|  |
| --- |
| **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

Table des matières

[1 Analyse préliminaire 4](#_Toc167862486)

[1.1 Introduction 4](#_Toc167862487)

[1.2 Objectifs 4](#_Toc167862488)

[1.3 Planification initiale 4](#_Toc167862489)

[2 Analyse / Conception 5](#_Toc167862490)

[2.1 Concept 5](#_Toc167862491)

[2.2 Méthode de projet 5](#_Toc167862492)

[2.3 Technologies du projet 6](#_Toc167862493)

[2.3.1 Typescript 6](#_Toc167862494)

[2.3.2 Node JS 6](#_Toc167862495)

[2.3.3 Express JS 6](#_Toc167862496)

[2.3.4 MySQL 6](#_Toc167862497)

[2.3.5 Prisma (ORM) 7](#_Toc167862498)

[2.3.6 Vue JS 7](#_Toc167862499)

[2.3.7 Figma 7](#_Toc167862500)

[2.3.8 Draw.io 7](#_Toc167862501)

[2.3.9 VS code 7](#_Toc167862502)

[2.3.10 Docker 7](#_Toc167862503)

[2.4 Fonctionnalités 7](#_Toc167862504)

[2.4.1 Authentification 7](#_Toc167862505)

[2.4.2 CRUD 7](#_Toc167862506)

[2.4.3 Consulter les activités par année 8](#_Toc167862507)

[2.4.4 Consulter les activités d’une ruche ou d’un rucher 8](#_Toc167862508)

[2.4.5 Ergonomie des interfaces 8](#_Toc167862509)

[2.5 Maquette 8](#_Toc167862510)

[2.5.1 Login 9](#_Toc167862511)

[2.5.2 Page principale 9](#_Toc167862512)

[2.5.3 Activités 10](#_Toc167862513)

[2.5.4 Détails ruche 11](#_Toc167862514)

[2.5.5 Détails rucher 12](#_Toc167862515)§

[2.6 Base de données 12](#_Toc167862516)

[2.6.1 MCD 13](#_Toc167862517)

[2.6.2 MLD 14](#_Toc167862518)

[2.6.3 MPD 15](#_Toc167862519)

[2.7 API 16](#_Toc167862520)

[2.7.1 Architecture 16](#_Toc167862521)

[2.8 Stratégie de test 19](#_Toc167862522)

[3 Réalisation 19](#_Toc167862523)

[3.1 Tests 19](#_Toc167862524)

[3.1.1 API 19](#_Toc167862525)

[3.1.2 Interface 27](#_Toc167862526)

[4 Conclusion 32](#_Toc167862527)

[4.1 État finale de l’application 32](#_Toc167862528)

[4.2 Objectifs atteints 32](#_Toc167862529)

[4.3 Objectifs non-atteints 32](#_Toc167862530)

[4.4 Problèmes rencontrés 32](#_Toc167862531)

[4.5 Amélioration possible 32](#_Toc167862532)

[4.6 Bilan de la planification 32](#_Toc167862533)

[4.7 Bilan personnel 32](#_Toc167862534)

[5 Sources – Bibliographie 32](#_Toc167862535)

[6 Glossaire 32](#_Toc167862536)

[7 Annexes 33](#_Toc167862537)

[8 Résume 33](#_Toc167862538)

# Analyse préliminaire

## 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

## 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 lier
* 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 vue 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.

## Planification initiale

La planification initiale est fournie en annexe

# Analyse / Conception

## 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).

## 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 la partie « décider » je dois choisir la façon dont laquelle je vais réaliser ce projet. Les technologies utilisées pour ce projet ont déjà été décidée 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 ça fait, on peut 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. Conclusion qui contient tous les bilans du projet, le problème rencontrer, l’état finale de l’application, etc.

## Technologies du projet

### 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

### Node JS

Node et un environnement d’exécution pour javascript qui permet de faire du code javascript en dehors du navigateur. Il sera utilisé pour la réalisation du backend.

### Express JS

Express est un Framework javascript qui aide à la réalisation d’API en fournissant les éléments de base pour leurs création (route, middlewares, etc.).

### MySQL

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

### 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 traduit en Typescript et ainsi me donné plusieurs avantages comme l’autocomplétions dans mon IDE, requêtes simplifiée, migrations, etc.

### Vue JS

Vue JS et 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 template (HTML), style (CSS) et script (Typescript), facilitant la création d’interface réactive

### Figma

Figma est un éditeur graphique, permettant la réalisation de maquette pour site web principalement. C’est avec lui que je vais designer mon site

### Draw.io

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

### VS code

VS code est l’IDE que j’ai utilisé pour ce projet

### Docker

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

## Fonctionnalités

### 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.

### CRUD

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

### Consulter les activités par année

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

### Consulter les activités d’une ruche ou d’un rucher

Les activités liées à une ruche ou un rucher doit pouvoir être consulter depuis la page détails de celui-ci.

### Ergonomie des interfaces

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

## Maquette

Pour la conception de la maquette avec Figma j’ai décidé de me basé 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’atome, comme un bouton (texte avec une couleur en background).

Un organisme et une combinaison de molécule et la page finale une combinaison d’organisme. 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 le jaune comme couleur principale et c’est avec ces éléments que je vais créer les pages du site.

### Login

Une image contenant ruche, rayon de miel, abeille, rucher

Description générée automatiquement

### Page principale

Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Description générée automatiquement

### Activités

Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement

### Détails ruche

Une image contenant texte, capture d’écran, logiciel, Page web

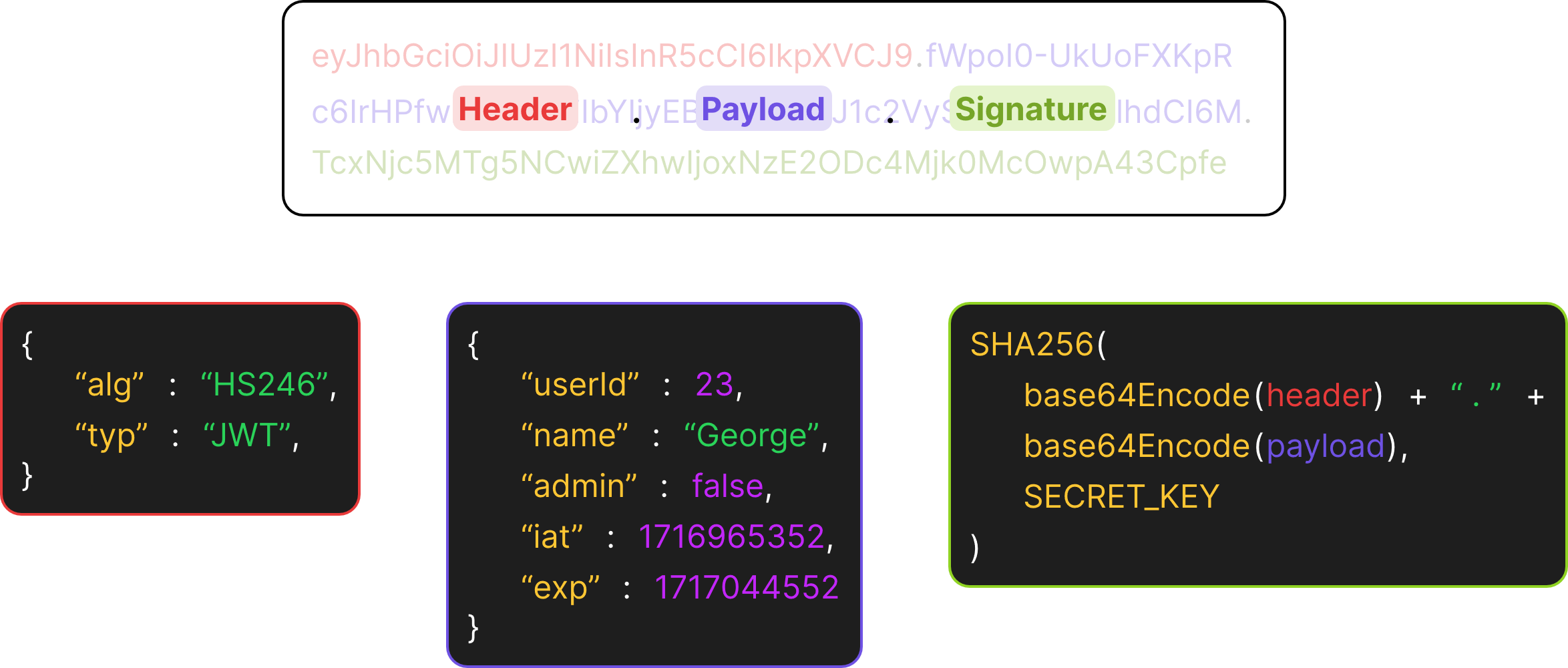
Description générée automatiquement

### Détails rucher

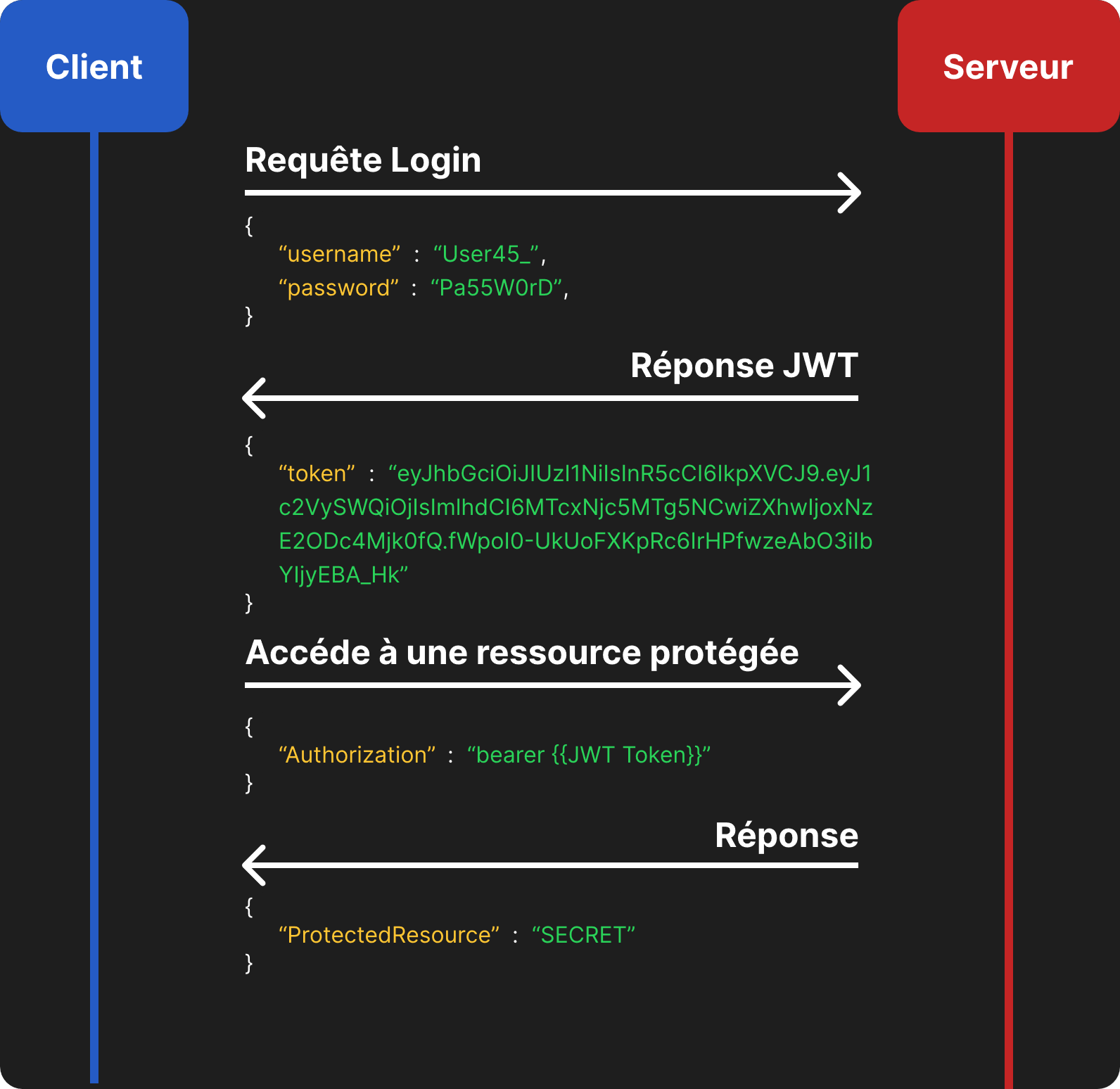
Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Description générée automatiquement

## Authentification



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

## 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 ruche qui le compose, les activités sont uniquement liées aux ruches en base de données.

Une table « t\_reine » a été créer pour associer l’année de naissance d’une reine à une couleur.

Une table « t\_couleur » a été créer pour associer le nom d’une couleur à son code hexadécimale, permettant ainsi d’afficher la bonne couleur en CSS si nécessaire.

Une table « t\_catégorie » a été créer pour modéliser la liste prédéfinie des catégories d’activité disponible, 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-incrémente.

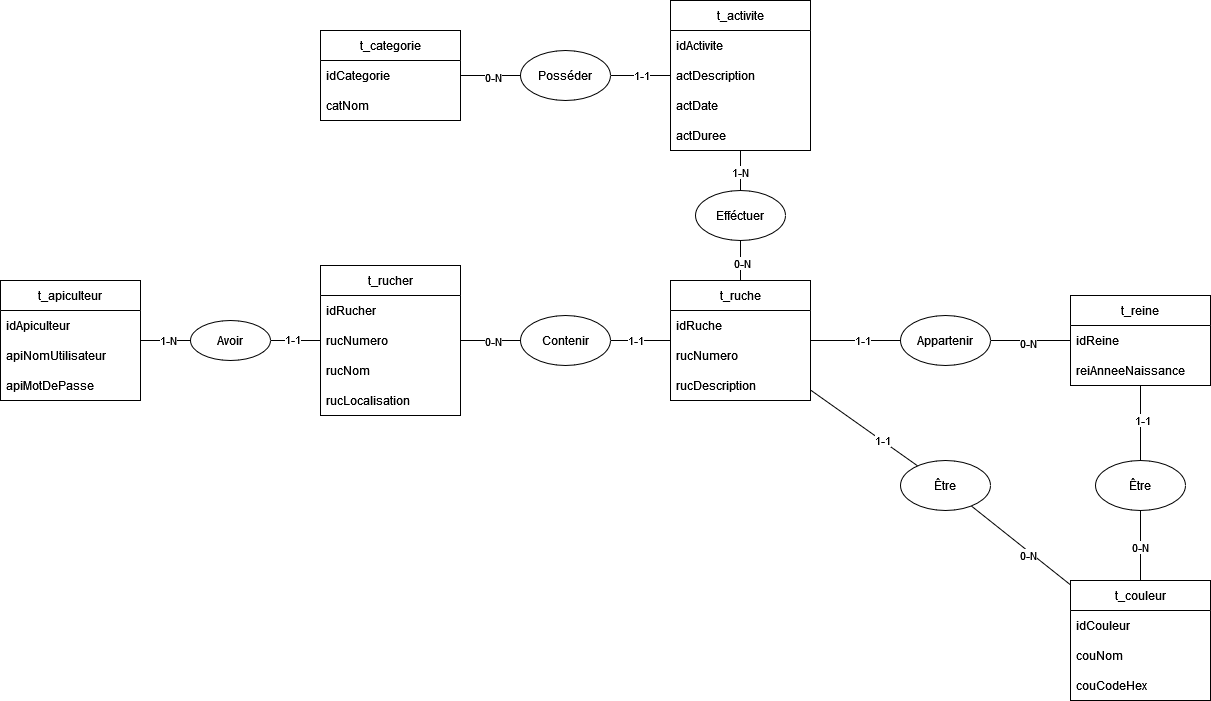
Tous les textes (sauf description) en base de données sont des varchar(255) laissant suffisamment de place ce qui va être stocké (nom d’utilisateur, mot de passe haché, etc.). Les descriptions sont stockées avec un type varchar(1000) laissant un bon paragraphe pour décrire quoi que ce soie

L’année de naissance d’une reine est stockée avec le type « year » de MySQL qui permet de contenir une année sur le format « YYYY », parfait pour nos besoins.

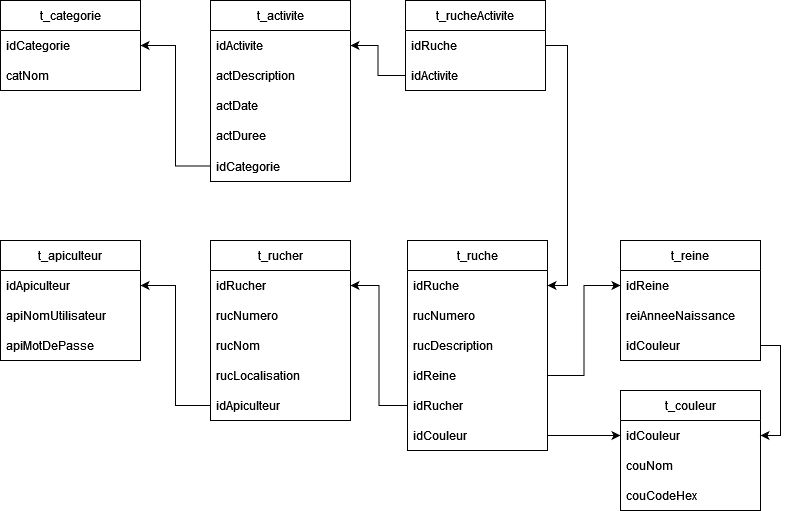
La date et la durée sont stocké avec leur type correspondant en MySQL (Date, Time).

Le MPD est réaliser avec Prisma (<https://www.prisma.io/docs/orm/prisma-schema/overview>) qui fournis ça propre façon de décrire un schéma de base de données. Prisma va également générer 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 sont générée automatiquement par Prisma)

### MCD



### MLD



### MPD (Modèle Prisma)



## 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.

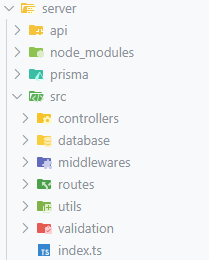
Une image contenant capture d’écran, noir, obscurité

Description générée automatiquement

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 récupérer pour profiter pleinement des capacités de Typescript.

### Architecture

Pour la création de l’api j’ai établi une architecture qui va nous servir de fondement pour le code de l’API.



* **Server :**

Contient tout le code du serveur.

* **Api**

Contient des fichiers « .rest » qui sont utilisé par l’extension vscode « REST client ». Cette extension permet de décrire des requête http et de les exécuter. Je l’utilise pour tester mon api

* **Node\_modules**

Contient module et package de node.js

* **Prisma**

Ce dossier et créer automatiquement par Prisma, il contient le code SQL de toutes les migrations réaliser

* **Src**
  + **Controllers**

Contient tous les contrôleurs de l’api. Les contrôleurs sont appelés dès qu’une requête les concernant à été faites et les gèrent.

* + **Database**

Contient toutes les requêtes faites à la base de données pour créer, lire, modifier ou supprimer une donnée.

* + **Middlewares**

Contient tous les middlewares de l’api

* + **Routes**

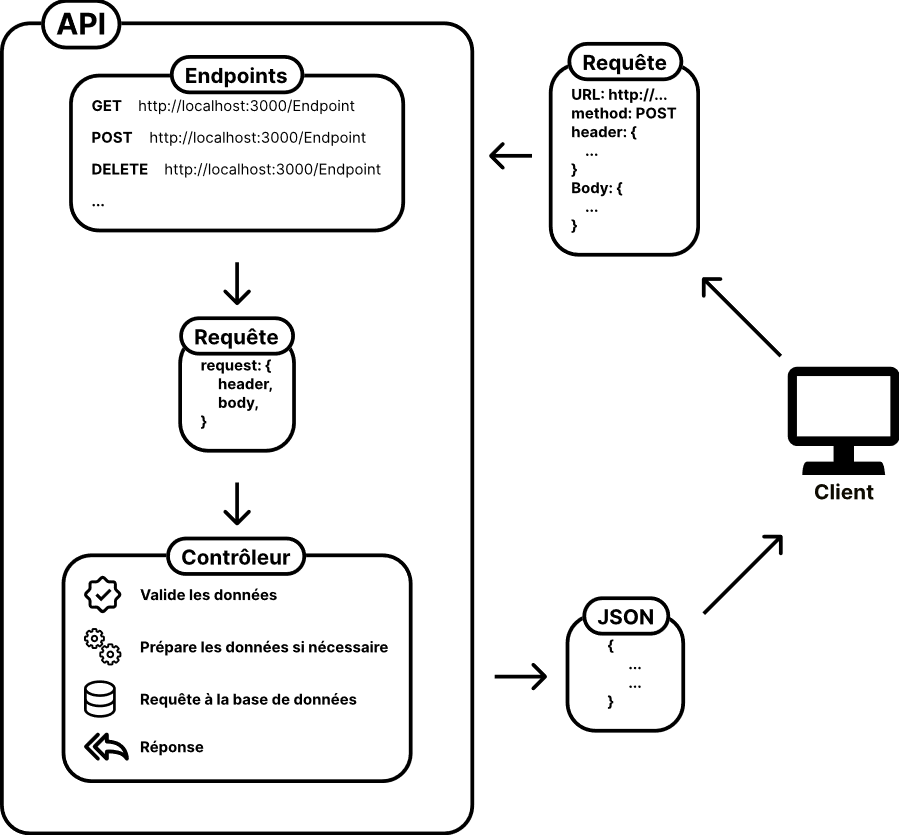
Contient toutes les routes de l’api

* + **Utils**

Contient du code qui peut être utile à l’api et qui est utilisé dans plusieurs endroit de celle-ci.

* + **Validation**

Contient du code pour la validation des données reçue depuis les requêtes faites à l’api.



À chaque fois que l’utilisateur va faire une requête à l’API, un contrôle lier à l’Endpoint qui a été appeler par le client va prendre en charge la requête.

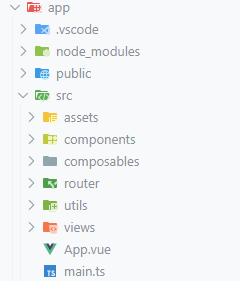
Celui-ci va faire tout d’abord une validation des données reçue par le client, si celles-ci sont erronée et/ou non conforme à ce que le contrôleur attend de recevoir, une erreur sera retournée avec un message et un code http approprié.

Ensuite, le contrôleur va préparer les données reçues si nécessaire, par exemple, mettre le bon format de date sur les données reçue pour pouvoir ensuite les stocké en base de données.

Pour continuer, le contrôleur va faire la/les requêtes en base de données nécessaire et finalement retourner une réponse au client sous forma JSON.

## Frontend

### Architecture



* **app :**

Contient tout le code du frontend.

* **Public**

Contient des ressources statiques (images, icône, etc.)

* **Src**
  + **assets**

Contient le CSS globale de l’application

* + **Components**

Contient tous les composants de l’applications (navbar, footer, etc.)

* + **Composables**

Dans vue.js les composable sont des fonctions/code contiennent de la logique avec état accessible globalement. Ceci permet à plusieurs composants d’avoir une même source de donnée.

* + **Router**

Contient le code relatif au router

* + **Utils**

Contient du code qui peut être utile à l’application et qui est utilisé dans plusieurs endroit de celle-ci.

* + **Views**

Contient toutes les pages de l’applications.

## 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 teste 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.

# Réalisation

## Dépôt GIT

Pour pouvoir commencer à réaliser mon projet, il me fallait un endroit pour le stocker et c’est git qui va être utilisé.

Pour mon dépôt ([TPI-GestionApiculture](https://github.com/Exy-lophone/TPI-GestionApiculture)) il est diviser en deux dossier, « doc » et « sourceCode » l’un contient la documentation, l’autre tout le code source.

## Base de données

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

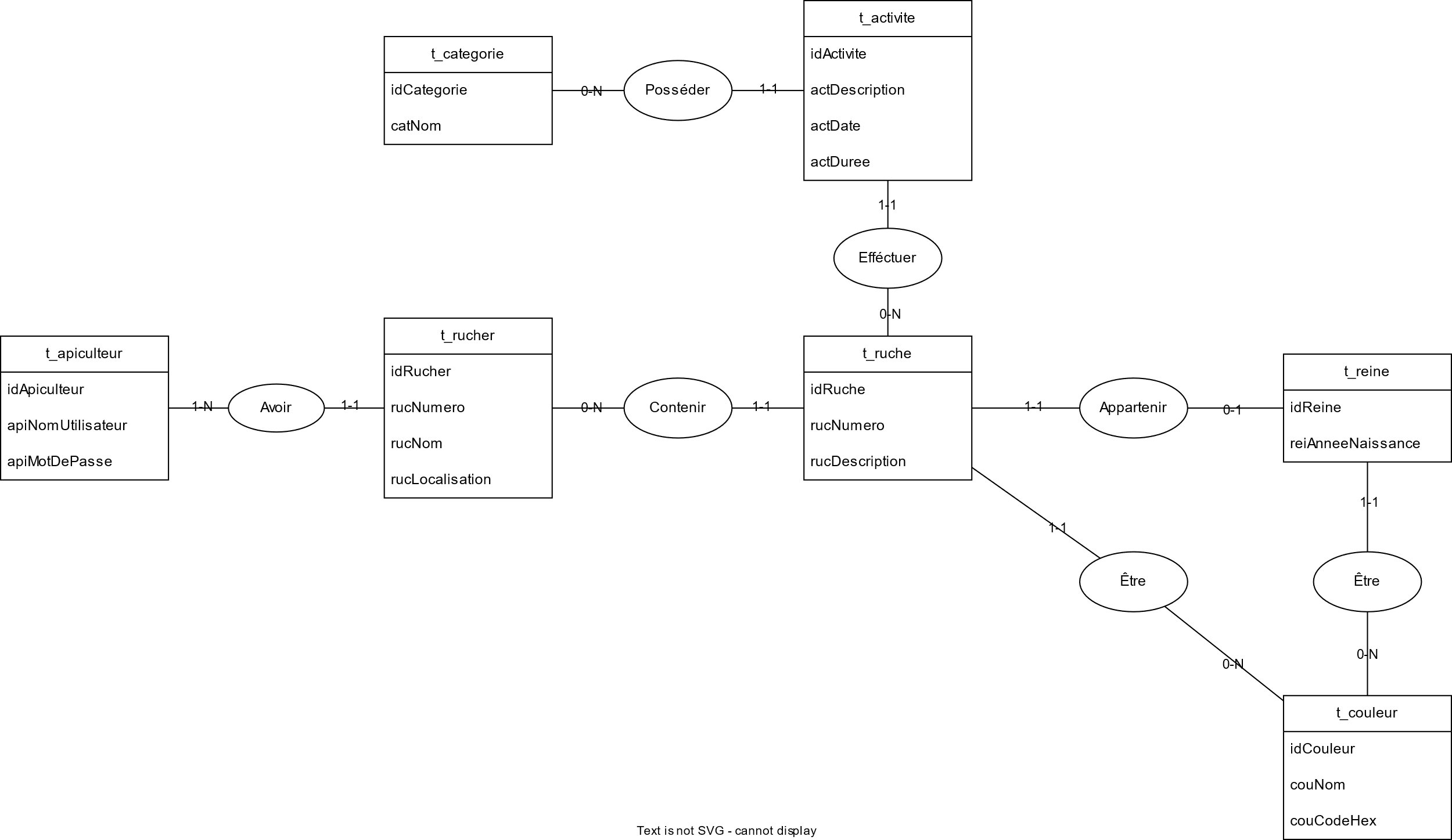
Au cours de la réalisation de mon TPI, j’ai dû changer le schéma de la base de données établi plus-haut.

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 vue 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.

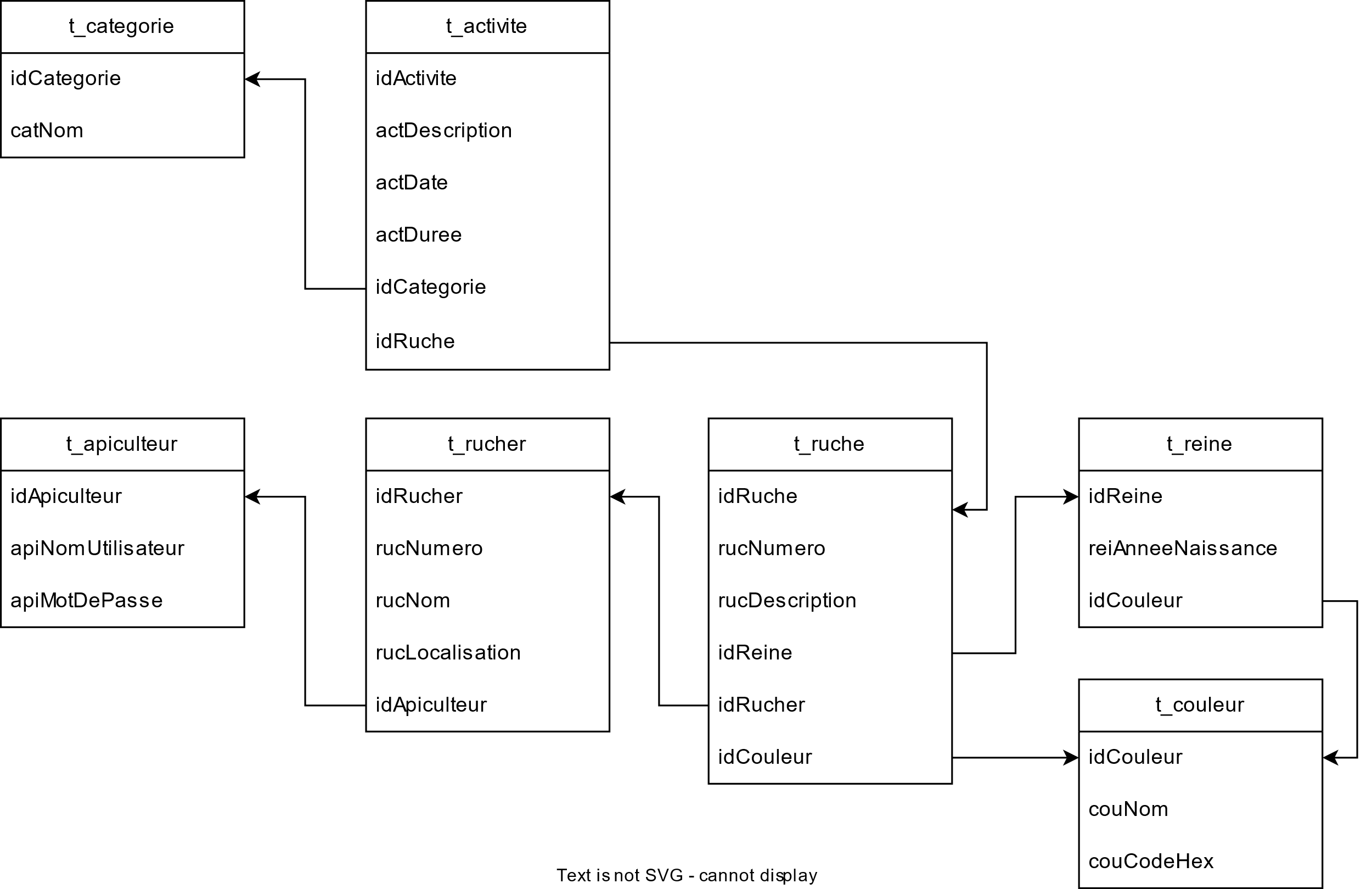
J’ai donc décider de modifier le schéma et maintenant les activités ne peuvent avoir qu’une seule ruche. L’encodage du lien entre ruche et activité ne se fait plus par la table pivot mais par une nouvelle colonne sur la table des activités qui contient la clé étrangère d’une ruche. Ceci facilite la récupération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

Voici les nouveaux schémas :

### MCD



### MLD



### MPD (Modèle Prisma)



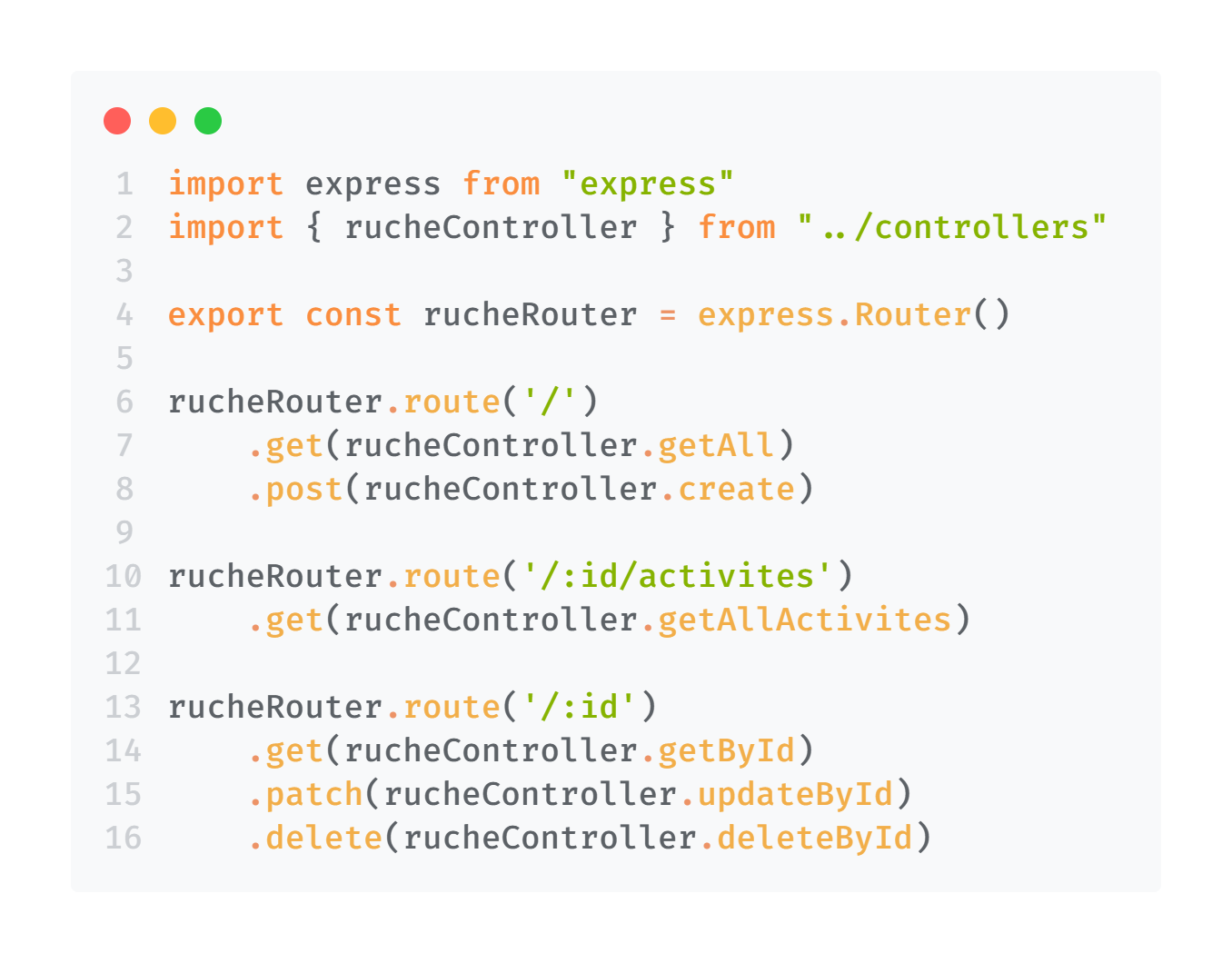
## API

### Routeurs

L’API définie plusieurs routes pour accéder à diverse ressources. J’ai utilisé plusieurs routeurs pour regrouper certaines routes qui fournissent des ressources similaires.



Le code importe les différents routeurs nécessaires à l’API et les ajoutes dans l’application express.js. Maintenant toutes les routes des routeurs sont accessible depuis le chemin spécifié pour chacun d’entre eux. Certaine routeurs sont précédé de « verifyToken » qui est un middleware.



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

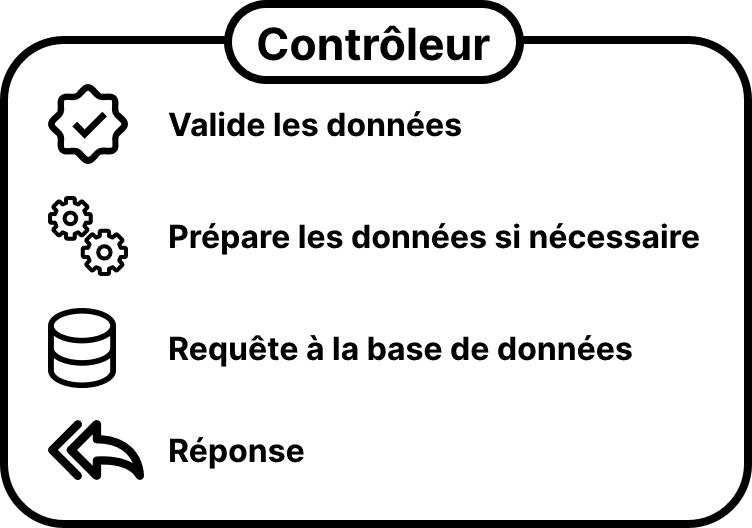
### Contrôleurs

Avec express.js un contrôleur est une fonction qui prend en paramètre une requête et une réponse qui seront fournis par la route qui appel 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 prends une fonction en paramètre et retourne cette même fonction enveloppée d’un bloc try/catch.



Le bloc try/catch qui enveloppe la fonction reçue en paramètre de la fonction « asyncHandler » appel la fonction « resWithErr » qui s’occupe des possible erreurs arrivées dans le bloc try. La fonction « resWithErr » vérifie le type de l’erreur reçue et réponds avec le code d’erreur 500 ou bien avec celui fourni par l’erreur si celle-ci est de type « ErrorStatus » qui est un type créé par moi.

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

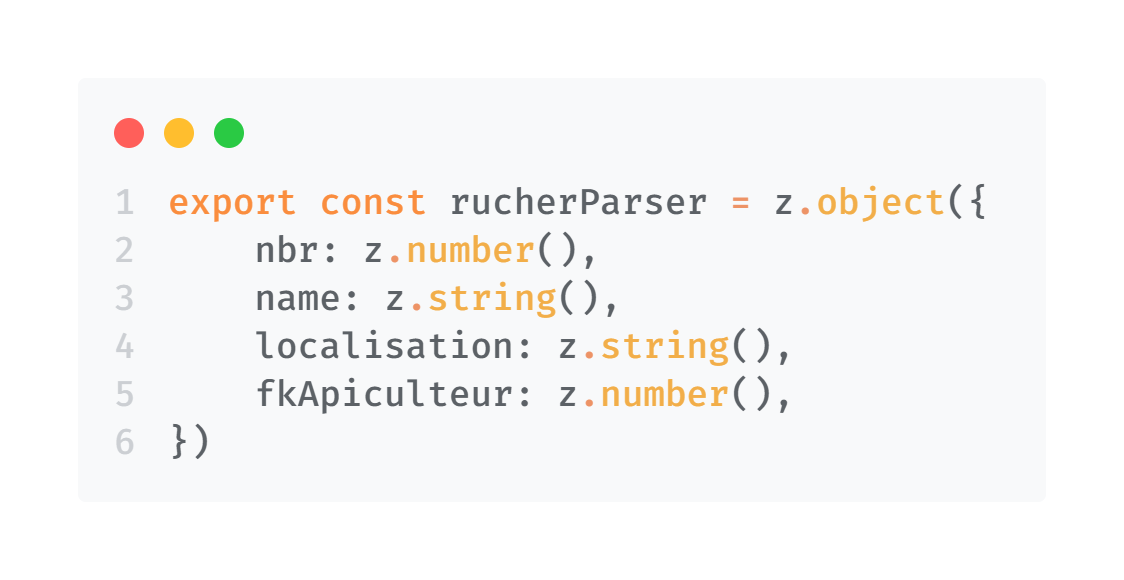


Voyons maintenant le fonctionnement d’un contrôleur. Je vais prendre ici le contrôleur qui s’occupe de gérer la requête pour créer un rucher.



**Valider les données**

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



« rucherParser » est un schémas zod qui est une librairie typescript. Ici on définie le type de ce que l’on souhaite avoir, dans notre cas un objet avec 4 propriétés qui un chacun un type défini. Lorsque la fonction « .parse() » et appelée, l’éléments reçus en paramètre est vérifié par zod. Si l’élément reçu en paramètre ne corresponds par à nôtre schémas alors une erreur et lancé et gérée par « asyncHandler ».

**Préparation des données**

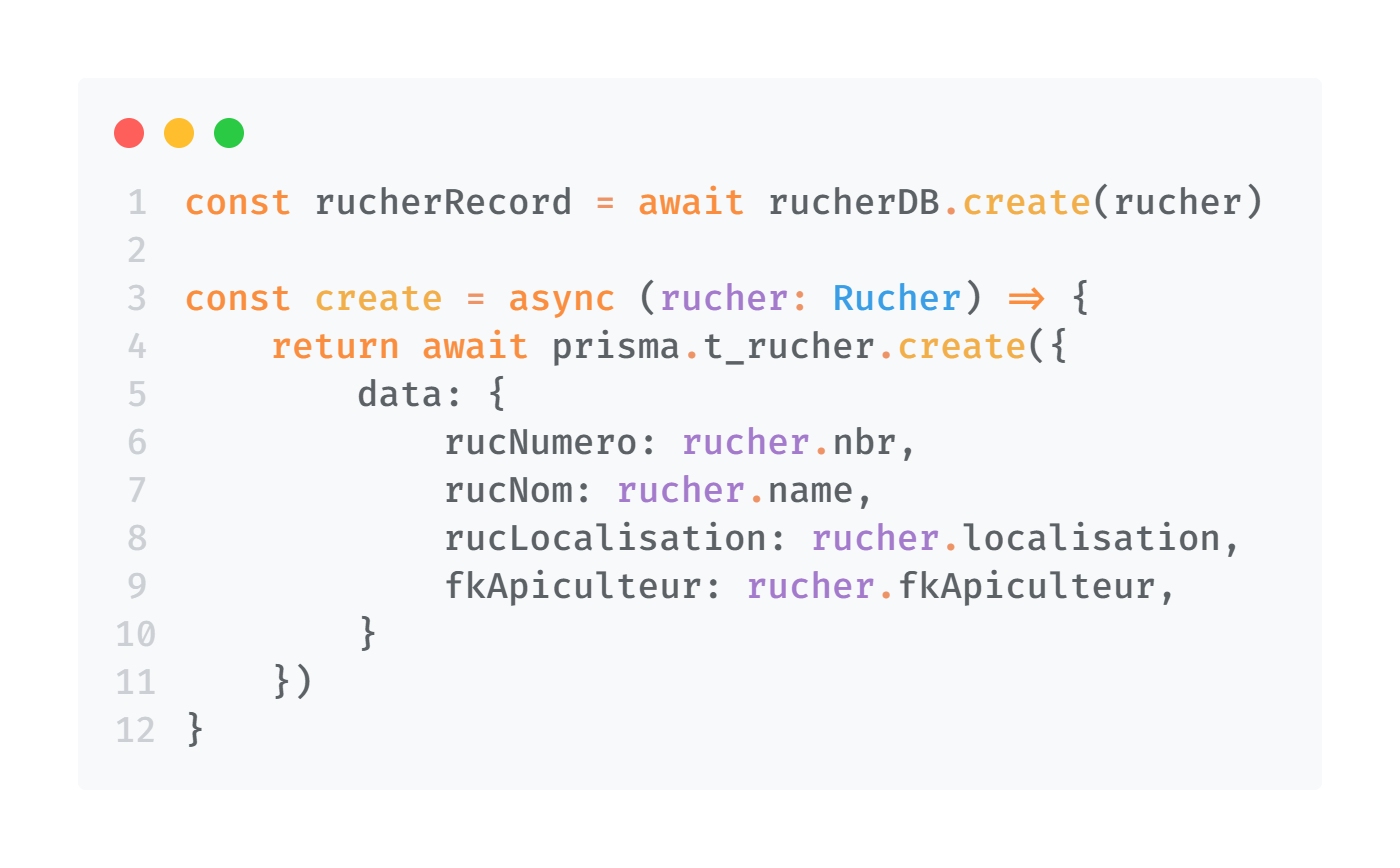
Ici pas de préparation nécessaire. Mais dans le contrôleur qui gère la requête qui permet d’obtenir toutes les activités par années par exemple, une préparation des données est nécessaire pour formatter les dates.



La requête ne contient que l’année 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**

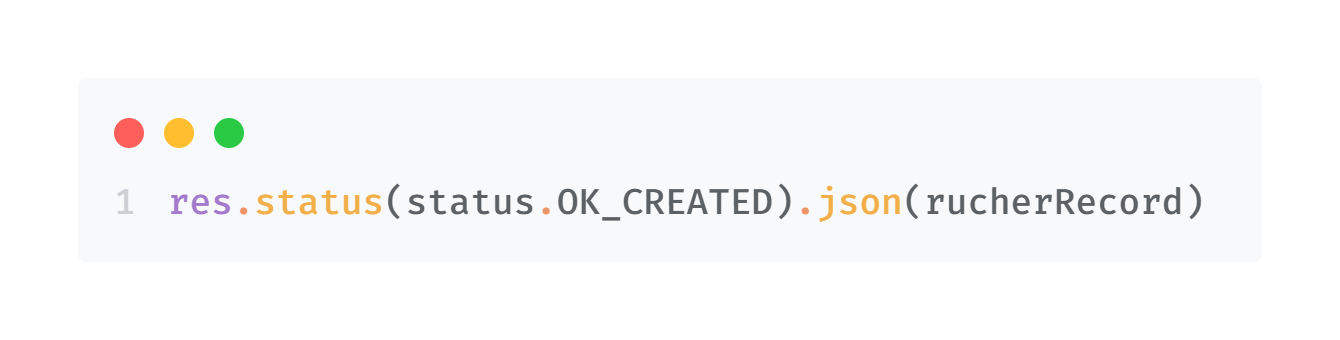
Maintenant que les données sont validées et préparées si nécessaire. On peut effectuer la requête en base de données.



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



« status()» prends en paramètre le code d’erreur à renvoyer et « json() » le json à renvoyer.

Tous les contrôleurs suivent ce même schéma avec un différend validateur, une différente requête Prisma, une différente réponse et possiblement une préparation de données.



J’ai créé un type pour les codes d’erreur http les plus récurant ainsi qu’un objet « status » qui contient tout ces codes d’erreurs, simplifiant l’utilisations de ceux-ci.

### Authentification

Pour s’authentifier l’API fournis un endpoint (**POST** /auth/login) qui permet au client d’envoyer ses identifiants pour recevoir un JWT



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 créer grâce à la librairie « jsonwebtoken »

### VerifyToken

Comme vue plus-haut, certains routeurs sont précédés de « verifyToken » avant d’être ajouter à l’application express.js. Ce « verifyToken » est un middleware qui s’exécute avant le contrôleur.



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 ». Si non 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.

## Frontend

### Requête API



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. « Ref<> » et un type spécifique à vue.js qui représente une fonction qui retourne une variable réactive. Vue.js réagis 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 schémas zod qui permet de validé les données reçue par la requête. Avoir un modèle pour les données reçue permet de connaitre le type de la donnée et d’exploiter pleinement Typescript. « CreateFetchResult<T>() » est un constructeur de « FetchResult<T> ».

Tous les éléments récupérer depuis l’API sont stocké sous forme de « FetchResult<T> » dans un composable et mis à disposition globalement dans toutes l’applications. De cette façons lorsqu’une action est faites par l’utilisateurs sur une page, cette page peut aller mettre à jours les infos dans d’autre 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 dépende.

### Page Login

## Tests

### 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 à été faites en local, donc l’url de base est : [*http://localhost:3000/*](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 | | 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 | | 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 |
| **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 |
| **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 |
| **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 |
| **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 / 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 jours (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 |
| **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 |
| **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 |
| **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 |
| **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 |
| **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 |
| **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 |
| **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 |
| **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 |
| **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 |
| **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 |
| **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 |
| **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 / 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 |
| **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 |
| **JSON** | Rucher supprimée | | Message d’erreur |
| **Résultat** | | | |
| Correspond au résultat attendu | | | |

### Interface

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

**Tests**

|  |  |
| --- | --- |
| **Test N° 1** | |
| **Fonctionnalité** | Authentification |
| **Actions** | Lorsque l’utilisateurs est sur la page de login, il doit renseigner son nom d’utilisateurs ainsi que son mot de passe et se connecter |
| **Résultat attendu** | Il devrait être rediriger 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 rucher/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** | Tout les rucher/ruches devrait être afficher 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 moyens des flèches gauche/droite qui entoure l’années actuelle |
| **Résultat attendu** | Toutes les activités avec l’année correspondante doivent être affiché |
| **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** | Redirections sur la page détails de la ruche ou du rucher, qui contient toutes les informations de la ruche ou du rucher. Toutes les activités liée sont également visible |
| **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 visible 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** | Pas implémenté |

# Conclusion

## État finale de l’application

## Objectifs atteints

## Objectifs non-atteints

## Problèmes rencontrés

## Amélioration possible

## Bilan de la planification

## Bilan personnel

# Sources – Bibliographie

# Glossaire

* **API** (**A**pplication **P**rogramming **I**nterface) : 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 fourni les fonctionnalités nécessaires à son fonctionnement
* **CRUD** (**C**reate **R**ead **U**pdate **D**elete) : L’acronyme CRUD définie les opérations de base sur les données stockées. Créer une donnée, la lire, la mettre à jour où 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 vas interagir pour bénéficier des fonctionnalités disposées par le backend
* **IDE** (**I**ntegrated **D**evelopment **E**nvironment) : un environnement de développement et une combinaison d’outils qui facilite la création de programme (éditeur de texte, débuggeur, etc.)
* **Intégration**: assemblages 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 faites par l’utilisateur au serveur et le traitement final de la requête par le serveur
* **Migrations**: transitions de schéma de base de données. Les base de données doivent changer au fil du temps pour s’adapter aux nouvelles exigences, le changement de schémas (structure de la base de données) est une migration
* **ORM**(**O**bject **R**elational **M**apping) : 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** (**S**ystème de **G**estion de **B**ase de **D**onnées) : logiciel permettant la gestion d’une base de données

# Annexes

# Résume