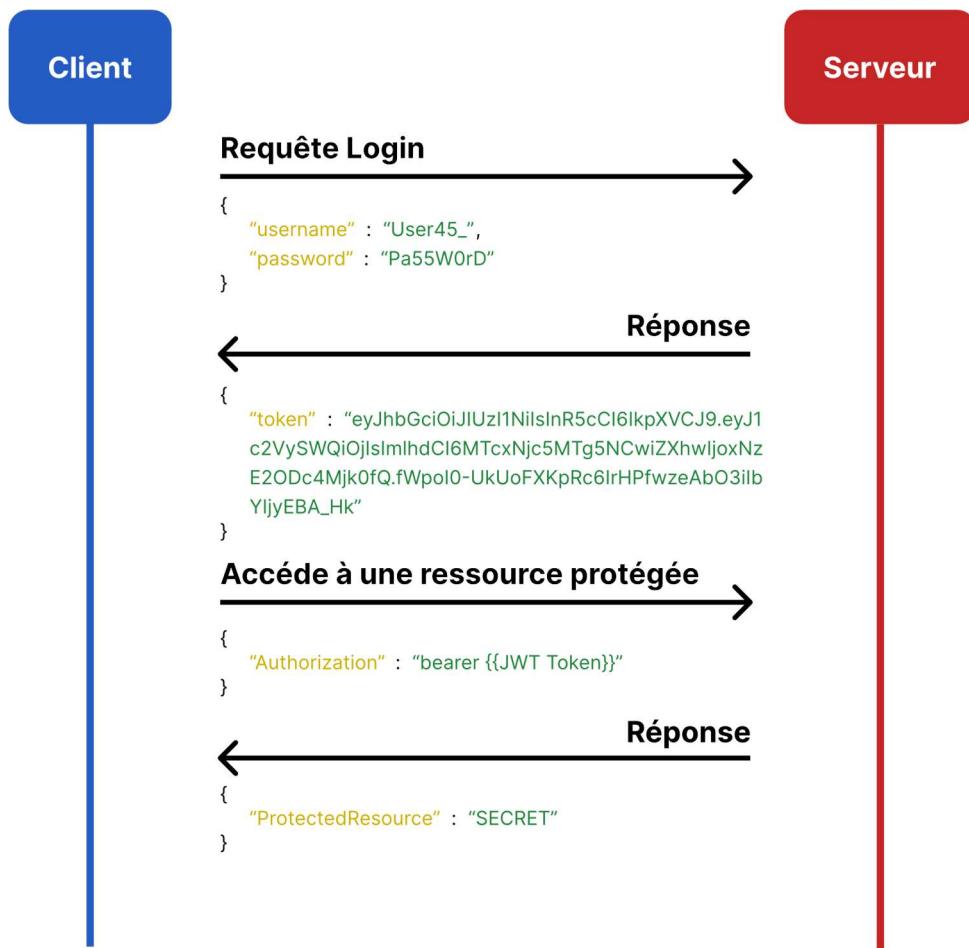
Candidat Kevin Daddy Avdylaj | Chef de projet Gaël Sonney | Experts Nicolas Borboën et Pascal Benzonana | Lieu ETML Vannes

2.6 Authentification



Pour la connexion j'utilise le « Json Web Token » (JWT). Celui-ci est un standard RFC 7519. Le JWT est un JSON en trois parties : en-tête, contenu, et sa signature. Il est représenté par une chaîne de caractères séparée en trois par des points, chacune des parties de cette chaîne de caractères est l'encodage en base 64 de l'une des composantes du JWT (en-tête, contenu, signature).

L'en-tête (Header) indique que le token est un « JWT » ainsi que l'algorithme de hachage utilisé pour générer la signature. Le contenu (Payload) contient tous types d'informations que l'on souhaite (ID de l'utilisateur, date de création, date d'expiration, etc.). La signature correspond au hachage de l'en-tête encodé en base 64 combiné au contenu également encodé en base 64.



Grâce à la signature du JWT, il peut être utilisé comme moyen d'authentification. L'utilisateur envoie ses identifiants (mot de passe, nom d'utilisateur) au serveur, le serveur vérifie la validité de ceux-ci et crée un JWT qui sera retourné au client. À chaque fois que le client souhaitera accéder à une ressource protégée, il devra envoyer son JWT qui sera utilisé comme moyen pour confirmer l'identité de l'utilisateur.

2.7 Base de données

La base de données est composée de 8 tables dont une table pivot servant à lier les activités aux ruches.

Il n'y a aucun lien entre une activité et un rucher. Comme l'application d'une activité sur un rucher consiste à appliquer l'activité à toutes les ruches qui le composent, les activités sont uniquement liées aux ruches en base de données.

Une table « t_reine » a été créée pour associer l'année de naissance d'une reine à une couleur.

Une table « t_couleur » a été créée pour associer le nom d'une couleur à son code hexadécimal, permettant ainsi d'afficher la bonne couleur en CSS si nécessaire.

Une table « t_catégorie » a été créée pour modéliser la liste prédéfinie des catégories d'activités disponibles, évitant ainsi le risque d'entrer en base de données une catégorie qui n'existe pas.

Tous les identifiants sont des entiers non signés qui s'auto-incrémentent.

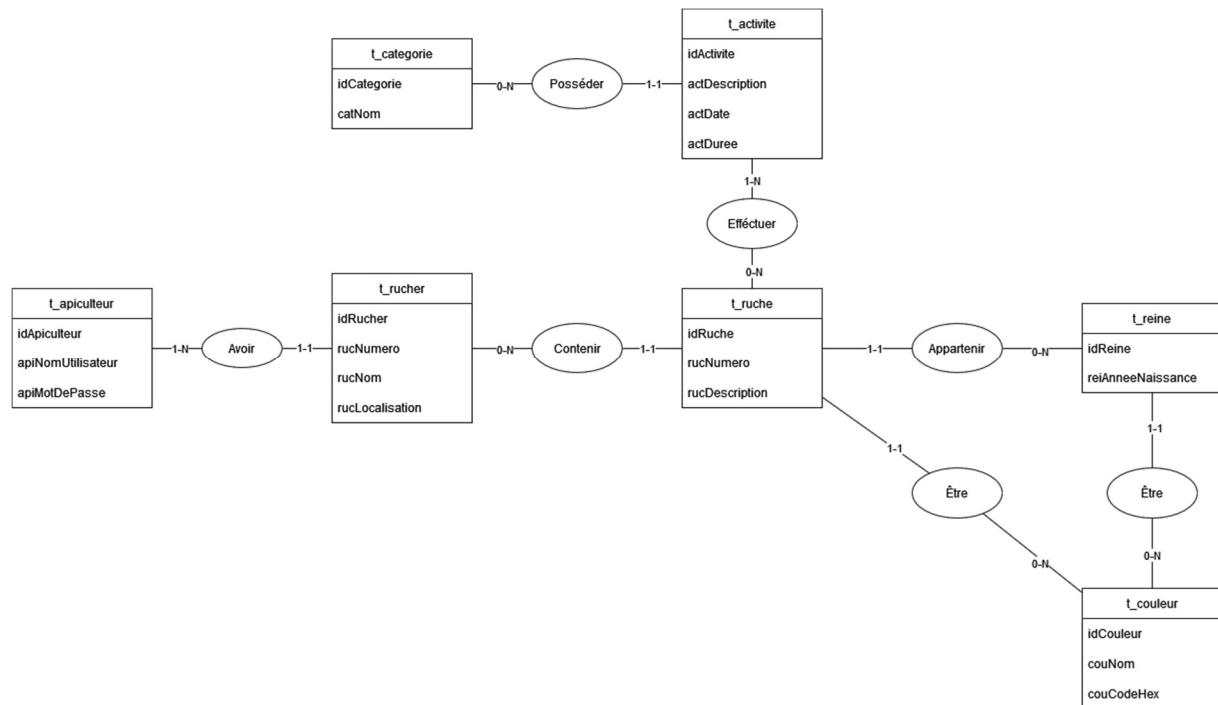
Tous les textes (sauf description) en base de données sont des varchar(255) laissant suffisamment de place pour ce qui va être stocké (nom d'utilisateur, mot de passe haché, etc.). Les descriptions sont stockées avec un type varchar(1000), ce qui est suffisant pour accueillir la donnée.

L'année de naissance d'une reine est stockée avec le type « year » de MySQL qui permet de contenir une année avec le format « YYYY », ce qui est conforme à nos besoins.

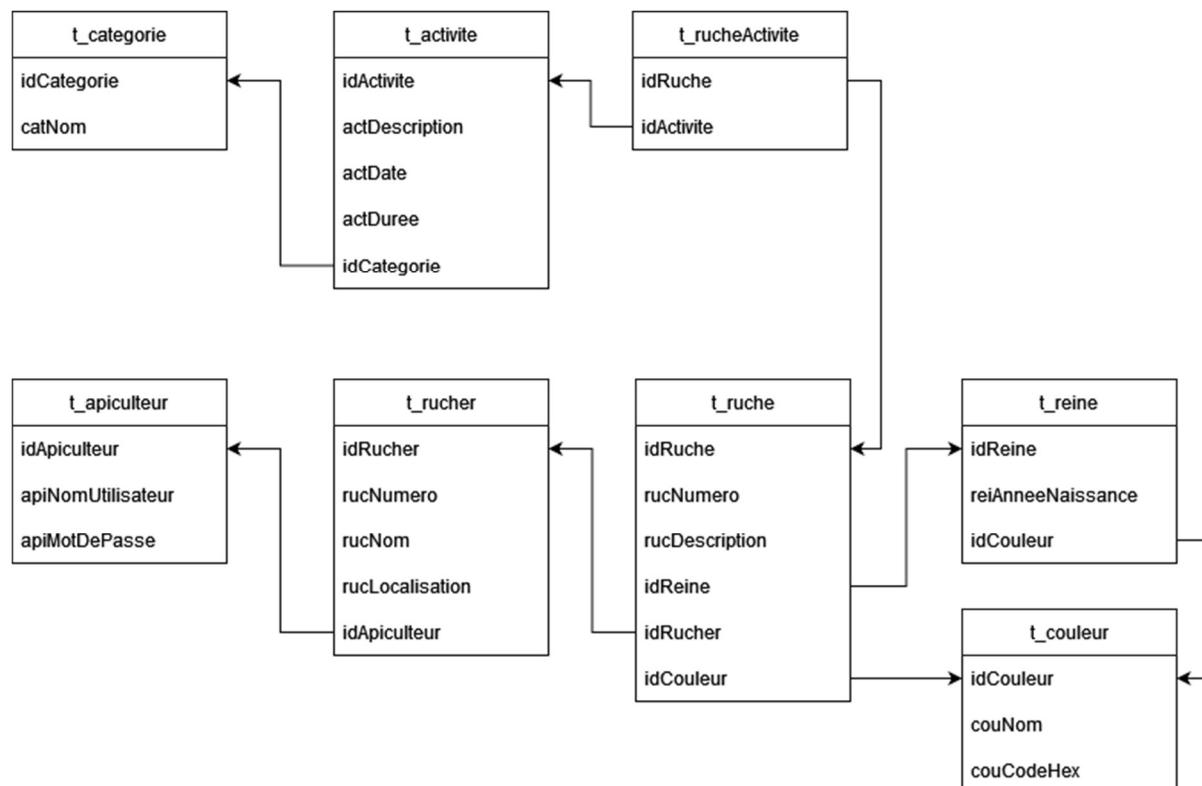
La date et la durée sont stockées avec leur type correspondant en MySQL (Date, Time).

Le MPD est réalisé avec Prisma (<https://www.prisma.io/docs/orm/prisma-schema/overview>) qui fournit sa propre façon de décrire un schéma de base de données. Prisma génère également le code SQL pour effectuer la migration et enfin avoir une base de données MySQL avec notre schéma. (La table pivot « t_rucheActivite » n'apparaît pas dans le schéma car ce genre de table est généré automatiquement par Prisma)

2.7.1 MCD



2.7.2 MLD



2.7.3 MPD (Modèle Prisma)

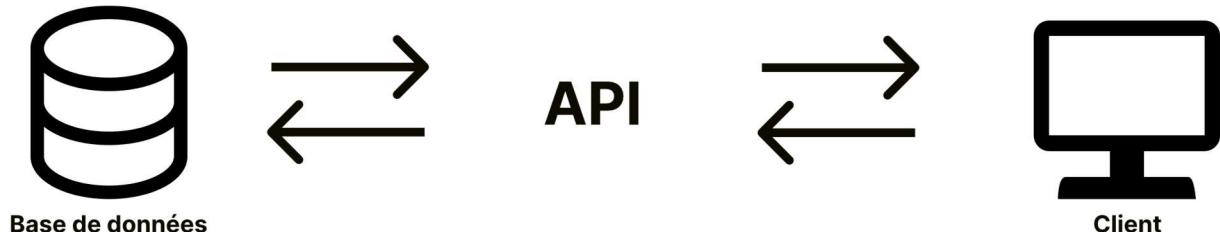
```

1 model t_categorie {
2   idCategorie Int      @id @default(autoincrement()) @db.UnsignedInt
3   catNom String    @unique @db.VarChar(255)
4   t_activite t_activite[]
5 }
6
7 model t_activite {
8   idActivite Int      @id @default(autoincrement()) @db.UnsignedInt
9   actDescription String @db.VarChar(1000)
10  actDate DateTime @db.Date()
11  actDuree DateTime @db.Time()
12  categorie t_categorie @relation(fields: [fkCategorie], references: [idCategorie])
13  fkCategorie Int @db.UnsignedInt
14  ruches t_ruche[]
15 }
16
17 model t_apiculteur {
18   idApiculteur Int      @id @default(autoincrement()) @db.UnsignedInt
19   apiNomUtilisateur String @unique @db.VarChar(255)
20   apiMotDePasse String @db.VarChar(255)
21   t_rucher t_rucher[]
22 }
23
24 model t_rucher {
25   idRucher Int      @id @default(autoincrement()) @db.UnsignedInt
26   rucNumero Int      @unique @db.UnsignedInt
27   rucNom String @db.VarChar(255)
28   rucLocalisation String @db.VarChar(255)
29   apiculteur t_apiculteur @relation(fields: [fkApiculteur], references: [idApiculteur])
30   fkApiculteur Int @db.UnsignedInt
31   t_ruche t_ruche[]
32 }
33
34 model t_couleur {
35   idCouleur Int      @id @default(autoincrement()) @db.UnsignedInt
36   couNom String    @unique @db.VarChar(255)
37   couCodeHex String @db.VarChar(6)
38   t_reine t_reine[]
39   t_ruche t_ruche[]
40 }
41
42 model t_reine {
43   idReine Int      @id @default(autoincrement()) @db.UnsignedInt
44   reiAnneNaissance Int @db.Year
45   couleur t_couleur @relation(fields: [fkCouleur], references: [idCouleur])
46   fkCouleur Int @db.UnsignedInt
47   t_ruche t_ruche[]
48 }
49
50 model t_ruche {
51   idRuche Int      @id @default(autoincrement()) @db.UnsignedInt
52   rucNumero Int      @unique @db.UnsignedInt
53   rucDescription String @db.VarChar(1000)
54   reine t_reine @relation(fields: [fkReine], references: [idReine])
55   fkReine Int @db.UnsignedInt
56   rucher t_rucher @relation(fields: [fkRucher], references: [idRucher])
57   fkRucher Int @db.UnsignedInt
58   couleur t_couleur @relation(fields: [fkCouleur], references: [idCouleur])
59   fkCouleur Int @db.UnsignedInt
60   activites t_activite[]
61 }

```

2.8 API

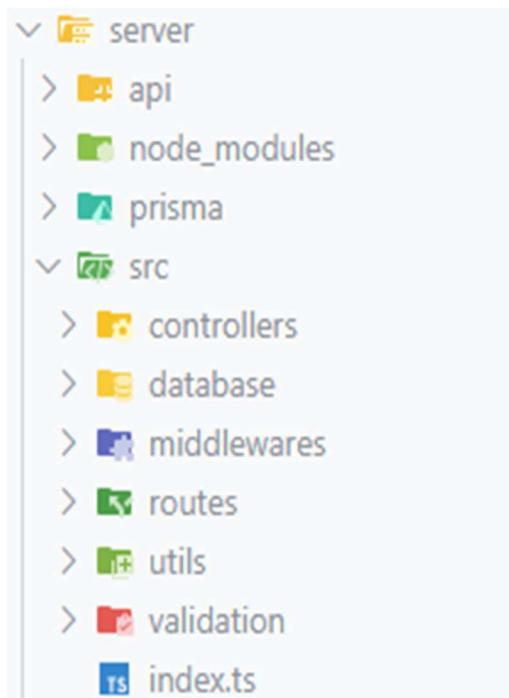
Pour permettre à notre application web d'accéder aux données dans la base de données, il nous faut une API pour faire le pont entre ces deux éléments.



La communication entre la base de données et l'API sera simplifiée avec l'ORM Prisma qui s'occupera de générer les requêtes SQL et d'inférer les types des données récupérées pour profiter pleinement des capacités de TypeScript.

2.8.1 Architecture

Pour la création de l'API, j'ai établi une architecture de dossiers qui va me servir de base pour l'organisation du code de cette API.



- **server**

Contient tout le code du serveur.

- **api**

Contient des fichiers « .rest » qui sont utilisés par l'extension VSCode « REST client ». Cette extension permet de décrire des requêtes HTTP et de les exécuter. Je l'utilise pour tester mon api.

- **node_modules**

Contient les modules et les packages de Node.js.

- **prisma**

Ce dossier est créé automatiquement par Prisma, il contient le code SQL de toutes les migrations réalisées.

- **src**

- **controllers**

Contient tous les contrôleurs de l'API. Les contrôleurs sont des fonctions appelées pour gérer les différentes requêtes.

- **database**

Contient toutes les requêtes Prisma qui seront effectuées sur la base de données pour créer, lire, modifier ou supprimer une donnée.

- **middlewares**

Contient tous les middlewares de l'API.

- **routes**

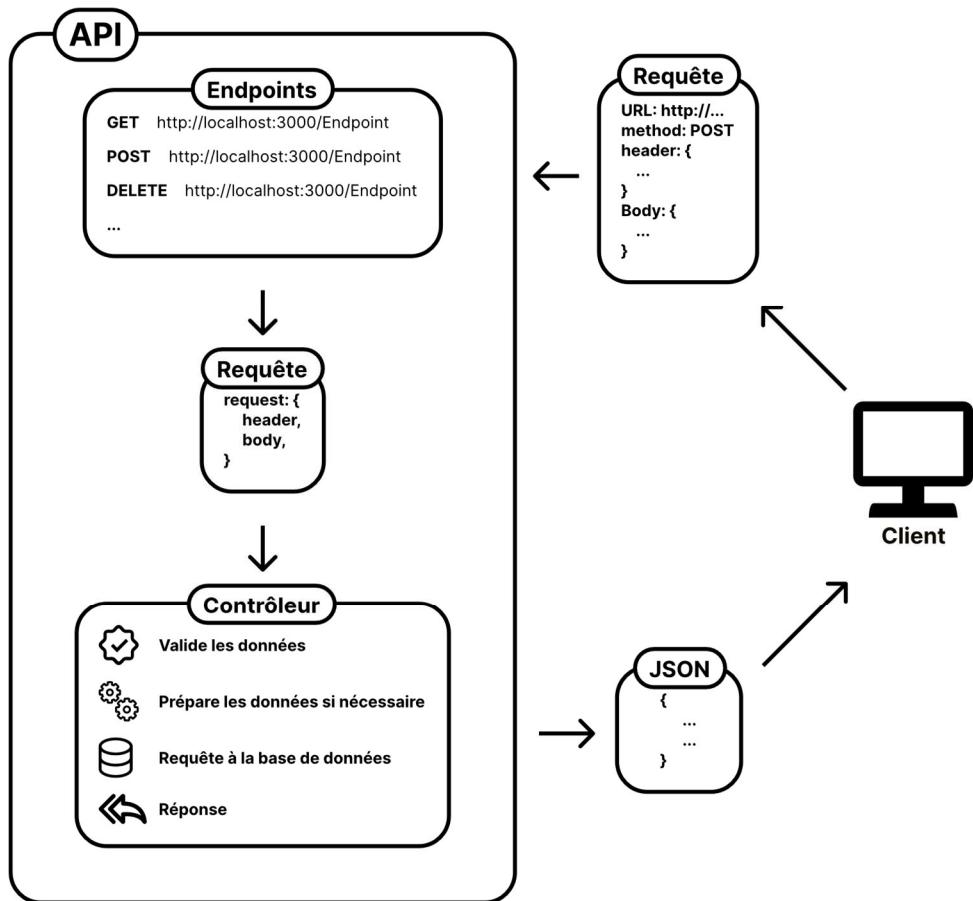
Contient toutes les routes de l'API.

- **utils**

Contient des fonctions génériques qui sont utiles à plusieurs endroits de l'API.

- **validation**

Contient du code pour la validation des données reçues depuis les requêtes exécutées auprès de l'API.



À chaque fois que l'utilisateur va faire une requête à l'API, un contrôleur lié à l'endpoint qui a été appelé par le client va prendre en charge la requête.

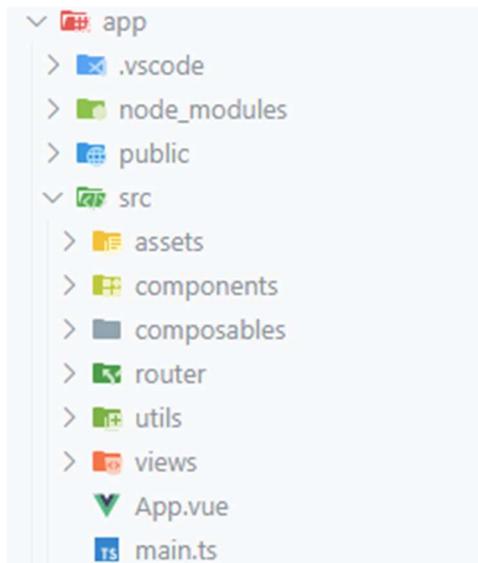
Celui-ci va tout d'abord réaliser une validation des données reçues par le client, si celles-ci ne sont pas conformes à ce que le contrôleur attend de recevoir, une erreur sera retournée avec un message et un code HTTP approprié.

Ensuite, le contrôleur va préparer les données reçues si c'est nécessaire : par exemple, mettre le bon format de date pour pouvoir ensuite les stocker en base de données.

Pour continuer, le contrôleur va envoyer les requêtes nécessaires en base de données et finalement retourner une réponse au client sous format JSON.

2.9 Frontend

2.9.1 Architecture



- **app :**
Contient tout le code du frontend.
- **public**
Contient des ressources statiques (images, icônes, etc.)
- **src**
 - **assets**
Contient le CSS global de l'application.
 - **components**
Contient tous les composants de l'application (navbar, footer, etc.)
 - **composables**
Dans Vue.js, les composables sont des fonctions accessibles globalement qui contiennent de la logique à états. Ceci permet à plusieurs composants d'avoir une même source de données.
 - **router**
Contient le code relatif au routeur.
 - **utils**
Contient des fonctions génériques qui sont utiles à plusieurs endroits de l'application frontend.
 - **views**
Contient toutes les pages de l'application.

2.10 Stratégie de test

Dans les points techniques évalué dans le TPI (point A14 à A20) il y a :

- L'apiculteur peut se loguer dans l'application et afficher ses ruchers et ruches.
- Les opérations CRUD sur un rucher et une ruche.
- Les opérations CRUD sur une activité.
- L'utilisateur peut afficher la liste des activités pour une année spécifique.
- Les activités concernant un rucher ou une ruche sont affichées dans les détails du rucher ou de la ruche

Je vais donc concentrer mes tests sur tous ces points.

En premier lieu, je vais tester l'API en testant chaque endpoint et en vérifiant que l'on obtient le résultat attendu. Le test de tous les endpoint de l'API va me permettre de déterminer la validité des CRUD et de l'authentification.

Il va également falloir tester l'interface. Pour cela, je vais décrire les étapes que l'utilisateur doit effectuer et décrire le résultat attendu, c'est-à-dire, décrire les informations qu'il devrait voir à l'écran.

3 Réalisation

3.1 Dépôt GIT

Pour pouvoir commencer à réaliser mon projet, il me fallait un endroit pour le stocker et avoir une traçabilité de toutes mes avancé. C'est GitHub qui va être utilisé.

Mon dépôt ([TPI-GestionApiculture](#)) est divisé en deux dossiers, « doc » et « sourceCode », l'un contient la documentation, l'autre contient tout le code source.

3.2 Base de données

Les schémas de la base de données ont été conceptualisés avec l'application web « Draw.io ».

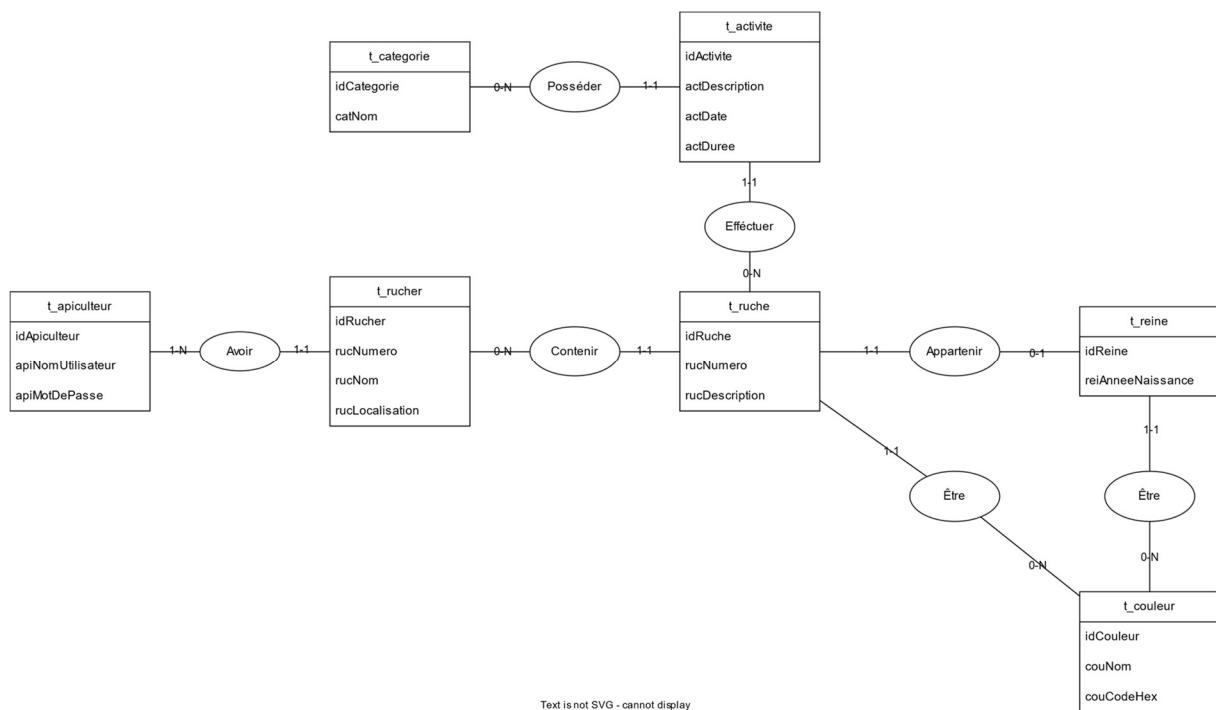
Au cours de la réalisation de mon TPI, j'ai dû changer le schéma de la base de données établi plus haut.

En effet, pendant l'intégration du site web en Vue.js, lorsque je souhaitais supprimer une activité je devais le faire à deux endroits en base de données, sur la table pivot et sur la table des activités. Avec le schéma de base vu plus haut, chaque activité peut avoir plusieurs ruches, ce qui n'est pas très pratique. Lorsque je souhaite lister toutes les activités existantes, je devais récupérer chaque activité ainsi que les liens entre elles et les ruches sur la table pivot.

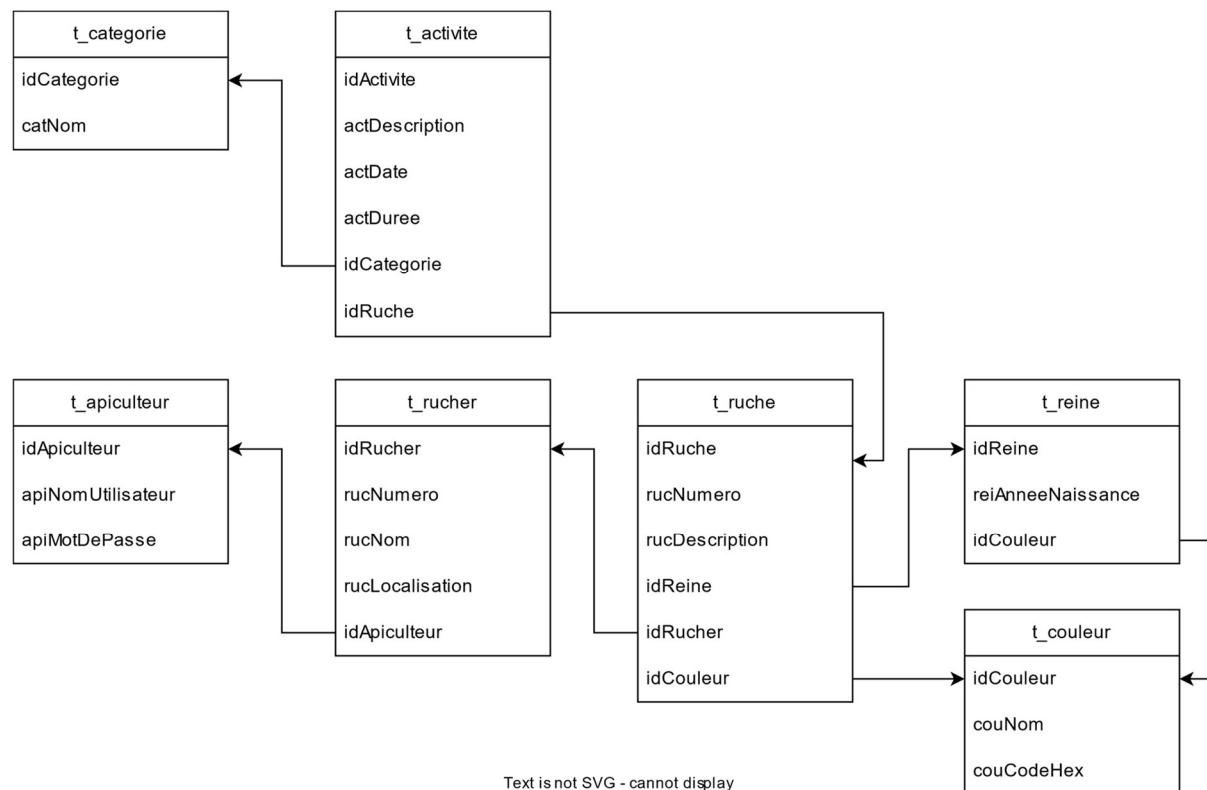
J'ai donc décidé de modifier le schéma et les activités peuvent désormais n'avoir qu'une seule ruche. L'encodage du lien entre ruche et activité ne se fait plus par la table pivot mais par une nouvelle colonne sur la table des activités qui contient la clé étrangère d'une ruche. Ceci facilite la récupération de toutes les activités et leurs suppressions.

Voici les nouveaux schémas :

3.2.1 MCD



3.2.2 MLD



3.2.3 MPD (Modèle Prisma)

```

1 model t_categorie {
2   idCategorie Int      @id @default(autoincrement()) @db.UnsignedInt
3   catNom String    @unique @db.VarChar(255)
4   t_activite t_activite[]
5 }
6
7 model t_activite {
8   idActivite Int      @id @default(autoincrement()) @db.UnsignedInt
9   actDescription String @db.VarChar(1000)
10  actDate DateTime @db.Date()
11  actDuree DateTime @db.Time()
12  categorie t_categorie @relation(fields: [fkCategorie], references: [idCategorie])
13  fkCategorie Int @db.UnsignedInt
14  ruches t_ruche @relation(fields: [fkRuche], references: [idRuche])
15  fkRuche Int @db.UnsignedInt @default(5)
16 }
17
18 model t_apiculteur {
19   idApiculteur Int      @id @default(autoincrement()) @db.UnsignedInt
20   apiNomUtilisateur String @unique @db.VarChar(255)
21   apiMotDePasse String @db.VarChar(255)
22   t_rucher t_rucher[]
23 }
24
25 model t_rucher {
26   idRucher Int      @id @default(autoincrement()) @db.UnsignedInt
27   rucNumero Int      @unique @db.UnsignedInt
28   rucNom String @db.VarChar(255)
29   rucLocalisation String @db.VarChar(255)
30   apiculteur t_apiculteur @relation(fields: [fkApiculteur], references: [idApiculteur])
31   fkApiculteur Int @db.UnsignedInt
32   t_ruche t_ruche[]
33 }
34
35 model t_couleur {
36   idCouleur Int      @id @default(autoincrement()) @db.UnsignedInt
37   couNom String    @unique @db.VarChar(255)
38   couCodeHex String @db.VarChar(6)
39   t_reine t_reine[]
40   t_ruche t_ruche[]
41 }
42
43 model t_reine {
44   idReine Int      @id @default(autoincrement()) @db.UnsignedInt
45   reiAnneNaissance Int @db.Year
46   couleur t_couleur @relation(fields: [fkCouleur], references: [idCouleur])
47   fkCouleur Int @db.UnsignedInt
48   t_ruche t_ruche[]
49 }
50
51 model t_ruche {
52   idRuche Int      @id @default(autoincrement()) @db.UnsignedInt
53   rucNumero Int      @unique @db.UnsignedInt
54   rucDescription String @db.VarChar(1000)
55   reine t_reine @relation(fields: [fkReine], references: [idReine])
56   fkReine Int @db.UnsignedInt
57   rucher t_rucher @relation(fields: [fkRucher], references: [idRucher])
58   fkRucher Int @db.UnsignedInt
59   couleur t_couleur @relation(fields: [fkCouleur], references: [idCouleur])
60   fkCouleur Int @db.UnsignedInt
61   activites t_activite[]
62 }

```

3.3 API

3.3.1 Routeurs

L'API définit plusieurs routes pour accéder à diverses ressources. J'ai utilisé plusieurs routeurs pour regrouper certaines routes qui fournissent des ressources similaires.

```
● ● ●
1 import { config } from './utils/index'
2 import express from 'express'
3 import cors from 'cors'
4 import { authRouter, rucherRouter, categorieRouter, activiteRouter, couleurRouter, reineRouter, rucheRouter } from './routes'
5 import { verifyToken } from './middlewares'
6
7 //init
8 const app = express()
9
10 //Set up middlewares
11 app.use(cors())
12 app.use(express.json())
13 app.use('/auth', authRouter)
14 app.use('/ruche', verifyToken, rucheRouter)
15 app.use('/rucher', verifyToken, rucherRouter)
16 app.use('/categorie', verifyToken, categorieRouter)
17 app.use('/activite', verifyToken, activiteRouter)
18 app.use('/couleur', verifyToken, couleurRouter)
19 app.use('/reine', verifyToken, reineRouter)
20
21 //Start listening
22 app.listen(process.env.PORT, () => console.log(`Server started at: http://localhost:${config.PORT}`))
23 app.get('/', (req, res) => res.status(200).json({message:"Server is running !"}))
```

Le code importe les différents routeurs nécessaires à l'API et les ajoute dans l'application Express.js. Maintenant, toutes les routes des routeurs sont accessibles depuis le chemin spécifié pour chacun d'entre eux. Certains routeurs sont précédés de « verifyToken » qui est un middleware.

```
 1 import express from "express"
 2 import { rucheController } from "../controllers"
 3
 4 export const rucheRouter = express.Router()
 5
 6 rucheRouter.route('/')
 7   .get(rucheController.getAll)
 8   .post(rucheController.create)
 9
10 rucheRouter.route('/:id/activites')
11   .get(rucheController.getAllActivites)
12
13 rucheRouter.route('/:id')
14   .get(rucheController.getById)
15   .patch(rucheController.updateById)
16   .delete(rucheController.deleteById)
```

Sur l'image du dessus on a l'exemple d'un routeur, un chemin y est défini avec différentes méthodes HTTP et un contrôleur qui s'occupera de répondre à la requête reçue.

3.3.2 Contrôleurs

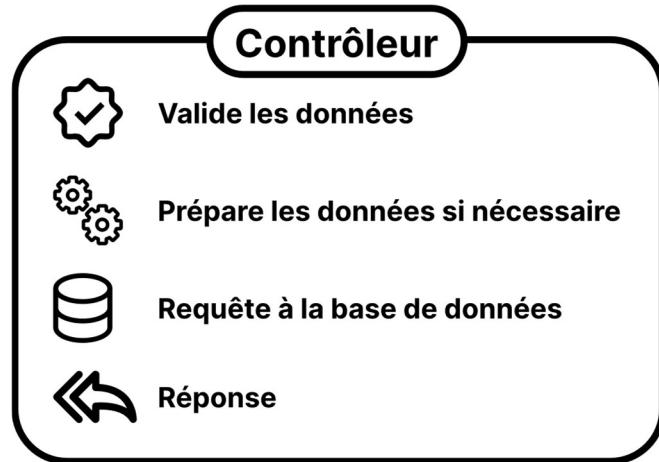
Avec Express.js, un contrôleur est une fonction qui prend en paramètre une requête et une réponse qui seront fournis par la route qui appelle le contrôleur. Ces fonctions peuvent être asynchrones et c'est le cas de tous mes contrôleurs car ils font des appels à la base de données.

Pour faciliter la création de contrôleurs j'ai créé une fonction « asyncHandler » qui prend une fonction en paramètre et retourne cette même fonction enveloppée d'un bloc try/catch.

```
● ● ●
1  export function asyncHandler(reqhndlr: RequestHandler): RequestHandler {
2      return async function (req, res, next) {
3          try {
4              await reqhndlr(req, res, next)
5          } catch (err) {
6              resWithErr(err, res)
7          }
8      }
9  }
10
11 export function resWithErr(err: unknown, res: Response) {
12     if(!err || typeof err !== "object") {
13         console.log('[resWithErr] not a valid error')
14         return
15     }
16     if(err instanceof Error) {
17         console.error(err.message)
18         res.status(500).json({error:err.message})
19         return;
20     }
21     const errStatus = (err as ErrorStatus)
22     if(!errStatus) return
23     console.error(errStatus.message)
24     res.status(errStatus.status).json({error:errStatus.message})
25 }
26
27 export type ErrorStatus = {
28     status: HttpStatusCode
29     message: string
30 }
```

Le bloc try/catch qui enveloppe la fonction reçue en paramètre de la fonction « asyncHandler » appelle la fonction « resWithErr » qui s'occupe des possibles erreurs survenues dans le bloc try. La fonction « resWithErr » vérifie le type de l'erreur reçue et répond avec le code d'erreur 500 ou bien avec celui du type « ErrorStatus ».

Grâce à « asyncHandler » je n'ai alors pas besoin de réécrire le bloc try/catch et la gestion d'erreur.



Voyons maintenant le fonctionnement d'un contrôleur. Je vais prendre ici le contrôleur qui s'occupe de gérer la requête pour créer un rucher.

```
● ● ●  
1 const create = asyncHandler(async (req, res) => {  
2   const rucher = rucherParser.parse(req.body)  
3   const rucherRecord = await rucherDB.create(rucher)  
4   res.status(status.OK_CREATED).json(rucherRecord)  
5 })
```

Valider les données

Tout d'abord, la validation. Ici « rucherParser » vient effectuer la validation du body de la requête.

```
● ● ●  
1 export const rucherParser = z.object({  
2   nbr: z.number(),  
3   name: z.string(),  
4   localisation: z.string(),  
5   fkApiculteur: z.number(),  
6 })
```

« rucherParser » est un schéma créé avec Zod, qui est une librairie TypeScript. Ici, on définit le type d'un objet avec 4 propriétés qui ont chacun un type également défini. Lorsque la fonction « .parse() » est appelée, l'élément reçu en paramètre est vérifié par Zod. Si l'élément reçu en paramètre ne correspond pas à notre schéma, alors une erreur est lancée et gérée par « asyncHandler ».

Préparation des données

Pour le contrôleur qui s'occupe de créer les ruchers, il n'y a pas de préparation nécessaire. Mais dans le contrôleur qui gère la requête permettant d'obtenir toutes les activités par années par exemple, une préparation des données est nécessaire pour formatter les dates.

```
● ● ●  
1 const year = idParser.parse(req.params.year)  
2   const start = year + "-01-01"  
3   const end = year + "-12-31"
```

La requête ne contient que l'année en paramètre et pour la requête en base de données il nous faut un format « YYYY-MM-DD » alors ce contrôleur prépare les données.

Requête à la base de données

Maintenant que les données sont validées et préparées si cela a été nécessaire, on peut effectuer la requête en base de données.

```
● ● ●  
1 const rucherRecord = await rucherDB.create(rucher)  
2  
3 const create = async (rucher: Rucher) => {  
4   return await prisma.t_rucher.create({  
5     data: {  
6       rucNumero: rucher.nbr,  
7       rucNom: rucher.name,  
8       rucLocalisation: rucher.localisation,  
9       fkApiculteur: rucher.fkApiculteur,  
10    }  
11  })  
12 }
```

Ici on effectue une requête SQL avec Prisma. On crée un nouveau rucher avec son numéro, nom, localisation et l'id de l'apiculteur auquel il appartient.

Réponse

La réponse se fait grâce à l'objet « res » reçue en paramètre.

```
● ● ●  
1 res.status(status.OK_CREATED).json(rucherRecord)
```

« status() » prend en paramètre le code d'erreur à renvoyer et « json() » le JSON à renvoyer.

Tous les contrôleurs suivent ce même schéma avec de différents validateurs, requêtes Prisma, ou réponses et possiblement une préparation de données.

```
● ● ●  
1 type HttpStatusCode = | 200 | 201 | 204 | 207 | 400 | 401 | 404 | 500  
2  
3 const OK : HttpStatusCode = 200  
4 const OK_CREATED : HttpStatusCode = 201  
5 const OK_NOCONTENT : HttpStatusCode = 204  
6 const MULTI_STATUS : HttpStatusCode = 207  
7 const BAD_REQUEST : HttpStatusCode = 400  
8 const UNAUTHORIZED : HttpStatusCode = 401  
9 const NOTFOUND : HttpStatusCode = 404  
10 const INTERNALERR : HttpStatusCode = 500  
11  
12 export type {  
13     HttpStatusCode  
14 }  
15  
16 export const status = {  
17     OK,  
18     OK_CREATED,  
19     OK_NOCONTENT,  
20     MULTI_STATUS,  
21     BAD_REQUEST,  
22     UNAUTHORIZED,  
23     NOTFOUND,  
24     INTERNALERR  
25 }
```

J'ai créé un type pour les codes d'erreur HTTP les plus récurrents ainsi qu'un objet « status » qui contient tout ces codes d'erreurs, simplifiant l'utilisation de ceux-ci.

3.3.3 Authentification

Pour s'authentifier, l'API fournit un endpoint (**POST /auth/login**) qui permet au client d'envoyer ses identifiants afin de recevoir un JWT.

```
● ● ●
1 const login = asyncHandler(async (req, res) => {
2   const { username, password } = loginParser.parse(req.body)
3   const user = await apiculteur.getByUsername(username)
4   if(!user) throw wrongCredErr
5   const match = await bcrypt.compare(password, user.apiMotDePasse);
6   if(!match) throw wrongCredErr
7   const token = jwt.sign({userId: user.idApiculteur}, config.SECRET_KEY, {expiresIn: '24h'})
8   res.status(status.OK).json({token, userId: user.idApiculteur})
9 })
```

Le contrôleur qui s'occupe de l'authentification récupère les identifiants de l'utilisateur, hash le mot de passe, vérifie avec celui en base de données et retourne le JWT créé grâce à la librairie « jsonwebtoken ».

3.3.4 VerifyToken

Comme vu plus haut, certains routeurs sont précédés de « verifyToken » avant d'être ajoutés à l'application Express.js. Ce « verifyToken » est un middleware qui s'exécute avant le contrôleur.

```
● ● ●
1 export const verifyToken = (req: Request, res: Response, next: NextFunction) => {
2   const authHeader = req.header('Authorization');
3   if(!authHeader) return res.status(status.BAD_REQUEST).json({error: "Request must contain authorization header"});
4   const bearer = authHeader.split(' ');
5   if(bearer.length !== 2) return res.status(status.BAD_REQUEST).json({error: "Authorization header must contain bearer token"});
6   const token = bearer[1];
7   try {
8     const decoded = jwt.verify(token, config.SECRET_KEY);
9     next();
10   } catch (err) {
11     res.status(status.UNAUTHORIZED).json({error: "Access denied"});
12   }
13 }
```

Ce middleware récupère le token dans l'en-tête de la requête, vérifie sa validité. S'il n'est pas valide, il répond avec un code d'erreur 401 et un message « Access denied ». S'il est valide, il passe à la suite avec la fonction « next() » qui va permettre d'exécuter le prochain middleware s'il y en a un ou bien de passer au contrôleur.

3.4 Frontend

3.4.1 Requête API

```
● ● ●
1 export type FetchRequest<T> = {
2   url: string,
3   req: RequestInit,
4   parser: z.ZodType<T>
5 }
6
7 export type FetchResult<T> = {
8   data: Ref<T> | Ref<null>,
9   loading: Ref<boolean>,
10  error: Ref<unknown>,
11  load: (fetchRequest: FetchRequest<T>, callback?: (data: T) => void) => Promise<void>
12 }
13
14 export const createFetchResult = <T>(): FetchResult<T> => {
15   return {
16     data: ref(null),
17     loading: ref(true),
18     error: ref(null),
19     load: async function (fetchRequest: FetchRequest<T>, callback?: (data: T) => void) {
20       this.data.value = null
21       this.loading.value = true
22       this.error.value = null
23       try {
24         const response = await fetch(fetchRequest.url, fetchRequest.req)
25         const json = await response.json()
26         if(!(response.status ≥ 200 && response.status < 300)) {
27           throw Error (json.error)
28         }
29         this.data.value = fetchRequest.parser.parse(json)
30         if(callback) {
31           callback(this.data.value);
32         }
33       } catch (err) {
34         this.error.value = err
35         console.log(err)
36       } finally {
37         this.loading.value = false
38       }
39     }
40   }
41 }
```

Pour simplifier la consommation de l'API dans le frontend, j'ai créé un type « FetchResult<T> » qui représente le résultat d'une requête. « Data » contient les données reçues par la requête, « loading » indique l'état de la requête, « error » contient l'erreur de la requête s'il y en a une, « load » permet d'exécuter la requête et possiblement un callback s'il a été renseigné, « Ref<> » est un type spécifique à Vue.js qui représente une fonction qui retourne une variable réactive. Vue.js réagit au changement des variables réactives et met à jour l'affichage.

« FetchRequest » représente une requête. « url » contient l'url de la requête, « req » contient la requête, c'est-à-dire, l'en-tête (authorization, Content-Type, etc.), la méthode (POST, GET, PATH, DELETE, etc.) et le contenu (body) et « parser » contient un schéma Zod qui permet de valider les données reçues par la requête. Avoir un modèle pour les données reçues permet de connaître le type de la donnée et d'exploiter pleinement TypeScript. « CreateFetchResult<T>() » est un constructeur de « FetchResult<T> ».

Tous les éléments récupérés depuis l'API sont stockés sous forme de « FetchResult<T> » dans un composable et mis à disposition globalement dans toute l'application. De cette façon, lorsqu'une action est faite par l'utilisateur sur une page, cette page peut aller mettre à jour les infos dans d'autres pages en rechargeant les données avec la fonction « load » disponible dans un « FetchResult<T> ».

Ce genre de manipulation est nécessaire lorsqu'on souhaite ajouter un nouvel élément en base de données et qu'il faut mettre à jour les parties du site qui en dépendent.

3.4.2 Authentification

Lorsque l'utilisateur n'est pas connecté, une redirection automatique sur la page de login s'effectue, grâce à la fonction « beforeEach » de Vue.js. Cette fonction prend en paramètre une autre fonction qui sera exécutée avant chaque changement de page.

Dans mon cas, je vérifie si le token de connexion (JWT) existe dans le « local storage ». Si ce n'est pas le cas, la redirection s'effectue.

```
● ● ●
1 router.beforeEach((to, from) => {
2   if(to.name === 'login') return true
3   const token = window.localStorage.getItem('token')
4   if(!token) return { path: '/login' }
5 })
```

La page login présente un simple formulaire comprenant deux zones de texte pour pouvoir y entrer un nom d'utilisateur et un mot de passe. Si ceux-ci sont incorrects, un petit message d'erreur apparaît.



Vue.js permet de renseigner un attribut « v-model » dans une balise HTML de type `input` avec une variable réactive comme valeur. Vue.js va automatiquement remplir la variable réactive avec la valeur contenue dans la balise `input`, ce qui nous permet de récupérer ce que l'utilisateur a entré.

```
● ● ●
1 <script setup lang="ts">
2   const username = ref('')
3   const password = ref('')
4 </script>
5
6 <template>
7   <form @submit.prevent class="login-form d-flex">
8     <p v-if="auth.error.value" style="color: var(--error-300);">Mauvais nom d'utilisateur ou mot de passe !</p>
9     <div class="label-input d-flex">
10       <label class="font-size-h6 font-bold">Nom d'utilisateur</label>
11       <input v-model="username" type="text" placeholder="Nom d'utilisateur ... " >
12     </div>
13     <div class="label-input d-flex">
14       <label class="font-size-h6 font-bold">Mot de passe</label>
15       <input v-model="password" type="password" placeholder="Mot de passe ... " >
16     </div>
17     <button class="btn-green" @click="authentify()">Se connecter</button>
18   </form>
19 </template>
```

Une fois que l'utilisateur a cliqué sur le bouton « Se connecter », la fonction « authentify() » se lance et envoie une requête à l'API pour récupérer un JWT et le stocker en « local storage ».

```

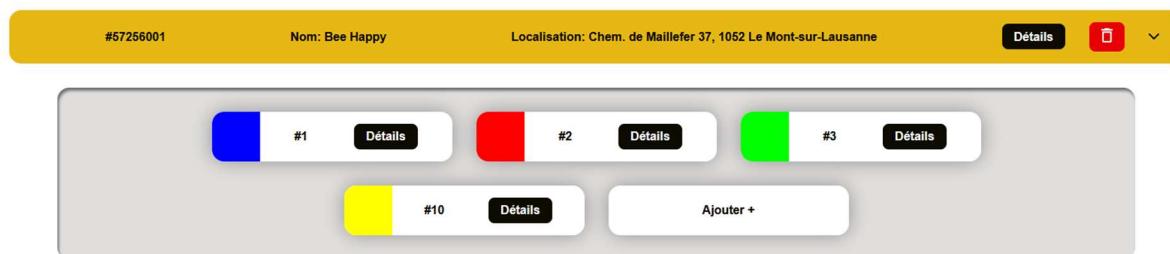
● ● ●
1 const authentify = () => {
2   auth.load({
3     url: BASE_URL + '/auth/login',
4     req: {
5       method: 'POST',
6       headers: {
7         'Content-Type': 'application/json'
8       },
9       body: JSON.stringify({username: username.value, password: password.value})
10    },
11    parser: loginParser
12  },
13  (x) => {
14    window.localStorage.setItem('token', x.token)
15    window.localStorage.setItem('user_id', `${x.userId}`)
16    loadAllCategories()
17    loadAllColor()
18    loadAllQueen()
19    router.push('/')
20  });
21 }

```

3.4.3 Consultation des ruches/rucher

Rucher(s):

Ajouter +



Les ruchers ainsi que les ruches sont tous deux implémentés au moyen des SFCs (Single File Components) mis à disposition par Vue.js.

Un SFC est représenté par un fichier avec l'extension « .vue ». Ce fichier contient en général 3 balises : script, template et style. Les composants Vue.js peuvent accepter des « props » qui sont des valeurs reçues par un composant parent et envoyé par attribut HTML.

```
● ● ●
1 export type RucherItemProps = {
2   id: number,
3   nbr: number,
4   name: string,
5   localisation: string,
6 }
7
8 const props = defineProps<RucherItemProps>()
```

Avec TypeScript, il est possible de définir les props qu'un composant doit recevoir grâce à un type.

```
● ● ●
1 <rucher-item
2   v-for="rucher in ruchersFetch.data.value"
3   v-bind={}
4     id: rucher.idRucher,
5     nbr: rucher.rucNumero,
6     name: rucher.rucNom,
7     localisation: rucher.rucLocalisation,
8   "
9 ></rucher-item>
```

L'image ci-dessus montre le composant « rucher-item » qui reçoit 4 props au moyen de l'attribut « v-bind ».

L'attribut « v-for » est également présent et sert à créer plusieurs mêmes composants grâce à une boucle. Ici, on boucle sur les ruchers reçus par l'API afin d'afficher chacun d'entre eux.

```
● ● ●
1 <div v-if="showRuche" class="ruche-item-container d-flex inner-shadow">
2   <ruche-item
3     v-for="ruche in ruches.data.value"
4     v-bind={}
5       nbr: ruche.rucNumero,
6       color: ruche.couleur.couCodeHex,
7       id: ruche.idRuche
8     "
9   ></ruche-item>
10  <div class="outline-shadow btn-add-ruche d-flex font-bold" @click="showRucheModal()">
11    Ajouter +
12  </div>
13 </div>
```

L'affichage des ruches suit le même principe de boucle. Mais dans ce cas particulier, les ruches ne doivent pas être visibles constamment, mais

seulement quand l'utilisateur sélectionne la flèche pointant vers le bas, qui est intégrée dans chacun des ruchers affichés, permettant d'afficher les ruches qui leur appartiennent.

C'est pour cela que la balise div qui enveloppe les ruches possède un attribut « v-if » avec « showRuche » comme valeur, c'est la façon dont Vue.js rend possible l'affichage conditionnel.

```
● ● ●
1 <script setup lang="ts">
2 const showRuche = ref(false);
3
4 const toggleShowRuche = () => {
5     showRuche.value = !showRuche.value
6 }
7 </script>
8
9 <template>
10    <arrow-icon direction="down" @click="toggleShowRuche()"></arrow-icon>
11 </template>
```

Lorsqu'on clique sur la flèche, la fonction « toggleShowRuche() » s'exécute et inverse la valeur de « showRuche » se qui cachera ou affichera la liste des ruches.

3.4.4 Consultations des activités

Activité(s)				
Ajouter +				
mise des hausse	2024-07-21	03:21	Ruche numéro: 2°	Modifier  
mise des hausse	2025-07-21	04:20	Ruche numéro: 3°	Modifier  
nourrissement	2024-06-21	03:30	Ruche numéro: 1°	Modifier  
nourrissement	2024-06-21	03:30	Ruche numéro: 2°	Modifier  
nourrissement	2024-06-21	03:30	Ruche numéro: 3°	Modifier  
nourrissement	2024-06-21	03:30	Ruche numéro: 10°	Modifier  

Les activités sont toutes affichées grâce au composant « ActivityList » qui génère une liste d'activités représentées par le composant « ActivityItem ».

```
● ● ●
1 export type ActivityListProps = {
2   from: 'rucher' | 'ruche' | 'all'
3 }
4
5 const props = defineProps<ActivityListProps>()
```

Le composant « ActivityList » requiert un props qui lui permet de savoir de quelle façon il doit afficher les activités, car les activités peuvent être consultées à 3 endroits, dans les détails d'une ruche ou d'un rucher, ou dans la page activité qui liste les activités par années.

```

● ● ●
1 switch (props.from) {
2   case 'all':
3     activities = activitiesFetch;
4     loadAllActivities(year.value);
5     break;
6   case 'ruche':
7     activities = activitiesByRucheFetch;
8     loadAllActivitiesByRucheId(z.coerce.number().parse(route.params.id));
9     break;
10  case 'rucher': activities = activitiesByRucherFetch;
11    loadAllActivitiesByRucherId(z.coerce.number().parse(route.params.id));
12    break;
13 }

```

Selon le props reçu, la variable « activities » qui contient les activités à afficher prend une valeur différente. Les valeurs que « activities » peut prendre sont toutes des variables globales mises à disposition dans le fichier « useActivity.ts » qui est un composable.

```

● ● ●
1 export const activitiesFetch = createFetchResult<Activity[]>()
2
3 export const activitiesByRucheFetch = createFetchResult<Activity[]>()
4
5 export const activitiesByRucherFetch = createFetchResult<Activity[]>()
6
7 export const year = ref(2024)
8
9 export const loadAllActivities = (year: number) => {
10   activitiesFetch.load({
11     url: BASE_URL + `/activite/year/${year}`,
12     req: {
13       headers: {
14         'Authorization': `bearer ${getToken()}`
15       }
16     },
17     parser: z.array(activityParser)
18   })
19 }
20
21 export const loadAllActivitiesByRucherId = (id: number) => {
22   activitiesByRucherFetch.load({
23     url: BASE_URL + `/rucher/${id}/activites`,
24     req: {
25       headers: {
26         'Authorization': `bearer ${getToken()}`
27       }
28     },
29     parser: z.array(activityParser)
30   })
31 }
32
33 export const loadAllActivitiesByRucheId = (id: number) => {
34   activitiesByRucheFetch.load({
35     url: BASE_URL + `/ruche/${id}/activites`,
36     req: {
37       headers: {
38         'Authorization': `bearer ${getToken()}`
39       }
40     },
41     parser: z.array(activityParser)
42   })
43 }

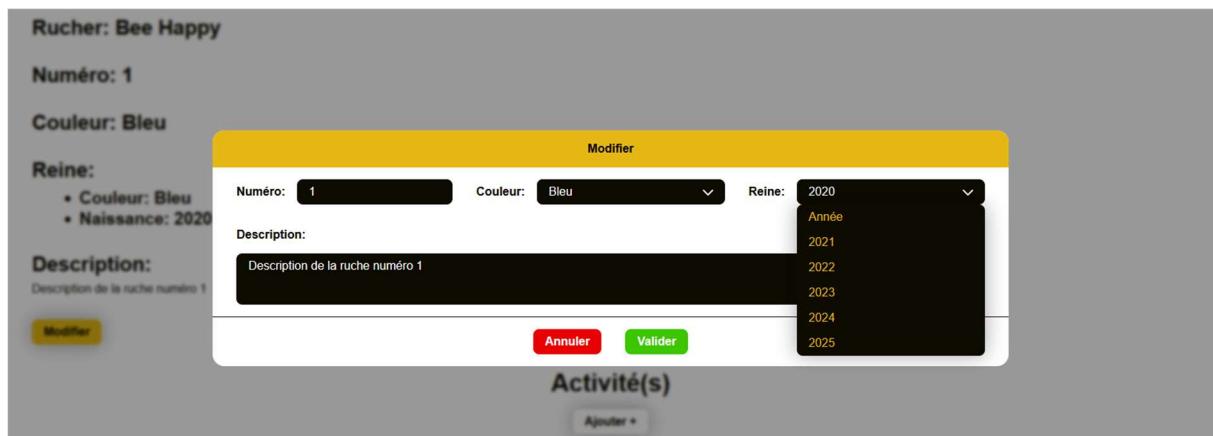
```

De cette façon, on peut mettre à jour la liste des activités depuis n'importe quel endroit du code. Il est désormais possible d'avoir un composant qui

ajoute ou modifie une activité et de rafraîchir la liste des activités, ce qui aurait été plus compliqué à faire si les activités étaient stockées directement dans le composant « ActivityList ».

Les composants ne peuvent communiquer qu'au travers de leurs parents et enfants directs, il devient vite compliqué de gérer l'accès à une ressource qui est stockée loin dans l'arbre (les composants forment une structure d'arbre), surtout quand cette ressource est sollicitée par plusieurs composants.

3.4.5 Modals



Pour s'occuper de la modification ou de l'ajout d'un élément (rucher, ruche, activité), un modal contenant un formulaire apparaît à l'écran lorsqu'un bouton de modification ou d'ajout a été sélectionné. Le formulaire est vide si c'est un ajout ou rempli avec les informations de l'élément à modifier si c'est une modification.

```

● ● ●
1 <script setup lang="ts">
2 export type ModalProps = {
3   title: string
4 }
5
6 export type ModalEmits = {
7   (e: 'canceled'): void,
8   (e: 'validated'): void,
9   (e: 'background-clicked'): void
10 }
11
12 const props = defineProps<ModalProps>()
13 const emit = defineEmits<ModalEmits>()
14 </script>
15
16 <template>
17   <div class="modal-background d-flex" @click.self="emit('background-clicked')">
18     <div class="modal">
19       <div class="modal-header font-bold">{{ props.title }}</div>
20       <slot></slot>
21       <div class="modal-footer d-flex">
22         <button class="btn-red" @click="emit('canceled')">Annuler</button>
23         <button class="btn-green" @click="emit('validated')">Valider</button>
24       </div>
25     </div>
26   </div>
27 </template>

```

Étant donné qu'il y aura 3 modals différents avec chacun 2 versions (modifier/ajouter). Pour éviter la répétition, j'ai établi le composant « modal » qui met en place tous les éléments similaires à tout autre modal disponible dans l'application, c'est-à-dire :

- Un en-tête, avec un titre
- La partie principale (elle est définie avec la balise « <slot> »)
- Un pied-de-page, avec deux boutons (annuler et valider)

La balise « slot » dans Vue.js est une balise qui définit un endroit qui va accueillir des éléments HTML donnés par le parent. Ce qui va me permettre d'y injecter un formulaire différent pour une ruche, un rucher ou une activité.

Des « emits » ont également été définis. Ce sont des événements qui peuvent être écoutés par le parent du composant ([Vue.js events](#)).

```

● ● ●
1 <template>
2   <modal-ruche v-if="modals.ruche.show"></modal-ruche>
3   <modal-rucher v-if="modals.rucher.show" ></modal-rucher>
4   <modal-activity v-if="modals.activity.show" ></modal-activity>

```

Les 3 modals sont tous définis dans « App.vue » (composant racine).

Les modals ne sont pas affichés par défaut, il faut changer la valeur booléenne d'une des variables « show » pour rendre un modal visible.

```
● ● ●
1 export const currentRucher: Ref<Rucher | null> = ref(null)
2 export const currentRuche: Ref<RucheCreate | null> = ref(null)
3 export const currentActivity: Ref<Omit<Activity, "ruches"> | null> = ref(null)
4
5
6 export const showModal = (name: keyof Modal, mode: ModalMode['mode']) => {
7   modals[name].mode = mode
8   modals[name].show = true
9 }
10
11 export const closeModal = (name: keyof Modal) => {
12   modals[name].show = false
13 }
```

Un composable « useModal.ts » permet d'afficher ou de fermer un modal grâce aux fonctions « showModal » et « closeModal ».

Les variables « currentRucher », « currentRuche » et « currentActivity » contiennent l'élément couramment utilisé pour opérer une modification.

Voyons l'implémentation du modal pour les ruchers afin de comprendre son fonctionnement (les 2 autres modals sont implémentés de façon similaire).

```
● ● ●
1 const getTitle = () => {
2   switch (modals.rucher.mode) {
3     case 'add': return 'Ajouter'
4     case 'modify': return 'Modifier'
5   }
6 }
```

Une fonction « getTitle » est définie dans chaque modal afin d'afficher le bon titre selon le mode du modal (modification ou ajout).

```
● ● ●  
1 const nbr = ref('')  
2 const name = ref('')  
3 const localisation = ref('')
```

Des variables sont instanciées pour contenir ce que l'utilisateur va entrer.

```
● ● ●  
1 const create = () => {  
2   createRucher({  
3     idRucher: 1,  
4     rucNumero: z.coerce.number().parse(nbr.value),  
5     rucNom: name.value,  
6     rucLocalisation: localisation.value,  
7     fkApiculteur: z.coerce.number().parse(getUserId()),  
8   })  
9   closeModal('rucher')  
10 }  
11  
12 const update = () => {  
13   if(!currentRucher.value) {  
14     console.error('[ModalRucher] currentRucher is null')  
15     return  
16   }  
17   updateRucher({  
18     idRucher: currentRucher.value.idRucher,  
19     rucNumero: z.coerce.number().parse(nbr.value),  
20     rucNom: name.value,  
21     rucLocalisation: localisation.value,  
22     fkApiculteur: z.coerce.number().parse(getUserId()),  
23   })  
24   closeModal('rucher')  
25 }  
26  
27 const validate = () => {  
28   switch (modals.rucher.mode) {  
29     case 'add': create(); break  
30     case 'modify': update(); break  
31   }  
32 }
```

Deux fonctions « create » et « update » sont créés et peuvent créer ou mettre à jour un rucher en base de données au travers de l'API. La fonction « validate » est appelée lorsque l'utilisateur appuie sur le bouton de validation. La fonction « validate » va exécuter « create » ou « update » selon le contexte du modal.

```
● ● ●  
1 onMounted(() => {  
2   if(modals.rucher.mode === 'modify') {  
3     nbr.value = currentRucher.value?.rucNumero ? `${currentRucher.value.rucNumero}` : ''  
4     name.value = currentRucher.value?.rucNom ? currentRucher.value.rucNom : ''  
5     localisation.value = currentRucher.value?.rucLocalisation ? currentRucher.value.rucLocalisation : ''  
6   }  
7 })
```

Lorsque le composant est monté, c'est-à-dire qu'il a été ajouté dans le DOM, la fonction fléchée reçue par « onMounted » va être exécutée. Dans ce cas, si le modal est en mode modification, les « input » vont être initialisés avec les valeurs du « currentRucher ».

3.5 Tests

3.5.1 API

Pour tester l'API, je vais définir l'endpoint testé, la forme que doit avoir la requête et le résultat attendu.

Le développement de l'application a été faite en local, donc l'URL de base est : <http://localhost:3000/>

Le serveur renvoie toujours un JSON contenant un message d'erreur si la requête est invalide.

Test N° 1		
Endpoint	GET /auth/verify	
Paramètres	Aucun	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Vérifie la validité du token JWT	
Résultat attendu		
	OK	Erreur
Statut	200	500 / 401
JSON	Aucun	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 2		
Endpoint	GET /auth/login	
Paramètres	Aucun	
Headers	Content-type : application/json	
Body	Nom d'utilisateur + mot de passe	
Description	Authentifie l'utilisateur	
Résultat attendu		
	OK	Erreur
Statut	200	500 / 401
JSON	Token JWT + id de l'utilisateur	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 3		
Endpoint	GET /activite/year/ :year	
Paramètres	:year => une année	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Récupère toutes les activités d'une année	
Résultat attendu		
	OK	Erreur
Statut	200	500 / 401
JSON	Tableau d'activités	Message d'erreur
Résultat		
Correspond au résultat attendu		

Test N° 4		
Endpoint	POST /activite/onRuche/	
Paramètres	Aucun	
Headers	Authorization : bearer <JWT token> Content-Type: application/json	
Body	Description + durée + date + catégorie (clé étrangère) + ruche (clé étrangère) de l'activité	
Description	Crée une activité sur une ruche	
Résultat attendu		
	OK	Erreur
Statut	201	500 / 401
JSON	Activité créée	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 5		
Endpoint	POST /activite/onRucher/	
Paramètres	Aucun	
Headers	Authorization : bearer <JWT token> Content-Type: application/json	
Body	Description + durée + date + catégorie (clé étrangère) + ruche (clé étrangère) de l'activité	
Description	Crée une activité sur un rucher	
Résultat attendu		
	OK	Erreur
Statut	201	500 / 401
JSON	Activité créée	Message d'erreur
Résultat		
Correspond au résultat attendu		

Test N° 6		
Endpoint	GET /activite/	
Paramètres	Aucun	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Récupère toutes les activités	
Résultat attendu		
	OK	Erreur
Statut	200	500 / 401
JSON	Tableau d'activités	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 7		
Endpoint	GET /activite/ :id	
Paramètres	:id => id d'une l'activité	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Récupère l'activité avec l'id correspondant	
Résultat attendu		
	OK	Erreur
Statut	200	500 / 401 / 400
JSON	Activité	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 8		
Endpoint	PATCH /activite/ :id	
Paramètres	:id => id d'une l'activité	
Headers	Authorization : bearer <JWT token> Content-Type: application/json	
Body	Valeurs à mettre à jour (description ou durée ou date ou catégorie (clé étrangère) ou ruche (clé étrangère) de l'activité)	
Description	Modifie l'activité avec l'id correspondant	
Résultat attendu		
	OK	Erreur
Statut	200	500 / 401
JSON	Activité	Message d'erreur
Résultat		
Correspond au résultat attendu		

Test N° 9		
Endpoint	DELETE /activite/ :id	
Paramètres	:id => id d'une activité	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Supprime l'activité avec l'id correspondant	
Résultat attendu		
OK		Erreur
Statut	200	500 / 401
JSON	Activité supprimée	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 10		
Endpoint	GET /ruche/	
Paramètres	Aucun	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Récupère toutes les ruches	
Résultat attendu		
OK		Erreur
Statut	200	500 / 401
JSON	Tableau de ruches	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 11		
Endpoint	POST /ruche/	
Paramètres	Aucun	
Headers	Authorization : bearer <JWT token> Content-Type : application/json	
Body	Numéro + description + reine (clé étrangère) + rucher (clé étrangère) + couleur (clé étrangère) de la ruche	
Description	Crée une ruche	
Résultat attendu		
OK		Erreur
Statut	201	500 / 401
JSON	Ruche créée	Message d'erreur
Résultat		
Correspond au résultat attendu		

Test N° 12		
Endpoint	GET /ruche/ :id/activites	
Paramètres	:id => id d'une ruche	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Récupère toutes les activités liées à la ruche avec l'id correspondant	
Résultat attendu		
OK		Erreur
Statut	200	500 / 401
JSON	Tableau d'activité	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 13		
Endpoint	GET /ruche/ :id	
Paramètres	:id => id d'une ruche	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Récupère la ruche avec l'id correspondant	
Résultat attendu		
OK		Erreur
Statut	200	500 / 400 / 401
JSON	Ruche	Message d'erreur
Résultat		
Correspond au résultat attendu		

Test N° 14		
Endpoint	PATCH /ruche/ :id	
Paramètres	:id => id d'une ruche	
Headers	Authorization : bearer <JWT token> Content-Type : application/json	
Body	Valeurs à modifier (numéro ou description ou reine (clé étrangère) ou rucher (clé étrangère) ou couleur (clé étrangère) de la ruche)	
Description	Modifie la ruche avec l'id correspondant	
Résultat attendu		
OK		Erreur
Statut	200	500 / 401
JSON	Ruche	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 15		
Endpoint	DELETE /ruche/ :id	
Paramètres	:id => id d'une ruche	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Supprime la ruche avec l'id correspondant	
Résultat attendu		
OK		Erreur
Statut	200	500 / 401
JSON	Ruche supprimée	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 16		
Endpoint	GET /rucher/	
Paramètres	Aucun	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Récupère tous les ruchers	
Résultat attendu		
OK		Erreur
Statut	200	500 / 401
JSON	Tableau de rucher	Message d'erreur
Résultat		
Correspond au résultat attendu		

Test N° 17		
Endpoint	POST /rucher/	
Paramètres	Aucun	
Headers	Authorization : bearer <JWT token> Content-Type : application/json	
Body	Numéro + nom + localisation + apiculteur (clé étrangère)	
Description	Crée un rucher	
Résultat attendu		
OK		Erreur
Statut	201	500 / 401
JSON	Rucher créée	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 18		
Endpoint	GET /rucher/ :id/ruches	
Paramètres	:id => id d'un rucher	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Récupère toutes les ruches liées au rucher avec l'id correspondant	
Résultat attendu		
OK		Erreur
Statut	200	500 / 401
JSON	Tableau de ruches	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 19		
Endpoint	GET /rucher/ :id/activites	
Paramètres	:id => id d'un rucher	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Récupère toutes les activités liées à un rucher avec l'id correspondant	
Résultat attendu		
OK		Erreur
Statut	200	500 / 401
JSON	Tableau d'activité	Message d'erreur
Résultat		
Correspond au résultat attendu		

Test N° 20		
Endpoint	GET /rucher/ :id	
Paramètres	:id => id d'un rucher	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Récupère le rucher avec l'id correspondant	
Résultat attendu		
OK		Erreur
Statut	200	500 / 401 / 400
JSON	Rucher	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 21		
Endpoint	PATCH /rucher/ :id	
Paramètres	:id => id d'un rucher	
Headers	Authorization : bearer <JWT token> Content-Type : application/json	
Body	Valeurs à modifier (numéro ou nom ou localisation ou apiculteur (clé étrangère))	
Description	Modifie le rucher avec l'id correspondant	
Résultat attendu		
OK		Erreur
Statut	200	500 / 401
JSON	Rucher	Message d'erreur
Résultat		
Correspond au résultat attendu		
Test N° 22		
Endpoint	DELETE /rucher/ :id	
Paramètres	:id => id d'un rucher	
Headers	Authorization : bearer <JWT token>	
Body	Aucun	
Description	Supprime le rucher avec l'id correspondant	
Résultat attendu		
OK		Erreur
Statut	200	500 / 401
JSON	Rucher supprimée	Message d'erreur
Résultat		
Correspond au résultat attendu		

3.5.2 Interface

Pour tester l'interface, je vais définir les actions que l'utilisateur doit effectuer et les informations qui devrait être affichées une fois les actions réalisées.

Tests

Test N° 1	
Fonctionnalité	Authentification
Actions	Lorsque l'utilisateur est sur la page de login, il doit renseigner son nom d'utilisateur ainsi que son mot de passe et se connecter
Résultat attendu	Il devrait être redirigé vers la page principale. Aucune ressource (activités, ruches, rucher, etc.) ne devrait être accessible sans être connecté
Résultat	Correspond au résultat attendu
Test N° 2	
Fonctionnalité	Consulter les ruchers/ruches
Actions	Se rendre sur la page principale (« rucher » sur la barre de navigation) et cliquer sur les flèches pointant vers le bas de chaque rucher pour y afficher leurs ruches respectives
Résultat attendu	Tous les ruchers/ruches devraient être affichés à l'écran
Résultat	Correspond au résultat attendu

Test N° 3	
Fonctionnalité	Consulter les activités par année
Actions	Se rendre sur la page « Activités » grâce à la barre de navigation et sélectionner une année au moyen des flèches gauche/droite qui entourent l'année actuelle
Résultat attendu	Toutes les activités avec l'année correspondante doivent être affichés
Résultat	Correspond au résultat attendu
Test N° 4	
Fonctionnalité	Consulter les détails d'un rucher/d'une ruche
Actions	Se rendre sur la page principale et cliquer sur le bouton « détails » d'une ruche ou d'un rucher
Résultat attendu	Redirection sur la page détails de la ruche ou du rucher, qui contient toutes les informations de la ruche ou du rucher. Toutes les activités liées sont également visibles
Résultat	Correspond au résultat attendu
Test N° 5	
Fonctionnalité	Ajout d'une activité
Actions	Se rendre sur la page détails d'une ruche ou d'un rucher, cliquer sur le bouton « Ajouter + » en blanc, remplir le pop-up et cliquer sur valider
Résultat attendu	L'activité devrait s'ajouter sur la ruche ou le rucher et devrait être visibles dans la liste d'activités
Résultat	Correspond au résultat attendu

Test N° 6	
Fonctionnalité	Modifier une activité
Actions	Se rendre sur la page d'activités, choisir une activité, cliquer sur le bouton « Modifier », effectuer les modifications dans le pop-up et cliquer sur valider
Résultat attendu	Les modifications devraient être visibles sur l'activité modifiée
Résultat	Correspond au résultat attendu
Test N° 7	
Fonctionnalité	Suppression d'une activité
Actions	Se rendre sur la page d'activités, choisir une activité, cliquer sur le bouton avec l'icône de poubelle en rouge
Résultat attendu	L'activité ne devrait plus être visible dans la liste
Résultat	Correspond au résultat attendu
Test N° 8	
Fonctionnalité	Ajout d'un rucher
Actions	Se rendre sur la page principale et cliquer sur le bouton « Ajouter + » en jaune, remplir toutes les informations demandées par le pop-up et cliquer sur valider
Résultat attendu	Un nouveau rucher devrait apparaître dans la liste
Résultat	Correspond au résultat attendu
Test N° 9	
Fonctionnalité	Modifier un rucher
Actions	Se rendre sur la page détails d'un rucher, cliquer sur le bouton « Modifier » en jaune et effectuer les modifications souhaitées dans le pop-up puis cliquer sur valider
Résultat attendu	Les modifications devraient être visible sur la page détails
Résultat	Correspond au résultat attendu

Test N° 10	
Fonctionnalité	Suppression d'un rucher
Actions	Se rendre sur la page principale et cliquer sur le bouton avec l'icône de poubelle en rouge correspondant au rucher que l'on souhaite supprimer
Résultat attendu	Le rucher ne devrait plus apparaître dans la liste
Résultat	La suppression ne fonctionne que si le rucher ne contient aucune ruche
Test N° 11	
Fonctionnalité	Ajout d'une ruche
Actions	Se rendre sur la page principale, cliquer sur la flèche pointant vers le bas du rucher ou l'on veut y ajouter la ruche, cliquer sur le bouton « Ajouter + » qui est apparu, remplir le pop-up et cliquer sur valider
Résultat attendu	Une nouvelle ruche devrait apparaître dans la liste
Résultat	Correspond au résultat attendu
Test N° 12	
Fonctionnalité	Modifier une ruche
Actions	Se rendre sur la page détails d'une ruche, cliquer sur le bouton « Modifier » en jaune et effectuer les modifications souhaitées dans le pop-up puis cliquer sur valider
Résultat attendu	Les modifications devraient être visible sur la page détails
Résultat	Correspond au résultat attendu

Test N° 13	
Fonctionnalité	Suppression d'une ruche
Actions	Se rendre sur la page principale et cliquer sur le bouton avec l'icône de poubelle en rouge correspondant à la ruche que l'on souhaite supprimer
Résultat attendu	La ruche ne devrait plus apparaître dans la liste
Résultat	La suppression ne fonctionne que si la ruche ne possède aucune activité

4 Conclusion

4.1 Objectifs atteints

Point technique CDC	Résultat
Modélisation et intégration de la base de données	Objectif atteint
Qualité, lisibilité du code et convention de nommage	Objectif atteint.
Authentification de l'apiculteur et affichage des ruches/rucher	Objectif atteint
CRUD sur ruche et rucher	Objectif atteint
CRUD sur une activité	Objectif atteint
Affichage des activités par année	Objectif atteint
Les ruches/ruchers affichent les activités les concernant dans leurs pages de détails	Objectif atteint

4.2 Objectifs non-atteints

Aucun, tous les points techniques ont été validés.

4.3 Problèmes rencontrés

Il n'y a pas eu d'énormes problèmes qui m'ont réellement empêché d'avancer dans mon projet. Cependant, il y a quand même eu quelques soucis qui m'ont ralenti dans mon élan.

Tout d'abord la gestion des « états » dans le frontend. Au début les composants faisaient eux-mêmes les requêtes GET à l'API et s'occupaient également de stocker le résultat. Cela n'était pas idéal, car lorsque j'effectuais une action qui mettait à jour les données de la DB je n'avais pas moyen d'accéder facilement aux données des composants pour pouvoir les rafraîchir. Un changement de l'implémentation du code a dû être effectué et actuellement les requêtes et leurs résultats sont gérés par des composables.

Enfin, le changement du schéma de la base de données. Comme expliqué plus haut dans la partie réalisation, j'ai modifié une relation dans la base de données et cela m'a fait perdre un peu de temps, car j'ai dû modifier certaines parties du code qui dépendaient de la relation qui a été modifiée.

4.4 Améliorations possibles

Actuellement l'utilisateur n'a aucun moyen de savoir si les actions qu'il a effectué ont réussi ou non sans devoir ouvrir la console de son navigateur. Par exemple lorsqu'on essaie de supprimer un rucher alors qu'il contient encore des ruches, rien ne se passe.

Il n'y a aucun feedback sur les actions que l'on peut faire sur l'application. Une bonne amélioration serait alors d'ajouter un système de pop-up qui apparaîtrait après chaque action pour indiquer la réussite ou l'échec de celle-ci.

Pour encore plus améliorer l'expérience utilisateur, il faudrait refactoriser le CSS de l'application afin d'ajuster les soucis d'alignement. L'ajout d'animations pourrait également donner à l'application un sentiment de fluidité.

Une autre amélioration possible serait de changer la structure de fichiers pour passer à une architecture « folder-by-features ». En effet, le backend et le frontend ont tous deux une architecture « folder-by-type » ce qui rend compliquée la navigation à travers le code source ([folder by type vs folder by features](#)).

4.5 Bilan de la planification

Il n'y a eu aucun souci avec la planification. Le P_APPRO 1 et 2 m'ont permis de bien pouvoir jauger le temps nécessaire pour la production d'une application fullstack. Au début du projet j'ai même été souvent en avance sur la planification (la planification détaillée ainsi que le journal de travail sont fournies en annexe).

4.6 Bilan personnel

Le projet m'a permis d'approfondir mes connaissances de développement web en me laissant l'occasion d'organiser un projet sur plusieurs semaines, le tout en utilisant des technologies qui n'ont pas été étudiées pendant ces 4 ans à l'ETML.

Je suis satisfait de ce que j'ai produit lors de ce TPI, ma planification était bien pensée et je n'avais jamais à me stresser sur ce que je devais faire ou non. Tout était décidé à l'avance grâce à la méthode des six pas, il me suffisait juste d'exécuter ce que j'avais planifié de faire pour emmener ce projet jusqu'à aboutissement.

5 Sources – Bibliographie

Fonction empruntée

- <https://github.com/ykdr2017/ts-deepcopy>

Documentations

- <https://vuejs.org/guide/introduction.html>
- <https://expressjs.com/en/guide/routing.html>
- <https://www.prisma.io/docs/orm>

P_APPRO 2 (Code réutilisé)

- https://github.com/Exy-lophone/MEVN_Skeleton

Folder by type vs folder by features

- <https://softwareengineering.stackexchange.com/questions/338597/folder-by-type-or-folder-by-feature>

6 Glossaire

- **API (Application Programming Interface)** : L'API agit comme une interface permettant la communication entre un logiciel et un service.
- **Backend** : Tout ce que l'utilisateur ne voit pas. C'est ce qui se passe dans les coulisses de l'application et fournit les fonctionnalités nécessaires à son fonctionnement.
- **CRUD (Create Read Update Delete)** : L'acronyme CRUD définit les opérations de base sur les données stockées. Créer une donnée, la lire, la mettre à jour ou la supprimer.
- **Endpoint** : Élément de l'interface d'une API qui permet la communication avec celle-ci. Représenté par une URL sur laquelle on peut y faire une requête HTTP.
- **Environnement d'exécution** : logiciel qui s'occupe de l'exécution d'un programme pour un langage de programmation donné.
- **Framework** : Structure de code sur laquelle on va construire notre application.
- **Frontend** : Tout ce que l'utilisateur voit. C'est ce avec quoi l'utilisateur va interagir pour bénéficier des fonctionnalités disposées par le backend.
- **IDE (Integrated Development Environment)** : un environnement de développement et une combinaison d'outils qui facilite la création de programme (éditeur de texte, débuggeur, etc.).
- **Intégration** : assemblage des différents éléments qui constitue une application web (textes, API, images, vidéos, etc.).
- **Middlewares** : Un middleware est une fonction qui s'exécute entre la requête faite par l'utilisateur au serveur et le traitement final de la requête par le serveur.
- **Migrations** : transitions de schémas de bases de données. Les bases de données doivent changer au fil du temps pour s'adapter aux nouvelles exigences, le changement de schéma (structure de la base de données) est une migration.
- **ORM (Object Relational Mapping)** : interface entre la base de données et le langage de programmation (orienté objet). Simplifie la communication entre les deux.

- **Route** : Une route et la combinaison entre une URL, une méthode HTTP (GET, POST, PUT, DELETE, etc.) et une fonction qui sera appelé lorsqu'une requête sera effectuée sur la route.
- **SGBD (Système de Gestion de Base de Données)** : logiciel permettant la gestion d'une base de données.

7 Annexes

Dépôt Github

<https://github.com/Exy-lophone/TPI-GestionApiculture>

Journal de travail - X-TPI-KevinAvdylaj-JDT.pdf

<https://github.com/Exy-lophone/TPI-GestionApiculture/blob/main/doc/PDFs/X-TPI-KevinAvdylaj-JDT.pdf>

Planification initiale - X-TPI-KevinAvdylaj-PlanificationInitial.pdf

<https://github.com/Exy-lophone/TPI-GestionApiculture/blob/main/doc/PDFs/X-TPI-KevinAvdylaj-PlanificationInitial.pdf>

Planification détaillée - X-TPI-KevinAvdylaj-PlanificationDetaillee.pdf

<https://github.com/Exy-lophone/TPI-GestionApiculture/blob/main/doc/PDFs/X-TPI-KevinAvdylaj-PlanificationDetaillee.pdf>

Manuel d'installation - X-TPI-ManuellInstallation.pdf

<https://github.com/Exy-lophone/TPI-GestionApiculture/blob/main/doc/PDFs/X-TPI-KevinAvdylaj-ManuellInstallation.pdf>

Base de données (Fichier SQL) - db_gestionApiculteur.sql

https://github.com/Exy-lophone/TPI-GestionApiculture/blob/main/sourceCode/db/db_gestionApiculteur.sql

Maquettes & schémas

<https://www.figma.com/design/PsJDDB3NPGUnkPbfqNpMrK/TPI-GestionApiculture?node-id=0-1&t=IIPW75JB5Ct8fi3U-1>

Code source - /sourceCode

<https://github.com/Exy-lophone/TPI-GestionApiculture/tree/main/sourceCode>

Résumé du rapport

Situation de départ

Ce TPI a comme objectif la création d'une application qui répond aux besoins d'un apiculteur, afin que celui-ci puisse organiser et gérer ces ruchers. L'application est full-stack avec une base de données relationnelle, une API et un frontend réactif. L'apiculteur doit pouvoir s'authentifier, ajouter/supprimer/modifier/consulter ces ruches, ces ruchers et ces activités.

Mise en œuvre

Le projet suit la méthode des six pas pour mener à bien sa réalisation. La réalisation de ce projet a nécessité la création d'un modèle d'une base de données ainsi que la création d'une maquette de l'interface du site. Le code est séparé en deux, le code du serveur et celui du client. Le serveur contient une API qui utilise Express.js et Prisma pour permettre la communication entre la base de données et le client. Le client est une application Vue.js. Tous les éléments sont divisés en composants avec chacun un template (HTML), un script (TypeScript) et un style (CSS).

Résultats

L'application finale a pu être finalisée dans les temps. Toutes les fonctionnalités demandées par le cahier des charges ont été implémentées. La planification a été respectée, avec une légère avance au départ et aucun retard particulier. Il manque cependant encore quelques fonctionnalités pour que l'application puisse être considérée comme terminée et livrable. Notamment, une amélioration du style global de l'application et un meilleur feedback pour l'utilisateur lorsqu'il effectue des actions qui peuvent échouer.