

Dossier Projet

**Développeur web et web mobile
2022 / 2023**

Araujo Nicolas

Sommaire

Accueil

1 Résumé du dossier projet

2 Compétences couvertes par le projet

3 Cahier des charges

3.1 Front-end

3.2 Back-end

4 Spécification techniques du projet

4.1 Front-end

4.1.1 Maquette Figma

4.1.2 Framework Vue.js

4.2 Back-end

4.2.1 Framework Symfony

4.2.2 Framework ApiPlatform

4.3 Outils Communs aux deux parties

4.3.1 Git

4.3.2 Github

5 Réalisation du projet

5.1 Application Vue.js (Front-end)

5.1.1 Introduction

5.1.2 Maquettage de l'application

5.1.3 Initialisation d'un projet Vue.js

5.1.4 Présentation des stores

5.1.5 Interface d'authentification

5.2 Back-end

5.2.1 Conception base de donnée

5.2.2 Installation de Symfony et remplissage de la base de donnée

5.2.3 Sécurité

5.2.4 API

6 Présentation du jeu d'essai de la fonctionnalité la plus représentative (ajout manga au suivi ?)

7 Veille sur les vulnérabilités de sécurité

8 Situation ayant nécessité une recherche à partir d'un site anglophone

9 Extrait du site anglophone et sa traduction

Résumé du dossier projet

N'ayant pas eu l'opportunité de réaliser un stage, mon dossier projet portera sur un projet entamé avant la période du stage et poursuivie durant celle-ci.

Le but de mon projet est de mettre en place une bibliothèque en ligne dédiée aux mangas / scans, offrant aux visiteurs la possibilité de suivre leur progression (en cours, abandonnée ou terminée), de connaître le nombre de chapitres disponibles, et de maintenir un suivi personnel. Les utilisateurs auront la possibilité de créer un compte pour ajouter les mangas de leur choix à leur liste de suivi, attribuer des notes, ou tout simplement en mettant un 'like'. Les notations et le nombre de 'like' seront également visibles sur la page des mangas, permettant ainsi aux utilisateurs de découvrir les mangas les plus populaires.

La base de données de ce projet est conçue en utilisant MySQL. Le projet se décompose en deux parties distinctes :

- Le back-end, réalisé en PHP à l'aide du framework Symfony, suit le modèle MVC (Modèle-Vue-Contrôleur). Cette partie se consacre à l'API et à l'administration du site, facilitant la gestion des utilisateurs ainsi que des mangas.
- Le front-end, quant à lui, est construit avec le framework Vue.js en tant qu'application monopage (Single Page Application ou SPA). Il correspond à l'interface visible par l'utilisateur.

Pour établir la liaison entre le back-end et le front-end, le framework ApiPlatform a été implémenté, qui fournit une API REST permettant de configurer les différentes URL (endpoints), les données renvoyées, ainsi que mettre des restrictions sur certains accès.

Compétences couvertes par le projet

Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité :

- Maquetter une application
- Réaliser une interface utilisateur web statique et adaptable
- Développer une interface utilisateur web dynamique

Développer la partie back-end d'une application web ou web mobile en intégrant les recommandations de sécurité :

- Développer les composants d'accès aux données
- Développer la partie back-end d'une application web ou web mobile

Cahier des charges

Front-end

- Recréer le design du site avec la maquette.
- Pouvoir filtrer les recherches par titre (barre de recherche) ou par catégorie(s).
- Avoir une interface de connexion et un système d'authentification.
- Avoir une interface permettant à l'utilisateur de modifier ses informations.
- Pouvoir ajouter un manga au suivi, possibilité de le noter, mettre un 'like' et modifier le chapitre ou en est actuellement l'utilisateur.

Back-end

1. Modération

- Accès seulement avec certaines adresses IP
- Avoir une interface de connexion pour vérifier les permissions de l'utilisateur.
- Permet la lecture, la création, la modification et la suppression des données des différentes entités (CRUD).

2. API

- L'API permet la lecture, la création, la modification et la suppression des données de certaines entités.
- L'API doit assurer l'intégrité des données lors des opérations de création ou de modification de celle-ci.
- L'API doit hasher les mots de passe utilisateur pour les sécuriser.
- L'API doit posséder un système d'authentification afin de vérifier l'utilisateur pour les opérations autre que celle de lecture de manga.

Spécifications techniques du projet

Front-end

Maquette: Figma

Figma est un éditeur graphiques et un outil de prototypage, servant également à créer des maquettes. Il a été choisi pour les raisons suivantes:

- Il s'agit d'un logiciel présenté pendant la formation.
- Il est simple à prendre en main.
- Il est possible de simuler des interactions entre les éléments.
- Il possède un outil permettant de voir un prototype de la maquette aidant à constater le résultat.

Framework: Vue.js

Général:

J'ai choisi le framework Vue.js pour les raisons suivantes:

- Il s'agit d'un des framework front-end vus durant la formation, et c'est celui où je suis le plus à l'aise.
- Vue.js offre d'excellentes performances grâce à la légèreté de son paquet et l'utilisation d'un DOM virtuel performant, grâce à quelques optimisations dont: **Static Hoisting, Patch Flags, Tree Flattening**.
- Vue.js dispose d'une architecture en **MVVM** (Model-Vue-Vue Model), constituée de divers composants réutilisables, permettant un développement plus rapide et efficace.
- Vue.js bénéficie d'une documentation détaillée et d'une communauté assez active, ce qui permet de trouver des ressources et des solutions facilement.
- Vue.js offre une utilisation plus simple et une meilleure lisibilité en permettant à chaque modèle d'avoir son propre style, HTML et JavaScript.

Component:

Vue.js utilise un système de composants permettant la séparation du code en modèles réutilisable, chacun contenant trois parties: CSS, HTML, JavaScript.

Un ou plusieurs modèles peuvent être combinés afin de créer un composant qui sera affiché côté client lorsque le routeur appellera celui-ci.

SPA:

Une application en Vue.js peut choisir de disposer d'un routage côté client (SPA: Single Page Application) ou côté serveur (SSR: Server Side Rendering).

Ce projet est une application mono page (Single Page Application); il utilise le routage côté client pour intercepter la navigation, récupérer dynamiquement de nouvelles données et mettre à jour la page actuelle sans la recharger complètement. Cela permet une expérience utilisateur plus rapide dans le cas où l'utilisateur doit effectuer de nombreuses interactions.

Lorsque l'utilisateur entre une URL dans la barre de recherche, celle-ci est interceptée par le router (fichier JavaScript), si elle correspond à l'une des routes disponibles alors le composant correspondant est chargé et affiché.

```
const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [
    {
      path: '/',
      name: 'home',
      component: HomeView
    },
    {
      path: '/manga/:id',
      name: 'mangaInfo',
      component: mangaDetail
    },
  ],
})
```

Capture d'écran d'une partie du router de l'application.

Les routes sont composées d'un chemin d'accès (path), d'un nom (name) et d'un composant qui sera chargé (component).

Pinia:

Vue.js dispose de *Pinia* une librairie permettant d'initialiser des 'stores' (magasins) contenant des propriétés et des fonctions afin de partager leur état (state) à divers endroit dans l'application. Ces 'stores' facilitent la gestion de l'état globale de l'application, ce qui est particulièrement utile pour partager des données entre différents composants de manière efficace et maintenable.

Back-end

Framework Symfony

Symfony est un framework PHP open-source, il dispose de nombreux plugins / librairies, permettant de gagner du temps de développement, de mieux sécuriser l'application. Étant un projet open-source et disposant d'une énorme communauté, Symfony dispose de beaucoup de ressources pour résoudre les problèmes rencontrés ou les soucis de sécurité.

Ce framework a été choisi pour les raisons suivantes:

- Symfony dispose d'easyAdmin permettant de créer une interface de modération modifiable.
- Symfony dispose d'un SecurityBundle, permettant de créer une interface d'authentification, de sécuriser l'application avec des pare-feu (Firewall) et d'ajouter d'autres protections.
- Symfony, étant un projet open-source, ne risque pas de devenir payant, et disposant de certaines versions en support à long terme, permet une certaine stabilité pour le développement de sites internet.

Symfony utilise le package Doctrine pour gérer la relation entre les entités et la base de donnée. Grâce à diverses commandes et bundles, Doctrine permet de faciliter les actions suivantes:

- La connexion à la base de donnée.
- Les requêtes à la base de données, les changements d'architecture sont faits avec migrations permettant de stocker les divers changements.
- Il utilise le DQL (Doctrine Query Language), qui est une version plus simple et plus lisible du langage SQL (Structured Query Language) permettant de communiquer avec la base de donnée afin de récupérer ou modifier des données.

ApiPlatform

Le framework ApiPlatform permet de générer rapidement une API REST modifiable en fonction des besoins pour les différentes entités. Il dispose de plusieurs avantages notamment:

- La facilité de mise en place. Une ligne dans chaque entité suffit pour créer 6 endpoints pour celle-ci (un pour chaque opération).
- La possibilité de créer des groupes de propriétés et de les attribuer aux endpoints afin de n'afficher que les données pertinentes pour certaines opérations.
- Une sécurité peut être ajoutée pour limiter l'accès à certaines opérations.

L'API utilise les JWT (JSON Web Token) pour sécuriser certaines routes et certaines opérations. Ils permettent d'identifier l'utilisateur en cours afin de renvoyer, modifier ou supprimer seulement les données liée à l'utilisateur courant.

Outil communs aux deux parties

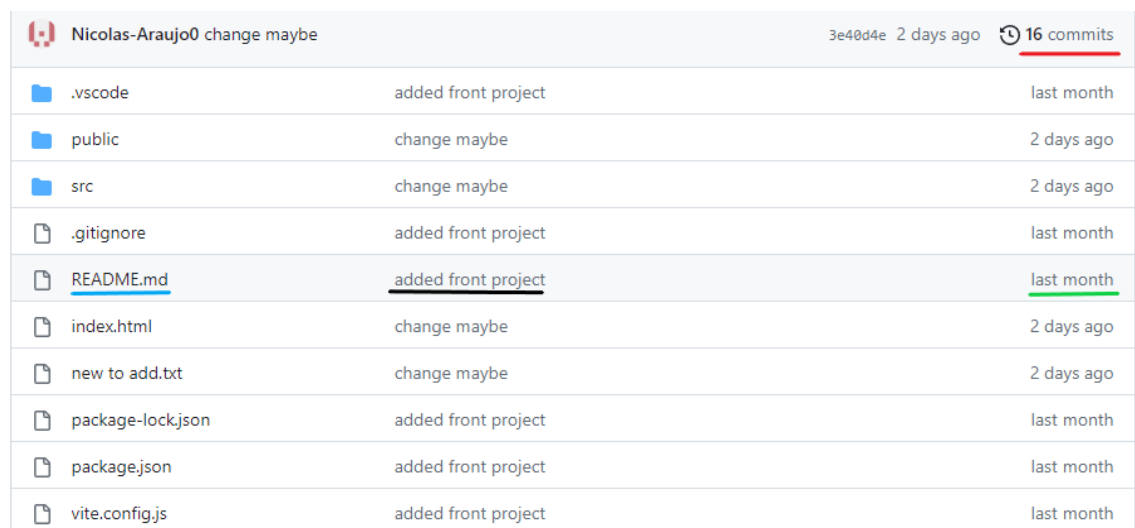
Git













Git est un logiciel de versions décentralisée. Un outil de versionnage comme celui-ci propose de nombreux avantages lors du développement d'un projet:

- La possibilité de créer une branche indépendante de la principale permettant d'implémenter des fonctionnalités et ainsi d'éviter de créer des conflits sur la principale en cas d'erreur ou d'abandon de celle-ci.
- Plusieurs personnes peuvent travailler sur des branches différentes et tout fusionner(merge) ensuite pour mettre en commun leur travail.
- La possibilité d'utiliser les versions antérieures de manière simple en cas de soucis.

Github

Github est une plateforme d'hébergement de projets et de code pour la gestion de version et la collaboration. La création d'un dépôt sur Github permet de conserver différentes versions d'un projet sur internet, facilitant le transfert entre différents postes de travail ou différentes personnes.



 Nicolas-Araujo0 change maybe	3e40d4e 2 days ago	 16 commits
 .vscode	added front project	last month
 public	change maybe	2 days ago
 src	change maybe	2 days ago
 .gitignore	added front project	last month
 <u>README.md</u>	<u>added front project</u>	<u>last month</u>
 index.html	change maybe	2 days ago
 new to add.txt	change maybe	2 days ago
 package-lock.json	added front project	last month
 package.json	added front project	last month
 vite.config.js	added front project	last month

Capture d'écran du répertoire github hébergeant le projet

Il s'agit du dépôt contenant la partie front-end de mon projet. Soulignées en différentes couleurs sont les différentes parties du dépôt: en bleu, le nom du fichier / dossier; en noir, le nom; en vert, la date de la dernière version ayant affecté l'élément; et en rouge, le nombre total de versions pour le dépôt (une liste des versions est accessible en cliquant dessus).

Réalisation du projet

Application Vue.js

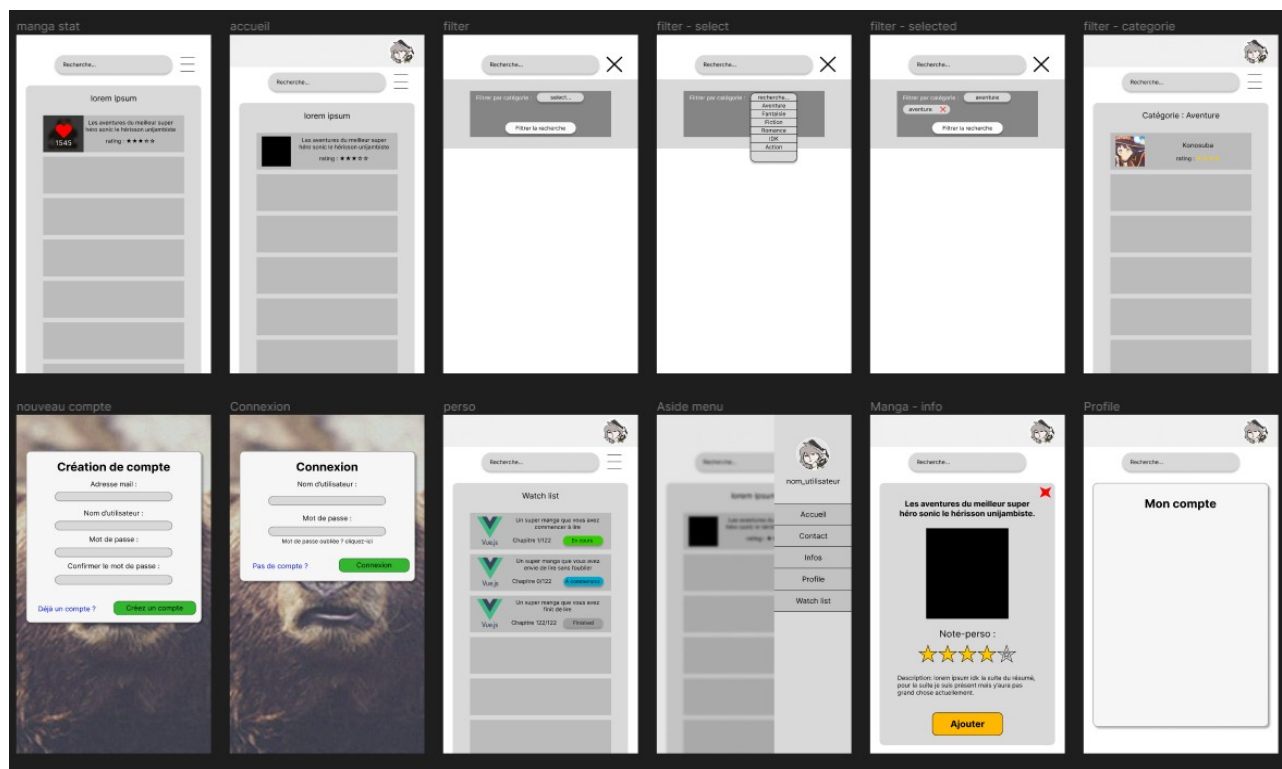
Introduction:

Avant de commencer à coder le projet, il est important de fixer les objectifs, le design et les restrictions du projet. Cela permet par la suite de ne devoir que coder les éléments sans avoir à réfléchir à ces diverses problématiques.

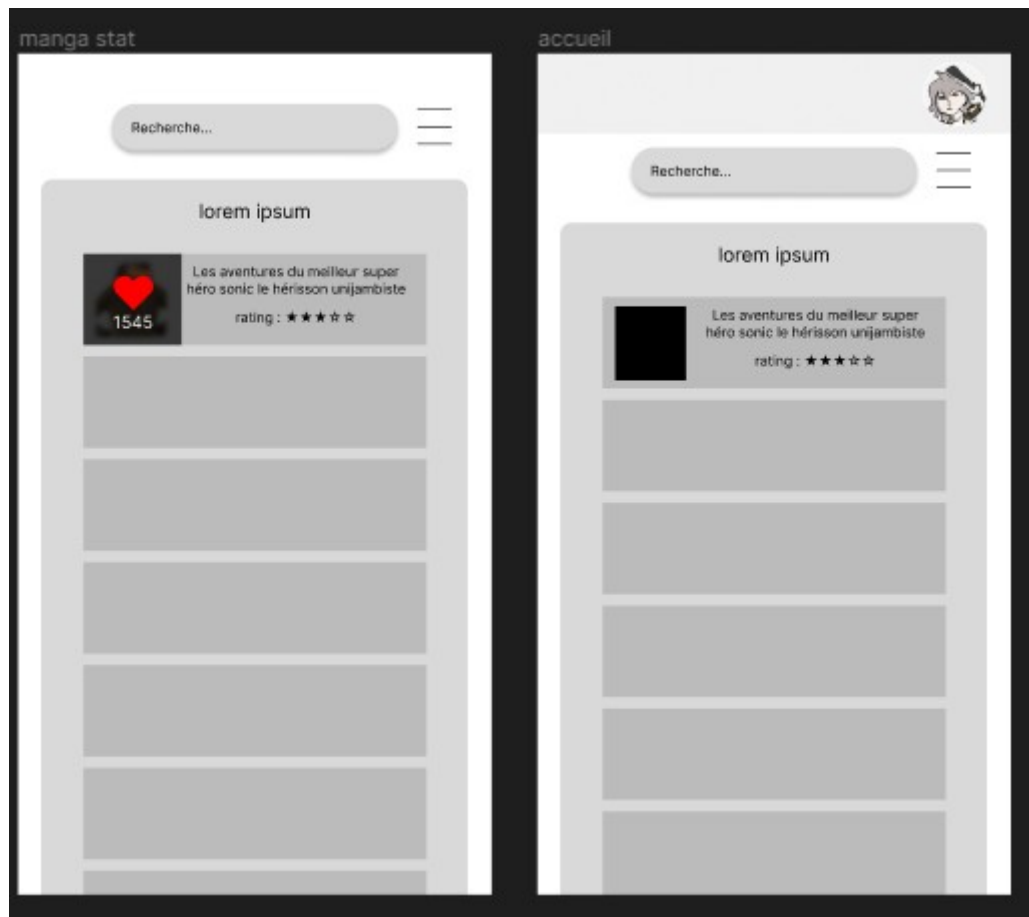
j'ai commencé par faire un cahier des charges de l'application afin de noter tous les besoins de celle-ci. Cela m'a permis de connaître les différentes pages, fonctionnalités et restrictions que j'aurais besoin de créer.

Maquette de l'application:

Ensuite, j'ai utilisé Figma pour réaliser la maquette de l'application en prenant en compte les besoins du cahier des charges.

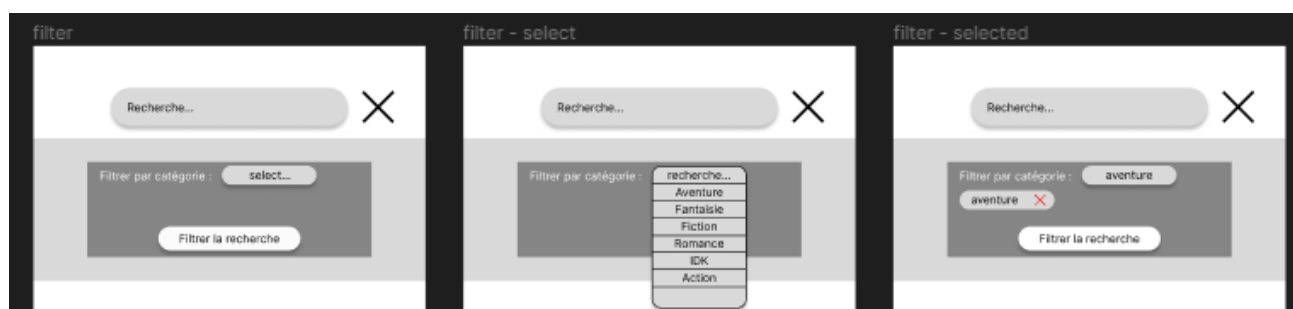


Différentes maquettes représentant le design mobile des différentes pages de l'application.



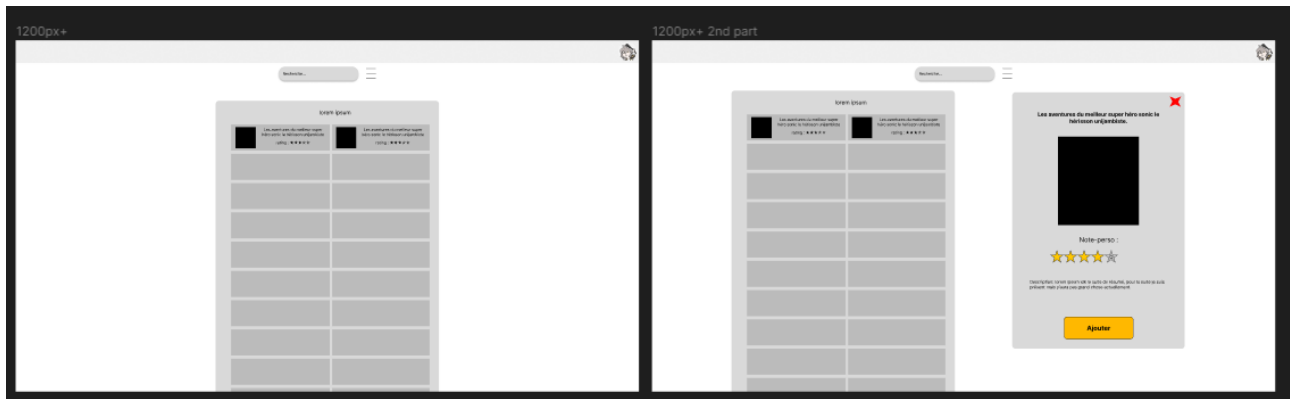
Capture d'écran des maquettes montrant les états de la page d'accueil

Ces deux maquettes représentent la page d'accueil de l'application. On peut y voir une liste qui contient les mangas récupérés avec l'API, quelques informations les concernant, telles que le nom, l'image et la moyenne des notes actuelles. Pour éviter une barre de défilement trop longue, une pagination sera présente en bas de la liste.



Capture d'écran montrant les différents états du menu permettant de filtrer les mangas par catégories.

Pour le responsive de l'application il n'y a eu aucun changement dans le design, à l'exception de la page d'accueil (celle affichant la liste des mangas).



Maquettes montrant les différents états de la page d'accueil sur pc

Le nombre de mangas affichés est doublé, et il est possible de voir les informations d'un manga sur la même page (en mobile, une redirection est faite).

Initialisation d'un projet Vue.js:

Pour créer une application Vue.js, il est nécessaire d'avoir Node.js installé, ainsi que NPM (gestionnaire de paquet JavaScript).

Ensuite, en utilisant la commande '`npm create vue@latest`' dans un invité de commande, un projet Vue.js sera créé. Plusieurs questions apparaîtront permettant d'ajouter un certain nombre de fonctionnalités optionnelles.

```
✓ Project name: ... <your-project-name>
✓ Add TypeScript? ... No / Yes
✓ Add JSX Support? ... No / Yes
✓ Add Vue Router for Single Page Application development? ... No / Yes
✓ Add Pinia for state management? ... No / Yes
✓ Add Vitest for Unit testing? ... No / Yes
✓ Add an End-to-End Testing Solution? ... No / Cypress / Playwright
✓ Add ESLint for code quality? ... No / Yes
✓ Add Prettier for code formatting? ... No / Yes

Scaffolding project in ./<your-project-name>...
Done.
```

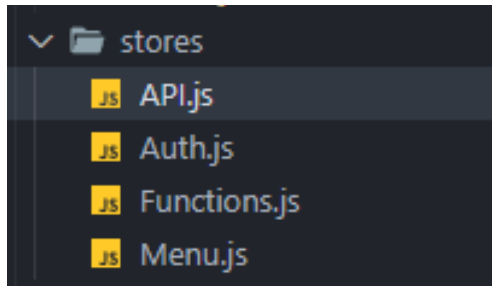
Maintenant, il suffit d'aller dans le dossier créé, et d'utiliser les commandes suivantes:

1. '`npm install`', cela va permettre l'installation des différents paquets que l'application a besoin. Ils sont visibles dans le fichier 'package.json'.
2. '`npm run dev`' va permettre de lancer l'application sur un serveur local en utilisant le port 5173 par défaut.

Lorsque le projet est prêt à être lancé en production, il suffit de faire la commande '`npm run build`', pour créer une version de votre application prête pour la production dans un fichier 'dist' du projet.

Stores:

Dans mon projet, des stores ont été utilisés afin de pouvoir accéder a des propriétés et des méthodes dans différentes parties de mon application.



Capture d'écran du dossier stores de mon application Vue.js

Mon projet possède 4 stores, l'un gérant les fonctionnalités liées à l'API, un autre concernant l'authentification, un troisième avec des fonctions réutilisables et le dernier gérant l'affichage de mon menu.

Lorsque que les propriétés appartiennent au store en cours d'utilisation, '**this.**' doit être ajouté afin de le spécifier.

Exemple:

```
import { defineStore } from 'pinia'
import { useAPIStore } from '../API'
import router from '../router'

export const useAuthStore = defineStore('auth', {
  state: () => ({
    refreshToken: "",
    errorMessage: "",
    currentUser: {},
    isAuthenticated: false,
  }),
  actions: {
    async login(emailInput, passwordInput) { ...
    },
    async createUser(emailInput, passInput, repeatPassInput,
      usernameInput) { ...
    },
    updateInfo() { ...
    },
    storeUserLocal(data) { ...
    },
    logout() { ...
    }
  }
})
```

Capture d'écran du store gérant l'authentification séparé en plusieurs blocs.

Pour mieux présenter l'architecture d'un store, je l'ai séparée en plusieurs blocs:

- Dans le bloc rouge, on peut y voir les fichiers qui seront importés pour être utilisés. 'defineStore' provient de Pinia et permet la création du store, 'useAPIStore' permet d'accéder aux propriétés et fonctions du store gérant l'API, et 'router' permet d'accéder aux méthodes du router, ici utilisé pour la redirection.
- Le bloc orange correspond au store en lui même, qui sera exporté, avec le nom 'useAuthStore', 'defineStore' est la méthode créant un store. Elle prend 2 paramètres, l'un correspondant au nom du store, ici 'auth' (il est affiché en utilisant l'outil développeur de Vue.js.).
- Le bloc jaune représente 'state' (état) représente un objet composé d'une ou plusieurs propriétés. Ici, 4 propriétés sont stockées: 'refreshToken' et 'errorMessage' sont des chaînes de caractères, 'currentUser' est un objet vide et isAuth est un booléen.
- Le bloc bleu contient un objet 'actions' contenant les méthodes qui seront propres à ce store. Ici 'login' et 'createUser' sont des méthodes asynchrone (l'exécution de la fonction ne bloque pas la lecture du code et retourne une promesse), updateInfo, storeUserLocal et logout sont des méthodes normales.

Interface authentication:

Pour présenter la façon dont fonctionne l'interface d'authentification, je vais la diviser en deux parties: la partie visuelle montrant ce que voit l'utilisateur, et la partie code avec des captures d'écran des fichiers VScode concernés.

Côté code:

L'architecture d'une application en Vue.js est répartie en **components** (composant), représentant une fonctionnalité ou un ensemble (ex: un menu) et les **views** (vues) sont des **components** utilisée par le router afin de gérer les redirections.

```
<script setup>...  
</script>  
  
<template>  
  <Menu />  
  <SearchFilter />  
  <operationMessage />  
  <RouterView />  
</template>  
  
<style scoped>  
</style>
```

*Capture d'écran représentant l'architecture d'un **composant** en Vue.js celui-ci ayant plusieurs **composants** à l'intérieur.*

```
<script setup>
import { useAuthStore } from '../stores/Auth'
import { ref } from 'vue';
import { useFunctionsStore } from '../stores/Functions';
const pass = ref('');
const confirmPass = ref('');
const show = ref(false);
const store = useAuthStore()

async function request() {
  const email = document.querySelector('#mail')
  const pass = document.querySelector('#pass')
  const confirmPass = document.querySelector('#repeatPass')
  const username = document.querySelector('#username')
  store.createUser(email, pass, confirmPass, username)
}

const funcsStore = useFunctionsStore();
const debouncePass = funcsStore.debounce(() => {
  show.value = pass.value !== confirmPass.value && confirmPass.value !== ''
});

</script>
```

Ce bloc est englobé dans une balise `<script>`, indiquant au navigateur qu'il s'agit de code exécutable, et ici, c'est du JavaScript. Plus particulièrement, il s'agit d'une balise `<script>` ayant l'attribut `setup`. Celle-ci est utilisée avec Vue.js pour plusieurs raisons:

- Le code à l'intérieur de la balise sera compilé à l'intérieur de la fonction **setup** du composant, permettant l'exécution du code à chaque fois que le composant est appelé, au lieu d'être exécuté seulement la première fois.
- Il est possible d'utiliser les variables et les fonctions déclarées dans la balise à l'intérieur du **template** (le code HTML qui sera affiché). Cela permet d'afficher directement la valeur d'une variable dans le rendu, ou de lier une fonction à un événement (au lieu de devoir créer un `addEventListener`).
- Utilisation de **Props** et d'**Emit** permet de communiquer des informations du parent à l'enfant et inversement.

On peut voir à l'intérieur de celle-ci des méthodes qui sont importées de certains fichiers. Plusieurs variables et constantes sont déclarées, ainsi des fonctions.

Les constantes dont la valeur est égale à `ref(...)` permettent à Vue.js de créer un objet ayant une valeur réactive. Pour accéder à celle-ci pour la constante `pass` ci-dessus, il faudra faire **pass.value**. La constante `store` ayant la valeur de `useAuthStore()` permet maintenant d'accéder aux propriétés et méthodes de celle-ci. La constante `debouncePass` a pour valeur la fonction `debounce` (voir page suivante) provenant de `funcsStore`, le store gérant certaines fonctions. Cette fonction prend en paramètres une autre fonction et le délai du `debounce`. Si ce délai n'est pas fourni, il aura comme valeur par défaut 500ms.

```
debounce(func, timeout = 500) {
  let timer;
  return (...args) => {
    clearTimeout(timer);
    timer = setTimeout(() => { func.apply(this,
      args); }, timeout);
  };
},
```

*Fonction debounce provenant du **store** gérant les fonctions*

Ici, la fonction est anonyme et va simplement modifier la valeur de 'show' en fonction du fait que les mots de passe correspondent et que la confirmation du mot de passe n'est pas vide. Chaque fois que la fonction va être appelée, un délai de 500ms est lancé. Si elle est appelée avant la fin du délai, celui-ci sera réinitialisé. Cela permet à l'utilisateur de taper la confirmation du mot de passe avant de faire la vérification pour voir s'ils correspondent, car il est logique qu'ils soient différents si il n'a pas fini de l'écrire.

```
<input type="password" name="repeatPass" id="repeatPass" placeholder="Confirmer le mot de passe"
  pattern="^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[a-zA-Z\d]{8,}$" v-model="confirmPass" @keyup="debouncePass">
```

Un input dans la partie template du composant

La fonction est appelée lorsque l'utilisateur relâche une touche lorsqu'il est sur l'input du mot de passe ou de la confirmation de celui-ci.

La fonction asynchrone 'request', sélectionne les 4 champs du formulaire afin de les passer en paramètres à la fonction 'createUser' provenant de 'useAuthStore', utilisable grâce à la constante 'store' créée ci-dessus.

Explication de la fonction createUser:

Pour plus de lisibilité le code sera fracturé en morceaux et expliqué, une capture d'écran avec la fonction entière est disponible à la fin de ce segment.

```
async createUser(emailInput, passInput, repeatPassInput, usernameInput) {
  const email = emailInput.value;
  const pass = passInput.value;
  const confirmPass = repeatPassInput.value;
  const username = usernameInput.value;
  const strongPassword = pass.match(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[a-zA-Z\d]{8,}$)/u);
```

*Première partie de la fonction **createUser**.*

Les 4 champs du formulaires sont récupérés grâce aux paramètres de la fonction, et leur valeur est stockée dans une constante (email, pass, confirmPass, username).

Pour des raisons de sécurité, la fonction 'match(regex)' est utilisée sur le mot de passe. Cela permet de vérifier si le mot de passe suit bien le pattern du regex (expression régulière). Celui utilisé ici demande à l'utilisateur d'avoir 8 caractères, dont au moins une minuscule, une majuscule et un chiffre. Cela permet d'assurer une meilleure protection du compte.

Ensuite, les champs sont vérifiés si certains sont vides. La propriété `errorMessage` va prendre une valeur et afficher le message d'erreur pour l'utilisateur. Dans ce cas, il sera indiqué que tous les champs doivent être remplis.

Si tout les champs sont remplis et que le mot de passe valide le pattern, la requête est préparée avec la constante `requestPost`. Celle-ci contient un objet avec 3 propriétés:

1. **method:** Il s'agit de l'opération qui va être effectuée, ici `POST`, car on veut insérer des données dans la base de données.
2. **headers:** Cela permet de spécifier des informations sur les données envoyées. Ici, on précise que le contenu est de type `application/json`.
3. **body:** Il correspond au contenu de la requête, contenant l'e-mail, le mot de passe et le nom d'utilisateur qui sont dans un objet, puis transformée en **JSON** grâce à la méthode `JSON.stringify()`.

```
if (email.length > 0 && username.length > 0 && (pass.length > 0 && pass === confirmPass) && strongPassword !== null) {  
  const requestPost = {  
    method: "POST",  
    headers: { "Content-Type": "application/json" },  
    body: JSON.stringify({  
      email: email,  
      plainPassword: pass,  
      username: username  
    })  
  }  
}
```

Seconde partie de la fonction `createUser`

Par la suite, `await` est utilisé pour demander l'arrêt de la lecture de la fonction `createUser` tant que la fonction `fetch` n'aura pas retourné une promesse. La fonction `fetch` prend en compte deux paramètres, le premier étant l'URL à contacter pour la requête, et le second, optionnel, permet de spécifier des informations. Dans ce contexte, on lui donne la constante `requestPost` définie ci-dessus.

```
await fetch('http://127.0.0.1:8000/api/user/add', requestPost)
```

La méthode `.then` est utilisée pour gérer la réponse de la promesse de la fonction `fetch`. Une fonction fléchée est utilisée, prenant en paramètre `response`. La constante `data` est créée et aura pour valeur `response` une fois que celle-ci sera analysée (parse) au format JSON.

```
.then(async response => {  
  const data = await response.json();  
  // ...  
})
```

Une condition vérifie le code HTTP renvoyé par la requête est OK (status 200 à 299). Si le statut est différent de OK, alors la constante `error` est initialisée et va essayer d'extraire le message d'erreur de `response`. Si la réponse contient une propriété `message`, elle sera utilisée, sinon, elle utilisera la propriété `statusText` par défaut.

La promesse va ensuite être rejetée en passant `error` comme motif. Cela permettra à la méthode `catch()` ci-dessous de capturer l'erreur et de changer la valeur de `errorMessage`, qui sera ensuite l'affichée à l'utilisateur.

```
if (!response.ok) {
  // get error message from body or default to response.statusText
  const error = (data && data.message) || response.statusText;
  return Promise.reject(error);
}
```

Cinquième partie de la fonction *createUser*

Si le code renvoyé est OK, alors l'utilisateur a bien été créé, et ne rentrant pas dans la condition d'erreur, la suite du code est exécutée.

Sixième partie de la fonction *createUser*

```
this.refreshToken = data['refreshToken']
this.isAuth = true;
const store = useAPIStore();
store.token = data['token'];
this.storeUserLocal(this.parseJwt(data['token']))
router.push('/');
```

L'utilisateur va être connecté directement, donc la propriété *isAuth* prendra la valeur *true*, *refreshToken* prendra la valeur de la clé *refreshToken* de *data*.

Une nouvelle constante *store* est créée, ayant pour valeur le store gérant l'API, afin de pouvoir utiliser ses propriétés et donner une valeur à la propriété *token* lui appartenant, elle prendra la valeur de la clé *token* de *data*. La clé *token* de *data* va aussi être **parse** (analysée) afin de récupérer les informations concernant l'utilisateur et les stocker dans une session.

Ensuite, l'utilisateur est redirigé vers la route */* correspondant à la page d'accueil.

```
async createUser(emailInput, passInput, repeatPassInput, usernameInput) {
  const email = emailInput.value;
  const pass = passInput.value;
  const confirmPass = repeatPassInput.value;
  const username = usernameInput.value;
  const strongPassword = pass.match(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[a-zA-Z\d]{8,}$)/u);
  if (email.length > 0 && username.length > 0 && (pass.length > 0 && pass === confirmPass) && strongPassword !== null) {
    const requestPost = {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({
        email: email,
        plainPassword: pass,
        username: username
      })
    }
    await fetch('http://127.0.0.1:8000/api/user/add', requestPost)
      .then(async response => {
        const data = await response.json();
        if (!response.ok) {
          // get error message from body or default to response.statusText
          const error = (data && data.message) || response.statusText;
          return Promise.reject(error);
        }
        this.refreshToken = data['refreshToken']
        this.isAuth = true;
        const store = useAPIStore();
        store.token = data['token'];
        this.storeUserLocal(this.parseJwt(data['token']))
        router.push('/');
      })
      .catch(error => { ...
    })
  } else {
    this.errorMessage = "Tout les champs doivent êtres remplis."
  }
},
updateToken() {
```

Capture d'écran de la fonction *createUser* au complet.

Template:

La balise ‘*template*’, basée sur la syntaxe du HTML, permet de lier directement des données au DOM. Il s’agit du contenu qui sera affichée à l’utilisateur.

```
<template>
  <h1>Création de compte</h1>
  <form @submit.prevent="request">
    <div>
      <label for="mail">Adresse mail</label>
      <input type="email" name="mail" id="mail" pattern="^[a-zA-Z0-9_\-\.]+\@([a-zA-Z0-9_\-\.]+\.)\.[a-zA-Z]{2,5}$">
    </div>
    <div>
      <label for="username">Nom d'utilisateur</label>
      <input type="text" name="username" id="username" pattern="^[a-zA-Z0-9]{4,20}$">
    </div>
    <div>
      <label for="pass">Mot de passe</label>
      <input type="password" name="pass" id="pass" pattern="^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[a-zA-Z\d]{8,}$"
        v-model="pass" @keyup="debouncePass">
    </div>
    <div>
      <input type="password" name="repeatPass" id="repeatPass" placeholder="Confirmer le mot de passe"
        pattern="^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[a-zA-Z\d]{8,}$" v-model="confirmPass" @keyup="debouncePass">
    </div>
    <span v-if="show">
      Les deux mots de passe doivent correspondre
    </span>
    <div class="flex">
      <RouterLink to="login">Déjà un compte ?</RouterLink>
      <input type="submit" value="Créer un compte">
    </div>
  </form>
</template>
```

Capture d’écran du template provenant du composant FormNewUser.vue

Dans l’image ci-dessus on peut voir plusieurs balises:

- Un **<h1>** qui représente le titre principal de la page, ici ‘Création de compte’.
- Un **<form>** représentant un formulaire, possédant un attribut provenant de Vue.js, ‘@submit.prevent’ ayant pour valeur ‘request’. Cet attribut va empêcher la validation normale du formulaire et déclencher la fonction ‘request’ à la place, qui va gérer les données et leur envoi.
- Des **<div>** permettant de regrouper des éléments ensemble, servant principalement en tant que conteneur pour d’autres balises.
- Un **<label>** est liée à un **input** et permet d’associer du texte à un **input** afin de donner plus d’information à l’utilisateur. L’attribut ‘for’ est utilisé pour spécifier l’**input** auquel il est lié.
- Des **<input>** créant un espace où l’utilisateur peut entrer des données. Plusieurs attributs sont disponibles:
 - ➔ L’attribut **type** pour modifier leur fonctionnement (email, text, password, number, ...)
 - ➔ L’attribut **name** servant de ‘clé’ pour récupérer la valeur envoyée.

- L'attribut **id** permet de cibler un élément spécifique, afin modifier son comportement ou son style. Il doit avoir une valeur unique.
- L'attribut **pattern** permet de spécifier un **regex** (expression régulière), un pattern à respecter pour que l'**input** soit considéré valide et que l'envoi du formulaire soit valide.
- L'attribut **v-model** est intrinsèque à Vue.js, il permet de relier une variable à un input.
- **** représente un conteneur générique de ligne. Dans l'image ci-dessus, il possède un attribut de Vue.js '**v-if**' ayant pour valeur '*show*'. Cela permet d'afficher l'élément dans le DOM si la condition vaut **true**.
- **<RouterLink>** provient de Vue.js. Il possède un attribut '*to*' ayant pour valeur '*login*' correspondant à la propriété **name** de l'une des route disponibles dans le **router**. Il représente un **<a>** dans le langage HTML classique.

Style:

Il s'agit de ligne de code permettant de modifier l'affichage des éléments. Il est possible de modifier la taille, la couleur, la police, l'ombrage, et bien d'autres paramètres.

Pour que les changements s'appliquent à une balise, il suffit d'utiliser son nom. Afin d'affecter un élément en particulier et non toutes les balises portant le même nom, il est important d'ajouter à la balise un attribut **class** ou **id** ayant pour valeur ce qui représente l'élément, et de l'utiliser pour sélectionner celui-ci. À noter que l'attribut **class** peut avoir la même valeur sur plusieurs balises différentes, alors que **id** doit posséder une valeur unique.

Le langage CSS se lit de haut en bas et utilise un système de spécificité (**Specificity**) calculant la précision du sélecteur. S'il y a un conflit (différentes valeurs pour la même propriété) entre 2 règles, celle définit le plus bas écrasera l'autre à moins que le sélecteur ne soit plus spécifique.

Selector	Specificity Value	Calculation
p	1	1
p.test	11	1 + 10
p#demo	101	1 + 100
<p style="color: pink;">	1000	1000
#demo	100	100
.test	10	10
p.test1.test2	21	1 + 10 + 10
#navbar p#demo	201	100 + 1 + 100
*	0	0 (the universal selector is ignored)

Tableau récupéré sur la site w3schools représentant le poids des différents sélecteurs. Lien: https://www.w3schools.com/css/css_specificity.asp

Avec Vue.js, il est possible d'ajouter l'attribut **scoped** à la balise **style**. Cela va permettre d'encapsuler le style dans le composant où il est écrit en ajoutant l'attribut **data-v** ayant pour valeur une série de lettres et de chiffres à chaque sélecteur présent dans la balise.

Côté visuel:

Page de création de compte de l'application.

Le formulaire est composé de 4 champs: adresse e-mail, nom d'utilisateur, mot de passe, et la confirmation du mot de passe.

Il est important de faire confirmer le mot de passe par l'utilisateur afin de s'assurer que celui-ci est écrit correctement, sans erreur de typographie, pour qu'il puisse se connecter sans erreur par la suite.

Lorsque l'information entrée dans l'un des champs ne correspond pas au format attendu, la couleur de fond change pour une couleur 'rosée' pour l'indiquer à l'utilisateur.

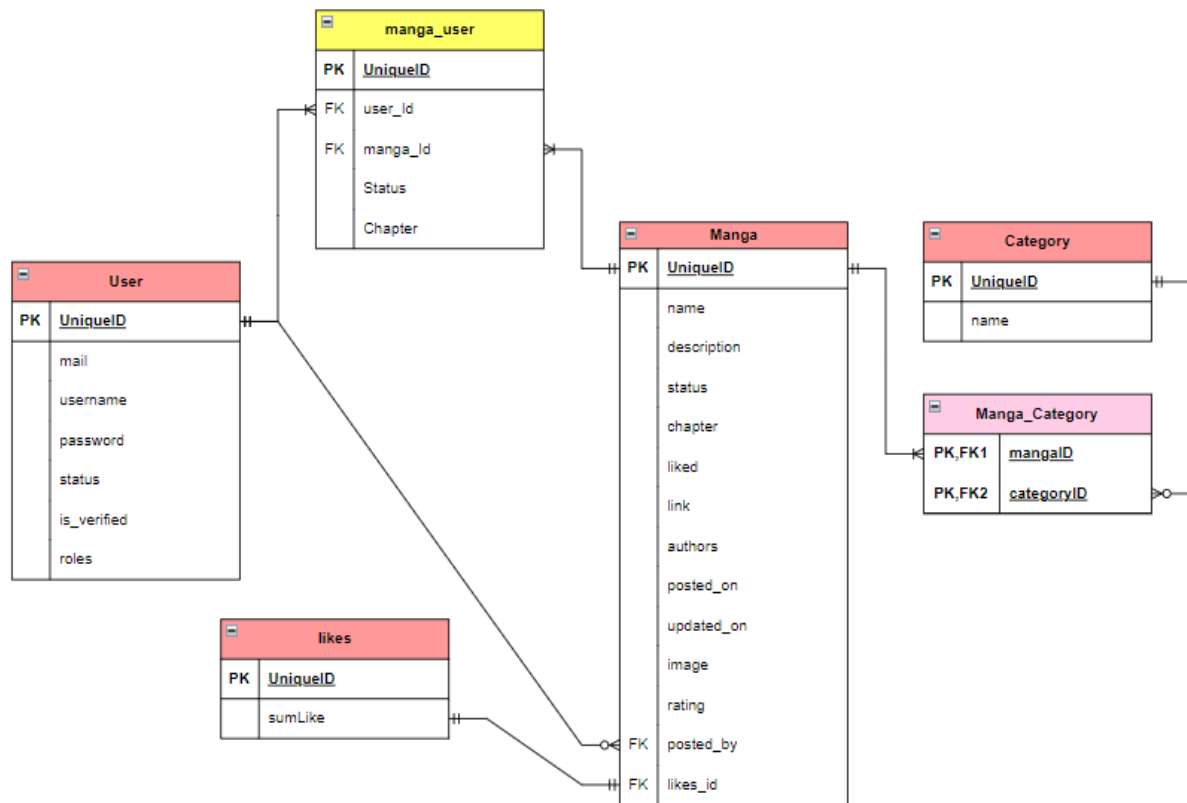
Un message apparaît également lorsque les 2 mots de passe rentrés ne sont pas identiques.

Back-end

Conception la base de donnée

Avant de créer ma base de donnée, j'ai utilisé le logiciel de dessin graphiques **draw.io**, qui peut être utilisé pour créer des diagrammes. Il m'a permis de représenter la base de données et m'a aidé à penser aux entités, à leurs champs, à leurs propriétés et aux relations les reliant.

Deux tables seront utilisées pour développer leurs champs et leurs relations.



*Image: diagramme représentant ma base de donnée faite sur **draw.io***

La table 'User' correspond aux utilisateurs qui seront enregistrés et contient les champs suivants:

- **mail** (adresse mail) sera utilisée pour identifier l'utilisateur et vérifier leur compte. Elle doit être unique.
- **username** (nom d'utilisateur) sera affiché côté client. Il doit être unique
- **password** (mot de passe) permettra à l'utilisateur de s'authentifier. La version enregistrée est hachée par mesure de sécurité.
- **Status** permet de savoir si le compte de l'utilisateur est actif ou non.
- **is_verified** (est vérifié) permet de savoir si l'e-mail de l'utilisateur a été vérifié.

- **roles** permet de définir les rôles disponibles (admin ou utilisateur).

La table '*manga_user*' correspond aux mangas que l'utilisateur ('*user*') a décidé de suivre. Elle possède des clés étrangères (champs uniques permettant de faire référence à une entité d'une autre table, en général l'**id**).

Cette table contient les champs suivants:

- **user_id** est une clé étrangère de la table '*user*'.
- **manga_id** est une clé étrangère de la table '*manga*'.
- **status** correspond à l'état de la lecture (en cours ou finie).
- **chapter** (chapitre) correspond au chapitre où l'utilisateur s'est arrêté dans la lecture.

Pour les relations de cette table, on peut voir que:

- Un **user** peut être associé à plusieurs instances de **manga_user**. Un utilisateur peut avoir plusieurs suivis de mangas, mais un suivi n'appartient qu'à un seul utilisateur.
- Un **manga** peut être associée à plusieurs instances de **manga_user**. Un manga peut être dans plusieurs suivis, mais un suivi ne peut avoir qu'un seul manga.

Installation de Symfony et remplissage de la base de donnée:

La base de donnée a été créée avec **phpMyAdmin**, une application web de gestion pour les systèmes de gestion de bases de donnée en **MySQL** et **MariaDB**.

Lien vers phpMyAdmin: <https://www.phpmyadmin.net/>

Avant de créer une application avec Symfony, il faut installer une version de **PHP 8.1** ou supérieure, ainsi que **Composer** (un gestionnaire de dépendance pour **PHP**).

Lien vers Composer: <https://getcomposer.org/download/>.

Scoop est un installateur en ligne de commande pour Windows. Il permet d'installer **Symfony-CLI**, un outil binaire qui permet d'accéder à des outils pour les développeur et gérer des applications **Symfony** en local. Pour ce faire, vous pouvez exécuter la commande: '*iwr -useb get.scoop.sh | iex*' dans une fenêtre **PowerShell** (invité de commande windows).

Lien vers Scoop: <https://github.com/ScoopInstaller/Scoop#readme>

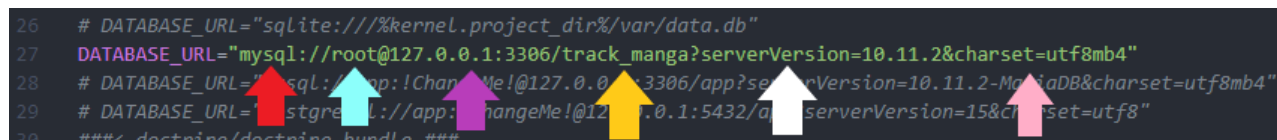
Une fois **Scoop** installé, il vous suffit de saisir la commande '*scoop install symfony-cli*' pour installer Symfony-Cli. Cela vous permettra de construire, créer et gérer vos applications Symfony directement depuis le **terminal**.

Pour créer un projet d'application web classique, vous pouvez utiliser la commande '*symfony new --webapp nom_du_projet*'.

Pour lancer l'application, il suffit d'ouvrir un terminal dans le dossier du projet et faire la commande '*symfony server:start*'. Par défaut, le port 8000 va être utilisé.

Base de donnée

Pour faire la connexion avec la base de donnée, il suffit maintenant d'ouvrir le fichier `.env` situé directement dans le dossier, de dé-commenter le système de base de donnée utilisée, et de modifier les informations de la chaîne de caractères.



```
26 # DATABASE_URL="sqlite://%kernel.project_dir%/var/data.db"
27 DATABASE_URL="mysql://root@127.0.0.1:3306/track_manga?serverVersion=10.11.2&charset=utf8mb4"
28 # DATABASE_URL="mysql://user:password@127.0.0.1:3306/app?serverVersion=10.11.2-MariaDB&charset=utf8mb4"
29 # DATABASE_URL="mysql://stgre@127.0.0.1:5432/app?serverVersion=15&charset=utf8"
30 ###< doctrine/doctrine-bundle ###
```

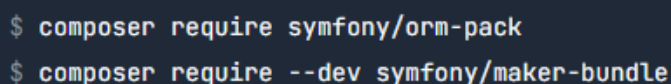
Image: fichier `.env` de l'application Symfony du projet.

Les différentes flèches correspondent à :

- Rouge: le système de gestion de bases de données.
- Bleu: le couple nom d'utilisateur et mot de passe (ici seulement 'root' car il n'y a pas de mot de passe).
- Violet: est le port de connexion (par défaut celui de **MySQL**).
- Jaune: le nom de la base de données, ici 'track_manga'.
- Blanc: représente la version du serveur (valeur par défaut).
- Rose: l'encodage des caractères (valeur par défaut).

Maintenant que la base de donnée est créée, et que la connexion avec l'application Symfony est établie, il faut la remplir. Pour cela, **Doctrine** va être utilisée. Il s'agit d'un ensemble de **librairies PHP** concentré sur le stockage en base de donnée et le **mapping objet-relationnelles** (le lien entre les programmes orientés objets et les bases de données relationnelles.) Plusieurs commandes vont être nécessaires pour l'installer.

Lien: <https://symfony.com/doc/current/doctrine.html>



```
$ composer require symfony/orm-pack
$ composer require --dev symfony/maker-bundle
```

*Image: Commandes pour installer **Doctrine** provenant de la documentation **Symfony**.*

La commande 'composer install' va être utilisée pour installer le package.

La prochaine commande sera 'php bin/console make:entity', une série de questions apparaîtra, permettant de définir le **nom** de l'entité, les **champs** de celle-ci, leur **type**, leur **taille**, et si il est **obligatoire**. Si un champ doit avoir une **relation**, il suffit d'entrer celle-ci dans son type (ex: OneToMany) et de donner le nom de l'entité concernée. (Voir image page suivante). Lorsque l'on souhaite créer une entité **user**, il est préférable d'utiliser la commande 'php bin/console make:user' à la place, les raisons seront évoquées dans la partie sécurité.


```
PS C:\Users\Nicolas\Desktop\TEST\manga> php bin/console make:entity

Class name of the entity to create or update (e.g. VictoriousGnome):
> Manga

Mark this class as an API Platform resource (expose a CRUD API for it) (yes/no) [no]:
> no

created: src/Entity/Manga.php
created: src/Repository/MangaRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):
> name

Field type (enter ? to see all types) [string]:
> string

Field length [255]:
>

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/Manga.php

Field type (enter ? to see all types) [string]:
> ManyToOne

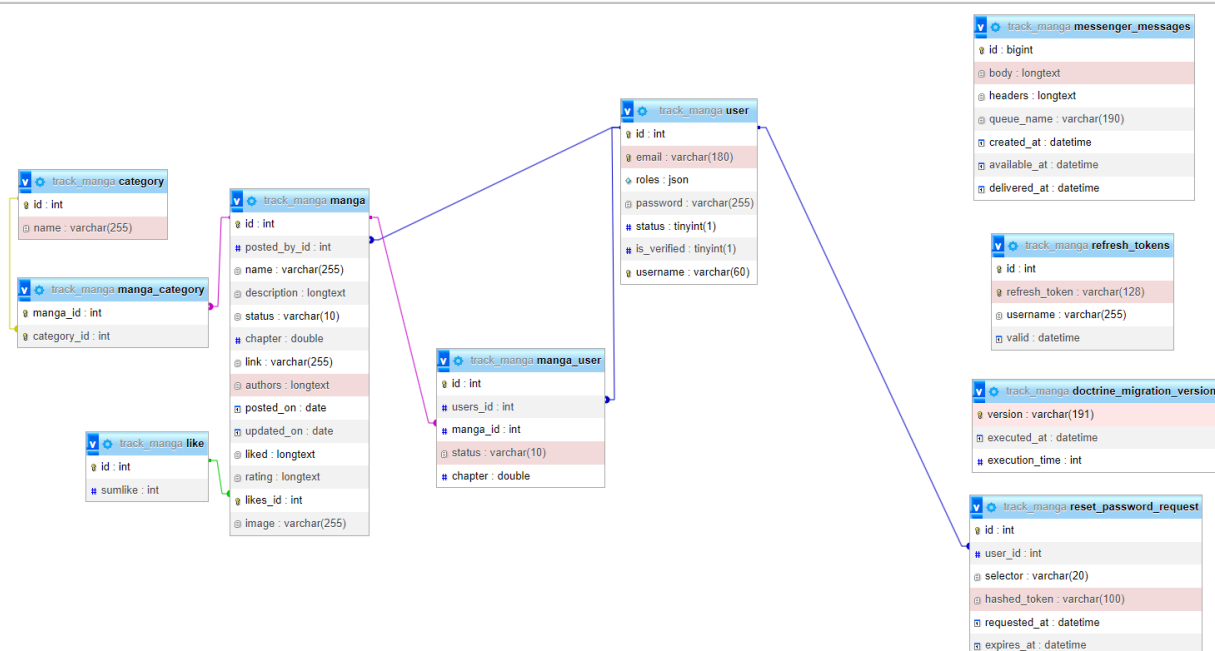
What class should this entity be related to?:
> User
```

Image: Extrait de code provenant du terminal lors de la création d'une entité et d'ajout de relation.

Lorsque que toutes les entités et leurs champs ont étaient créés, il suffit de lancer la commande `'php bin/console make:migration'`, cela va générer un fichier de migration en PHP dans le dossier des **migrations**. Ce fichier contiendra les requêtes SQL générées à partir de toutes les entités créées et des modifications à apporter à la base de données. Ensuite la commande `'php bin/console doctrine:migrations:migrate'` sera exécutée pour appliquer tous les fichiers de migrations qui n'ont pas encore été utilisés.

Maintenant, en utilisant **phpMyAdmin**, précédemment installée, on peut voir l'état de la base de données `'track_manga'` (page suivante).

Vous remarquerez que certains packages ont créé de nouvelles tables dans la base de données, d'où la présence de tables telles que `'refresh_tokens'`, `'doctrine_migration_version'`, `'messenger_messages'`, et `'reset_password_request'`.



Capture d'écran de la base de donnée du projet.

Sécurité

Symfony dispose de nombreux outils permettant de sécuriser une application, et d'autres peuvent être installés en supplément, tels que le **SecurityBundle**. Ce bundle nous donnera accès à des fonctionnalités avancées de gestion de l'authentification et d'autorisations dans l'application.

Make:user

En utilisant la commande '`php bin/console make:user`', une entité **user** va être créée de la même façon que décrit précédemment, en répondant à différentes questions pour définir les propriétés de l'entité. Ces questions varieront légèrement, certaines demanderont une propriété unique (autre que l'**id**) pour identifier l'utilisateur, comme l'adresse e-mail dans ce projet.

Une autre question va nous demander si le mot de passe a besoin d'être hashé. Si vous choisissez de le faire, chaque fois qu'un nouvel utilisateur sera enregistré en utilisant l'interface du back-office, son mot de passe sera hashé avant d'être enregistré en base de données. (La fonctionnalité était activée, mais pour ce projet, les utilisateurs s'inscriront via l'API et passeront donc par un contrôleur différent.)

security.yaml:

Il s'agit d'un fichier de configuration qui gère la sécurité de l'application.

Les pare-feu, **firewalls**, servent à appliquer une politique de sécurité dans le réseau. Dans le contexte d'une application **Symfony**, ils permettent de définir quelle partie de l'application est sécurisée et comment les utilisateurs doivent s'authentifier. Ils sont définis dans le fichier **security.yaml** provenant du **SecurityBundle**.

```
firewalls:
  dev:
    pattern: ^/(_(profiler|wdt)|css|images|js)/
    security: false
  login:
    pattern: ^/api/login
    stateless: true
    lazy: true
    provider: app_user_provider
    json_login:
      check_path: api_login # or api_login_check as defined in config/routes.yaml
      success_handler: lexik_jwt_authentication.handler.authentication_success
      failure_handler: lexik_jwt_authentication.handler.authentication_failure
    logout:
      path: api_token_invalidate
  api:
    pattern: ^/api
    provider: app_user_provider
    stateless: true
    jwt: ~

  main:
    lazy: true
    provider: app_user_provider
    custom_authenticator: App\Security\AppAuthenticator
    logout:
      path: app_logout
      # where to redirect after Logout
      target: app_login
    remember_me:
      secret: '%kernel.secret%'
      lifetime: 604800
      path: /
      always_remember_me: true

  json_login:
    check_path: /api/login
```

*Capture d'écran provenant du fichier **security.yaml** de mon application Symfony*

Un seul **firewall** est actif à la fois, **pattern** est utilisé afin de repérer à quelle requête actuelle il appartient. Dans Symfony, ils sont composés de diverses **key** (clés) permettant de modifier leur fonctionnement. Le **firewall** 'dev' sert à éviter accidentellement le blocage de l'outil développeur de Symfony.

Le **firewall** 'login' est celui qui gère la connexion à l'API. Il est composée des **key** suivantes:

- **pattern** indique la route pour accéder à ce **firewall**, ici c'est './api/login'.
- **stateless** permet d'indiquer s'il est nécessaire de stocker les informations de connexion dans la session; la valeur 'true' indique que non.
- **lazy** permet d'empêcher le démarrage de la session si il n'y a pas besoin d'autorisation.
-

- **provider** ici fait référence à 'app_user_provider', lui-même faisant référence à l'entité user et utilisant la propriété 'email' afin de vérifier la connexion en utilisant Doctrine.

```
app_user_provider:
  entity:
    class: App\Entity\User
    property: email
```

- **json_login** permet de créer un **endpoint** (point d'accès) pour l'authentification et récupérer un **tokenJWT**, qui sera utilisé pour interagir avec certaines opérations de l'API. La clé **check_path** utilise le nom de l'une des routes de **SecurityController**.
- **logout** permet d'invalidier le token généré précédemment.

Le **firewall 'api'** gère les opérations de l'API, la **key 'jwt'** indique l'utilisation d'un **tokenJWT** pour la vérification des autorisations.

Le **firewall 'main'** est le pare-feu responsable du reste du back-office et possède plusieurs **key** dont:

- **custom_authenticator** est lié au fichier **AppAuthenticator** généré par le **SecurityBundle**, (je ne connais pas le fonctionnement de celui-ci à part le fait qu'il permet la connexion et la génération d'un 'passport').
- **logout** avec la 'sous-clé' **path** indique le nom de la route pour la déconnexion, la 'sous-clé' **target** indique le nom de la route vers laquelle l'utilisateur sera redirigé après sa déconnexion. Ici il sera redirigé vers la page de connexion.
- **remember_me** permet de sauvegarder les informations de connexion de l'utilisateur dans la session pendant un certain laps de temps. Ici, la valeur de **lifetime** est de 604800 secondes, ce qui signifie que l'utilisateur restera connecté pendant 7 jours avant de devoir s'authentifier à nouveau.

Un autre moyen de sécuriser l'application est de bloquer certaines routes en fonction des autorisations de l'utilisateur connecté. Pour cela, **access_control** est utilisé.

Il suffit de préciser un chemin d'accès (**path**) où la restriction sera appliquée et les **roles** nécessaires pour y accéder. La notation **^/admin** permet de cibler toutes les routes commençant par **^/admin**. Il est aussi possible de limiter l'accès en utilisant des attributs telles que **IP**, **port**, **host**, **methods**.

```
access_control:
- { path: ^/admin, roles: ROLE_ADMIN }
- { path: ^/manga, roles: ROLE_ADMIN }
- { path: ^/user, roles: ROLE_ADMIN }
- { path: ^/category, roles: ROLE_ADMIN }
- { path: ^/login, roles: PUBLIC_ACCESS }
- { path: ^/api$, roles: PUBLIC_ACCESS, ips: [127.0.0.1, ::1, 192.168.0.1/24] }
- { path: ^/api$, roles: ROLE_NO_ACCESS }
- { path: ^/api/(login|token|refresh), roles: PUBLIC_ACCESS }
```

API

Afin d'établir la relation entre le système de base donnée et l'application front, une API est utilisée. Dans ce projet, ApiPlatform a été utilisé pour créer une API REST configurable. Pour l'utiliser, vous devez l'installer avec la commande `'composer require api'`. Ensuite, pour exposer les entités souhaitées, il faut leur ajouter l'attribut **ApiResource**. Cet attribut dispose de nombreuses propriétés permettant de configurer les opérations disponibles, les propriétés de l'entité à envoyer, d'appliquer des mesures de sécurité, de définir des critères de tri, etc.

La création d'un fichier **OpenApiFactory** permet de **décorer** des services d'ApiPlatform, c'est à dire de définir des nouveaux comportements et éventuellement de les remplacer. Dans mon projet, il est utilisé afin de créer un schéma de sécurité utilisant un tokenJWT pour l'authentification, créer une route de connexion et de déconnexion.

```
$schemas = $openApi->getComponents()->getSecuritySchemes();  
$schemas['bearerAuth'] = new ArrayObject([  
    'type' => 'http',  
    'scheme' => 'bearer',  
    'bearerFormat' => 'JWT'  
]);
```

*Capture d'écran du schéma de sécurité du fichier **OpenApiFactory**.*

La première ligne attribut la valeur de `'$openApi->getComponents()->getSecuritySchemes()'` à la variable `'$schemas'`, qui représente désormais les schémas de sécurité de l'API.

La seconde ligne permet d'ajouter un nouveau schéma de sécurité nommée `'bearerAuth'`. Celui-ci est de type `'http'`, utilise le `'scheme'` `'bearer'` comme méthode d'autorisation. L'utilisation de `'bearerFormat'` permet de spécifier qu'il s'agit d'un tokenJWT. Désormais, lorsque **ApiResource** aura la propriété `'openapiContext'` et la valeur `'security => [['bearerAuth'=>[]]]'`, un tokenJWT sera requis pour accéder à l'opération.

```
#[ApiResource(  
    openapiContext: [  
        "security" => [["bearerAuth" => []]]  
    ],  
    security: "is_granted('ROLE_USER')",  
    securityMessage: "Vous devez être connecté pour accéder à ces infos.",  
    ...  
)]
```

*Exemple de l'attribut **ApiResource** de l'entité **MangaUser** (suivi des mangas).*

Comme expliqué ci-dessus, `'openapiContext'` permettra ici de demander un tokenJWT permettant authentifier l'utilisateur et de récupérer des informations, y compris l'identifiant, afin de restreindre ses opérations. La propriété `'security'` servira comme deuxième couche de protection pour n'autoriser que les utilisateurs ayant le rôle `'USER'`. `'securityMessage'` correspond au message renvoyé à l'utilisateur en cas de refus d'accès.

```
operations: [  
  new GetCollection(  
    uriTemplate: '/bookmark',  
    paginationItemsPerPage: 10,  
    order: [  
      'status' => 'DESC',  
      'id' => 'DESC'  
    ],  
    normalizationContext: ['groups' => ['read:bookmark']],  
  ),  
]
```

*Exemple d'une opération de l'attribut **ApiResource** de l'entité **MangaUser** (suivi des mangas)*

Le tableau 'operations' est composée des diverses opérations disponible sur l'API concernant l'entité. Ici, il contient 'GetCollection' qui permet de récupérer un tableau composé des suivis de l'utilisateur actuel.

Cette opération possède diverses propriétés, telles que:

- **uriTemplate**, qui représente l'**uri** pour y accéder. Elle sera ajoutée après './api', ce qui signifie que pour y accéder il faudra donc faire './api/bookmark'.
- **paginationItemsPerPage**, qui permet de définir le nombre de résultats par requête. Une pagination sera créée pour accéder aux résultats suivants ou précédants. Ici, un maximum de 10 résultats par requête est demandé.
- **order**, permet de définir dans quel ordre les éléments du tableau de résultat seront agencés. Ici, 'status' ayant pour valeur 'DESC', cela va placer en premier ceux ayant le status 'ongoing' (en cours), et 'id' ayant la valeur 'DESC' va mettre les derniers éléments ajoutés en premier.
- **normalizationContext**, qui permet d'indiquer quel groupe de propriétés doit être renvoyé suite à la requête. Ici, le groupe est 'read:bookmark' (la syntaxe *operation:nom* permet de repérer plus facilement le groupe d'opération concerné). Ce groupe renverra les informations concernant l'id, le status, le chapitre, et le manga concerné.

En implémentant 'QueryCollectionExtensionInterface' et 'QueryItemExtensionInterface' provenant du bundle **ApiPlatform** et en utilisant **Doctrine** dans un fichier php, il est possible de modifier les requêtes DQL.

Dans ce contexte, l'objectif est de restreindre ce que l'utilisateur peut récupérer et modifier la requête effectuée afin de sécuriser les données.

Deux fonctions sont disponibles, l'une pour les opérations affectant une entité, et l'autre pour les collections d'entités. Le processus est similaire pour les deux fonctions. Les fonctions ont plusieurs paramètres, mais ici, seulement deux nous seront utiles ici, '\$resourceClass' et '\$queryBuilder', qui représentent respectivement le type d'entité et la requête à exécuter.

Dans cet exemple, la fonction qui s'applique aux collections d'entités sera utilisée.

```
public function applyToCollection(QueryBuilder $queryBuilder, QueryNameGeneratorInterface $queryNameGenerator,
    string $resourceClass, ?Operation $operation = null, array $context = []): void
{
    if ($resourceClass === MangaUser::class) {
        $this->addWhere($resourceClass, $queryBuilder);
    }
}
```

Capture d'écran provenant du fichier CurrentUserExtension du projet.

Cette fonction affecte toutes les opérations récupérant une collection d'entités, mais son but est de n'affecter que celles nécessitant une authentification. Dans notre cas, il s'agit des opérations provenant de l'entité 'MangaUser'. Une condition est donc créée pour vérifier à quelle entité appartient l'opération. Si l'opération appartient à l'entité 'MangaUser', alors la fonction 'addWhere' sera déclenchée.

```
private function addWhere(string $resourceClass, QueryBuilder $queryBuilder)
{
    $alias = $queryBuilder->getRootAliases()[0];
    $user = $this->security->getUser();
    if ($user instanceof User) {
        $queryBuilder
            ->andWhere("$alias.users = :current_user")
            ->setParameter('current_user', $user->getId());
    }
}
```

Capture d'écran de la fonction addWhere du fichier CurrentUserExtension

La fonction 'addWhere' permet d'ajouter une condition pour vérifier que les données que l'utilisateur souhaite récupérer ou modifier lui appartiennent.

La variable '\$alias' prend la valeur de l'alias de la table racine de la requête SQL générée par la variable '\$queryBuilder', tandis que la variable '\$user' prend la valeur de l'entité 'User' récupérée en décryptant le tokenJWT grâce au **SecurityBundle** de Symfony.

Une vérification est effectuée pour s'assurer qu'il s'agit bien d'une **instance** de l'entité 'User'. Si c'est le cas, alors la variable '\$queryBuilder' est modifiée, '→andWhere' permet d'ajouter une condition, ici, nous vérifions que le champ 'users' de la table racine correspond à la variable ':current_user'. Ensuite, '→setParameter' est utilisée pour attribuer une valeur à la variable ':current_user' de manière sécurisée, afin de prévenir les injections SQL.

Présentation du jeu d'essai de la fonctionnalité la plus représentative.

La fonctionnalité la plus représentative d'une application de suivi de lecture en Vue.js est la capacité à ajouter ou retirer des mangas du suivi de lecture de manière dynamique.

Sans cette fonctionnalité, il serait impossible de suivre la progression de la lecture, et le projet se réduirait simplement à une application de recensement de mangas, perdant tout son objectif.

Fonctionnalité:

```
<div class="bookmark" v-if="auth.isAuthenticated">
  <button v-if="!store.specificMangaBookmarked" @click="store.addBookmark">Ajouter au suivi</button>
  <button v-else @click="store.deleteOneBookmark">Retirer du suivi</button>
</div>
```

Code gérant l'affichage du bouton permettant l'ajout/suppression d'un manga au suivi

En utilisant Vue.js, il est possible d'ajouter des conditions à l'affichage de balise dans le DOM. Dans cet exemple, une condition est appliquée à la balise parente du bouton permettant l'ajout/suppression d'un manga au suivi. Cette condition utilise le **store** d'**authentification** pour accéder à la propriété `isAuth` (booléen). Si cette propriété vaut **true** (vrai), alors la balise est créée. Lorsque l'utilisateur se connecte, cette propriété prend la valeur **true**, et le bouton devient disponible lorsqu'il arrive sur la page d'un manga. Pour permettre l'ajout ou la suppression d'un manga au suivi, une autre condition est vérifiée, selon si le manga a déjà été ajouté ou non au suivi.

Lorsque l'utilisateur arrive sur la page d'un manga, la fonction **getOneManga** du **store** de l'**API** est appelée pour récupérer les informations le concernant.

La fonction **getOneManga** est asynchrone et utilise un appel **fetch** pour récupérer les informations. Seul le résultat du **fetch** de cette fonction nous intéresse.

```
// Les données reçues sont stockées dans une variable
this.specificManga = data;
// Si l'utilisateur est sur la page 'bookmark', on vérifie si le manga recherché est dans la liste.
const res = this.bookmarks.findIndex(e => e.manga.id == data.id);
if (res >= 0) {
  // il y est on peut tout simplement récupérer les informations.
  this.specificMangaBookmarked = this.bookmarks[res];
} else {
  // il n'y est pas, il faut chercher les informations
  this.currentMangaBookmark();
}
```

*Résultat du **fetch** de la fonction **getOneManga**.*

La variable `'specificManga'` prend la valeur des informations stockées dans `'data'` concernant le manga actuel. La fonction `'findIndex'` est utilisée sur la variable `'bookmarks'`, qui est un tableau. Cette variable prend une valeur lorsque l'utilisateur va sur la page **bookmark** (suivi de lecture). Cette fonction permet de récupérer l'**index** d'un élément d'un tableau si une condition est remplie. Ici, on vérifie si un manga possède une propriété `'id'` ayant la même valeur que l'id de `'data'`.

Si c'est le cas, l'**index** est renvoyé (chiffre supérieur ou égale à 0), et la variable `'specificMangaBookmarked'` prend pour valeur l'élément du tableau `'bookmark'` ayant comme **index** (`'res'`) instancier juste au dessus. Si ce n'est pas le cas la fonction `'currentMangaBookmark'` est appelée pour le récupérer.

La fonction `'currentMangaBookmark'` est une fonction asynchrone utilisant un appel **fetch** avec l'**id** du manga actuel afin de voir si l'utilisateur l'a déjà ajoutée à sa liste de suivi. La réponse du **fetch** est la seule partie qui nous intéresse.

```
if (data['hydra:member'].length == 0) {  
  this.specificMangaBookmarked = false;  
} else {  
  this.specificMangaBookmarked = data['hydra:member'][0];  
}
```

Morceau de code de la fonction currentMangaBookmark

Les données renvoyée par cet appel **fetch** sont sous forme de tableau et sont contenues dans une clé `'hydra:member'`. Une condition est utilisée pour vérifier la taille de tableau, en utilisant la propriété `'length'`. Si le tableau à une taille de 0, cela signifie qu'il ne contient aucune donnée, ce qui implique que l'utilisateur n'a pas ajouté ce manga à son suivi. Dans ce cas, la variable `'specificMangaBookmarked'` prend la valeur `'false'`.

Un bloc `'else'` est utilisé pour gérer les autres cas de figures. Si des informations sont renvoyées, elles sont stockées dans la variable `'specificMangaBookmarked'`.

Grâce à Vue.js, lorsque que la variable `'specificMangaBookmarked'` est modifiée, le bouton affiché du côté utilisateur change, permettant d'ajouter le manga au suivi si il n'y est pas, et le supprimer si il y est déjà.

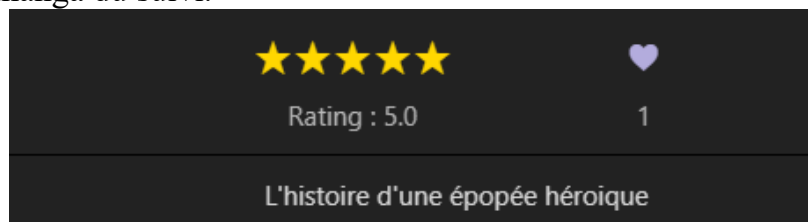
Jeu d'essai

Différents scénarios doivent être testés pour valider la fonctionnalité.

1. L'utilisateur n'est pas connecté, et par conséquent, il ne doit pas pouvoir accéder à la fonctionnalité.
2. L'utilisateur est connecté et ajoute un manga à sa liste de suivi. Cela doit être effectué de manière dynamique pour montrer que l'action a bien été réalisée.
3. L'utilisateur est connecté et retire un manga de sa liste de suivi. Cela doit également être fait de manière dynamique pour montrer que l'action a bien été effectuée.

Scénario 1, Utilisateur non connecté

L'utilisateur n'est pas connecté, il ne peut donc pas voir le bouton permettant d'ajouter ou de retirer un manga du suivi.

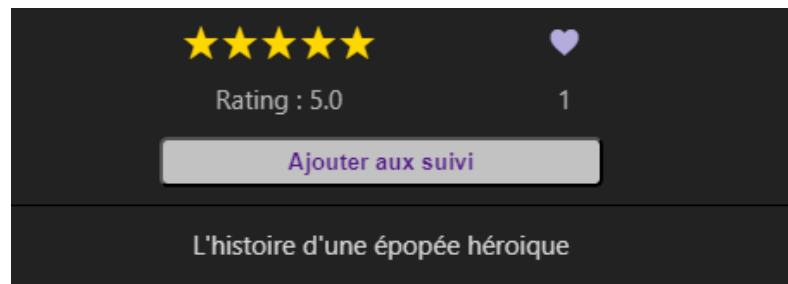


Interface utilisateur d'un manga lorsque celui-ci n'est pas connecté

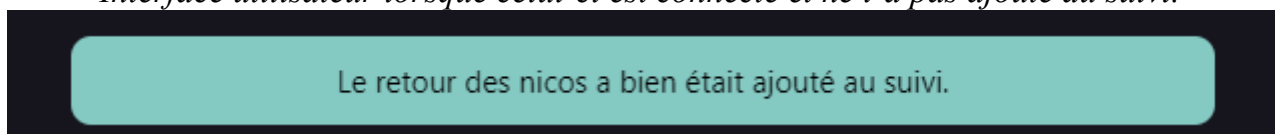
Aucune donnée n'est attendue en entrée, et aucune donnée n'est attendue ou obtenue.

Scénario 2, Ajouter au suivi

L'utilisateur s'est connecté, il peut donc voir le bouton permettant d'ajouter un manga à sa liste de suivi.



Interface utilisateur lorsque celui-ci est connecté et ne l'a pas ajouté au suivi.



Message indiquant que le manga a bien était ajouté au suivi.

La donnée en entrée est l'**@id** du manga à ajouter, spécifiée dans le body de la requête. (propriété d'**ApiPlatform** servant a relier a un objet directement)

```
body: JSON.stringify({  
  "manga": mangaId,  
})
```

Les données attendues comprennent un objet représentant l'entité **mangaUser** pour le manga actuel, avec les informations suivantes: **@context – @id – @type - id – chapter – objet manga – status**

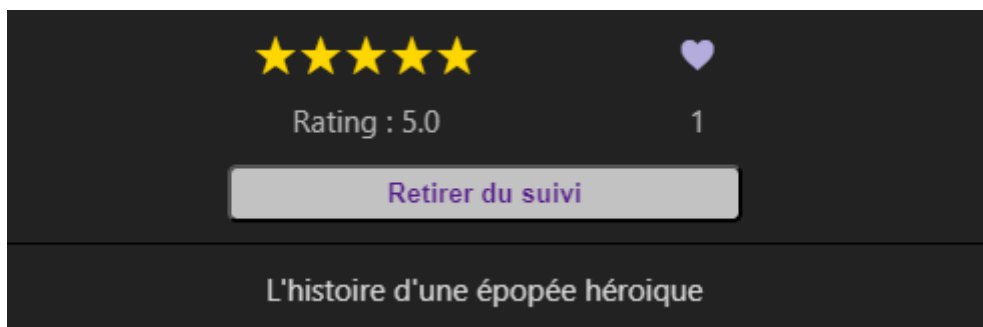
Les données récupérées correspondent aux informations spécifiées (voir image ci-dessous)

```
@context: "/api/contexts/MangaUser"  
@id: "/api/bookmark/22"  
@type: "MangaUser"  
chapter: 1  
id: 22  
▶ manga: {@id: '/api/manga/19', @type: 'Manga', id: 19, name: 'Pol', status: 'finished', ...}  
status: "ongoing"  
▶ [[Prototype]]: Object
```

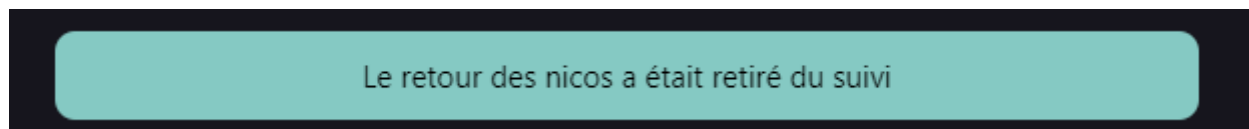
Console.log() des données reçues pour l'ajout d'un manga au suivi.

Scénario 3. Retirer du suivi

L'utilisateur est connecté et a déjà ajouté le manga à son suivi, il peut donc accéder au bouton permettant de le retirer.



Interface utilisateur lorsqu'il est connecté et que le manga est déjà dans son suivi.



Message indiquant que le manga a bien été retiré du suivi.

La donnée en entrée est l'**id** du manga à retirer, spécifiée dans l'URL de la requête.

```
http://127.0.0.1:8000/api/bookmark/remove/${id}
```

La seule donnée attendue est un **status HTTP 200**, indiquant le succès de la requête.

La donnée reçue est un **status HTTP 200**.

```
200 'OK'
```

Veille sur les vulnérabilités de sécurité.

Injection SQL

Définition

L'injection SQL ou **SQLi**, est une technique où un attaquant exploite les failles dans le code d'application responsable de la création de requêtes SQL dynamiques. Cette faille permet de faire exécuter une requête créée par l'attaquant, afin de manipuler la base de donnée.

Cette vulnérabilité est causé par un manque de précaution du développeur lors de la création de requête SQL et compromet toute l'application web car si l'attaquant le désire il peut récupérer les diverses informations, ou tout simplement supprimer des entités de la base de donnée.

Exemple de requête

```
$sql = "SELECT * FROM user WHERE name = '" . $_GET['name'] . "'";
```

Dans cet exemple de requête SQL, '`$_GET['name']`' est une variable provenant d'un formulaire envoyé par l'utilisateur, ici cette variable n'a pas été vérifiée, ni assainie (**sanitize**), cette variable peut potentiellement être dangereuse.

```
$_GET['name'] = '0; DROP TABLE user';
```

Si un utilisateur envoie comme valeur '`0; DROP TABLE user`' pour le champ '`name`' alors celle-ci sera concaténée à la requête SQL, et à son exécution si il y a une base de donnée '`user`' celle-ci sera supprimée.

Comment s'en protéger

Pour se protéger des **injections SQL**, plusieurs moyens sont disponibles:

- L'utilisation de déclarations préparées (**prepared statements**) avec des liaisons de variables, ce qui permet de lier les données de manière sécurisée pendant l'exécution et de les envoyer séparément de la requête SQL.
- La création d'une liste composée des différentes valeurs possibles pour une variable, puis la comparaison de ces valeurs avec les données envoyées par l'utilisateur pour garantir leur conformité.

Ces pratiques sont essentielles pour protéger une application contre les vulnérabilités d'injection SQL

Protection du projet

Lors de l'exécution d'une requête dans l'API comportant des données envoyées par l'utilisateur, la déclaration préparée '`→setParameter`' est utilisée afin de lier les données à la requête SQL.

```
->andWhere("$alias.users = :current_user")  
->setParameter('current_user', $user->getId());
```

Situation ayant nécessité une recherche à partir d'un site anglophone

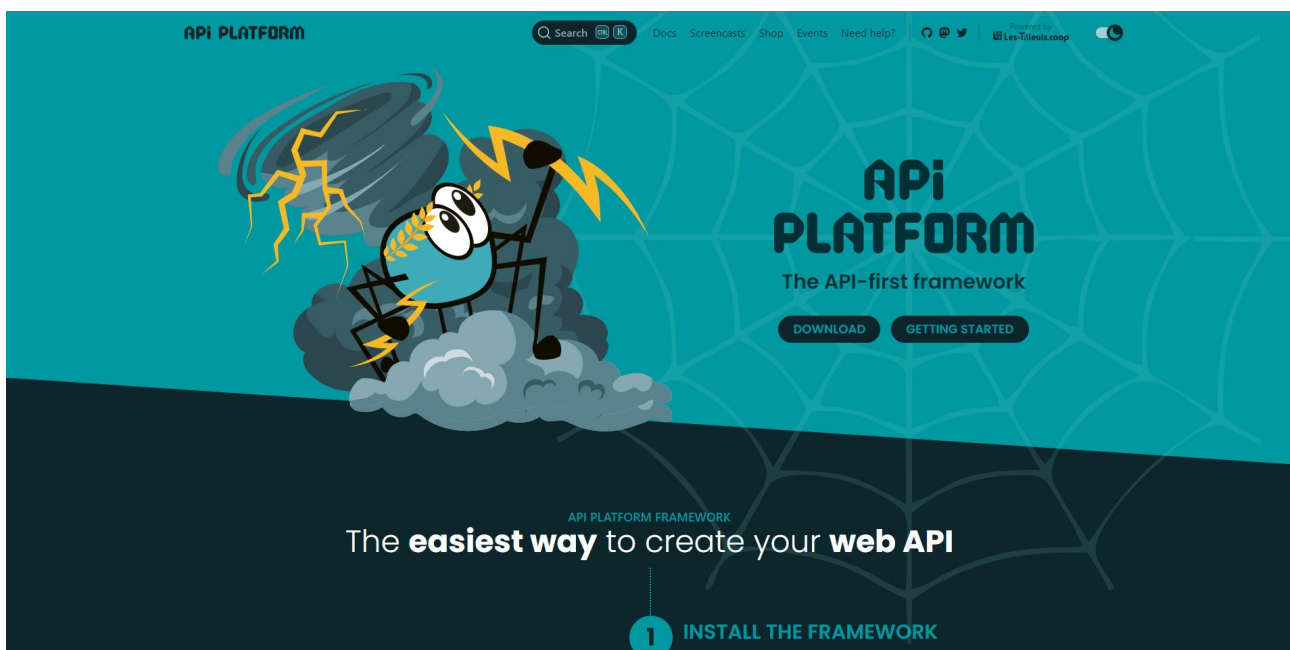
Description de la situation

Lors de la création du back, j'ai eu besoin de m'informer sur la façon d'authentifier l'utilisateur faisant des requêtes à l'API créée par le framework ApiPlatform.

Plus précisément sur la manière d'identifier un utilisateur en utilisant un token JWT afin de modifier les requêtes nécessitant une authentification, en ajoutant une restriction pour s'assurer que les données qu'il essaye de modifier ou de récupérer lui appartiennent.

Je suis allé sur le site officiel du framework ApiPlatform.

Lien: <https://api-platform.com/>



Page d'accueil du site du framework ApiPlatform.

J'ai ensuite utilisé la barre de recherche afin de trouver la documentation concernant l'authentification JWT.

Lien: <https://api-platform.com/docs/core/jwt/>

Extrait du site anglophone et sa traduction.

Configuring the Symfony SecurityBundle

It is necessary to configure a user provider. You can either use the [Doctrine entity user provider](#) provided by Symfony (recommended), [create a custom user provider](#) or use [API Platform's FOSUserBundle integration](#) (not recommended).

If you choose to use the Doctrine entity user provider, start by [creating your User class](#).

Then update the security configuration:

Configurer le paquet sécurité Symfony

Il est nécessaire de configurer un fournisseur utilisateur. Tu peut soit utiliser le fournisseur d'entité utilisateur Doctrine prévu par Symfony (recommander), créer un fournisseur utilisateur sur mesure ou utiliser le paquet d'intégration utilisateur 'FOS' d'API Platform.

Si tu choisis d'utiliser le fournisseur d'entité utilisateur Doctrine, commence en créant ta classe Utilisateur.

Puis met à jour la configuration de la sécurité.

If you want to avoid loading the **User** entity from database each time a JWT token needs to be authenticated, you may consider using the [database-less user provider](#) provided by LexikJWTAuthenticationBundle. However, it means you will have to fetch the **User** entity from the database yourself as needed (probably through the Doctrine EntityManager).

Refer to the section on [Security](#) to learn how to control access to API resources and operations. You may also want to [configure Swagger UI for JWT authentication](#).

Si tu veut éviter de charger l'entité **Utilisateur** depuis la base de donnée à chaque fois qu'on token JWT a besoin d'être authentifiée, tu peut considéré utiliser le fournisseur utilisateur sans base de donnée prévu par le paquet d'authentification 'LexikJWT'. Cependant, cela veut dire que vous devrez avoir à récupérer l'entité **Utilisateur** depuis la base de donnée toi-même si nécessaire (probablement à travers le manager d'entité Doctrine).

Ce référé à la section sur la sécurité pour apprendre comment contrôle l'accès aux ressources de l'API et aux opérations. Tu peut aussi vouloir configurer pour l'interface Utilisateur Swagger pour l'authentification JWT.