

# DOSSIER DE PROJET



# **LES GARDIENS DE LA LEGENDE**

## **Club de jeux**

Présenté et soutenu par :

Romain Boudet



En vue de l'obtention du  
Titre Professionnel Développeur Web et Web mobile

## Table des matières

|   |    |
|---|----|
| REMERCIEMENT .....  | 4  |
| CONTEXTE ET PRÉSENTATION DU PROJET .....  | 5  |
| COMPÉTENCES DU RÉFÉRENTIEL .....  | 6  |
| Activité type 1 - Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité ..... | 6  |
| CP 1 - Maquetter une application .....  | 6  |
| CP 2 - Réaliser une interface utilisateur web statique et adaptable .....   | 6  |
| CP 3 - Développer une interface utilisateur web dynamique .....   | 7  |
| Activité type 2 - Développer la partie back-end d'une application web ou web en intégrant les recommandations de sécurité.....          | 7  |
| CP 5 - Créer une base de données.....   | 7  |
| CP 6 - Développer les composants d'accès aux données .....  | 8  |
| CP 7 - Développer la partie back-end d'une application web ou web mobile .....  | 9  |
| CAHIER DES CHARGES .....  | 10 |
| La conceptualisation de l'application .....   | 10 |
| Fonctionnalités et Minimum Viable Product .....   | 10 |
| Le Workflow de l'application.....   | 11 |
| La mise en dimension de l'application .....   | 12 |
| Les Wireframes de l'application .....   | 12 |
| La création de la charte graphique.....   | 13 |
| Utilisateurs et user stories.....   | 14 |
| Le public visé et les rôles.....  | 14 |
| Les User Stories .....  | 15 |
| La Roadmap de l'application .....   | 16 |
| SPÉCIFICATIONS TECHNIQUES.....  | 18 |
| Le Versionning .....  | 18 |
| Les technologies du Back.....   | 19 |
| Les technologies du Front .....   | 19 |
| La sécurité de l'application.....   | 19 |
| Autres services utilisés .....  | 22 |
| ORGANISATION DE L'ÉQUIPE .....  | 22 |
| Présentation de l'équipe .....  | 22 |
| Organisation de travail .....   | 23 |
| Outils utilisés .....   | 23 |

|  |    |
|--|----|
| Répartition et détail des rôles .....  | 24 |
| Réalisations personnelles .....  | 24 |
| Sprint 0 – Construction du Cahier des charges – MCD / MLD + réalisation de wireframe. .... | 25 |
| Sprint 1 – Mise en place des accès CRUD et du mécanisme d'inscription .....                | 25 |
| Sprint 2 – Authentification, mise en place de la méthode reset password, .....             | 32 |
| Sprint 3 – Mise en place du cache, configuration d'axios, CSS .....                        | 36 |
| Jeu d'essai.....   | 41 |
| Test divers via Postman.....   | 41 |
| VEILLE ET TROUBLESHOOTING .....  | 47 |
| Veille technologique et veille sur les vulnérabilités de sécurité .....                    | 47 |
| Comprendre l'utilisation des JWT .....   | 47 |
| Veille sur les vulnérabilités de sécurité .....  | 48 |
| Problématique – Recherche et traduction d' un extrait .....                                | 48 |
| CONCLUSION .....   | 50 |
| ANNEXES.....   | 51 |
| WIREFRAMES DESKTOP .....   | 51 |
| WIREFRAMES MOBILE .....  | 53 |
| MCD .....  | 54 |
| MLD .....  | 55 |
| SCRIPT BACKUP BDD.....   | 56 |
| RECAP COMMANDE GIT et SQITCH .....   | 59 |

## REMERCIEMENTS

Ces derniers six mois furent intenses et riches. J'ai fait le choix d'une reconversion professionnelle et ce projet en est le fruit.

Mais ce projet est également le résultat d'encouragements et de soutiens, sans lesquels tout cela n'aurait peut-être pas vu le jour.

Merci à l'école O'clock, qui a réussi le pari d'une formation téléprésentielle chaleureuse, humaine et de qualité. Je remercie particulièrement l'équipe pédagogique d'Oclock, toujours là pour œuvrer dans un intérêt commun, nous armer au mieux pour notre futur métier. Et ce dans une bonne ambiance, permettant un apprentissage serein et structuré. Un remerciement particulier pour Ben qui a ouvert la voie d'un long apprentissage (et sa passion du FOFF qu'il a essayé de transmettre en vain...), à Tony et Luc pour nous avoir lancé dans l'aventure du JS et à JEAN pour m'avoir mis sur la voie d'une passion au quotidien, avec la mise en place d'une partie Back-End toujours au top. Et bien évidemment à Marc, toujours là pour nous épauler, que l'on soit au sommet ou dans le creux de la vague.

Merci à l'équipe administrative et à celle lié à la certification, notamment Céline et Gauthier, toujours aux petits soins et avec ce brin d'humour qui fait tant de bien.

Tout ceci n'aurait pas eu la même saveur sans la team de champions qu'a été l'ensemble de la promotion Némo, avec laquelle l'entraide et l'échange n'ont pas été des mots vains.

Mention spéciale pour mes 4 co-équipiers pour ce projet avec qui ça été un plaisir de travailler et d'échanger au quotidien.

Et enfin, merci à mes proches, amis et parents, qui de pars leurs encouragements et soutien tout au long de ce projet, m'ont permis de me sentir prêt pour cette formation et cette reconversion.

## CONTEXTE ET PRESENTATION DU PROJET

Le projet « Les gardiens de la légende » est né d'un besoin, celui du club de jeux éponyme de la Maison des Jeunes et de la Culture d'Avrillé (49 – Nord d'Angers) de posséder un espace en ligne pour échanger, choisir des jeux, définir leurs prochaines soirées jeux et connaître les participants à ces soirées organisées. La demande était aussi d'être plus visible, et d'améliorer la notoriété du club.

Le site possédait un petit site sous Wordpress fraîchement mis en place, et son administrateur et membre du club, Franck Cappon, également étudiant à l'école O'Clock, nous a mobilisé, Lorène, Marion et moi-même, pour la création d'un site internet plus complet et permettant davantage de fonctionnalité, le tout avec un potentiel d'évolution dépendant de nos simples compétences.

Ce club est ouvert à tous les publics et est composé d'une trentaine de joueurs se réunissant de manière aléatoire, le mardi, le vendredi ou le samedi, souvent par groupe d'une dizaine. Le club propose et pratique des jeux de rôle, des jeux grandeur nature, des jeux de plateau, des jeux de cartes, des jeux de figurines, et quelques jeux vidéo. Avant tout abonnement, trois essais gratuits au club sont possibles. Le site possède un local au sein de la Maison des Jeunes et de la Culture d'Avrillé, avec une clé qui est demandée à l'administrateur quand des membres veulent s'y rendre pour jouer.

L'esprit du club est convivial et familial, mettant en avant le partage et le plaisir du jeu.

## COMPÉTENCES DU RÉFÉRENTIEL

Les spécifications techniques sont décrites plus en détail dans le chapitre spécifications techniques de ce rapport (cf. page 18), néanmoins un petit rappel afin de faciliter la lecture et la compréhension des compétences du référentiel en lien avec notre projet. Celui-ci a été réalisé via Javascript avec un front React et back sous Node.js .

Activité type 1 - Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité

CP 1 - Maquetter une application

Après avoir échangé avec Franck, le product owner de ce projet (Voir l'organisation de l'équipe) chargé de satisfaire les besoins du club, les besoins du club ont mis en avant la nécessité de mettre en place un site avec un design responsive. Des wireframes sous format smartphone et sous format fenêtre d'ordinateur ont donc été conçus. Cette étape de la conception est indispensable pour structurer notre interface et vérifier que tous s'articulent correctement. Cette étape est aussi un bon moyen de se concentrer particulièrement sur les versions responsives.

Une arborescence de l'application a également été maquettée via gloomap (<https://gloomaps.com>), pour qu'on se représente au mieux ce qu'allait être notre futur site et afin d'articuler correctement les pages entre elles selon les interactions. Les pages en vert dans l'arborescence ci-dessous représentent les parties développées dans la V2 et en rouge, les pages avec accès sécurisé.



Arborescence de notre application réalisée via Gloomap

CP 2 - Réaliser une interface utilisateur web statique et adaptable

Nous avons réalisé nos pages conformément à nos maquettages. Bien que la plupart des pages du projet soient dynamiques, les pages « Les gardiens » celles présentant le RGPD ou le profil de l'utilisateur sont des statiques et adaptables.

Pour que les pages soient responsives, et donc s'adaptent au média de l'utilisateur, le framework Sémantic UI est utilisé dans le menu avec les balises spécialisées : `only="computer"` ou `only= "tablet mobile"` (correspondant à 760px et 991px) et en CSS avec des media queries (des balises `@media`) en définissant des breakpoints pertinents et logiques en pixel.

### CP 3 - Développer une interface utilisateur web dynamique

Pour une interface réactive et interactive nous avons utilisé REACT avec ses composants.

React gère les données de l'application (le state) de façon dynamique. Au moindre changement de ce state (l'utilisateur clique sur un lien, une donnée est mise à jour, un timer se déclenche...), les éléments du DOM virtuel de React sont recalculés et le DOM mis à jour est affiché. Le DOM (Document Object Model) est une représentation de la structure de l'affichage à l'écran. C'est le DOM qui est interprété par le moteur de rendu des navigateurs internet pour être ensuite affiché. React est une librairie flexible et performante, notamment grâce à un DOM virtuel et en ne mettant à jour le rendu qu'en cas de nécessité.

De manière pratique, des pages comme celles des évènements, permettant de s'ajouter à un évènement, d'en créer un, sont des pages dynamiques. Pages qui varient aux gré de ce que rentrent les utilisateurs dans la base de données.

### Activité type 2 - Développer la partie back-end d'une application web ou web en intégrant les recommandations de sécurité

#### CP 5 - Créer une base de données

Avant la création à proprement parlé d'un script DDL (Data Définition Language), nous avons réfléchi à la structure de notre Base de Données (BDD).

Nous avons élaboré les Users Stories du projet, puis le Model Conceptuel de Données (MCD) via Mocodo Online (<http://mocodo.wingi.net>) pour notre Minimum Viable Product (MVP), puis nous avons effectué un passage du MCD au Modèle Logique de Données (MLD). Bien que nous ayons conçu notre MCD pour notre MVP, un second MCD pour la V2 a également été conçu.

Le passage du MCD au MLD s'est effectué en suivant certaines règles :

- Chaque entité du MCD devient une table, son identifiant devient la clé primaire de la relation. Les autres propriétés deviennent les attributs de la relation.
- Une association avec la cardinalité 1:N (avec les cardinalités maximales positionnées à "1" d'un côté de l'association et à "N" de l'autre côté) va se traduire par la

création d'une clé étrangère dans la relation correspondante à l'entité coté 1 (c'est cette table qui porte la relation). Cette clé étrangère référence la clé primaire de la relation correspondante à l'autre entité.

- Une association avec la cardinalité N:N, va se traduire par la création d'une table spéciale dont les attributs seront des clés étrangères référençant les relations correspondant aux entités liées par l'association.
- Dans le cas d'une cardinalité 1:1, la clé étrangère pourra être dans l'une ou l'autre des deux tables.

Un dictionnaire de données a également été conçu pour lister toutes les colonnes de nos tables de manière visuelle.

Associé à Node.js il nous fallait un SGBDR efficace. Nous avons fait le choix de PostgreSQL (libre sous licence BSD). Parce qu'il est reconnu pour son comportement stable et ses possibilités de programmation étendues, et aussi parce qu'il s'agissait du SGBDR avec lequel nous avions le plus de pratique.

Conformément à un bon usage de PostgreSQL, nous avons respecté les contraintes d'intégrité avec des clés (en pratique une clé "id" ayant une valeur numérique auto-incrémentée dans notre cas avec le type IDENTITY) et des clés étrangères, afin d'assurer la cohérence du modèle de données.

Pour respecter l'unicité de la donnée, de nombreuses clés ont été caractérisées avec la valeur UNIQUE afin de nous préserver de tout doublon non souhaité (la présence de la valeur UNIQUE sur la clé "email" nous assure qu'un seul utilisateur avec cet email sera présent en base). Pour maintenir une qualité des données en base, un typage fin des colonnes a permis d'apporter un premier niveau de contrôle (NOT NULL si on ne veut pas de valeur null, CHECK pour ajouter des contrôles sur le contenu des données.etc.). Avec PostgreSQL, tous les ordres de type DDL sur des objets d'une base étant transactionnels, nous n'avons pas oublié d'encadrer nos script DDL par les ordres BEGIN et COMMIT, nous assurant de ne rien écrire en BDD en cas de problème à la lecture du script.

Enfin, d'un point de vue sécurité en production, nous avons donné un mot de passe à un rôle (ALTER ROLE monrôle WITH PASSWORD "..."). Tandis que d'un point de vue pratique, en back, j'ai travaillé avec psql (interface en mode texte pour PostgreSQL) qui permet de saisir les requêtes de manière interactive et qui permet l'utilisation de méta-commande. (<https://www.postgresql.org/docs/13/app-psql.html>).

La BDD, une fois déployée, a pu évoluer (création de vue, de domaine, etc). via sqitch (cf. page 16 "Les technologies du back").

## CP 6 - Développer les composants d'accès aux données

L'architecture choisie a été celle de l'Active Record (et non du dataMapper), où le contrôleur interroge les modèles, avec chaque modèle qui dispose de ses propres accès

CRUD (Create Read Update Delete) à la BDD, puis le modèle renvoie les infos obtenues au contrôleur.

Dans notre application, les modèles gèrent l'accès aux données manipulées par l'application et s'occupent du stockage (ils constituent l'ensemble des classes qui définissent les objets manipulés par l'application), tandis que nos contrôleurs gèrent l'action demandées, récupèrent les données via le model, et renvoient en format json au front l'information voulue.

Nous avons fait le choix de ne pas utiliser d'ORM comme Sequelize dans ce projet. Les requêtes SQL ont été réalisées par nos soin dans les modèles.

#### CP 7 - Développer la partie back-end d'une application web ou web mobile

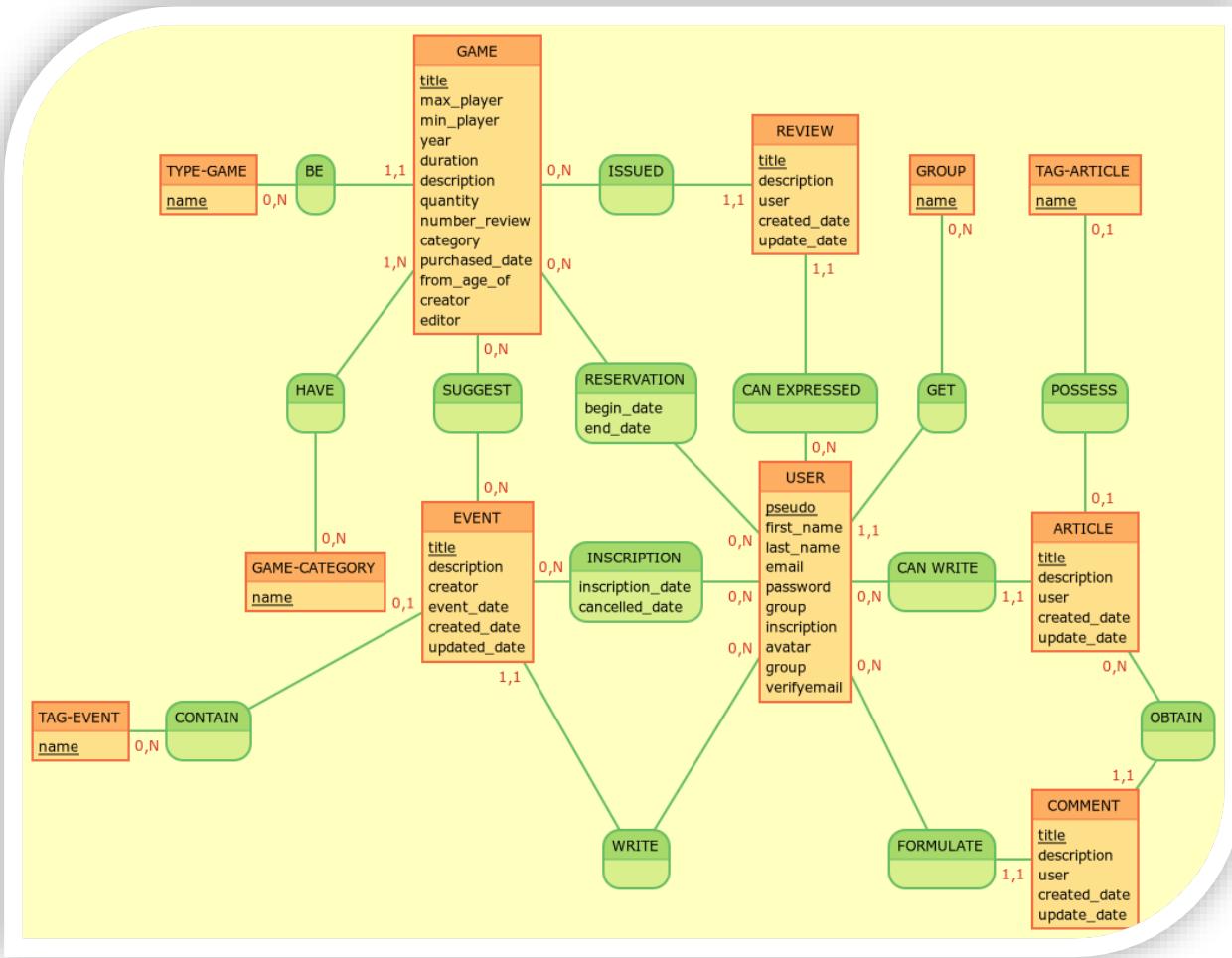
Le back repose sur un serveur Express HTTPS qui renvoie des données en format JSON à un front codé via la librairie REACT. La logique back-end de l'application est située dans les contrôleurs. Nous avons 7 contrôleurs pour 5 modèles, certaines parties de la logique ayant été déportées dans un contrôleur dédié (comme l'authentification qui n'est pas dans le "userController").

Nos méthodes dans nos contrôleurs s'occupent de récupérer, traiter et renvoyer l'information nécessaire au front. Dans notre application, les données renvoyées au front sont variées : articles, données sur un utilisateur, données sur un jeu, données sur un événement, autorisation d'authentification, etc.

# CAHIER DES CHARGES

## La conceptualisation de l'application

### Fonctionnalités et Minimum Viable Product



Afin de définir la structure de notre BDD, après un travail de réflexion nous avons créé notre MCD via mocoDo Online pour notre Minimal Viable Product :

Et son MLD associé, en suivant les 4 règles précédemment éditées (cf. page 5 "CP 5 – Créer une base de données") :

- ❖ group(id, name)
- ❖ tag\_article(id, name)
- ❖ category(id, name)
- ❖ type\_game(id, name)
- ❖ tag\_event (id, name)

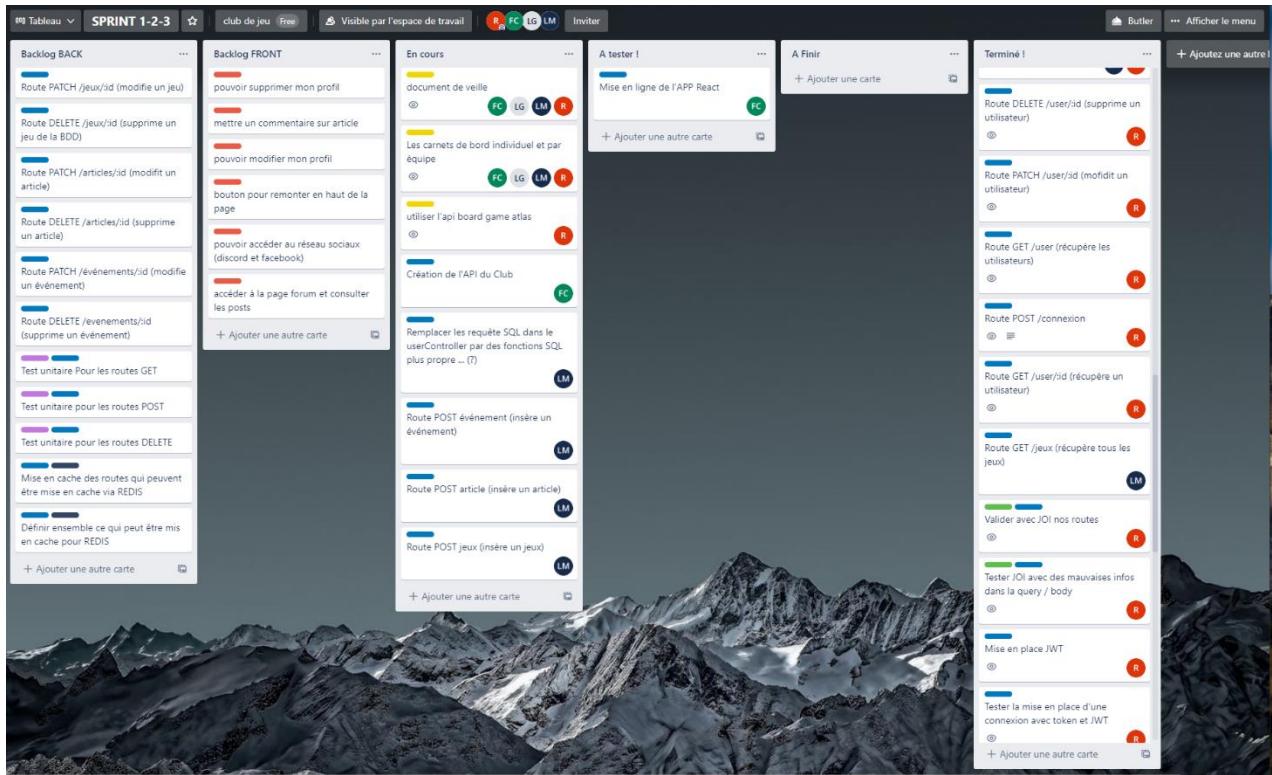
- ❖ game(id,title, max\_player, min\_player, year, duration, description, quantity, number\_review, category, purchased\_date, from\_age\_of, creator, editor)
- ❖ article(id, title, description, created\_date, update\_date , #user(id))
- ❖ review(id, title, description, created\_date , update\_date, #user(id), #game(id))
- ❖ comment(id, title, description, created\_date , update\_date, #user(id), #article(id))
- ❖ event(id ,title, description, event\_date, created\_date ,updated\_date, #user(id), #game(id))
- ❖ user(id, pseudo, first\_name, last\_name, email, password, avatar, verifyemail, #group(id))
- ❖ event\_has\_user(id, inscription\_date, cancelled\_date,#event(id), #user(id))
- ❖ event\_has\_game(id, #event(id), #game(id))
- ❖ category\_has\_game(id, #category(id),#game(id)): ##

## Le Workflow de l'application

Pour la réalisation du site, l'équipe, composée de 4 personnes, a utilisé la méthode agile SCRUM basée sur la communication, et un développement évolutif. Dans le cadre de cette méthode, on a réalisé un backlog priorisé (via Trello) où nous avons organisé nos idées, des sprints, (soit des intervalles de temps pendant lesquels l'équipe à complété un certain nombre de tâches du backlog), des réunions quotidiennes matinales pour suivre l'avancement de chacun et proposer son aide sur des points précis si nécessaire, et une rétrospective en fin de sprint. Au fur et à mesure que le statut des tâches fixées dans le backlog évoluaient (en cours, à tester, fait), on déplaçait les tâches dans le trello.

Ce projet a été réalisé sur 4 sprints :

- ❖ Sprint 0 : Initialisation du projet, réflexion technique, Wireframes et élaboration des documents liés à la BDD avant la création de script de création de la BDD (5 jours).
- ❖ Sprint 1 et 2 : Mise en place des principales fonctionnalités, temps dédié au code (10 jours).
- ❖ Sprint 3 : Finition des dernières fonctionnalités en cours, test, refactoring, correction des bugs, test finaux, déploiement sur le serveur d'hébergement (5 jours).



Notre Trello et son backlog

## La mise en dimension de l'application

### Les Wireframes de l'application

Les wireframes ont été réalisé via l'outil whimsical (<https://whimsical.com>). 14 Wireframes légendés ont été réalisés pour la partie dekstop et mobile.

Chaque wireframe reprend ce à quoi correspond chaque composant dans la page, détaille le lien s'il s'agit d'un bouton, et décrit succinctement l'image si le wireframe en contient. Chacun a réalisé en moyenne 3 wireframes pour la partie dekstop et 3 pour la partie mobile. Nous avons travaillé ensemble afin d'homogénéiser nos créations après avoir défini ensemble une charte visuelle commune pour tous nos wireframes.

Le détail de ces wireframes est disponible en annexe. En voici un aperçu succinct :



Wireframe présentant la page contact

## La création de la charte graphique

Le club ne possède pas à proprement parler de charte graphique que l'on aurait pu suivre, mais uniquement un logo. Nous avons repris leur logo et afin de se rapprocher de l'univers parfois fantastique et aventure des jeux de plateau, nous nous sommes mis d'accord pour un thème médiéval en adéquation avec le logo du club, représentant un bouclier et des épées.



La bannière du site



Le logo du club

#### Le fond d'écran du site

Une fois les avis échangés, le product owner a validé cette idée d'un décor médiéval pour le site. Les images ne sont pas libres de droits. Les droits nous ont été délivré par écrit, à la suite d'une demande de notre part, par le studio de production de jeux vidéo anglais Firefly Studios propriétaire de ces images, issues d'un de leur jeux vidéo : "Stronghold".

### Utilisateurs et user stories

#### Le public visé et les rôles

Ce club est ouvert à tous les publics et est composé d'une trentaine de joueurs se réunissant de manière aléatoire, le mardi, le vendredi ou le samedi, souvent par groupe d'une dizaine. Les joueurs ont tous les âges mais la plupart sont majeurs et adultes, et de nombreuses personnes viennent en famille.

Le site est configuré pour 4 types de rôles. Toutes les fonctionnalités reliées ne sont pas encore mises en place dans le Front.

- ❖ **Les utilisateurs non connectés**, qui ont accès aux articles, aux événements et à la liste des jeux. Mais ils ne peuvent avoir accès à leur page de profil, créer un événement, s'inscrire ou se désinscrire à un événement.
- ❖ **Les utilisateurs connectés (membres)**, qui peuvent faire tout ce que peut faire un utilisateur non connecté mais également avoir accès à leur profil (prénom, pseudo, nom, email), créer un événement, s'inscrire ou se désinscrire à un événement.
- ❖ **Les modérateurs**, qui lorsque le forum sera créé pourront supprimer et modifier des messages écrits par des utilisateurs connectés et supprimer des articles.
- ❖ **Les administrateurs**, qui pourront faire tout ce que font les rôles précédents mais aussi supprimer des comptes de membres, écrire des articles, ajouter ou enlever des jeux de la base de données, donner des droits supplémentaires à certains membres.

## Les User Stories

En vue de définir les différentes fonctionnalités du projet qu'il allait falloir développer, nous avons établi des User Stories. Avec des User Stories de base pour le MVP et des Users Stories pour la V2 du projet.

Nos Users Stories :

### En tant que visiteur je peux :

- ❖ Accéder la page home et visualiser des événements, des jeux de sociétés, articles
- ❖ Accéder à la page Événement et visualiser les évènements
- ❖ Accéder à la page forum et consulter les articles
- ❖ Accéder à la page liste des jeux mis à disposition
- ❖ Accéder à la page d'un jeu de société
- ❖ Accéder à la page contact du club
- ❖ Accéder à la page présentation du Club
- ❖ Accéder à la page RGPD
- ❖ Peut s'inscrire pour devenir utilisateur
- ❖ Bouton pour remonter en haut de la page\*
- ❖ Pouvoir accéder aux réseaux sociaux (Discord et Facebook)\*

### En tant qu'utilisateur je peux

- ❖ Faire tout ce que fait un visiteur
- ❖ Pouvoir modifier mon profil \*
- ❖ Pouvoir supprimer mon profil\*
- ❖ Mettre un commentaire sur article\*
- ❖ Écrire un article
- ❖ Supprimer ou modifier un article\*
- ❖ Proposer un évènement
- ❖ Supprimer ou modifier un évènement\*
- ❖ M'inscrire ou me désinscrire à un évènement
- ❖ Vérifier un email avant connexion

Actuellement, de nombreuses fonctionnalités sont présentes en back et ne sont pas encore implémentées en front, elles sont représentées avec une astérisque\*. Ces fonctionnalités seront intégrées prochainement.

## La Roadmap de l'application

J'ai mis en place une documentation Swagger sur le router. Pour rappel, Swagger UI est un logiciel permettant de générer une documentation en utilisant les spécifications d'OpenAPI.

Pour la documentation du router, et la description précise des routes présentes et de leurs caractéristiques majeures (comme le format de données qu'elles acceptent, leurs réponses, etc.), une documentation auto-formattée par Swagger est donc disponible sur la route : /v1/ quand l'API est en production (sur le nom de domaine localhost).

Cette implémentation, mise à jour au fil du développement du router, permet ainsi d'avoir une idée détaillée des routes présentes et exploitables pour le front à tout moment.

L'API possède 29 routes opérationnelles :

The screenshot shows the Swagger UI interface for an API titled "Les gardiens de la légende" version 1.0.0. The base URL is set to "localhost:3000/". The interface includes a navigation bar with links like Applications, GitHub Guides, O-clock-NN-G..., APOTHEOSE, APO travail, PSQL, NPM, REACT, TITRE PRO, Autres favoris, and Liste de lecture. The main content area displays various API endpoints categorized by endpoint type (acceuil, connexion, inscription, articles) and method (GET, POST, PATCH, DELETE). Each endpoint is described with its path and a brief description. For example, the "acceuil" section has a single GET endpoint at "/v1/" with the description "Une magnifique documentation swagger :)". The "articles" section contains several methods: GET for "/v1/articles", POST for "/v1/articles", GET for "/v1/articles/:id", PATCH for "/v1/articles/:id", and DELETE for "/v1/articles/:id". The "Authorize" button is visible in the top right corner of the interface.

|   |  |
|---|--|
| <b>jeux</b> gestion des jeux  |  |
| <b>GET</b> <code>/v1/jeux</code> Affiche tous les jeux en base de donnée. Route mise en cache (REDIS)   |  |
| <b>POST</b> <code>/v1/jeux</code> Insère un jeu en base de donnée. Route mise en flush (REDIS)*** Nécessite un rôle Membre***   |  |
| <b>GET</b> <code>/v1/jeux/:id</code> Affiche un jeu en base de donnée. Route mise en cache (REDIS)  |  |
| <b>PATCH</b> <code>/v1/jeux/:id</code> Mets à jour un jeu en base de donnée. Route mise en flush (REDIS)*** Nécessite un rôle Membre***   |  |
| <b>DELETE</b> <code>/v1/jeux/:id</code> Supprime un jeu en base de donnée. Route mise en flush (REDIS)*** Nécessite un rôle Membre***   |  |
| <b>evenement</b> gestion des événements   |  |
| <b>GET</b> <code>/v1/evenements</code> Affiche tous les événements en base de donnée. Route mise en cache (REDIS)   |  |
| <b>POST</b> <code>/v1/evenements</code> Insère un événement en base de donnée. Route mise en flush (REDIS)*** Nécessite un rôle Membre***   |  |
| <b>GET</b> <code>/v1/evenements/:id</code> Affiche un événement en base de donnée. Route mise en cache (REDIS)  |  |
| <b>PATCH</b> <code>/v1/evenements/:id</code> Mets à jour un événement en base de donnée. Route mise en flush (REDIS)*** Nécessite un rôle Membre***   |  |
| <b>DELETE</b> <code>/v1/evenements/:id</code> Supprime un événement en base de donnée. Route mise en flush (REDIS)*** Nécessite un rôle Membre***   |  |
| <b>Vérification du mail</b>   |  |
| <b>POST</b> <code>/resendEmail</code> Prend un mail en entrée et renvoie un email dessus si celui-ci est présent en BDD. Cliquer sur le lien dans l'email l'enverra sur la route 'llink'  |  |
| <b>GET</b> <code>/verifyEmail</code> Route qui réceptionne le lien de la validation du mail avec un token en query et valide le mail en BDD. Front géré par le serveur. Back power !  |  |
| <b>Changement du mot de passe</b>   |  |
| <b>POST</b> <code>/user/new_pwd</code> Prend un mail en entrée et renvoie un email dessus si celui-ci est présent en BDD. Cliquer sur le lien dans l'email l'enverra sur la route '/user/reset_pwd' ou l'attendent un formulaire            |  |
| <b>POST</b> <code>/user/reset_pwd</code> Reset du mot de passe. prend en entrée, newPassword, passwordConfirm et pseudo dans le body et userId et token en query: decode le token avec clé dynamique et modifie password (new hash + bdd) ! |  |
| <b>USER</b> gestion des utilisateurs  |  |
| <b>GET</b> <code>/v1/user</code> Affiche tous les utilisateurs en base de donnée. Route mise en cache (REDIS) *** Nécessite un rôle Admin***  |  |
| <b>GET</b> <code>/v1/user/:id</code> Affiche un utilisateur en base de donnée. Route mise en cache (REDIS)*** Nécessite un rôle Membre***   |  |
| <b>DELETE</b> <code>/v1/user/:id</code> Supprime un utilisateur en base de donnée. Route mise en flush (REDIS)*** Nécessite un rôle Admin***  |  |
| <b>PATCH</b> <code>/v1/user/:id</code> Modifie un utilisateur en base de donnée. Route mise en flush (REDIS)*** Nécessite un rôle Membre***   |  |
| <b>participant</b> gestion des participants   |  |
| <b>POST</b> <code>/v1/participants</code> Insère un participant en base de donnée. Route mise en flush (REDIS)*** Nécessite un rôle Membre***   |  |
| <b>PATCH</b> <code>/v1/participants</code> Annule une participation en base de donnée. Route mise en flush (REDIS)*** Nécessite un rôle Membre***   |  |
| <b>deconnexion</b> Pour se déconnecter  |  |
| <b>GET</b> <code>/v1/deconnexion</code> déconnecte un utilisateur - on reset les infos du user en session   |  |

## SPÉCIFICATIONS TECHNIQUES

### Le Versioning

Pour la programmation, on a utilisé Github pour sauvegarder notre code et centraliser les données. Avec comme organisation un monorepo et des branches : une branche Main toujours fonctionnelle représentant le code prêt à être mis en production, une branche dev-test et une dev-back, également fonctionnelle, dédiée au front ou au back, et enfin des branches par features. Les branches étaient mergées après une pull request validée. Git hub nous a également permis de nous essayer au code review, qui est un bon moyen d'obtenir un code plus performant.

Pour l'évolution de la base, on a utilisé Sqitch. Sqitch est un système de migrations permettant de versionner une base de données. Une migration Sqitch est l'équivalent d'un commit Git : elle recense les instructions SQL permettant de passer d'un état donné à l'état suivant (script SQL dans le dossier "deploy") ainsi que les instructions SQL pour revenir à l'état initial (script SQL dans le dossier "revert"). Il est aussi possible de fournir un script de test pour confirmer la validité du déploiement (dans le dossier "verify"). Une migration c'est donc 3 scripts SQL supplémentaire que l'on remplit après avoir fait la commande :

```
sqitch add <nom-de-la-migration> -n 'description'
```

Après seulement il nous est possible d'écrire nos script de déploiement que l'on pourra lancer avec la commande : sqitch deploy db:pg:<nom\_de\_la\_DB> et nos réversions si nécessaire seront toutes autant aisées avec la commande sqitch revert db:pg:<nom\_de\_la\_DB>.

L'efficacité de Sqitch réside dans sa simplicité. Un projet fonctionnant avec Sqitch possède seulement 2 petits fichiers indispensables :

- ❖ Un fichier "sqitch.conf" qui détaille la configuration du projet (SGBD utilisé et nom de la base de données, principalement)
- ❖ Un fichier "sqitch.plan" qui recense l'ordre des migrations du projet

Et toute modification sera détectée par Git et elles seront naturellement sauvegardées dans le prochain commit.

En bon Git Master, j'ai édité un pense bête pour l'équipe, afin de centraliser en un seul document les principales commandes github et Sqitch, nécessaire à tous dans ce projet.

Ce récapitulatif est disponible dans les annexes.

## Les technologies du Back

Pour la partie back nous avons utilisé l'environnement d'exécution Node.js créé par Ryan Dahl en 2009. Sur celui-ci, plusieurs autres technologies sont venues se greffer :

- ❖ PostgreSQL : comme SGBDR, pour la base de données.
- ❖ REDIS : base de données clé valeur, Nosql, très haute performance, utilisé pour la mise en cache de certaines routes et la mise en cache des sessions.
- ❖ Sqitch : système de versioning de BDD pour déployer plus facilement une BDD.
- ❖ PM2 : gestionnaire de processus Node.js, pour lancer notre serveur Express en production
- ❖ Express : Mise en place du serveur HTTPS
  - Joi : pour la validation de données d'une query ou d'un body.
  - Bcrypt : pour chiffrer et vérifier les mots de passe.
  - Express-sanitizer et Email-validator pour se protéger au mieux des attaques XSS.
  - Swagger : pour la documentation du router.
  - JWT : pour générer des tokens utilisés dans la création de liens temporaires.
  - NodeMailer : pour les envois d'emails.

## Les technologies du Front

Pour le front nous avons utilisé la librairie REACT avec :

- ❖ Webpack : pour une automatisation de tâches,
- ❖ Babel : Transpilation ES6/JSX -> ES5,
- ❖ React-redux : fait le lien entre le store et React, pour la gestion centralisée des états et des actions des composants react,
- ❖ Axios : pour faire les appels avec l'API,
- ❖ Redux : pour créer un store des variables dynamiques à modifier,
- ❖ SementiUI-React et SCSS pour le design,
- ❖ React-Router : pour afficher les pages sur une url sans recharger la page

## La sécurité de l'application

J'ai essayé d'être vigilant tout au long du projet à sécuriser l'API un minimum. En lien avec le top 10 des failles reconnues par le Open Web Application Security, j'ai essayé d'être vigilant contre 4 attaques fréquentes, à savoir : les failles XSS, CSRF, injection SQL et Brute-

Force. Dans cette partie à venir, je vous expliquerai rapidement ce en quoi consiste ces attaques, afin de mieux percevoir les stratégies mises en place pour s'en protéger.

1. Les attaques Cross Site Scripting (XSS), consistent à injecter du contenu dans une page, ce qui provoquera des actions de la part des navigateurs qui la liront. Pour s'en prémunir, nous avons suivi la règle NTUI, Never Trust User Input. C'est-à-dire la sécurisation de tout ce qui rentre dans l'API, avec un schéma Joi (toute donnée doit respecter un certain format défini par le schéma), le middleware Express-Sanitizer pour échapper tous les caractères spéciaux qui passent par le body et les query mais aussi Email-validator pour vérifier les formats des emails, et enfin nous avons mis en place une particularité d'express qui permet de mettre des mini regex sur nos routes avec params.
2. Les attaques Cross Site Request Forgery (CSRF), visent à faire faire une requête à l'insu d'un utilisateur, sur un site où il est déjà connecté. L'attaquant devra néanmoins connaître le endpoint et la query à modifier pour mener son attaque. Il est important de se rappeler qu'une propriété des navigateurs web est qu'ils incluent automatiquement et de manière invisible, tous les cookies utilisés par un domaine donné, dans toute demande web envoyée à ce domaine. Les attaquants exploitent ainsi cette propriété par des attaques CSRF puisque toute demande web effectuée par un navigateur inclura automatiquement tous les cookies (y compris les cookies de session et autres) créés lorsqu'une victime se connecte à un site Web. Pour s'en prémunir on envoie un token dans le local storage et on envoie ce même token dans un cookie (signé) supplémentaire. Lors d'une requête sur l'API, ces deux données devront être présentes, le token du local storage envoyés via le header et l'autre via le cookie. Si les deux tokens sont bien envoyés par ces deux biais distincts et qu'ils sont identiques en back, on en déduit l'absence d'attaque CSRF. Dans cette configuration, nous partons du principe que le local storage est peu sensible aux attaques CSRF, mais dans le cas où une attaque XSS aurait aboutit malgré nos défenses, alors celle-ci pourrait dérober le token du local storage et faire aboutir une attaque CSRF...

Afin de contrer efficacement ces deux attaques, un paramétrage précis des options de configuration pour les CORS et les cookies est également indispensable :

Configuration pour les cookies :

- ❖ Les cookies sont envoyés uniquement en HTTPS avec l'option "httpOnly: true" dans la configuration d'express-session.
- ❖ Les cookies sont envoyés uniquement sur HTTPS et pas au javascript client avec l'option "secure: true".
- ❖ L'envoi de cookie est interdit dans le cas d'un accès au site via un lien externe, avec l'option "SameSite: "strict"". En n'oubliant pas que les clients utilisant Firefox ne pourront pas bénéficier de cette option.
- ❖ L'utilisation de cookie signé, pour le cookie porteur du csrf token avec l'option "signed: true", (en oubliant pas de récupérer le cookie dans le MW d'authentification avec la méthode "req.signedCookies" dans ce cas..)

Configuration pour les Cross Origine Ressource Sharing (dans notre cas de figure avec l'utilisation du paquet NPM cors) :

- ❖ CORS : Cross Origine Ressource Sharing, le fait que d'autre site que notre front ne puisse pas interroger notre API, avec la valeur "origin: "mondomain.prod""", a un nom de domaine précis et unique correspondant à notre front. En local la valeur était "https://localhost:8080" sans oublier de passer également à true l'option "credentials", permettant d'envoyer des cookies et des entêtes d'autorisation et d'autoriser notre header à transporter notre xsrfToken avec "allowedHeaders : 'x-xxrsf-token'".
3. L'attaque par injection SQL est une attaque visant à injecter du langage SQL en vue d'essayer de communiquer directement avec la BDD. Pour s'en prémunir nous avons mis place des requêtes paramétrées dans nos modèles : "...WHERE id=\$1; [id]", où la valeur de la variable n'est pas disposée directement dans la query.
  4. L'attaque par Brute-Force vise à tester un très grand nombre de mots de passe pour tenter de trouver le bon. Pour s'en prémunir nous avons un module permettant de limiter, pour une adresse ip donnée, le nombre de tentative de connexion sur la route /connexion. Néanmoins, cette méthode serait contournable par un vpn qui pourrait changer d'adresse IP à volonté. Dans notre cas, le module est configuré pour permettre uniquement 100 connexions en 10 heures par adresse ip sur la route /connexion. Un mot de passe robuste a également été demandé à nos utilisateurs (8 caractères minimum, une lettre minuscule, une lettre majuscule, un nombre, et un caractère spécial parmi !\*#\$%^&). Il serait également préférable de ne lancer la règle des 100 connexions en 10h qu'après le premier essai infructueux uniquement.

Et d'une manière générale nous avons aussi opté pour certaines bonnes pratiques permettant d'accroître la sécurité générale de l'application :

- ❖ La mise en place du protocole HTTPS, avec un certificat auto-signé quand on travaille en développement via mkcert (pratique car accepté comme s'il était émis par une CA via les navigateurs) et signé avec une vrai CA (Certificate Authority) pour la mise en production, via Let's encrypt. Le protocole de sécurisation des données, entre la couche transport TCP/UDP et la couche application HTTP est le TLS 1.3 (dernière version du Transport Layer Security, remplaçant le Secure Socket Layer). L'API est également compatible HTTPS2 et Speedy (protocole de Google pour augmenter les capacités du http, notamment en multiplexant le transfert, intégré dans HTTP2). Seule l'API est en HTTP2, Webpack ne prenant pas en charge HTTP2 pour une version de Node.js supérieur à 10. Le front est resté en HTTPS1.
- ❖ Une configuration fine de nos en-têtes. Comme le fait de supprimer l'en-tête X-powered-by (pour que personnes ne sache qu'on utilise Express, et ainsi cibler les attaques), définir l'en-tête Content-Security-Policy pour la protection contre certaines attaques XSS et CSRF, imposer des connexions au serveur sécurisé via

- SSL/TLS via l'en-tête Strict-Transport-Security, etc. Pour en savoir plus, cf. page 44 "Jeu d'essai".
- ❖ Le rangement des variables sensibles et des "secrets" dans un fichier d'environnement. (.env).
  - ❖ Suivi et veille en termes de sécurité en auditant nos paquets npm et la configuration de nos headers. Pour en savoir plus, cf. page 48 "Veille sur les vulnérabilités de sécurité".
  - ❖ La mise à jour régulière des paquets qui le nécessitent.
  - ❖ Le stockage des mots de passe sous forme de hash en BDD.
  - ❖ L'utilisation d'algorithme de chiffrement sûr et reconnu, HS512 dans notre application via les JWT et SHA 256 pour l'algorithme de signature du certificat (DES, 3DES et fonction de hashage MD5, SHA1 sont à bannir, fonctionnalités obsolètes encore présentes dans le TLS 1.2).
  - ❖ Une clé secrète complexe, à plus de 100 caractères.
  - ❖ Vérifier l'intégrité des sous-ressources (pour être certain que les CDN reçus sont intègres) en front avec un plugin Webpack, et en back avec l'attribut nonce (dynamique) des balises styles dans les vues.
  - ❖ Un rôle utilisateur Postgres sur le serveur de production avec des droits réduits en accès sur la BDD.
  - ❖ Un backup régulier de la BDD via une commande pg\_dump associé à une tâche CRON qui envoie, via un email (en utilisant les paquets linux postfix mailutils) la BDD dans un fichier compressé et chiffré. Le script associé à la tâche cron écrit est disponible en Annexe.

## Autres services utilisés

Amazon Web Service a été utilisé pour la mise en ligne.

## ORGANISATION DE L'ÉQUIPE

### Présentation de l'équipe

Comme vu précédemment, nous avons respecté la méthodologie Agile scrum.

Dans cette méthodologie, chaque membre du groupe doit s'attribuer un ou plusieurs rôles (en plus de celui de développeur).

Nous avons donc Franck, le Product Owner, membre du club Les gardiens de la légende, qui connaît le produit et représente les intérêts et les besoins du clubs (purement fonctionnels, pas techniques).

Il tranche en cas de conflits fonctionnels, il a été le porteur du projet.

Laura en tant que Scrum master qui a été la garante de la méthode du projet : elle a géré le respect des conventions définies dans le groupe et notamment le bon remplissage du journal de bord par toute l'équipe et elle a animée la réunion du matin qui gère l'avancement du projet. De manière générale, elle s'est assurée que toutes les tâches étaient bien attribuées, suivies et accomplies.

Lorène a été notre Lead dev, elle a choisi les orientations importantes et choix techniques importants. Elle a eu le rôle de référent technique.

Pour ma part, j'ai été Git master, garant du bon fonctionnement du versionning avec Git, créateur des branches au démarrage pour que tout le monde puisse travailler, vérifier les pull request avant les merges et régler les conflits de branches quand cela a été nécessaire. J'ai rédigé un mémo des principales commandes git et leur utilisées, ainsi qu'un mémo sur les principales commande Sqitch pour les deploy et les revert de la BDD. J'ai été le référent technique de l'API, ou pour les connexions front - back.

## Organisation de travail

Nous avons travaillé avec un rythme classique de 9h – 17h, avec une réunion matinale commune. Certains d'entre nous ont largement travaillé après ces horaires. Le matin était souvent consacré à régler certains bugs en commun de la veille non résolus. Nous nous réunissions sous Discord pour les réunions matinales et nous étions tous connectés à Slack durant notre journée de travail pour des questions et coups de mains divers.

## Outils utilisés

Nous avons utilisé de nombreux outils au cours de nos 4 sprints.

Les outils collaboratifs utilisés pour la gestion projet :

- ❖ Trello
- ❖ Google drive

Les outils utilisés pour la communication entre nous :

- ❖ Slack (plateforme collaborative adopté par O'cock) avec un canal dédié à notre groupe de 4 personnes
- ❖ Discord, (salle de réunion vocale) pour les rendez-vous en vocal et la réunion matinale.

Les outils que j'ai utilisés pour coder :

- ❖ VSCode comme IDE
- ❖ Git et github pour le versionning et la sauvegarde du code

- ❖ Postman pour tester mon API sous tous ses angles avec envoi de cookies et headers
- ❖ PGadmin4 pour écrire et tester du SQL avant de l'injecter dans les modèles, interagir directement avec la BDD, vérifier le contenu des tables

Les outils que j'ai utilisé pour l'hébergement :

- ❖ Amazon Web Service, avec une instance EC2 pour la mise en production
- ❖ Un hébergement de certains fichiers statiques (photos) sur mon espace personnel en ligne.

## Répartition et détail des rôles

Le premier sprint a été réalisé en commun, constamment en direct sur Discord entre nous. Nous avons pris le temps de bien posé les bases, d'élaborer ensemble les wireframes, le MCD, MLD, les outils liés au projet, les trello, etc. Le script DDL de création de la BDD a été réalisé en commun avec Franck, Lorène et moi-même.

Pour les 2 autres sprints, lié à la rédaction du code nous avons travaillé de manière séparée :

Laura, seule à avoir fait la spécialité React, s'est consacrée au développement du front en consommant les endpoints que nous lui mettions à disposition.

Franck a secondé Laura dans le front, et s'est également chargé de la mise en production (à cette époque via surge et heroku pour la présentation du projet à l'école), et de l'installation de la carte google map.

Laurène et moi-même nous nous sommes chargés de l'API.

Laurene avec ces précieuses compétences en SQL, a réalisé de nombreux modèles, fonctions SQL et schémas de validation Joi. Tandis que j'ai pris en charge les contrôleurs pour l'authentification, et la gestion des utilisateurs de leur authentification (via les MW, leurs schéma Joi, au modèle), la gestion du cache avec REDIS, la configuration de la sécurité de l'api dans son ensemble et la connexion entre le front et le back notamment avec la gestion d'Axios en front pour qu'il envoie / reçoive / dispatche ce que je souhaitais en back.

Pour le dernier sprint consacré au débogage, nous avons travaillé principalement ensemble, à finir le CSS, tester les parcours utilisateurs et corriger ce qui ne fonctionnait pas.

## Réalisations personnelles

*Dans les extraits de code ci-dessous, la JS doc, les commentaires et certains console.log ont été retirés afin de rendre ces extraits plus compacts en vue de l'impression. Par ailleurs, l'indentation a été optimisé pour réduire le nombre de lignes ne contenant qu'une parenthèse et/ou accolade. Ils restent à votre entière disposition en format original dans VSCode le jour de l'examen.*

Sprint 0 – Construction du Cahier des charges – MCD / MLD + réalisation de wireframe.

Dans cette première partie, j'ai réalisé 4 wireframes, ceux dédiés à l'interface administrateur, pour la gestion desktop et mobile ainsi que ceux dédiés à la présentation du club, la page "A propos des gardiens". J'ai réalisé également la structure de l'API et du repo pour que tout le monde puisse commencer à coder en important une structure d'API sous serveur Express basique que j'avais déjà réalisé auparavant. Sous GitHub j'ai créé 4 branches, une "main", dédiée à du code clean et prêt pour la production, une branche "dev\_front" dédiée à héberger le code fonctionnel de la partie front, une branche "dev\_back" dédiée à accueillir le code fonctionnel de l'API, et une branche "dev\_test", visant à accueillir le nouveau code provenant du front et du back réunis. Chacun codant sur une branche portant le nom de la fonctionnalité qu'il/elle développait. Un récapitulatif des commandes GitHub et Sqitch a été édité et mis à disposition dans le Google Drive du groupe.

Nous avons par la suite tous réalisé des User Stories chacun de notre côté et une fois terminé nous avons partagé nos User Stories afin d'être certains que toutes les fonctionnalités que l'on souhaitait voir émerger de ce projet soient présentes.

Le MCD et le MLD ont été réalisé en commun via Discord et Mocodo Online.

Durant ce sprint, nous avons également fait en sorte que tout le monde ait les mêmes outils installés sur son ordinateur, (afin de faciliter l'aide et le support des autres, avec des commandes identiques) et nous avons guidé Laura pour qu'elle installe REDIS, PSQL et Sqitch sur sa machine virtuelle.

Pour l'API, Lorène, Franck et moi-même avons fait le choix d'une architecture avec Active Record ou c'est chaque classe de modèle qui va gérer sa propre logique de CRUD (accès multiple à la BDD et évite au passage un DataMapper avec une taille considérable...) et de ne pas utiliser d'ORM, pouvant ainsi prendre pleinement la main lors de nos requêtes SQL.

Durant ce sprint 0, j'ai également passé beaucoup de temps à comprendre le framework Oauth2.0 (et lire le RCF6749) pour échanger avec l'API Boardgame Atlas (API sécurisé permettant d'avoir accès à un très grand nombre de jeux de société et leurs caractéristiques), api gratuite mais sécurisé avec la notion de JWT (access token et refresh token) et les 4 rôles que définit ce protocole (détenteur de la ressource, serveur de ressource, le client, et le serveur d'autorisation). Après avoir réussi à comprendre et configurer correctement la possibilité de faire des appels sur cette api, le service est devenu payant la semaine qui a suivi (25\$ / mois), coupant court à toute tentative d'utilisation.

Sprint 1 – Mise en place des accès CRUD et du mécanisme d'inscription

Durant ce sprint 1, j'ai participé à la mise en place de la BDD, la création du script DDL, le fichier de seeding, et la recherche d'information sur un design. Je me suis également concentré à mettre en place la majorité des routes présentes et à gérer le formulaire d'inscription.

Mise en place du userController et du modèle associé :

Dans un premier temps j'ai mis en place le userController qui possède les méthodes dédiées aux utilisateurs pour les opérations classiques CRUD, de manière asynchrone avec une méthode asynchrone contenant un ou plusieurs await, (la fonction qu'on await va attendre toutes les fonctions asynchrone qu'elle contient) permettant de traiter en parallèle d'autres tâches. Et on l'encadre avec un try – catch pour obtenir l'erreur de la fonction asynchrone. Une fois l'information récupérée du modèle, si pas d'erreur, l'information est envoyée en format json avec un statut 200 indiquant que tout s'est bien passé.

```
● ● ●  
1  getAllUser: async (req, res) => {  
2    try {  
3      const users = await User.findAll();  
4  
5      res.status(200).json(users);  
6    } catch (error) {  
7      console.trace('Erreur dans la méthode getAllUser du userController :',  
8        error);  
9      res.status(500).json(error.message);  
10     }  
}
```

#### Méthode getAllUser du userController

Tandis que dans le modèle, qui va interroger notre BDD, on déclare la classe puis les propriétés de cette classe, allant de l'id à verifyemail pour nos users. Nos setters (accesseurs) permettent de passer du snake-case du langage sql, à l'écriture camel-case de javascript.

La méthode constructor permet de passer des arguments à la création et ainsi d'appeler new User avec des arguments (la méthode constructor est toujours présente mais l'appeler nous permet de lui passer des arguments), et ainsi créer une nouvelle instance à partir de notre classe User. On utilise une boucle for in (pour des objets énumérables) afin de boucler sur chaque propriété, et pour stocker les valeurs reçues en arguments dans l'objet final, on utilise le mot clé this pour y accéder. Ce qui nous permet de désigner l'object User en cours de "fabrication" (dans ce contexte).

```

1  class User {
2    id;
3    firstName;
4    lastName;
5    pseudo;
6    emailAddress;
7    password;
8    inscription;
9    avatar;
10   verifyemail;
11
12   set first_name(val) {
13     this.firstName = val;
14   }
15   set last_name(val) {
16     this.lastName = val;
17   }
18   set email_address(val) {
19     this.emailAddress = val;
20   }
21 /**
22 * @constructor
23 */
24 constructor(data = {}) {
25   for (const prop in data) {
26     this[prop] = data[prop];
27   }
28 }

```

Et on déclare nos méthodes d'instance sans utiliser de fonction fléchée pour des notions de portée, puisque this dans la méthode d'instance représente l'instance courante, alors que dans une méthode statique, elle représente la classe elle-même. Le mot static devant async indique une méthode statique, indiquant une méthode non pas rattachée à l'objet fabriqué mais à la class User elle-même, et elle sera partagée avec toutes ces instances. Avec await db.query j'attends le retour de ma requête sql lancé à ma BDD via le connecteur db qui va faire le lien avec ma BDD. On notera au passage la présence de requêtes paramétrées pour se prémunir de toutes attaques par injection sql. Query est une méthode de db, elle-même une nouvelle instance de Pool extrait du module "pg", dans le fichier database. S'il n'y a pas de ligne de résultats, donc si l'index 0 n'existe pas (s'il y en a au moins un qui existe, c'est celui-là), on l'envoie dans une nouvelle instance de la classe Error, avec son message en argument. Si tout va bien, Map va pour chaque élément du tableau, renvoyer un nouvel objet. Et pour chaque objet (utilisateur en BDD), on va créer une instance de User qui contient les infos de la BDD.

### Constructor et accesseur du user modèle

```

1  static async findAll() {
2    const {
3      rows
4    } = await db.query('SELECT * FROM "user" ');
5
6    if (!rows[0]) {
7      throw new Error("Aucun user dans la BDD");
8    }
9    console.log(
10      `les informations des ${rows.length} users a été demandé !`
11    );
12
13    return rows.map((user) => new User(user));
14  }

```

### Méthode static findAll dans le user modèle

On étudie le sens de l'information pour déterminer si on utilise une méthode d'instance ou une méthode statique : si on part de JS pour mettre à jour la BDD, alors on utilise une méthode d'instance. Si on part de la BDD pour mettre à jour des infos dans JS, ce sera une méthode statique. C'est pourquoi, seulement la méthode save, update, updatePwd et delete ne sont pas statiques, ce sont les seules méthodes d'instance qui mettent à jour la BDD. Toute les autres sont en statique.

```
● ● ●  
1  async save() {  
2    const {  
3      rows,  
4    } = await db.query(  
5      `INSERT INTO "user" (first_name, last_name, pseudo, email_address, password) VALUES ($1,$2,$3,$4,$5) RETURNING *;`,  
6      [this.firstName, this.lastName, this.pseudo, this.emailAddress, this.password]  
7    );  
8    this.id = rows[0].id;  
9    this.inscription = rows[0].inscription;  
10   console.log(  
11     `l'user ${this.id} avec comme nom ${this.firstName} ${this.lastName} a été inséré à la date du ${this.inscription} !`  
12  );  
};
```

#### méthode d'instance : save() du model user

Ce qui caractérise nos différentes opérations du CRUD, ce sont les différentes requêtes sql dans nos méthodes qui vont interagir de manière différente avec la BDD :

- ❖ INSERT INTO... pour insérer en BDD => Create
- ❖ SELECT \* FROM... pour aller chercher une info en BDD => Read
- ❖ UPDATE... pour mettre à jour la BDD => Update
- ❖ DELETE FROM... pour supprimer un enregistrement de la BDD => Delete

La logique entre les autres controllers et leurs modèles est identiques à celle décrite précédemment.

Maintenant que nous avons décrit les principales opérations de base que j'ai mises en place pour que notre userController interagisse avec notre BDD, voyons plus en détail la méthode handleSignupForm, du userController, qui va prendre en charge l'inscription en BDD d'un utilisateur qui souhaiterais s'inscrire. Cette méthode reçoit du schéma de validation Joi (middleware présent en amont du controller, pour s'assurer d'un format prédéfini, strict, des données reçues), 6 informations dans le body. Après avoir remplacé 4 d'entre elles par le résultat de la méthode sanitize (du paquet express-sanitizer, qui échappent tout caractères non autorisés), on réalise une ultime vérification concernant l'email reçu via le package email-validator spécialisé pour les emails (qui vérifie également les noms de domaines). A ce stade, nous sommes certains que les données envoyées sont valides. Mais certains points restent à vérifier :

- ❖ On vérifie que les deux mots de passe rentrés par l'utilisateur sont bien identiques avant de faire des appels à la BDD.

```
1 handleSignupForm: async (request, response) => {
2   try {
3     const firstName = request.sanitize(request.body.firstName);
4     const lastName = request.sanitize(request.body.lastName);
5     const pseudo = request.sanitize(request.body.pseudo);
6     const email = request.sanitize(request.body.emailAddress);
7
8     if (!validator.validate(email)) {
9       return response.json('Le format de l\'email est incorrect');
10    }
11    if (request.body.password !== request.body.passwordConfirm) {
12      return response.json(
13        'La confirmation du mot de passe est incorrecte');
14  }
15}
16
17
18
19
20
21
22
```

- ❖ Avec la méthode findByEmail de la classe User on vérifie que l'email proposé n'existe pas déjà en BDD, et on fait de même avec le pseudo proposé via findByPseudo. On retourne une erreur en json si c'est déjà le cas, avec un message clair.
- ❖ Si pas d'erreur, on convertit le mot de passe de notre utilisateur en hash via bcrypt.

```
1 const userInDb = await User.findByEmail(email);
2
3 if (userInDb.emailAddress) {
4   return response.json('Cet email n\'est pas disponible');
5 }
6 const pseudoInDb = await User.findByPseudo(pseudo);
7
8 if (pseudoInDb.pseudo) {
9   return response.json('Ce pseudo n\'est pas disponible');
10}
11 const hashedPwd = await bcrypt.hash(request.body.password, 10);
12 console.log(request.body.password, 'est devenu', hashedPwd);
13
14 const newUser = {
15   pseudo: pseudo,
16   emailAddress: email,
17   password: hashedPwd,
18   lastName: lastName,
19   firstName: firstName,
20 }
21 const userNowInDb = new User(newUser);
22 await userNowInDb.save();
```

- ❖ Je construis un objet qui reprend toutes les propriétés de notre nouvel utilisateur.
- ❖ Je créer une nouvelle instance de User qui possède désormais la méthode save que je peux utiliser.
- ❖ J'envoie les infos à la méthode d'instance save de manière asynchrone. Notre utilisateur est bien inscrit en BDD désormais.

- ❖ On va désormais envoyer un lien par email à l'utilisateur pour vérifier son email rentré en BDD récemment. Sans cette validation, notre nouvel utilisateur ne pourra pas se connecter.
  - On crée un Json Web Token JWT avec une date d'expiration de 24h, JWT que l'on aura pris soin de le construire avec l'option "issuer" égale à userId et l'option audience égal à la phrase de notre choix



```

1 const jwtOptions = {
2   issuer: userNowInDb.pseudo,
3   audience: 'Les gardiens de la légende',
4   algorithm: 'HS512',
5   expiresIn: '24h' };
6 const jwtContent = {
7   userId: userNowInDb.id,};
8 const newToken = jsonwebtoken.sign(jwtContent, jwtSecret, jwtOptions);

```

- On l'insère dans une variable "link" qui représente un lien dynamique qui inclut donc en query l'identifiant d'un utilisateur et un JWT signé. Ce lien, quand notre utilisateur cliquera dessus, ira chercher un endpoint que nous avons créé.
- J'utilise le transporter nodemailer pour envoyer un email personnalisé, que j'ai pris soin de mettre dans une fonction async, pour ne pas bloquer le flux.

```

async function main() {
  const host = request.get('host');
  const link = `https://${host}/v1/verifyEmail?userId=${userNowInDb.id}&token=${newToken}`;
  const transporter = nodemailer.createTransport({
    host: 'smtp.gmail.com',
    port: 465,
    secure: true,
    auth: {
      user: process.env.EMAIL,
      pass: process.env.PASSWORD_EMAIL,
    },
  });
  const info = await transporter.sendMail({
    from: 'lesgardiensdelalegende@gmail.com',
    to: `${userNowInDb.emailAddress}`,
    subject: `Les gardiens de la légende : merci de confirmer votre email`, // le sujet du mail
    text: `Bonjour ${userNowInDb.firstName} ${userNowInDb.lastName}, merci de cliquer sur le lien pour vérifier votre email auprès du club de jeu Les gardiens de la légende.`,
    html: emailheaderVerify + `

### Bonjour <span class="username"> ${userNowInDb.firstName} ${userNowInDb.lastName}, </span> </h3> <br> <p>Vous souhaitez vous inscrire au club de jeux des gardiens de la legende.</p> <br> <p>Merci de cliquer sur le lien pour vérifier votre email auprès du club de jeu Les gardiens de la légende. </p> <br> <a href="${link}">cliquez ici pour vérifier votre email. </a> <br>` + emailfooterVerify, }); main().catch(console.error); }


```

- ❖ On renvoie un message au front, lui donnant des informations sur notre nouvel utilisateur. Dans notre BDD nous avons bien pris soin de mettre la valeur de verifyemail à false par défaut. Tant que notre user ne clique pas sur le lien envoyé à son adresse email, la valeur de verifyemail restera à false, lui interdisant toute connexion sur la route /connexion de notre api.

Dans l'hypothèse où l'utilisateur clique sur le lien envoyé :

- ❖ Il va venir sur notre endpoint /verifyEmail avec dans la query, un userId et un JWT, préfiltré par le schéma de validation Joi que j'ai mis en amont.
- ❖ Je vérifie si l'id passé en query correspond bien à l'identifiant d'un utilisateur en BDD.
- ❖ Je vérifie la validité du JWT fourni, et si l'issuer, donc le pseudo, qui possède l'email à valider, correspond bien au pseudo de l'utilisateur qui est dans la query, et que le JWT n'est pas expiré.



```

1 verifyEmail: async (req, res, err) => {
2   try {
3     const { userId, token } = req.query;
4     const userInDb = await User.findOne(userId);
5     const decodedToken = await jsonwebtoken.verify(token, jwtSecret,
6       { audience: 'Les gardiens de la légende', issuer: `${userInDb.pseudo}` }, function (err, decoded) {
7       if (err) {
8         res.json("la validation de votre email a échoué", err)
9       return decoded;
10      }

```

- ❖ Je vérifie que son email n'est pas déjà validé.
- ❖ Alors dans ce cas, j'utilise la méthode emailverified du model que j'ai codé pour faire un UPDATE en sql, de la colonne verifyemail en BDD, pour la passer à true.



```

1 if (userInDb.verifyemail) {
2   res.status(200).render('verifyEmail', {userInDb});
3 } else if (!decodedToken.userId === userInDb.id && decodedToken.iss === userInDb.pseudo) {
4   console.log(`une erreur est apparue =>`, err)
5   return res.status(401).render('verifyEmailFail', {userInDb});
6 } else {
7   await User.emailverified(userInDb.id);
8   res.status(200).render('verifyEmailWin', {userInDb});
9 }
10 } catch (error) {
11   console.trace(
12     'Erreur dans la méthode verifyEmail du userController :',
13     error);
14   res.status(500).json(error.message);}}

```

- ❖ Je renvoie une vue au front confirmant que l'utilisateur a bien été connecté.

L'utilisateur a fini la procédure d'inscription et peut désormais se connecter. Si l'utilisateur n'a pas validé son email dans les 24h, le JWT ne sera jamais validé, mais je lui ai mis une autre méthode à disposition pour qu'il puisse vérifier son email : la route /resendEmailLink qui renvoie également sur /verifyEmail.

Sprint 2 – Authentification, mise en place de la méthode reset password,

*Dans ce prochain paragraphe, je vous ferai part de ma réflexion qui dans un premier temps m'a fait installer une méthode d'authentification basée sur le JWT, méthode qui a été implémentée jusqu'au milieu du sprint 3. Puis suite à d'autres lectures, mon point de vue a évolué et un mécanisme de session avec cookie stocké côté serveur a été mis en place.*

Avec ma petite expérience de la gestion des tokens et refresh tokens présents dans le framework Oauth2.0 (cf. page25, "sprint 0"), je me suis tourné vers une authentification via JWT, pour l'authentification des utilisateurs. En voyant les avantages des JWT intéressants pour notre projet :

- ❖ Ils n'ont pas à être stockés côté serveur, le client s'en charge.
- ❖ Ils sont peu gourmands en mémoire pour le serveur.
- ❖ Le JWT stocké dans le local Storage en front nous préserve d'une attaque CSRF, ce que ne fait pas un cookie envoyé automatiquement.
- ❖ Une authentification robuste et sécurisée, et qui, associée à l'envoi d'un refresh token, n'impose pas à l'utilisateur de se reconnecter à nouveau.
- ❖ Associée au package JWT-permission, la gestion des droits est très facile à implémenter.

C'est pourquoi, durant le sprint 2, la méthode d'authentification a été le JWT, avec un deploy Sqitch pour insérer une nouvelle table permettant de stocker le refresh token, (un simple crypt.randomByte) en BDD. Tandis que la gestion des droits était gérée par le paquet Express-JWT-permission, me permettant d'utiliser ces méthodes, avec guard.check('admin') par exemple pour restreindre l'accès d'une route au propriétaire d'un rôle précis.

Néanmoins j'avais sous-estimé l'importance de gérer l'intégralité du cycle de vie du token. Et mes premières interrogations sur la pertinence de cette méthode dans notre application, se sont révélées lors que j'ai souhaité gérer la révocation de la session. Jusqu'à présent, pour la déconnexion, le front s'occupait de détruire le token du local storage. Néanmoins, et même si j'avais défini un access token avec un temps d'expiration court, pour un refresh token avec un temps de vie long, si le local storage avait fait l'objet d'une attaque, ou si l'utilisateur l'avait sauvegardé, j'étais dans l'incapacité de lui refuser l'accès à mon API. La solution aurait pu consister à stocker les access tokens dans Redis, mais la gestion des refresh tokens ne faisant déjà pas partie de l'implémentation standard, cela me paraissait

une architecture bien plus volumineuse et contraignante que prévu à mettre en place. De plus, d'autres inconvénients se sont manifestés :

- ❖ Dans le cas où un utilisateur veut changer de mot de passe avec une authentification effectuée juste avant, le JWT aurait toujours été valide. Il ne pouvait certes plus se connecter avec son ancien mot de passe sur la page de connexion, néanmoins, le JWT généré avec l'ancien mot de passe sera toujours valide.
- ❖ Sans stockage des access tokens en BDD, pour qu'un utilisateur soit réellement bloqué par le serveur sur les routes où les ressources sont protégées, j'aurais dû attendre que le JWT expire.
- ❖ Comparé à un stockage de session sur Redis (qui peut avoir des délais quasi infimes de 5ms), il semblerait que les JWT, plus volumineux en octet, consomment plus de ressources du processeur pour calculer les signatures cryptographiques et soient en pratique, beaucoup plus lents que des sessions traditionnelles.
- ❖ Peu facilement compréhensible par les autres membres du groupe si on ne se plonge pas un peu dans le cycle de fonctionnement du token et du refresh token.

Et au final, pour un mécanisme que l'on dit stateless (pas besoin du serveur), il y avait beaucoup de choses à configurer dans le serveur pour une implémentation correcte ...

L'utilité d'un tel mécanisme, semble beaucoup résider dans le cas d'utilisation d'un même compte utilisateur sur plusieurs plateformes, ce qui pour notre club de jeux, n'a que peu d'intérêt actuellement. Par ailleurs, express-session fonctionne en production depuis des années sur des milliers de serveurs, bien utilisé et configuré, tout me pousse à penser que c'est une valeur sûre, or, l'implémentation que j'allais installer semblait en fin de compte se rapprocher d'une session classique comme pourrait le faire express-session avec cookie.

Au vue de la petite taille de notre application, de son architecture modeste, d'une sécurité complémentaire qu'il était possible de mettre en place pour lutter contre les attaque CSRF, de la possibilité de stocker les sessions via Redis, de la possibilité de gérer plus facilement l'authentification sans avoir à ajouter de tables à la BDD, de la robustesse d'express-session bien implanté, je suis revenu à un système de session avec cookie stocké côté serveur. Un mécanisme plutôt sûr, permettant de révoquer un utilisateur à tout moment, sécurisé avec des options robustes (décris dans la partie sécurité de l'application), et également plus simple à comprendre pour toute l'équipe, ce qui était également important pour moi.

Bien entendu, les MW pour la gestion des droits ont également évolué pour que l'on conserve une protection contre les attaques CSRF, la méthode est la suivante :

- ❖ Je vérifie la présence dans l'objet req d'un cookie signé, envoyé, par express lors de la connexion, renvoyé par le navigateur de l'utilisateur.

- ❖ Après récupération du header par déstructuration, Je récupère mon custom header qui a comme en tête "x-xsrf-token", un token qui provient du local storage de l'utilisateur. Au préalable, lors de la connexion, je l'ai envoyé dans le body et mis dans le local storage via "localStorage.setItem('xsrfToken', response.data.xsrfToken)".
- ❖ Je compare les deux. S'ils ne sont pas identiques, on pourrait être en présence d'une attaque CSRF, avec un cookie envoyé par le navigateur sans que l'utilisateur ne le veuille vraiment, on retourne une 401.

```

  ● ○ ●
1  const auth = async (req, res, next) => {
2    try {
3      const {
4        headers
5      } = req;
6      const cookieXsrfToken = req.signedCookies.xsrfToken;
7      if (!cookieXsrfToken) {
8        return res.status(401).json({
9          message: 'Il n\'y a pas de token dans le cookie de session'});
10     if (!headers || !headers['x-xsrf-token']) {
11       console.log(chalk.red('Il n\'y a pas de token CSRF dans le header (auth)'));
12       return res.status(401).json({
13         message: 'Il n\'y a pas de token CSRF dans le header'});
14     if (headerXsrfToken !== cookieXsrfToken) {
15       console.log(chalk.red('Problème de token csrf dans le auth MW'));
16       return res.status(401).json({
17         message: 'Problème de token csrf'});
18   }

```

- ❖ Je vérifie que mon utilisateur est bien connecté et a le bon rôle pour accéder à la ressource.
- ❖ Je laisse la main au MW suivant.

```

  ● ○ ●
1  if (!req.session.user) {
2    return res.status(403).json({
3      message: 'Vous n\'avez pas les droits nécessaires pour accéder à la ressource (auth).'});
4    if (req.session.user.role !== 'Modérateur' && req.session.user.role !== 'Administrateur' && req.session.user.role !== 'Membre') {
5      return res.status(403).json({
6        message: 'Vous n\'avez pas les droits nécessaires pour accéder à la ressource (auth).'});
7    next();
8  } catch (err) {
9    return res.status(500).json({
10      message: 'Erreur lors de l\'authentification'});
11 module.exports = auth;

```

Enfin, durant ce sprint, un des derniers passages qui a nécessité une certaine réflexion sur la méthode à utiliser, a été la mise en place d'une méthode pour l'utilisateur, de renouveler son mot de passe. Le mécanisme, en deux temps, est composé d'une première route reliée à un formulaire en front, qui renvoie une adresse email à l'API.

Si celle-ci appartient bien à un utilisateur de la BDD, un email est envoyé à l'utilisateur, lui offrant l'accès à une route qui le renvoie vers un formulaire en front pour rentrer son nouveau mot de passe (et sa confirmation). Et dans un second temps, ces infos

sont transmises à l'API sur une seconde route, qui aura pour but d'insérer ce nouveau mot de passe en BDD.

Le problème cette fois ci (comparé à faire un lien expirable pour la vérification des emails), était de pouvoir faire un lien dans l'email qui ne serait pas valide jusqu'à l'expiration du token, mais qui serait invalide à la seconde même où un utilisateur changerait son mot de passe en BDD (tout en mettant quand même, une date de validité). Ce qui signifierait que toute personne souhaitant utiliser ce lien à postériori serait bloqué. Pour ce faire, la solution a consisté à créer une clé secrète de JWT dynamique, et unique pour chaque utilisateur. Et pour ce faire, j'ai chiffré le JWT avec un secret composé du hash de son mot de passe actuel concaténé à sa date d'inscription sur le site. Cette clé secrète me permet de m'assurer qu'à la seconde où l'utilisateur changera son mot de passe dans la BDD, son nouveau hash écrasera l'ancien, qui est notre clé secrète. Le lien précédent devient caduque car le JWT ne pourra plus être déchiffré, et ainsi la route /reset\_pwd le renverra vers une 401.

Concaténer sa date d'inscription me permet de m'assurer, dans le cas où un utilisateur utiliserait un mot de passe unique pour plusieurs sites web, et que si son mot de passe aurait été découvert par un attaquant sur un autre site, absolument personne ne pourrait reconstituer la clé secrète car personne ne peut savoir l'heure et la seconde précise de création du compte de l'utilisateur.

Cette méthode est fonctionnelle en back, testé sous Postman, mais je ne l'ai pas encore implémentée en front. Ce sera fait dans les temps qui suivront.



```
1 new_pwd: async (req, res) => {
2   try {
3     const email = req.sanitize(req.body.emailAddress);
4     if (!validator.validate(email)) {
5       return res.json('Le format de l\'email est incorrect');
6     const userInDb = await User.findByEmail(email);
7     if (typeof userInDb.id === "undefined") {
8       res.status(404).json("cet email n'existe pas, assurez vous de l'avoir écrit correctement.");
9
10    const secret = `${userInDb.password}_${userInDb.inscription}`;
11    const jwtOptions = {
12      issuer: `${userInDb.pseudo}`, audience: 'envoyeretpwd', algorithm: 'HS512', expiresIn: '1h';
13    const jwtContent = {
14      userId: `${userInDb.id}`,
15      jti: userInDb.id + "_" + crypto.randomBytes(9).toString('base64'),};
16    const newToken = await jsonwebtoken.sign(jwtContent, secret, jwtOptions);
17    async function main() {
18      const host = req.get('host');
19      const link = `http://${host}/v1/user/new_pwd?userId=${userInDb.id}&token=${newToken}`;
```

Présentation de la première partie de la méthode new\_pwd.

### *Mise en place d'Axios*

Une partie du sprint 3 a été consacrée à faire correctement interagir l'API avec le front en paramétrant correctement Axios (et CORS).

Dans le fichier de configuration d'Axios (src/api/index.js), on définit la configuration par défaut de l'instance à sa création : notre url de base pour la connexion à l'API, le délai de réponse maximum et la possibilité d'envoyer et de recevoir des cookies :

```
● ● ●
1 import axios from 'axios';
2 export default axios.create({
3   baseURL: 'https://localhost:3000/v1/',
4   timeout: 10000,
5   withCredentials: true,
6 });


```

Puis dans le dossier middleware du front, chargé d'envoyer les requêtes Axios au back, on configure les requêtes sur des routes protégées en back par nos middlewares d'authentification avec l'envoi d'un header particulier (x-xsrf-token), qui contient le token que l'on est allé chercher dans le local storage. Pour l'envoi de la requête de connexion et d'inscription, ce header n'est pas présent, puisque le token n'est présent dans le local storage qu'après une connexion réussie. Ci-dessous un appel Axios dans le cas d'un envoi d'un nouvel article, sur une route sécurisée, qui nécessite donc un header particulier :

```
● ● ●
1 case SEND_ADD_ARTICLE: {
2   try {
3     const xsrfToken = localStorage.getItem('xsrfToken');
4     const options = {
5       mode: 'cors',
6       headers: {'x-xsrf-token': xsrfToken}, };
7     await axios.post('/articles', {
8       title: newTitle,
9       description: newDescription,
10      authorId: numberId,
11      tagId: newTagIdNumber, }, options);
12     store.dispatch(setAddNewArticle(true));
13   catch (error) {
14     store.dispatch(setError(error.response));}
15   return next(action);}
```

## Mise en place du cache via Redis

Afin de gagner du temps lors de la demande de certaines ressources à l'API, j'ai mis en place Redis pour du cache, avec un middleware composé de deux méthodes, une pour la mise en cache, et une pour vider les données (on parle de flush) afin que ce qu'il y a en cache dans Redis reflète bien ce qu'il y a dans PostgreSQL (on parle d'invalidation de cash événementiel et temporaire). De manière générale pour la mise en cache, je vais déjà vérifier s'il existe une clé dans Redis avec les données, si oui, je court-circuite PostgreSQL, je prends les données dans Redis et je les renvoie au front. S'il n'y a pas de données dans Redis, je vais les chercher dans PostgreSQL via Active Record (et les modèles), puis lors du renvoi de la réponse attendue au front, je récupère ces données au passage et je les stocke dans Redis, prêt à être envoyées depuis Redis à leur prochaine demande.

Voyons tout ça un peu plus en détail :

Pour connecter Redis à notre app, j'utilise un client Redis. Après avoir require Redis, on crée un client :

```
● ○ ●  
1 const {  
2   createClient  
3 } = require('redis');  
4 const client = createClient();
```

Les mots clés Redis sont disponibles sous la méthode du client Redis (del, get, set, etc...), par contre la V3 de Redis ne prenant pas en charge les promesses nativement, j'ai dû passer par "util.promisify" pour avoir des versions "promessifiées" des méthodes Redis. Promisify prend une fonction et retourne une fonction asynchrone qui retourne une promesse, (une version asynchrone de la fonction) mais ça retourne une fonction, pas la méthode, et du coup je perds "client", alors que les méthodes telles que "del", "get", etc, ont besoin des propriétés et de ce qu'il y a dans "client" pour fonctionner. J'utilise alors la méthode bind pour la relier à un contexte, pour que quand elle s'exécute, si elle a des mentions de "this" dans son code, celui-ci désignera "client", l'objet passé à bind.

```
● ○ ●  
1 const redis = {  
2   del: promisify(client.del).bind(client), // pour effacer une clé  
3   get: promisify(client.get).bind(client), // pour obtenir un clé  
4   set: promisify(client.set).bind(client), // pour insérer une nouvelle clé avec une nouvelle valeur  
5   setex: promisify(client.setex).bind(client), // je définis une clé avec un temps d'expiration (en sec)  
6   exists: promisify(client.exists).bind(client) // pour vérifier si une clé existe  
7 };  
8  
9 const keysIndex = new Set();
```

Les clés n'étant pas indexées dans Redis (Redis ne tient pas d'index des clés à jour), on va donc gérer nous même un index. La fonction client.keys ('\*') que l'on pourrait utiliser est très lente, et plus notre nombre de clés va grandir plus ce sera long. La commande Keys va parcourir toute la mémoire de Redis à la recherche de clés, ce qui pourrait être très long en cas de serveur en fonction depuis longtemps, avec un grand nombre de données. L'index des clés Redis va permettre de savoir à tout moment quelles sont les clés utilisées par notre application dans le cache de Redis. J'instancie un nouvel objet Set qui me permet de stocker des valeurs uniques pour créer mon index (comme un array mais avec une contrainte d'unicité). Un array n'aurait pas pu convenir ici, puisque on va avoir de nombreuses clés identiques, je ne veux surtout pas qu'il me stocke plusieurs valeurs avec la même clé...

Pour pouvoir continuer à utiliser le cache et nodemon ensemble en développement, j'ai dû créer un petit fichier de configuration pour nodemon (nodemon.json) à la racine, qui automatiquement, quand on démarre le back ou quand nodemon relance le back automatiquement, efface toutes les données dans Redis (ensemble clés - valeurs). Ainsi, on est toujours en adéquation entre ce qu'il y a dans notre index des clés (en js) et les clés réellement contenues dans Redis. Sinon Redis contient toujours des clés sans que notre index le sache, et s'il n'y pas de clés dans l'index, pas de flush qui boucle sur le Set d'index possible, puisqu'il est vide, et Redis retourne des données erronées, différentes de ce que contient PostgreSQL (Redis n'a pas pu flush et recharger avec des données fraîches).



```
1 {"events": {"restart":"redis-cli --scan --pattern 'mjc:' | xargs redis-cli DEL"}}
```

#### Le fichier nodemon.json

"Cache" et "flush" sont des méthodes de mon middleware, ils contiennent donc de manière universelle "request", "response" et "next".

Je définis une clé dynamique et paramétrable, qui correspond à chaque fois à une route différente via l'URL et l'option pour le préfix de la clé. Je vais chercher l'URL dans request, c'est une de ses propriétés (selon la doc d'Express). J'ai ici pris req.originalURL qui est l'URL demandée à la base mais attention à ne pas prendre req.baseUrl qui est l'URL sur laquelle j'ai monté le router (ici /v1, pour faciliter le passage vers une v2 si besoin à l'avenir). Je conserve la bonne pratique Redis qui consiste à mettre un préfix pour nos clés et je le définis comme un paramètre que je pourrai ainsi modifier à ma guise dans le router.

Si la clé existe, je la sors du registre et je réponds directement à l'utilisateur.

J'utilise await car tout ça retourne des promesses et je préfère attendre que Redis aie fait son travail, avant de renvoyer la réponse au front. Mon connecteur Redis va donc aller chercher la clé si elle existe. Redis ne stockant pas des objets mais des strings, on convertit en string avant de le stocker dans Redis et on reconvertis tout en json à la sortie de Redis pour le renvoyer à l'utilisateur en utilisant json.parse sur toutes nos données retournées (la version écrite est équivalente à : JSON.parse(await redis.get(theKey)); ).

```

1 const cacheGenerator = (options) => {
2
3   return {
4     cache: async (request, response, next) => {
5
6       const theKey = `${options.prefix}:${request.originalUrl}`;
7
8       if (await redis.exists(theKey)) {
9         const theValue = await redis.get(theKey).then(JSON.parse);
10        console.log(chalk.yellow(`la valeur ${theKey} est déjà dans Redis, on la renvoie depuis Redis`));
11        // et on répond directement à l'utilisateur
12        response.json(theValue);
13     } else {

```

Si la version n'existe pas dans le cache, on intercepte les données :

Je veux désormais que response.send, mette dans le cache et réponde, alors qu'avant cela ne faisait que répondre.

Je fais une copie de response.send et je le bind à "response" car je le stocke dans une fonction et non dans une méthode (pour préciser que quand j'utilise originalResponseSend il sera bindé à l'objet response). Je veux que cette fonction continue de fonctionner même si je la déracine de son contexte response. Après je redéfinis l'objet response.send comme étant un stockage dans le cache qui à la fin appelle la version originale de response.send. Ici, j'intercale une mise en cache avant de renvoyer response.send. Quand response.send sera exécuté dans le prochain middleware, c'est notre response.send qui sera en fait lancé.

Je garde mon index des clés à jour, en mettant cette nouvelle clé dans mon objet Set. Redis.setex permet de définir une clé avec une valeur (qui est notre variable theResponse) et un time to live (temps au delà duquel la donnée n'existe plus : invalidation temporaire) défini en option. La méthode redis.setex n'est pas await car sinon je dois async response.send et si je procède ainsi, je la dénature, ce que je ne veux pas, elle n'est pas par défaut async, donc pour la garder dans l'état, je ne l'async pas. Dans le cas où l'enregistrement du cache n'est pas terminé, si une deuxième requête arrive très rapidement, la clé n'existera pas encore et ça repassera par le "else" et response.send.

```

1 const originalResponseSend = response.send.bind(response);
2
3 response.send = (theResponse) => {
4
5   keysIndex.add(theKey);
6   redis.setex(theKey, options.ttl, theResponse);
7   console.log(chalk.yellow(`la valeur ${theKey} n'est pas dans Redis, on la renvoie depuis PostgreSQL`));
8   originalResponseSend(theResponse);
9
10  next();
11

```

Response.json applique JSON.stringify avant d'appeler response.send. En prenant response.send et non response.json, je n'ai plus besoin de JSON.stringify car c'est déjà appliqué.

Dès qu'on touche à la base de données (donc UPDATE, POST, PATCH, DELETE), on ne prend pas le risque de proposer un contenu périmé à l'utilisateur, aucune donnée invalide, je vide le cash via ma méthode "flush" :

```
● ○ ●  
1     flush: async (request, response, next) => {  
2  
3         for (const key of keysIndex) {  
4             await redis.del(key);  
5             console.log(chalk.hex('#16DE40')("on flush dans Redis"));  
6             keysIndex.delete(key);  
7         }  
8  
9         next();});};  
10  
11 module.exports = cacheGenerator;
```

Pour supprimer les données en cache, on parcourt les clés de notre index ("keysIndex"), on les supprime une par une et on supprime également la clé de l'index via la méthode "delete", toujours à chaque boucle.

## CSS

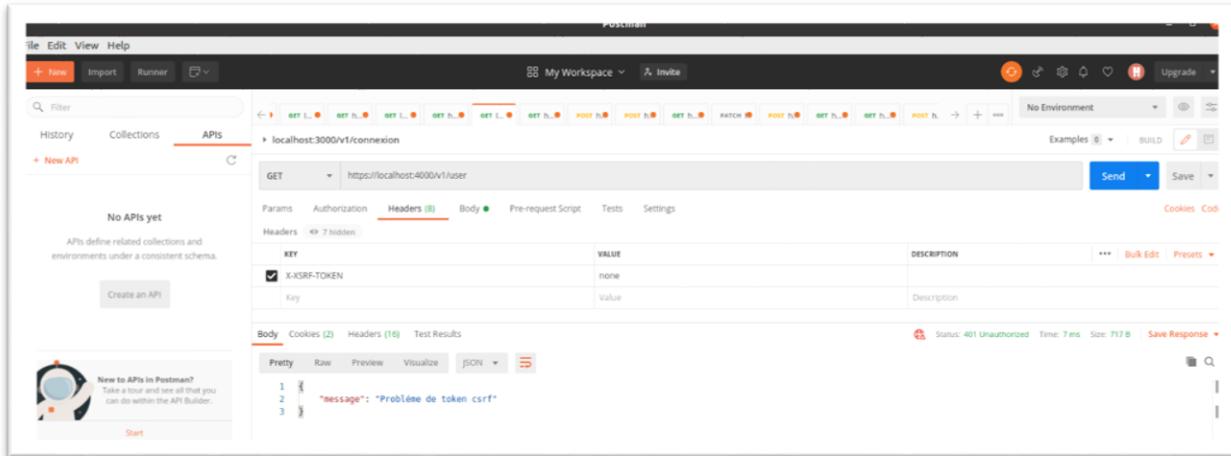
Enfin, une partie importante de ce sprint a consisté à finir le CSS du front, à enjoliver les formulaires, et les différentes pages du site en complétant les fichiers style.css des différents composants du front. Avec par exemple, la mise en place de media Query avec un breakpoint à 760 pixels de large avant de passer les balises div de la page contact en format colonne plutôt qu'en ligne via flexbox (technique utilisant des conteneurs à l'intérieur desquels on place plusieurs éléments) :

```
● ○ ●  
1 @media (min-width: 760px) {  
2     .contact_container{  
3         display: flex;  
4         flex-wrap: wrap;  
5         flex-direction: row;  
6     }
```

## Jeu d'essai

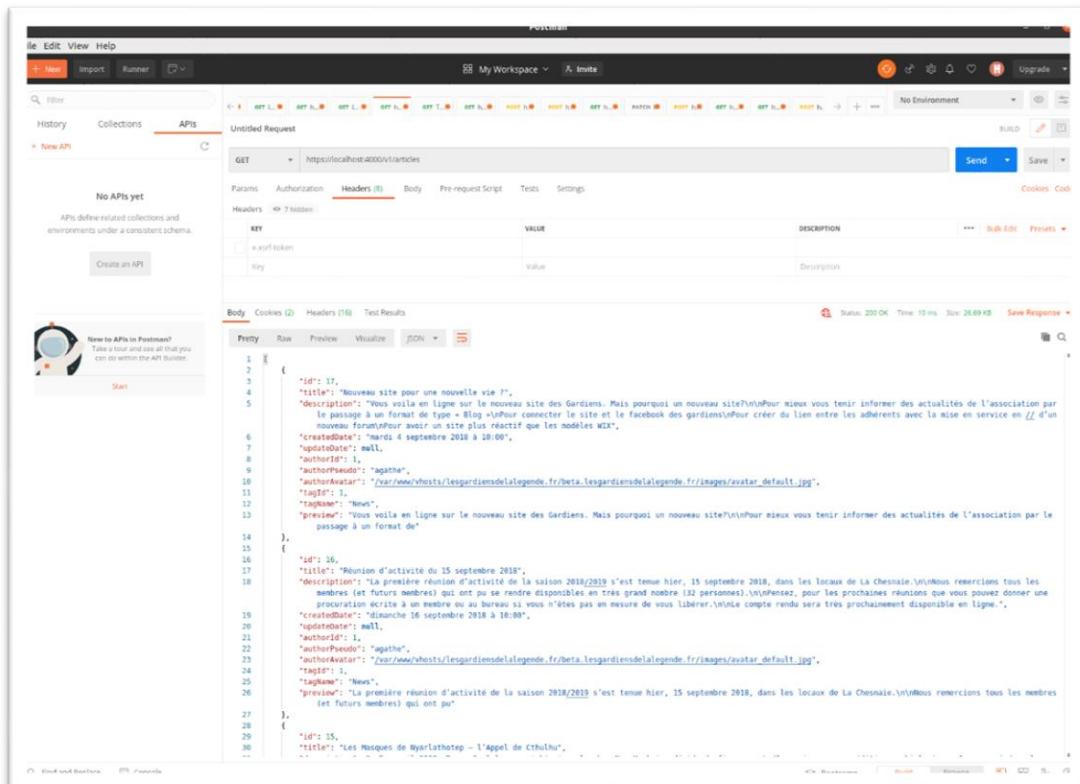
### Test divers via Postman

L'API contenant plus de routes que ne se que le front n'en utilise, Postman a été utilisé pour tester toutes les routes avec toutes les configurations possibles, en envoyant un header permettant de simuler une authentification avec un x-xsrf-token, et les cookies adéquats.



The screenshot shows a Postman interface with a failed connection attempt. The URL is `https://localhost:4000/v1/user`. The Headers tab shows a checked `X-XSRF-TOKEN` header with a value of `none`. The response status is 401 Unauthorized, with the message: "{'message': 'Problème de token csrf'}".

Test d'une connexion sans le header => nous sommes bien bloqués.



The screenshot shows a successful article retrieval. The URL is `https://localhost:4000/v1/articles`. The Headers tab shows an `x-xsrf-token` header with a value of `none`. The response status is 200 OK, with a JSON payload containing multiple articles. One article's preview is: "Vous voile en ligne sur le nouveau site des Gardiens. Mais pourquoi un nouveau site?\nPour mieux vous tenir informer des actualités de l'association par le passage à un format de type « Blog »\nPour connecter le site et le facebook des gardiens\nPour créer du lien entre les adhérents avec la mise en service en WIX d'un nouveau forum\nPour avoir un site plus réactif que les modèles WIX", and the date is "17 septembre 2018 à 10:00".

Test d'une récupération d'articles (route sans restriction) => nous sommes bien autorisés sans x-xsrf-token dans le header.

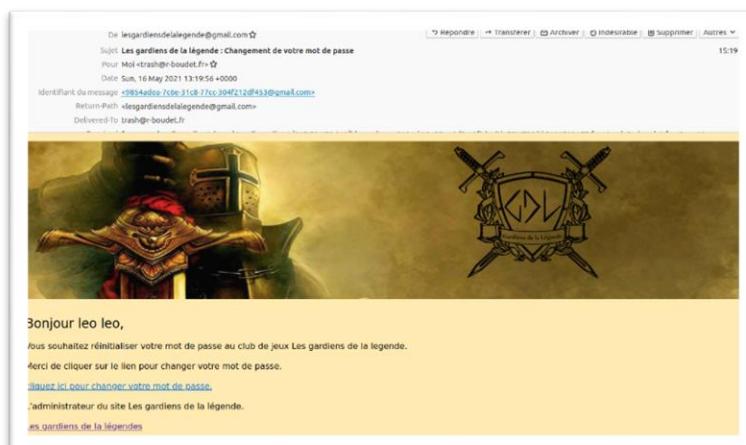
The screenshot shows the Postman interface with a failed API call. The URL is `https://localhost:4000/v1/user`. The Headers tab shows a `X-XSRF-TOKEN` header with a long value. The Body tab shows a JSON response with a single key `"message"` containing the string `"Vous n'avez pas les droits nécessaires pour accéder à la ressource."`. The status bar at the bottom indicates a 403 Forbidden status.

Test d'une récupération de données avec token mais sans les droits adéquats (bloqués par le MW admin) => nous sommes bien bloqués.

The screenshot shows the Postman interface with a successful API call. The URL is `https://localhost:4000/v1/user/reset_pwd`. The Headers tab shows an `emailAddress` header with the value `traph@boudet.fr`. The Body tab shows a JSON response with a single key `"message"` containing the string `"Merci de consulter vos emails et de cliquer sur le lien envoyé pour renouveler votre mot de passe."`. The status bar at the bottom indicates a 200 OK status.

Test de la méthode de reset du password, étape 1 : envoi d'un email et récupération d'un lien dans l'email pour renseigner son nouveau password.

**Donnée en entrée : email puis mot de passe. Donnée attendue : Changement du mot de passe.**



On récupère le lien envoyé par email

Etape 2 : envoi du body via Postman, avec le lien de connexion récupéré dans l'email.

Un email nous avertit de la bonne prise en compte du changement.

**Donnée obtenue : Notre mot de passe a changé en BDD.**

**Résultat : la donnée obtenue est identique à celle attendue.**

| Body                                | Cookies (2)  | Headers (19) | Test Results |
|-------------------------------------|--|--------------|--------------|
| KEY                                 | VALUE  |              |              |
| Content-Security-Policy ①           | default-src 'self'; img-src self filedn.eu; style-src 'nonce-fbc0424a-5baf-4df2-955f-52a62fa9a0c3'; upgr |              |              |
| X-DNS-Prefetch-Control ①            | on   |              |              |
| Expect-CT ①                         | max-age=0, enforce   |              |              |
| X-Frame-Options ①                   | SAMEORIGIN   |              |              |
| Strict-Transport-Security ①         | max-age=15552000; includeSubDomains  |              |              |
| X-Download-Options ①                | noopener   |              |              |
| X-Content-Type-Options ①            | nosniff  |              |              |
| X-Permitted-Cross-Domain-Policies ① | none   |              |              |
| Referrer-Policy ①                   | no-referrer  |              |              |
| X-XSS-Protection ①                  | 1; mode=block  |              |              |
| Access-Control-Allow-Origin ①       | https://localhost:8080   |              |              |
| Vary ①                              | Origin   |              |              |
| Access-Control-Allow-Credentials ①  | true   |              |              |
| Content-Type ①                      | application/json; charset=utf-8  |              |              |
| Content-Length ①                    | 479  |              |              |
| ETag ①                              | W/"1df-qNhHpbGdIAxjYzNCj9SYGSeU"   |              |              |
| Date ①                              | Sun, 16 May 2021 16:20:05 GMT  |              |              |
| Connection ①                        | keep-alive   |              |              |
| Keep-Alive ①                        | timeout=5  |              |              |

On remarque que la configuration de nos headers laisse peu de place à la récolte d'informations et offre une première défense contre les attaques les plus basiques, avec dans l'ordre présenté sous Postman :

- ❖ La CSP est strictement définie (le nom de domaine pour mes images est défini, toute requête http est upgradée en HTTPS, attribut "nonce" avec uuid dynamique).
- ❖ Optimisation des temps de chargement des pages (surtout sur mobiles) en autorisant la résolution de nom de domaine en parallèle de la récupération du contenu de la page via X-DNS-prefetch-control sur on.
- ❖ Expect-CT pour l'attente de la transparence de certificat SSL et TLS. Afin que le navigateur refuse toute future connexion qui viole la politique de transparence des certificats, s'il y a problème de certificat, pas de connexion.
- ❖ L'en-tête X-frame-Option est définie de façon à ce que seulement mon site puisse utiliser les balises frame, iframe, etc. (protection contre le clic-jacking).
- ❖ La Strict-Transport-Security qui prévient les navigateurs que sur ce site on préfère du HTTPS, avec l'option max-age= 15552000 pour que les navigateurs se remémorent cette préférence pendant 180 jours.
- ❖ X-download-Option est défini pour Explorer 8 (Il force l'enregistrement des téléchargements potentiellement dangereux, ce qui atténue l'exécution du HTML dans le contexte du site).
- ❖ Le reniflage de type MIME est désactivé via "nosniff".
- ❖ Une politique CORS strictement définie.
- ❖ L'en-tête Referrer (qui contient normalement l'adresse d'une demande) est annulé via la configuration "no-referrer" évitant toute fuite d'information potentiellement non désirée par un utilisateur.
- ❖ L'header X-powered-by a disparu, ainsi personne ne sait que j'utilise Express.

Voyons en détail une fonctionnalité du site, la connexion en tant qu'utilisateur :

Prenons l'exemple un peu plus détaillé d'une vérification d'un email, étape nécessaire à la connexion :

Si notre utilisateur a attendu plus de 24 heure après avoir fait son inscription, pour valider son email, son lien envoyé dans l'email n'est plus valide. Mais une méthode lui est fournie pour vérifier son email à volonté.

**Donnée en entrée** : l'email utilisateur.

**Donnée attendue** : l'utilisateur peut faire vérifier son email et ainsi se connecter au site par la suite.

En front :

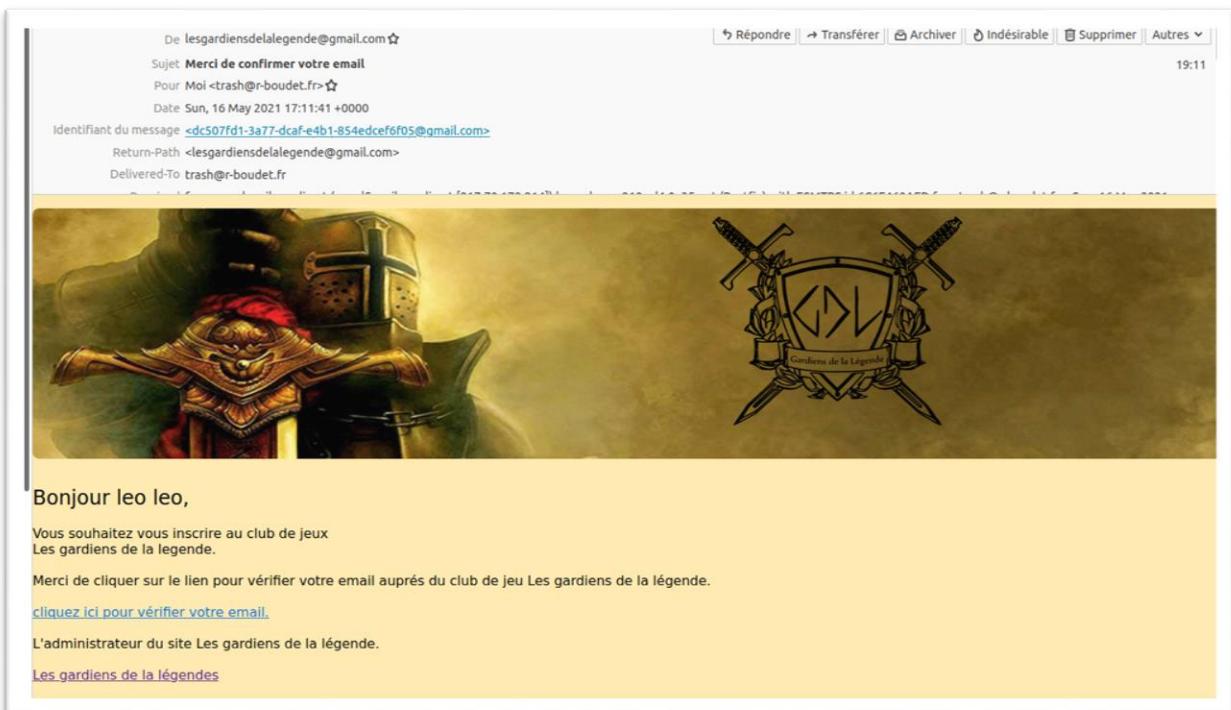
Lorsque l'utilisateur clique sur le bouton "vérifier mon email", ça déclenche la fonction handleSubmit email, qui lance emailSubmit, c'est une action, qui va mener dans le dossier Actions puis dans le dossier Middleware, qui va utiliser Axios pour faire une requête POST sur l'endpoint /resendEmailLink de notre API, avec dans le body, l'email à vérifier.

The top screenshot shows the main homepage of the website 'LES GARDIENS DE LA LÉGENDE'. At the top, there's a navigation bar with links for 'CONTACT', 'NOS JEUX', 'ÉVÉNEMENT', 'FORUM', and a key icon. On the left, there's a banner for 'LES GARDIENS'. In the center, there's a large 'CONNEXION' (Connection) form with fields for 'Votre pseudo' (Your nickname) and 'Votre mot de passe' (Your password), a 'Valider' (Validate) button, and a link 'vérifier un email' (Check an email). To the right of the form is a sidebar with links: 'Connexion', 'Inscription', and 'Profil'. At the bottom, there's a footer with text: 'Nous remercions FireFly Studios Games de nous autoriser à utiliser les photos à des fin non commerciale' and 'RGPD copyright © 2021'.

The bottom screenshot shows a modal window titled 'vérifier un email' (Check an email). It contains a text input field labeled 'votre adresse Email à vérifier' (Your email address to verify) and a 'Valider votre email' (Validate your email) button.

Dans le back :

Le code rentre dans index.js, les MW précédent le routeur sont lus, (CORS, sanitizer, le logger, helmet, MW express permettant de lire du JSON, les cookies, le body, etc.) et on arrive dans le routeur. Ça passe par la route appelée, le MW de Joi, vérifie si la donnée passée dans le body respecte le schéma de validation (REGEX). Si oui, on est renvoyé vers la méthode resendEmailLink du UserController. Express sanitizer double le travail de Joi en échappant tous les caractères spéciaux, dans le cadre de la politique NTUI (pour se prémunir de toute attaque XSS). On interroge la BDD en passant les données au modèle qui retourne les données du user avec cet email après avoir fait une requête SQL sur la BDD via le connecteur Postgres. Dans le contrôleur, on crée un JWT avec une date d'expiration de 24h (JWT que l'on aura pris soin de construire avec l'option "issuer" égale à userId), qu'on insère dans une variable "link" qui représente un lien dynamique qui inclut donc en query un identifiant d'un utilisateur et un JWT signé. Ce lien est intégré dans le texte d'un mail intégré à un transporteur mail, (Nodemailer pour nous) qui va envoyer le mail toujours de manière dynamique selon l'email de l'utilisateur.



A la réception du mail, l'utilisateur en cliquant sur le lien sur la route /verifyEmail du router, va récupérer les infos en query précédemment passées (s'ils passent le schéma de validation), vérifier que le userId en query est égal au "issuer" du JWT, que la signature du token est la bonne, puis le modèle va passer à true la valeur de la colonne verifyEmail en BDD via une requête SQL. Une vue est rendue par le serveur.

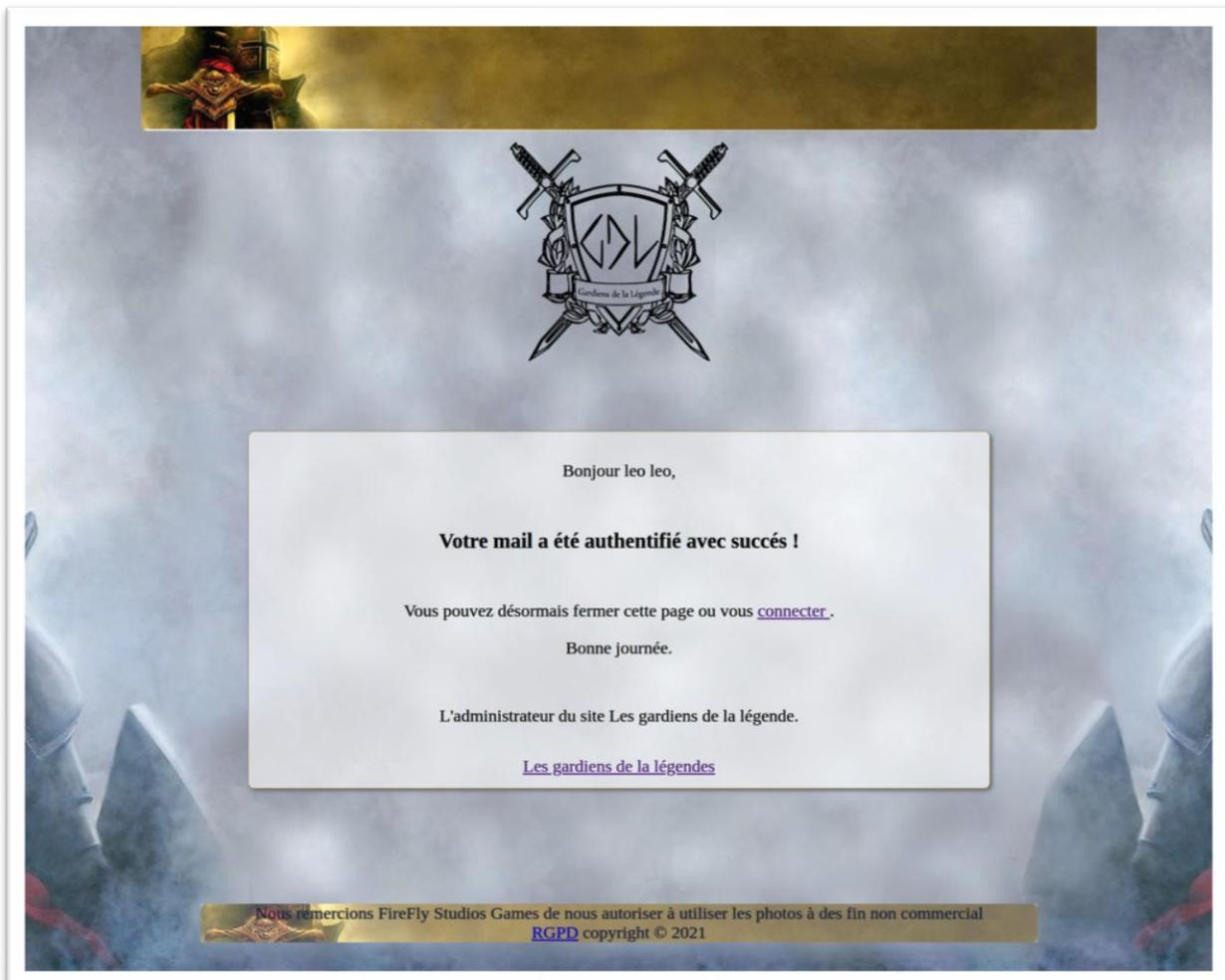


Image de la page de confirmation de vérification de l'email

A sa prochaine connexion l'utilisateur aura bien un email vérifié en BDD.

**Données obtenues :** l'utilisateur peut désormais se connecter.

**Résultats :** la donnée obtenue est identique à la donnée attendue, le test est réussi et notre utilisateur peut se connecter sans encombre.

## VEILLE ET TROUBLESHOOTING

Veille technologique et veille sur les vulnérabilités de sécurité

Comprendre l'utilisation des JWT

De manière générale, de très nombreuses ressources ont été utilisées pour la mise en place de l'application. Bien souvent, la lecture quasi-systématique de la documentation des

paquets npm, de la documentation d'Express, de Node.js, de REACT et la consultation de nombreux échanges sur Stackoverflow a été nécessaire.

Pour appréhender ce que sont les JWT et quand les utiliser à bon escient, de nombreuses ressources, la plupart en anglais, ont été consultées. Notamment la documentation officielle, des blogs spécialisés, et Stackoverflow :

- ❖ <https://jwt.io/>
- ❖ <https://www.vaadata.com/blog/fr/jetons-jwt-et-securite-principes-et-cas-utilisation/>
- ❖ <https://stackoverflow.com/questions/43452896/authentication-jwt-usage-vs-session>
- ❖ <https://ponyfoo.com/articles/json-web-tokens-vs-session-cookies>
- ❖ <https://developer.okta.com/blog/2017/08/17/why-jwts-suck-as-session-tokens>
- ❖ <http://crypto.net/~joepie91/blog/2016/06/13/stop-using-jwt-for-sessions/>
- ❖ <https://medium.com/devgorilla/how-to-log-out-when-using-jwt-a8c7823e8a6>
- ❖ <https://stackoverflow.com/questions/31919067/how-can-i-revoke-a-jwt-token>
- ❖ <https://fusionauth.io/learn/expert-advice/tokens/revoking-jwts/>

## Veille sur les vulnérabilités de sécurité

Je me suis appuyé sur site snyk.io (snyk.io/vuln/?type=npm) qui recense les failles de sécurité connues des packages npm. Suite à cette veille, tous les packages npm utilisés sont exempts de faille connue à ce jour. Il a été vérifié que le package express-jwt, était bien à la version 6.00 étant donné que toutes les versions précédentes, notamment celle publiée le 7 avril 2020 (version 5.3.3), possédaient de graves failles de sécurité (classées hight severity).

La bonne application de nos options de configuration pour les headers a été analysé via <https://observatory.mozilla.org/>.

## Problématique – Recherche et traduction d'un extrait en anglais

Extrait d'une recommandation du OWSAP concernant le paramétrage de l'option de configuration Samesite pour les cookies, extrait du site <https://owasp.org/www-community/SameSite> :

### Overview

SameSite prevents the browser from sending this cookie along with cross-site requests. The main goal is to mitigate the risk of cross-origin information leakage. It also provides some protection against cross-site request forgery attacks. Possible values for the flag are none, lax, or strict.

The strict value will prevent the cookie from being sent by the browser to the target site in all cross-site browsing contexts, even when following a regular link. For example, for a

GitHub-like website this would mean that if a logged-in user follows a link to a private GitHub project posted on a corporate discussion forum or email, GitHub will not receive the session cookie and the user will not be able to access the project.

A bank website however most likely doesn't want to allow any transactional pages to be linked from external sites so the strict flag would be most appropriate here.

The lax value provides a reasonable balance between security and usability for websites that want to maintain user's logged-in session after the user arrives from an external link. In the above GitHub scenario, the session cookie would be allowed when following a regular link from an external website while blocking it in CSRF-prone request methods (e.g. POST).

The none value won't give any kind of protection. The browser attaches the cookies in all cross-site browsing contexts.

The default value of the SameSite attribute differs with each browser, therefore it is advised to explicitly set the value of the attribute.

As of November 2017 the SameSite attribute is implemented in Chrome, Firefox, and Opera. Since version 12.1 Safari also supports this. Windows 7 with IE 11 lacks support as of December 2018, see caniuse.com below.

## TRADUCTION :

### Aperçu

L'option SameSite empêche le navigateur d'envoyer le cookie avec des requêtes provenant d'autres-sites. L'objectif principal est d'atténuer le risque de fuite d'informations suite à une requête d'une autre origine. Il offre également une certaine protection contre les attaques CSRF. Les valeurs possibles pour l'option sont "none", "lax" ou "strict".

La valeur stricte empêchera le cookie d'être envoyé par le navigateur au site cible, dans tous les contextes navigateur qui diffèrent du site d'origine, même en suivant un lien officiel. Par exemple, pour un site web comme GitHub, cela signifierait que si un utilisateur connecté suit un lien vers un projet GitHub privé, publié sur un forum de discussion d'entreprise ou par e-mail, GitHub ne recevra pas le cookie de session et l'utilisateur ne pourra pas accéder au projet.

Cependant, le site web d'une banque ne souhaite probablement pas autoriser les pages transactionnelles à être liées à partir de sites externes. L'option "strict" serait donc le plus approprié ici.

La valeur "lax" offre un équilibre raisonnable entre la sécurité et la convivialité pour les sites web qui souhaitent maintenir la session de connexion de l'utilisateur après que l'utilisateur soit arrivé via un lien externe. Dans le scénario GitHub ci-dessus, le cookie de session serait autorisé lors du suivi d'un lien régulier à partir d'un site Web externe tout en bloquant dans les méthodes de demandes sujettes à CSRF (par exemple POST).

La valeur "none" ne confère aucun type de protection. Le navigateur distribue les cookies dans tous les contextes de navigation intersite.

La valeur par défaut de l'attribut SameSite diffère avec chaque navigateur, il est donc conseillé de définir explicitement la valeur de l'attribut.

Depuis novembre 2017, l'attribut SameSite est implémenté dans Chrome, Firefox et Opera. Depuis la version 12.1, Safari le prend également en charge. Windows 7 avec IE 11 n'est plus pris en charge depuis décembre 2018, voir caniuse.com ci-dessous.

## CONCLUSION

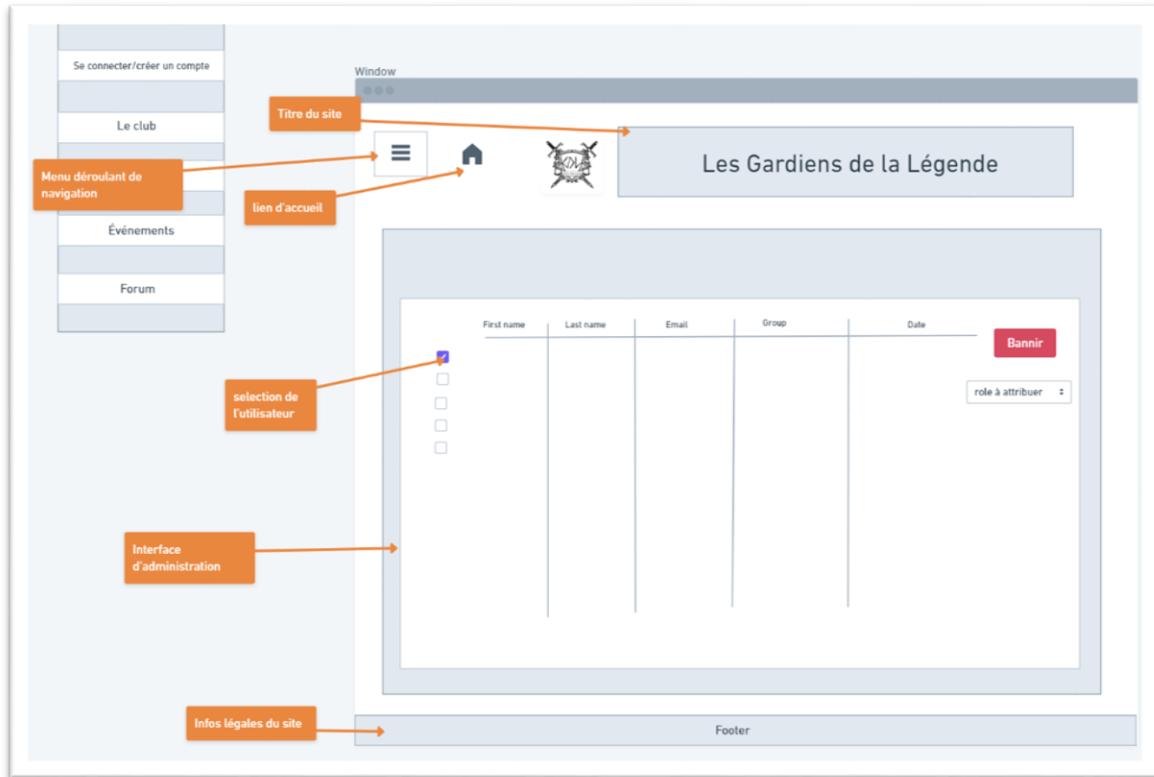
La participation à la création de ce site internet, qui n'est pas encore terminé, m'a permis de monter en compétence sur de nombreux points. En front, la compréhension de REACT et de sa configuration sont une bonne base pour perfectionner mes compétences en REACT dans les prochaines semaines, tandis que j'ai une vision claire du back, de sa structure et des points de sécurité importants sur lesquels il faut veiller.

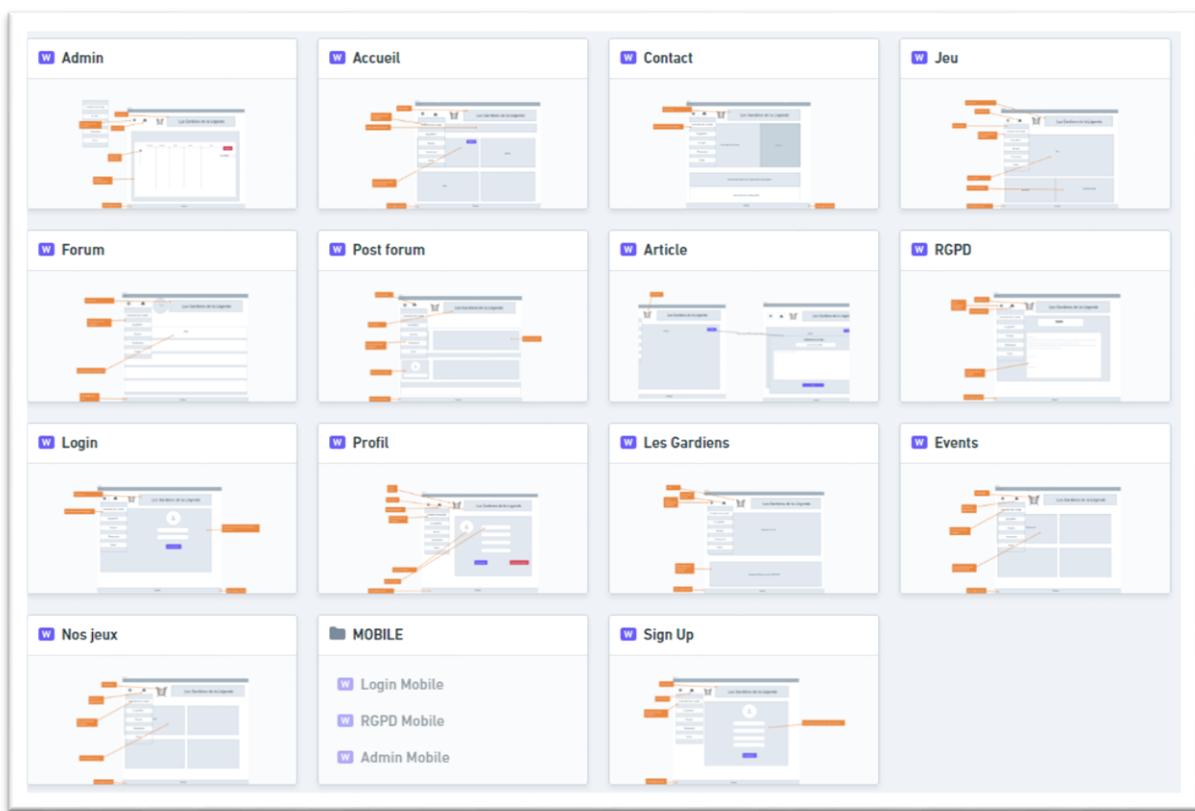
La mise en pratique de nos connaissances par la création d'un projet de bout en bout m'a donné un avant-goût de la vie de développeur que j'ai vraiment apprécié. Le travail en équipe, la coordination et la coopération que cela nécessite, allant de la réflexion à la conception jusqu'à la présentation, toutes ces étapes mises en pratique dans ce projet ont été très précieuses.

Il me reste de nombreuses choses à apprendre au cours de ma future vie de développeur junior, mais au vu du plaisir pris lors de cette phase de développement, je sais néanmoins que l'envie et la motivation seront toujours là.

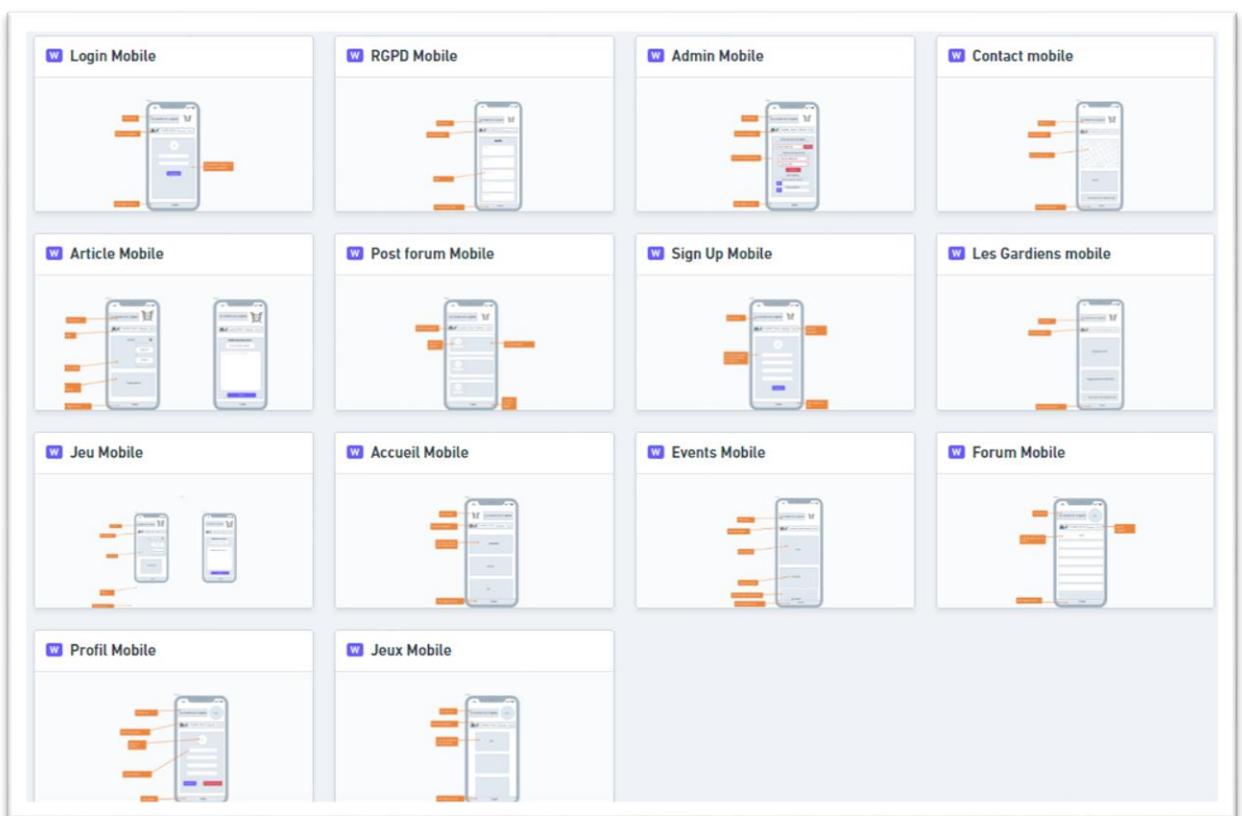
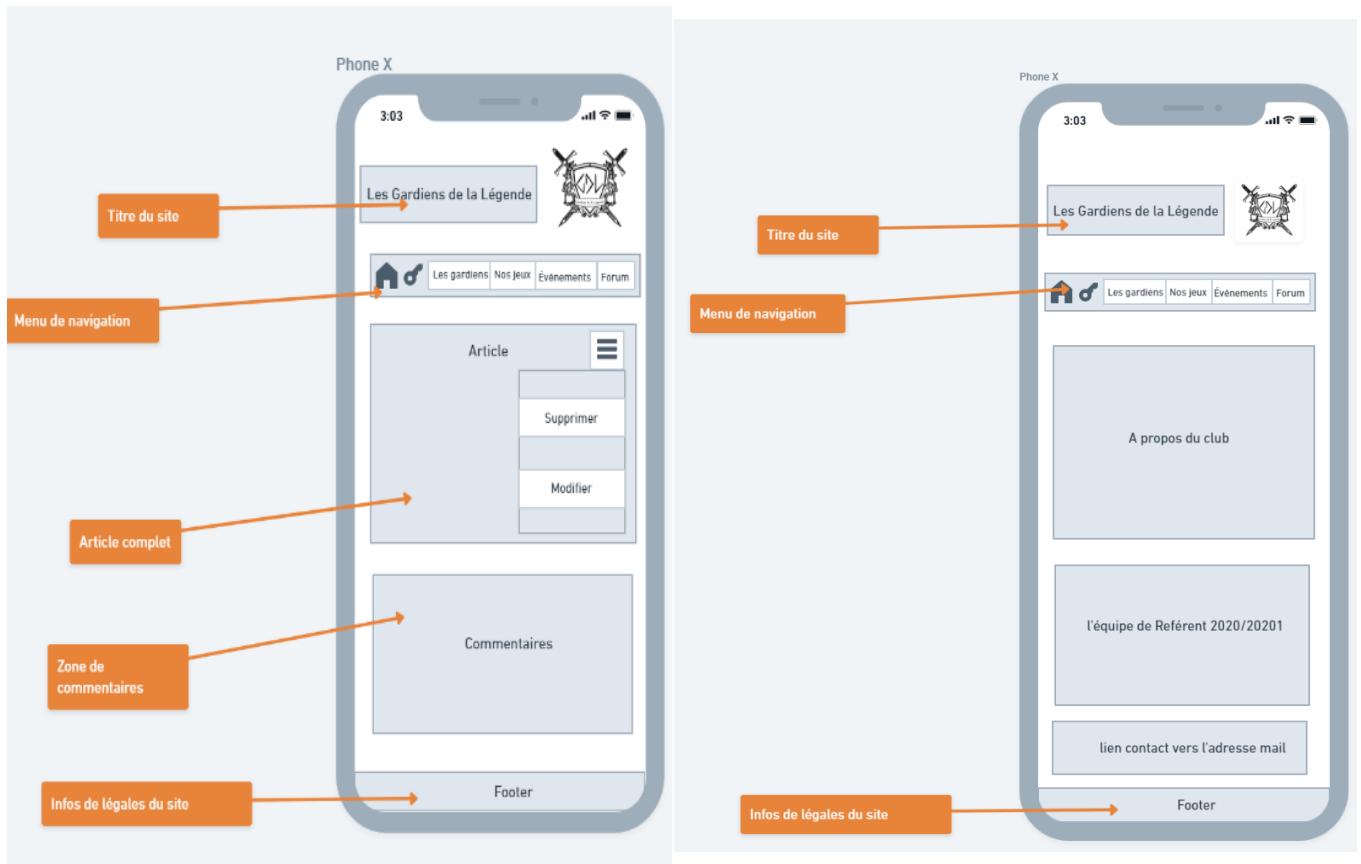
## ANNEXES

### WIREFRAMES DESKTOP

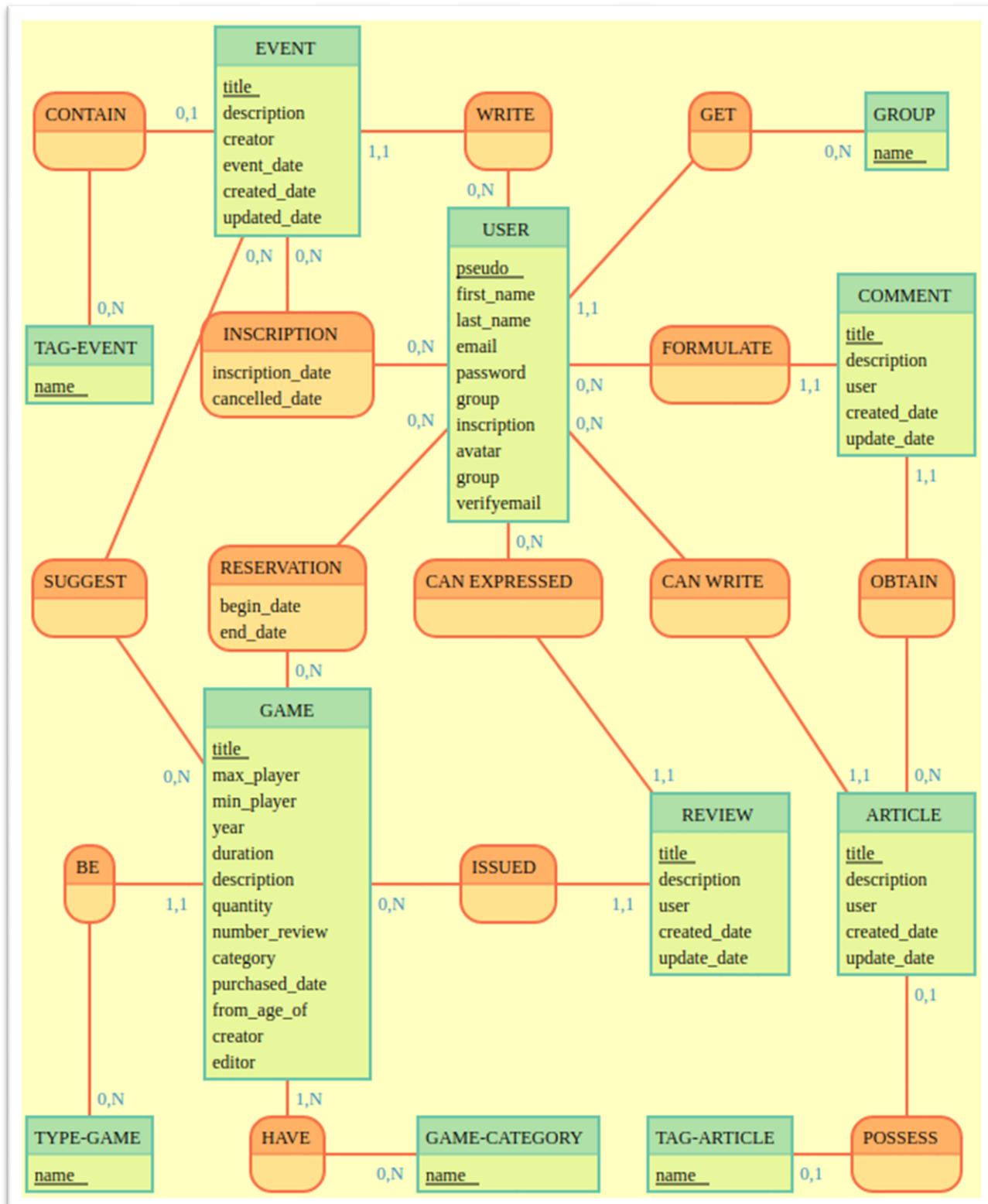




## WIREFRAMES MOBILE



## MCD



## MLD

### ( MVP)

- group(id, name)
- tag\_article(id, name)
- category(id, name)
- type\_game(id, name)
- tag\_event (id, name)
- game(id, title, max\_player, min\_player, year, duration, description, quantity, number\_review, category, purchased\_date, from\_age\_of, creator, editor)
- article(id, title, description, created\_date, update\_date , #user(id))
- review(id, title, description, created\_date , update\_date, #user(id), #game(id))
- event(id ,title, description, event\_date, created\_date ,updated\_date, #user(id), #game(id))
- user(id, pseudo, first\_name, last\_name, email, password, verifyemail, avatar, #group(id))
- event\_has\_user(id, inscription\_date, cancelled\_date, #event(id), #user(id))
- event\_has\_game(id, #event(id), #game(id))
- category\_has\_game(id, #category(id), #game(id))

## SCRIPT BACKUP BDD

```
#!/bin/bash

#####
## Script de backup pour notre BDD MJC ##
#####

# Un petit script maison fait par votre Git master préféré pour le backup de notre BDD

# Il y a surement mieux au vu de mes capacités limitées sur Linux, si vous avez des idées d'amélioration, you're welcome

# Pré-requis pour que ça fonctionne :

# - avoir installé postfix, mailutils, openssl sur le serveur (et toute leur dépendance)
# - avoir configuré correctement postfix

##### Pour faire tourner ce script, cette commande doit vous envoyé un mail

##### sur votre adresse mail => echo This is the body of the email | mail -s This is the subject line
your_email_address

##### des double quotes " sont nécessaire, encadrant chaque texte commençant par this-email et
this-line dans la commande précédente...

##### si vous ne recevez pas un email à chaque commande, il faut retourner configurer Postfix...

# - avoir installé openssl

# Que fait le script :

# 1) Il crée des variables utilisables par le script lui-même
# 2) Il crée un dossier
# 3) Il fait un dump de la base et en ressort un fichier SQL et envoie un mail si la tache est pas ok
# 4) Il compresses ce fichier SQL en .zip et envoi un mail si la tache est pas ok
# 5) Il supprime le fichier sql, on garde que l'archive.
# 6) Il chifre l'archive via openssl
# 7) Il envoie cette archive compressé et chiffré par mail, dans le mail qu'on a mis dans la variable
email
# 8) Il supprime les fichiers sur le serveur plus vieux que la date définie dans la variable keep_day
```

```

# et pour faire executer ce script on fait appel a CRON :

# ligne de commande sur le serveur=> crontab -e

##### et on colle une ligne cron.

##### Dans le cas d'un script script.sh situé dans le dossier /media/

##### voulant être effectué à midi et minuit

##### Sans être notifié par email => la ligne cron serait :

# dans la fenêtre qui s'est ouverte => 0 */12 * * * /media/script.sh >/dev/null 2>&1

```

```

# plus d'info sur cron => https://crontab-generator.org/ pour faire une ligne

# et un peu de doc, https://www.hostinger.fr/tutoriels/cron-job/

# les variables backupfolder, email et passw sont à configurer !

```

```

# Le nom de ma BDD

db_name=mjc

# Le dossier où va être sauvegarder ma BDD

backupfolder=

# L'email sur lequel va être envoyé la notification

email=

# Password pour le chiffrement

passw=

# Le nombre de jour où on stocke le backup de la BDD et le nom du fichier de backup à la date du
jour, jusqu'à la seconde du pg_dump

keep_day=3

sqlfile=$backupfolder/Mjc-database-$(date +%d-%m-%Y_%H-%M-%S).sql
zipfile=$backupfolder/Mjc-database-$(date +%d-%m-%Y_%H-%M-%S).zip
archivesecure=$backupfolder/Mjc-database-secure$(date +%d-%m-%Y_%H-%M-%S).zip

# Création du dossier de backup si jamais il n'est pas déjà créé

mkdir -p $backupfolder

# Création du backup

if pg_dump $db_name > $sqlfile ; then

```

```

echo 'Sql dump creer'

else

    echo 'pg_dump return un code d'erreur' | mailx -s 'ATTENTION aucun backup creer !' $email

    exit

fi

# Compression du backup

if gzip -c $sqlfile > $zipfile; then

    echo 'Le backup a ete compressé avec succès !'

else

    echo 'Erreur dans la compression du backup' | mailx -s 'ATTENTION aucun backup créé après la demande de compression !' $email

    exit

fi

# Suppression du fichier SQL, je garde seulement le .zip

rm $sqlfile

# Chiffrement du fichier compression via openssl (il s'agit de notre bien le plus précieux, pas envie de l'envoyer en clair...)

openssl enc -aes-256-cbc -salt -a -in $zipfile -out $archivesecure -iter 500000 -k $passw

# Envoie du fichier par pièce jointe pour le backup // ATTENTION fuite MAIL => consultation sur messagerie sécurisée par clé U2F

# (Et penser à bien configurer postfix pour TLS)

# ATTENTION si on utilise un mail google dans la configuration smtp (sudo apt-get install ssntp) Google refuse l'envoi de PJ chiffré => aucun mail ne sera envoyé

# Dans ce cas, remplacer $archivesecure par $zipfile dans la ligne suivante.

# on définit le type de fichier via --content ; le fichier à mettre en pièce jointe via --attach

echo $archivesecure | mailx --content-type=application/zip --attach=$archivesecure -s 'Backup de la BDD mjc créé avec succès !' $email

# Suppression des vieux backup

find $backupfolder -mtime +$keep_day -delete

```

## Un petit Récap des commandes git :

- `git status` : Affiche la liste des fichiers modifiés ainsi que les fichiers qui doivent encore être ajoutés ou validés
- `git add` : utilisée pour ajouter des fichiers à l'index
- `git commit -m "Nom de mon commit"` : Valide les modifications apportées au HEAD
- `git push` : Envoie les modifications locales à la branche associée
- `git checkout -b <nom-branche>` // `git branch <nom-branche>` : Crée une branche puis `git push --set-upstream origin <nom-branche>` pour pousser la branche courante
- `git checkout <nom-branche>` : Change de branche
- `git branch -d < nom-branch >` : supprimer une branche localement puis pour la supprimer en remote : `git push origin --delete <nom branch`
- `git branch` : List les branches
- `git log --oneline` : List les commit avec une seul ligne par commit pour plus de lisibilité
- `git checkout < code alphanumérique du commit >` : Permet de revenir au commit sélectionner.
- `gitk --all` : interface graphique pour voir l'avancement des commits => vous avez fait un `git checkout <numéro_de_commit>` et vous ne voyez plus les commit en amont avec `git log --oneline` : vous retrouvez ainsi les derniers commits. SousVM et ubuntu : `sudo apt-get install -y gitk`
- `git branch -d <nom-branche>` : Supprime une branche
- `git pull` : fusionne toutes les modifications présentes sur le dépôt distant dans le répertoire de travail local
- `git merge <nom-branche>` : Fusionne une branche dans la branche active
- `git diff --base <nom-fichier>` : Permet de lister les conflits
- `git diff <branche-source> <branche-cible>` : Affiche les conflits entre les branches à fusionner avant de les fusionner
- `git diff` : Permet de lister les conflits actuels
- `git stash` : Retour au précédents commit //casse les modif effectué
- `git stash list`
- `git stash drop // git stash drop <nom_fichier_a_dropper>`
- `git rebase master` : Réapplication des commits sur une autre branche
- 
- `git shortlog -s -n --all --no-merges` | Pour savoir le nombre de commit par personne

Aller plus loin avec un client git => <https://www.sublimemerge.com/>

avec le tuto qui va bien :

<https://blog.shevarezo.fr/post/2018/09/20/sublime-merge-client-git#:~:text=Il%20suffit%20d'ouvrir%20la,et%20filtrer%20ensuite%20par%20terme.>

[https://www.youtube.com/watch?v=ui4agRoFgwI&ab\\_channel=WatchandLearn](https://www.youtube.com/watch?v=ui4agRoFgwI&ab_channel=WatchandLearn)

# Recap des commandes Sqitch :

## Configuration du compte après installation

- `sqitch config --user user.name 'votre_nom'`
- `sqitch config --user user.email 'votre_mail'`
- `sqitch config -l`
- `sqitch config --user engine.pg.client psql`

## Principales commandes :

- `sqitch init game --engine pg --top-dir migration` => Pour initier le projet : on met tous les dossiers deploy, revert et verify dans un sous dossier migration : bcp plus propre ! ici game correspond au nom du projet et nom du premier deploy dans le .plan
- `sqitch add <monDossier>/<nom-de-la-migration> -n 'description'` permet d'ajouter un déploiement Sqitch qui aura pour nom 'Création des tables'

## On écrit nos deploy et nos revert dans les dossiers respectifs

- `sqitch deploy db:pg:<nom_de_la_DB>`
- `sqitch revert db:pg:<nom_de_la_DB>`