

DOSSIER DE PROJET



VOTRE RÉSERVOIR D'ÉVÈNEMENTS

PRÉSENTÉ PAR

FRÉDÉRIC GUIOU

EN VUE DE L'OBTENTION DU
TITRE PROFESSIONNEL

DÉVELOPPEUR WEB ET WEB MOBILE

Référents de formation

Jordan BRULL / Virginie LEMAIRE

Promotion CASSINI - 2022

ÉCOLE O'CLOCK



SOMMAIRE

I. INTRODUCTION	4
II. COMPÉTENCES COUVERTES PAR LE PROJET	5
A. Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité	5
• Maquetter une application :	5
• Développer une interface utilisateur web dynamique :	6
B. Développer la partie back-end d'une application web ou web mobile en intégrant les recommandations de sécurité	7
• Créer une base de données :	7
• Développer les composants d'accès aux données :	7
• Développer la partie back-end d'une application web ou web mobile :	7
III. SONOW - QU'EST CE QUE C'EST ?	8
IV. CAHIER DES CHARGES	9
A. Concepts théoriques de l'application	9
1. Détails des fonctionnalités de SoNow :	9
2. Minimum Viable Product (MVP) :	10
3. Le workflow de l'application :	11
B. Mise en oeuvre par la création des documents techniques	11
1. Les wireframes :	11
2. La charte graphique :	13
3. Les routes :	14
C. L'utilisateur et ses interactions	20
1. Le public visé et les rôles :	20
2. User stories :	21
D. Road Map de conception de l'application	22
V. SPÉCIFICATIONS TECHNIQUES DE SONOW	23
A. Gestion du versionning de projet :	23
1. Le versionning avec Git :	23
2. Versionning de la base de données :	24
B. Les technologies back-end :	26
C. Les technologies front-end :	27
D. La sécurité de l'application	28
VI. GESTION DE PROJET	32
A. Présentation de l'équipe	32
B. Organisation du travail	33
C. Outil de gestion utilisés	34
D. Répartition et détails des rôle de chacun	34
E. Programme des sprints	35
VII. PRODUCTION	36
A. Mes réalisations personnelles	36

1. Création des wireframes des pages d'inscription et de connexion :	36
2. Création de l'architecture back-end et du router :	38
3. Circulation de la donnée au travers des contrôleurs	39
4. Création du modèle dataMapper User	41
B. Test et jeu d'essai	43
VIII. VEILLE ET TROUBLESHOOTING	45
A. Veille technologique mise en place	45
B. Résolution des problèmes rencontrés	47
IX. CONCLUSION	48
X. ANNEXES	50

I. INTRODUCTION

Après des études et un parcours de comptable en cabinet d'expertise pendant près de vingt ans, j'ai commencé à réfléchir à l'aube de mes quarante ans à la possibilité de réaliser une reconversion professionnelle dans un secteur qui m'a toujours passionné : l'informatique.

En effet, j'ai sans cesse porté un intérêt aux nouvelles technologies, à l'assemblage et la mise en service de matériel, discipline que j'ai d'ailleurs exercée durant mes études à l'occasion de quelques stages non rémunérés à la fin des années 90, obtenus chez un professionnel de l'époque, sans pour autant envisager d'en faire mon métier un jour, mais également au software qui se tourne de plus en plus de nos jours vers le service en ligne.

Je me suis alors engagé dans la recherche des métiers d'aujourd'hui dans ce secteur. C'est de cette façon que j'ai découvert le développement et plus particulièrement celui des applications web, ainsi que l'école **O'clock**. Sa formation de développeur web de 6 mois en télé présentiel a retenu ma plus grande attention. J'ai alors tout mis en œuvre pour intégrer une promotion.

J'ai ainsi pu appréhender les concepts fondamentaux en front-end durant le socle de 3 mois, avec la découverte de langages qui m'étaient totalement inconnus comme le HTML et CSS pour toute la partie intégration statique ou encore le Javascript pour la dynamisation en manipulant le DOM avec la programmation orientée objet.
Le back-end était pour moi encore plus obscur à mon arrivée en cours car je ne me représentais pas en quoi cela consistait. Et pourtant cela a été une véritable révélation et un pur plaisir d'apprendre ce qu'était une architecture web, les bases de données et le langage SQL, ou encore une API. J'ai d'ailleurs choisi de suivre la spécialisation data.

A l'issue de ces quatre mois d'apprentissage, le cursus de l'école prévoit de remodeler l'ensemble du savoir acquis en développant un projet démarrant de zéro, en équipe de quatres à cinq personnes sur une durée d'un mois. Ce projet est appelé au sein de l'école "Apothéose". Il peut être issu de la proposition d'un des apprenants ou encore proposé par l'école elle-même et enfin, pour terminer, la demande également peut venir d'un intervenant externe.

C'est le cas du projet **SoNow** qui nous a été commandé par Jérémy Pyronnet, le product owner. Je vais donc vous présenter toutes les étapes de réalisation de l'application. Celle -ci doit couvrir les compétences du titre professionnel de développeur web et web mobile, titre pour lequel je me présente à vous, que je vais vous détailler dans la section suivante.

II. COMPÉTENCES COUVERTES PAR LE PROJET

Deux blocs de compétences front-end / back-end, chacun constitué de 3 sous-blocs sont à couvrir pour l'obtention du titre professionnel pour lequel je candidate aujourd'hui. Voici pourquoi je vous présente SoNow qui à mon sens permet de balayer chacune des attentes du référentiel.

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

- **Maquetter une application :**

Le projet à réaliser étant sans base de travail existante et n'ayant que quelques idées globales apportées par notre product owner externe, nous avons maquetté chacun les **wireframes** d'une page de l'application en version **desktop et mobile**. Chaque réalisation a été par la suite présentée à l'ensemble de l'équipe, présentation durant laquelle nous avons pu échanger et améliorer voir même redéfinir l'expérience utilisateur en pensant à de nouvelles fonctionnalités, tant sur le concept de l'interface que des actions possibles pour l'utilisateur. Nous avons utilisé des outils dédiés à cet effet pour obtenir des propositions conformes aux standards pour ce type d'exercice à réaliser.

- **Réaliser une interface utilisateur web statique et adaptable :**

La **diversité des caractéristiques de terminaux disponibles** pour accéder aux applications web de nos jours est telle qu'il est **nécessaire** que ces dernières **s'adaptent au format d'affichage** de chacun d'eux. (**Ordinateur, téléphone mobile, tablette...**)

Le product owner souhaitant une application web consultable depuis un terminal mobile, nous avons réalisé une **interface graphique simple d'accès pour tous les utilisateurs**. Cependant pour élargir au maximum la cible de l'application, celle-ci s'adapte

en fonction des contraintes d'affichage du terminal utilisé. Elle a été rendue parfaitement **responsive** par l'équipe et sa section front-end au moyen de **media queries**.

Exemple de media query appliquée au eventCardMain :

```
...  
@media screen and (max-width: 370px) {  
  .event-description {  
    &__details {  
      .header{  
        margin-bottom: 0 !important;  
      }  
      .description {  
        margin-bottom: 0 !important;  
        font-size: 0.7rem !important;  
      }  
  
      .content {  
        display: flex;  
        flex-wrap: wrap;  
        justify-content: center;  
      }  
    }  
  }  
  
.ui.label {  
  margin: 0 0.2em 0.1em 0;  
}  
}
```

Cette **média query** redéfinit certaines propriétés CSS des différentes **classes** du **composant eventCardMain** avec un **break point** de 370 pixels de largeur.

Le **display flex** et **flex-wrap combinés** à des **unités de valeur en em et rem** pour les marges et la taille de police **permettent** à l'élément de **s'adapter**.

- Développer une interface utilisateur web dynamique :

Afin de rendre l'expérience de l'utilisateur agréable et lui permettre un accès à l'information de façon **dynamique**, l'interface de l'application a été réalisée avec la librairie javascript **React**. Qui plus est, cet outil est enseigné dans le cadre de l'une des deux spécialisations proposées par l'école et a été choisi par tous les membres de l'équipe front-end. Sa logique de mise en œuvre est toutefois différente de Javascript Vanilla.

Elle repose sur des **composants** (components) **définis** par leur **état** (state) pouvant être **manipulés** à chaque tentative d'**interaction opérée** par **l'utilisateur** avec l'application par le biais de l'interface. Chaque **composant**, une fois son state défini, est alors **affiché** au travers d'un **Dom Virtuel**, propre à la librairie, lui-même **réconcilié** avec le **DOM "réel"** du navigateur pour être enfin affiché.

Le **DOM** pour Document Object Model est une **représentation dynamique orientée objet** du code de **l'application** assimilable à un **arbre** et ses **nœuds**.

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

- **Créer une base de données :**

Nous avons conçu toute la logique de la base de données de SoNow en élaborant en amont du développement différents documents tels qu'un **Modèle Conceptuel de Données**, un **Modèle Logique de Données**, un **dictionnaire des données**. Puis nous avons programmé les **outils pour la déployer** à l'aide du **langage SQL** avec un **Modèle Physique de Données** intégré au système de gestion de base de données **Postgres** et de **Sqitch** pour la gestion des versions.

- **Développer les composants d'accès aux données :**

Pour faire **communiquer la partie client** de SoNow **avec la base de données**, nous avons **développé** les composants permettant d'y **accéder** et de **la manipuler** suivant les demandes de l'utilisateur. Pool de connexion et Modèles ont été élaborés pour permettre une **gestion du CRUD de la BDD** avec la création de **requêtes SQL**. La **gestion de la sécurité** sur ce plan a été faite au moyen de **requêtes préparées** et de **jetons SQL**.

- **Développer la partie back-end d'une application web ou web mobile :**

Enfin pour **permettre la circulation de la données** de la BDD entre notre API et l'interface front-end nous avons **développé l'architecture back-end** de celle-ci en **respectant** au mieux les **règles** en matière de **sécurité**. Aussi nous avons procédé à l'implémentation d'un **serveur**, de **routeurs** pour l'appel aux **endpoints** et de **contrôleurs** qui **gèrent les trajets des requêtes et réponses** mais également certains **services** tels que **l'authentification**, la **gestion des erreurs**, ou encore la **validation** de certains **patterns** (emails, mots de passe) pour la création d'un compte.

III. SONOW - QU'EST CE QUE C'EST ?

Vous êtes-vous déjà retrouvé avec vos amis posés chez l'un d'entre vous autour d'une table dans votre canapé et tout à tout coup quelqu'un prononce cette phrase : "Bon qu'est ce qu'on fait ce soir ?" Tout le monde reste dans un silence qui veut bien évidemment dire "Aucune idée". Ou encore un dimanche après-midi un peu pénaud avec votre petit(e) ami(e) à vos côtés qui vous reproche souvent de ne jamais le/la surprendre lors d'une balade dans un lieu ou un évènement insolite ?

Et bien SoNow est la solution développée que vous pourrez utiliser pour vous éviter de vous retrouver sans inspiration. En effet grâce à elle vous aurez accès à un réservoir d'évènement autour de vous. Manifestations sportives, After work, Concerts, Expositions culturelles, vous ne manquerez pas d'idées. Notre application a été pensée mobile first car elle est avant tout prévue pour être utilisée lors de vos déplacements mais rien n'empêche de parcourir ses nombreux contenus confortablement installé(e) devant l'écran de votre ordinateur. Le concept va même au-delà d'une simple bibliothèque d'évènements. Notre product owner souhaitait y implémenter une dimension sociale en permettant à un utilisateur de s'abonner à l'actualité événementielle d'un autre utilisateur faisant partie du cercle familial ou amical. Nous avons ainsi pensé l'interface et les fonctionnalités de l'API dans cette direction. Les fonctionnalités étant peu à peu activées au fil des mises à jour.

L'application actuelle permet à l'utilisateur connecté de parcourir une bibliothèque d'évènements qu'il peut rechercher par catégories. Dans de prochaines versions il aura la possibilité d'en créer et les administrer lui-même pour les proposer à sa communauté de "SoNowistes", de s'abonner à d'autres utilisateurs ou évènements, de marquer sa participation afin d'en informer sa famille et ses amis. La personnalisation du profil sera également activée. Enfin, il est envisagé d'implémenter une messagerie pour pouvoir dialoguer directement avec ses abonnés ou les organisateurs d'un évènement. Beaucoup de ses fonctionnalités sont déjà prévues dans l'API et seront très prochainement déployées.

IV. CAHIER DES CHARGES

Pour mener à bien le développement du projet et dans l'optique de permettre à chaque intervenant de l'équipe un accès aux spécificités de l'application à tout moment mais également au product owner, nous avons élaboré un cahier des charges combinant les éléments fondamentaux pour nous permettre un lancement de la phase de développement du MVP.

A. Concepts théoriques de l'application

1. Détails des fonctionnalités de SoNow :

Nous avons à ce titre échangé avec Jérémy Pyronnet qui nous a fait part de son souhait concernant les fonctionnalités qu'il souhaitait pour l'application :

Les différentes pages :

-Page d'accueil

permettant à un utilisateur enregistré de s'authentifier avec un email et mot de passe

-Page de création d'un compte

en remplissant un formulaire d'inscription

-Page "A propos"

présentant l'équipe de développement

-Page Feed d'évènements

(swipe up et down pour passer d'un évènement à un autre). Chaque évènement affiche sa date, son nom, une description succincte, une photo, le nombre de participants, et la catégorie à laquelle il appartient.

Mettre des évènements en favoris

Mentionner sa participation à un évènement auprès de sa communauté

-Pages des évènements favoris

qui affiche la liste des évènements favoris de l'utilisateur

-Page de recherche des évènements

avec formulaire de saisie, propositions "À la une", filtrage par catégories

-Une page de profil utilisateur

Affichant les évènements auxquels il participe, un lien vers la création d'un évènement, une liste des utilisateurs de l'application, une page de modification de profil.

Créer, modifier et supprimer des évènements

S'abonner à d'autres utilisateurs pour connaître les évènements auxquels ils participent

Évolution potentielles pour des versions ultérieures

Uniquement pour un utilisateur authentifié :

- Messagerie instantanée** pour dialogues en messages privés (Positionné dans l'interface)
- Ajouter un commentaire** à un évènement (Positionné dans l'interface)
- Partager** un évènement
- Ajout** d'un évènement à son **calendrier personnel**
- Recevoir des **notifications** si un évènement est modifié (Changement de date, annulation etc...)
- Elargissement de la zone de couverture des évènements proposés en implémentant la géolocalisation.

2. Minimum Viable Product (MVP) :

Le minimum viable product (MVP) est la version minimale du produit proposée à l'utilisateur.

Aujourd'hui le projet offre les fonctionnalités suivantes en raison du temps limité autorisé lors de l'Apothéose pour son développement. La plupart des fonctionnalités sont fonctionnelles dans l'API mais n'ont pu être toutes raccordées à l'interface utilisateur.

Accessible pour un visiteur :

- Page d'accueil permettant à un utilisateur enregistré de s'authentifier
- Page de création d'un compte
- Page "A propos" présentant l'équipe de développement

Accessible pour un utilisateur après authentification :

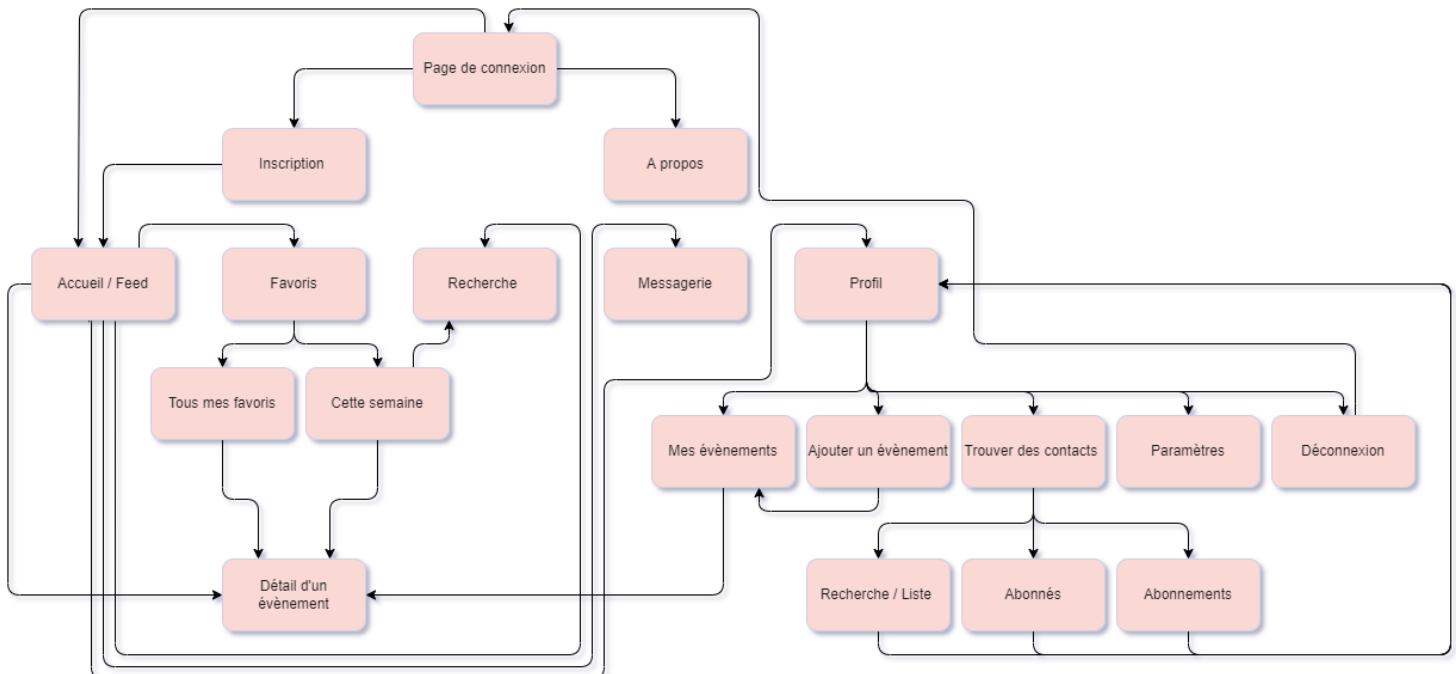
- Page Feed d'évènements (swipe up et down pour passer d'un évènement à un autre)
- Pages des évènements favoris
- Page de recherche des évènements avec formulaire de saisie, propositions "À la une", filtrage par catégories
- Une page de profil utilisateur avec bouton de déconnexion
- Mise des évènements en favoris (API uniquement)
- Mention de sa participation à un évènement (API uniquement)
- Créer, modifier et supprimer des évènements (API uniquement, interface à venir)
- Abonnement/désabonnement à d'autres utilisateurs (API uniquement)

Dans des versions futures :

- Page de paramètres qui donnera à l'utilisateur la possibilité de modifier des données telles que sa photo ou ses infos personnelles
- Liste des abonnés et abonnements utilisateurs
- Géolocalisation pour connaître les évènements les plus proches de soi.
- implémentation d'un light mode après avoir débattu avec le product owner.

3. Le workflow de l'application :

Afin d'organiser les différentes pages, nous avons réalisé un diagramme permettant d'avoir une vue d'ensemble de l'arborescence de l'application, couramment appelé "workflow". Ce type de diagramme permet de savoir dans quel niveau de l'application on se trouve.



B. Mise en oeuvre par la création des documents techniques

1. Les wireframes :

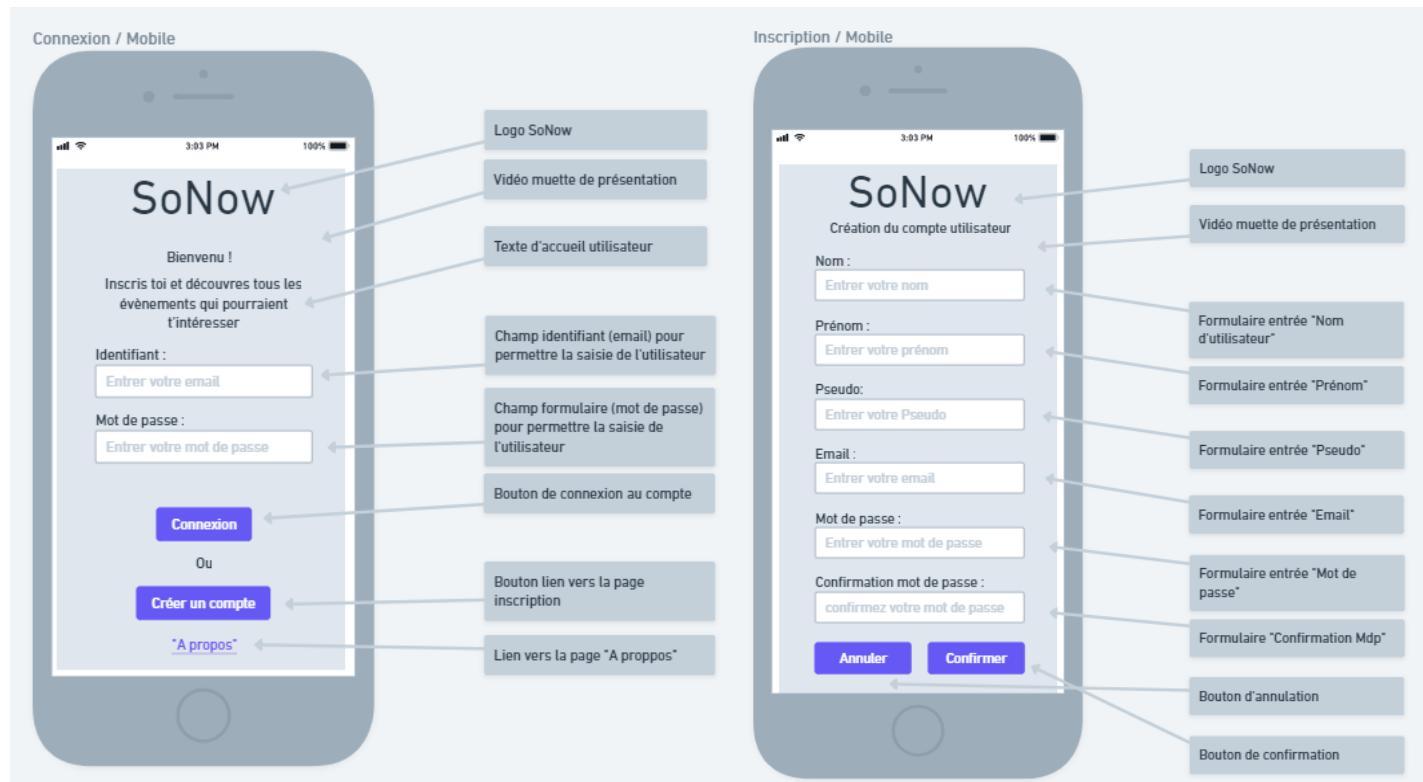
Une fois toutes les pages hiérarchisées, nous avons démarré la **création de maquettes** appelées **wireframes** qui ont pour but de présenter une **ébauche de l'organisation de l'interface** ainsi que les fonctionnalités prévues à chaque élément qui la compose.

La conception des wireframes est **une étape essentielle** avant le code dans le logiciel dédié. En effet, **passer rapidement cette étape** peut entraîner de **mauvaises prises de direction** dans la conception de l'interface et faire **perdre un temps précieux au client**. Nous avons donc demandé au product owner de participer à une réunion. L'inclure dans cette étape, en plus de lui prouver de l'intérêt en tant que client pour établir d'éventuelles bonnes relations commerciales, permet d'avoir sa vision de l'application et lui proposer des solutions adaptées en fonction de ses idées.

Les **wireframes** restent cependant des ébauches pouvant être amenées à des changements en cours de développement. Leur rendu est très sommaire (souvent en fil de fer), et n'est pas représentatif du résultat final de l'application. Ils sont agrémentés de **légendes** pour chaque élément afin de définir la **fonctionnalité attendue** lors d'une interaction avec l'utilisateur. À l'issue de la réunion nous avons avec **chacun de mes coéquipiers** pris en charge la conception de **deux wireframes**. J'ai pour ma part traité les pages de connexion et de création d'un compte utilisateur.

Plusieurs outils permettent de concevoir ces représentations, ne connaissant pas l'outil utilisé par le reste de l'équipe, et afin de gagner un temps précieux, j'ai utilisé **Whimsical** pour créer ces **deux pages en format mobile first et desktop**. (Plus de détails dans la section "production")

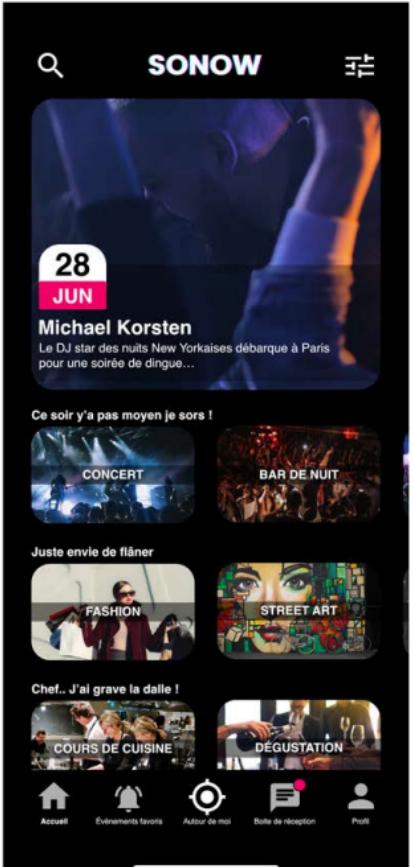
Pages Mobile First Connexion / Inscription :



(Pour le rendu des pages au format Desktop je vous renvoie aux annexes.)

2. La charte graphique :

Durant la réunion de conception des wireframes avec le **product owner**, et s'agissant de l'application qu'il nous a commandé, nous lui avons demandé de bien vouloir **nous fournir les éléments de la charte graphique** qu'il souhaitait être utilisé dans l'interface. Il faut savoir que ce dernier travaille dans une agence de communication, il avait **pensé en amont** de sa prise de contact avec nous au rendu éventuel. N'étant pas développeur, il nous a finalement demandé des **conseils sur les choses possibles en termes d'UX/UI avec les technologies que nous allions utiliser** et il nous a transmis le document ci-dessous que l'équipe front a validé comme réalisable dans le temps imparti.



Charte graphique

Code couleur

#000000
#f30067
#b3b3b3

Blanc de référence : #ffffff

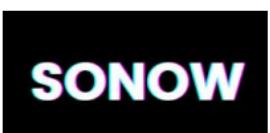
Police de caractère

Helvetica
Regular
Bold

Taille de police

Suite de Fibonacci
8 - 13 - 21

Logotype



Jérémy Pyronnet (pour rappel notre product owner), avait cependant **quelques demandes à respecter obligatoirement par l'équipe**, à savoir : **pas de light mode par défaut**, penser l'application en **mobile first** avant tout.

La **charte** se veut **moderne**, semblable aux **tendances des projets couramment utilisés** par toutes les générations mais plus particulièrement **Instagram et Tiktok** qui ont beaucoup de succès, Jérémie souhaitant que les **utilisateurs retrouvent les mêmes réflexes** pour ne pas les déstabiliser.

-Au niveau des couleurs, l'interface tournera autours de **4 couleurs principales** : le **noir** pour tous les **éléments globaux** tels que le fond du corps. Un **rouge rosé punchy acidulé** pour

apporter un peu de couleur et **mettre en avant certaines informations**. Un **gris** pour la **sélection**, comme par exemple mentionner la section active de l'application. Pour tous les **textes, un blanc**.

-La police retenue est **Helvetica**, **moderne et arrondie**.

-Le logo **SoNow** a été retenu avec un effet appelé “**distorsion chromatique**” conjuant le **blanc, le rose et le turquoise**. Cet effet a d'ailleurs été **repris** pour la réalisation de la **page 404**.

3. Les routes :

Les **routes** sont les **chemins** empruntés par le client lors d'un **appel à l'API**. Elles montrent comment une API **répond à des requêtes** pour accéder à des **endpoints**. Elles sont **définies** dans l'architecture de l'API suivant le **verbe de la méthode HTTP**, l'**adresse** utilisée lors de l'émission de l'appel (ex: '/login'), le **contrôleur** interrogé (ex : userController) et la **fonction** qui est lancée lors de cet appel (ex: loginUser).

Ci-dessous un extrait de la liste des routes établies lors de la mise en place de l'application :

URL	HTTP METHODS				CONTROLLERS	METHODS	CONTENTS
	DELETE	POST	PATCH	GET			
/user	✗	✗	✗	✓	user	getAllUsers	Récupérer tous les utilisateurs
/user/:user_id	✓	✓	✓	✓	user	getOneUserById, updateUser, deleteUser	Récupérer, modifier, supprimer un utilisateur par son ID
/login	✗	✓	✗	✗	user	loginUser	Se connecter à son compte utilisateur
/logout	✗	✗	✗	✓	user	logoutUser	Se déconnecter de son compte utilisateur
/signup	✗	✓	✗	✗	user	createUser	S'inscrire sur l'application en créant un compte utilisateur
/user/search	✗	✗	✗	✓	user	getOneUserByNickname	Rechercher un autre utilisateur par son surnom
/user/follow	✓	✓	✗	✗	user	followUser, unfollowUser	s'abonner / se désabonner à un utilisateur
/user/:user_id/followers	✗	✗	✗	✓	user	getFollowers	Récupérer la liste des abonnés d'un utilisateur par son ID
/user/:user_id/followed	✗	✗	✗	✓	user	getFollowed	Récupérer la liste des abonnements d'un utilisateur à d'autres comptes par son ID
/event	✗	✓	✗	✓	event	getAllEvents, createEvent	Récupérer la liste des évènements
/event/:event_id	✓	✗	✓	✓	event	getOneEventById, updateEvent, deleteEvent	Récupérer, modifier, supprimer un évènement par son ID
/event/tag/:tag_id	✗	✗	✗	✓	event	getByTagId	Récupérer les évènements d'une catégorie
/event/search	✗	✓	✗	✗	event	getEventsByTitle	Rechercher un évènement par son Titre
/event/getbookmarks	✗	✓	✗	✗	event	getEventsByPinUser	Récupérer la liste des évènements favoris d'un utilisateur
/event/getattend	✗	✓	✗	✗	event	getEventsByAttendUser	Récupérer la liste des évènements auxquels un utilisateur participe
/event/bookmarks	✓	✓	✗	✗	event	addToBookmarks, delToBookmarks	Ajouter / Supprimer un évènement en favoris
/event/attend	✓	✓	✗	✗	event	addAttendEvent, delAttendEvent	Participer / Ne plus participer à un évènement
/event/:event_slug	✗	✗	✗	✓	event	getOneEventBySlug	Récupérer un évènement par son slug
/tag/tag_id	✗	✗	✗	✓	tag	getOneTag	Récupérer un Tag par son ID
/tag	✗	✗	✗	✓	tag	getAllTags	Récupérer tous les tags
/withevents	✗	✗	✗	✓	tag	getAllTagWithEvents	Récupérer tous les tag avec les évènements associés

CRUD : Create = **POST** // Read = **GET** // Update = **PATCH** // Delete = **DELETE**

Légende : ✓ Route prévue ✗ Route non prévue

L'URL est le chemin de la route, son endpoint.

Les **méthodes HTTP** définissent l'action que l'utilisateur demande à effectuer au serveur. Nous en avons utilisées ici 4 (GET, POST, PATCH et DELETE).

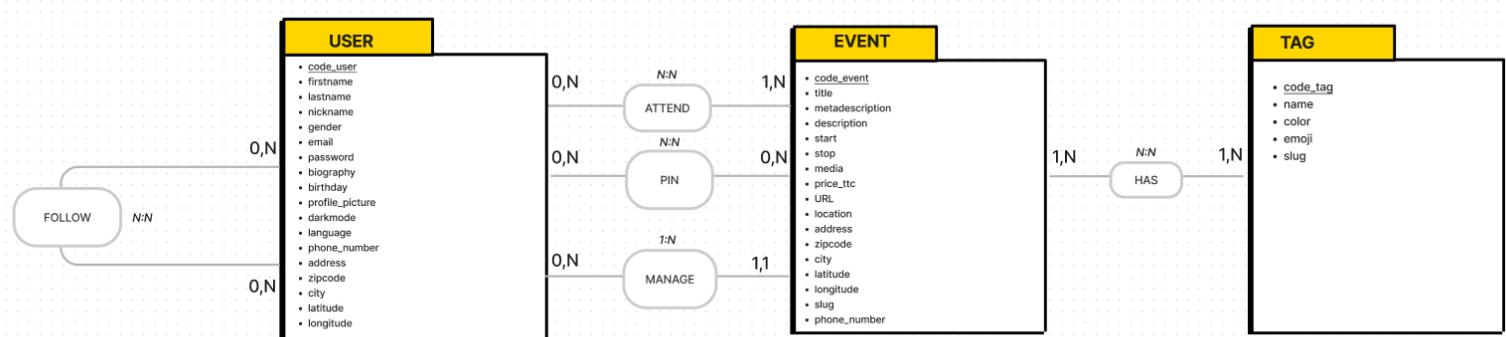
Le **contrôleur** permet de gérer la dynamisation de l'application, il est composé de **méthodes**. Celles - ci sont les interactions que les utilisateurs souhaitent opérer avec les objets. Elles peuvent recevoir des paramètres et retourner des données sous la forme de **tableaux d'objets JSON**.

Enfin la colonne **content** décrit en quelques mots les détails de l'action souhaitée par l'utilisateur.

4. MCD / Modèle Conceptuel de Données :

Le **modèle Conceptuel de Données** est une **représentation** des données **organisées en entités**, elles - même détaillant leurs **propriétés**, chaque entité peut être en **relation** avec une ou plusieurs autres. Chaque relation est symbolisée **au moyen** d'une **association**. De plus, on **quantifie** les **possibilités minimum et maximum** d'occurrences concernées par ces relations au moyen de **cardinalités**.

Pour notre application l'équipe s'est réunie et nous avons réalisé le MCD à l'aide de l'outil **Figma**. Ci-dessous sa représentation :



Après avoir **créé les entités** et les avoir **complété** de leurs **attributs** (les différentes propriétés qui les caractérisent), on définit les **associations** et les **cardinalités**.

Le MCD de SoNow est **composé de trois entités** : User, event, et tag. Chaque entité est faite d'**attributs** dont **un**, le **déterminant**, qui est **unique** et permet de l'**identifier**. Pour isoler cet attribut unique on le **souligne** dans l'entité et on le place en haut de la liste des attributs.

Nous avons par la suite placé les **associations**. On les établit au moyen de liaisons et de verbes.

Les **cardinalités** sont les suivantes :

- Un utilisateur peut suivre ou être suivi par aucun ou plusieurs autres utilisateurs. (0, n)
- Un utilisateur peut participer à zéro ou plusieurs évènements. (0, n)
- Un utilisateur peut mettre en favoris zéro ou plusieurs évènements. (0, n)
- Un utilisateur peut organiser à un ou plusieurs évènements. (1, n)
- Un évènement peut avoir un ou plusieurs participants.(1, n)
- Un évènement peut être mis en favori par zéro ou plusieurs utilisateurs.(0, n)
- Un évènement ne peut être organisé que par un et un seul utilisateur. (1, 1)
- Un évènement peut avoir un ou plusieurs tags.(1, n)
- Un Tag peut référencer un ou plusieurs évènements.(1, n)

5. MLD / Modèle Logique de Données :

Le modèle logique des données, MLD, est la continuité du MCD, à laquelle on ajoute une couche supplémentaire liée aux relations entre les entités.

Ci-dessous le MLD de l'application SoNow :

```
USER : code_user,firstname,lastname,nickname,gender,email,password,biography,birthday,profile_picture,darkmode,language,phone_number,address,zipcode,city,latitude,longitude
EVENT : code_event,title,metadescription,description,start,stop,location,address,zipcode,city,latitude,longitude,slug,phone_number,media,price_ltc,URL,#code_user_manager
TAG : code_tag,name,color,emoji,slug
USER_PIN_EVENT : #code_user,#code_event
USER_ATTEND_EVENT : #code_user,#code_event
EVENT_HAS_TAG : #code_event,#code_tag
USER_FOLLOW_USER : #code_user,#code_user
```

Les entités deviennent des **tables** et leurs déterminants respectifs des **clés primaires**, les propriétés des entités du MCD se transforment en **colonnes** des tables. Par ailleurs, les **cardinalités** peuvent **générer** des **clés étrangères** dans certaines tables **voire même** des tables supplémentaires dites **tables de liaisons**.

En effet, la **relation “One to many” 1:N** entre les tables **event** et **user** engendre la création d'une **clé étrangère** dans la table **event** (“*Un événement ne peut être organisé que par un et un seul utilisateur*”). Les clés étrangères sont **symbolisées** ainsi dans le MLD **#nom_clé_étrangère** et sont **positionnées** en **fin de table**.

De plus, on crée une **table de liaison** entre **deux tables** lorsque la **relation** établie est de type **“Many to many” N:N** c'est-à-dire lorsque les **cardinalités maximum** de chaque entité du MCD sont **“N”**. Les clés primaires des tables deviennent des clés étrangères dans les tables de liaison. **#code_event #code_user #code_tag**

Dans SoNow c'est le cas pour **quatre relations** :

- Pin
- Attend
- Has
- Follow

On a donc **4 tables de liaison** dans le MLD :

- User_pin_event
- User_attend_event
- User_follow_user
- Event_has_tag

6. Dictionnaire des données

Le dictionnaire des données est un recueil de l'**ensemble des tables** et de leurs **structures**. Son **importance est primordiale** car il permet d'**appréhender le vocabulaire** de base de données. Le **nom**, le **type**, les **spécificités** et la **description** des données y sont consignés sous forme de **tableau**. Il est établi suivant des **règles strictes** :

- Les données sont **organisées par table**
- Les **identifiants** sont **clairement mentionnés** comme tels
- Les **descriptions** doivent permettre une **compréhension rapide**.

Pour SoNow le dictionnaire des données a été **réalisé** durant les premières étapes de la conception de la documentation de l'application et bien entendu **en amont de la phase de développement**.

Nous avons commencé par **lister** dans un tableau toutes les **tables** composant notre **base de données** à partir du **MLD** car le dictionnaire des données ne prend **pas uniquement** en considération les **tables "physiques"** mais y compris les **relationnelles**.

Ensuite nous avons **listé** tous les **champs** de chaque table, pour enfin **remplir** les différentes **caractéristiques** de ces champs. Le dictionnaire des données se **rérite** de préférence en **anglais** car le **Modèle physique** des données et la création dans le SGBD qui en découle s'appuient sur celui - ci.

Voici un extrait du dictionnaire des données de trois tables dont une de liaison :

TABLE USER

CHAMPS	TYPE	SPECIFICITES	DESCRIPTIONS
id	INTEGER	GENERATED ALWAYS PRIMARY KEY	determining user
firstname	TEXT	NOT NULL	first name
lastname	TEXT	NOT NULL	last name
nickname	TEXT	NOT NULL UNIQUE	nickname
gender	TEXT		gender
email	TEXT	NOT NULL UNIQUE	email of the user
password	TEXT	NOT NULL	connection password
biography	TEXT		various informations the user
birthday	DATE		birthdate
defaut_profile_picture	TEXT	NOT NULL DEFAULT 'default_profile_picture.webp'	profile photo af the user
language	TEXT	NOT NULL DEFAULT 'FR'	language
darkmode	BOOLEAN	NOT NULL DEFAULT true	dark or light mode choice
address	TEXT		adress of the user
zipcode	TEXT		zip code of the user
city	TEXT		city of the user
latitude	FLOAT		determining lattitude set for position
longitude	FLOAT		determining longitude set for position
phone_number	TEXT		phone number of the user

TABLE TAG

CHAMPS	TYPE	SPECIFICITES	DESCRIPTIONS
id	INTEGER	GENERATED ALWAYS PRIMARY KEY	détermining tag
name	TEXT	NOT NULL	tag name
color	TEXT	NOT NULL	tag color
emoji	TEXT	NOT NULL	tag emoji
slug	TEXT	NOT NULL UNIQUE	slug of the tag

USER_PIN_EVENT

CHAMPS	TYPE	SPECIFICITES	DESCRIPTIONS
code_user	INTEGER	FOREIGN KEY REFERENCES user(id)	ID of the user in his table
code_event	INTEGER	FOREIGN KEY REFERENCES event(id)	Id of the event in his table

Nous avons utilisé les types suivants :

- **INTEGER** afin d'utiliser des **nombres entiers** pour les **identifiants et clés étrangères**
- **TEXT** appliqué à toutes les **données alphanumériques** car il n'y a **pas de limitation en nombre de caractères** pour ce type de données et certains champs tels que la biographie d'un utilisateur ou la description d'un événement sont sensibles à cette contrainte.
- **BOOLEAN** est choisi quand on souhaite une condition vraie ou fausse. Ici pour le choix du dark ou light mode de l'affichage de l'application.

A noter également, l'utilisation du type **TIMESTAMPTZ** pour les champs “start” et “stop” de la table event pour la gestion des dates et heures de départ et de fin d'un évènement, et du

type **FLOAT** pour les latitude et longitude en vue de l'implémentation future de la géolocalisation.

7. MPD / Modèle Physique de données

Le **modèle physique de données** est la **combinaison** du **MLD** et du **dictionnaire des données**. Il permet de **créer la base de données** dans la SGBB. Nous avons utilisé **Postgres** comme système de gestion de base de données combinés à **Sqitch** pour la **gestion du versionning** de la BDD. (Je reviendrai sur ce dernier point ultérieurement).

Le **MPD** est un **script SQL** qui s'articule ainsi :

```
• • •

BEGIN;

DROP TABLE IF EXISTS public.user, public.event, public.tag, public.event_has_tag,
public.user_pin_event, public.user_attend_event, public.user_follow_user;

CREATE TABLE IF NOT EXISTS "user"
(
    "id"                  integer NOT NULL UNIQUE GENERATED ALWAYS AS IDENTITY (
        INCREMENT 1 START 1 MINVALUE 1 CACHE 1 ),
    "firstname"            text NOT NULL,
    "lastname"             text NOT NULL,
    "nickname"             text NOT NULL UNIQUE,
    "gender"               text,
    "email"                text NOT NULL UNIQUE,
    "password"              text NOT NULL,
    "biography"             text,
    "birthday"              date,
    "profile_picture"       text DEFAULT 'default_profile_picture.webp',
    "language"              text DEFAULT 'FR',
    "darkmode"               boolean NOT NULL DEFAULT true,
    "phone_number"           text,
    "address"                text,
    "zipcode"                 text,
    "city"                   text,
    "latitude"                float,
    "longitude"               float,
    "created_at"              timestampz DEFAULT CURRENT_TIMESTAMP,
    "update_at"                timestampz
);
```

On **initialise** le script par la commande **BEGIN**.

En cas d'intervention nécessitant la **réinitialisation** de la BDD, on prévoit la **suppression** des **tables** existantes pour repartir sur une base "propre".

Avec la ligne de commande :

(DROP TABLE IF EXISTS nom_de_la_table).

Ceci fait, on passe à la **création de chaque table de la BDD**. Ici nous n'avons mis que l'extrait de la table **user** et des tables de liaisons **event_has_tag** et **user_follow_user**.

Nous créons donc la table **user** avec la commande **CREATE TABLE IF NOT EXISTS**. et détaillons chaque colonne de la table en définissant le type, et les spécificités. Ces informations sont consultables dans le dictionnaire des données expliqué dans la section précédente.

(...)

```
CREATE TABLE IF NOT EXISTS "event_has_tag"
(
    "code_event"      integer REFERENCES public.event (id),
    "code_tag"        integer REFERENCES public.tag (id),
    CONSTRAINT event_has_tag_pkey PRIMARY KEY (code_event, code_tag)
);

CREATE TABLE IF NOT EXISTS "user_follow_user"
(
    "code_user"       integer REFERENCES public.user (id),
    "code_user2"      integer REFERENCES public.user (id),
    CONSTRAINT user_follow_user_pkey PRIMARY KEY (code_user, code_user2)
);

COMMIT;
```

Pour les **tables de liaisons** la procédure est identique.
S'agissant de données en références aux Id des tables event tag et user la **REFERENCE** est précisée.

À la fin du script on lance le **COMMIT** si il n'y a pas eu de **ROLLBACK** en cas d'erreur détectée durant la **transaction**.

C. L'utilisateur et ses interactions

L'**élément pivot** du succès d'un **projet** d'application est l'**expérience proposée** à l'utilisateur. En effet, si celui-ci est **trop limité** dans ses possibilités d'interactions ou au contraire **si trop de possibilités** s'offrent à lui et qu'il ne sait pas par où commencer il va se **lasser rapidement**. Il est **essentiel de bien choisir le public visé et les actions** que tout utilisateur pourra réaliser.

1. Le public visé et les rôles :

Notre product owner étant issu du monde de la communication marketing il avait réalisé de son côté une étude de marché afin de déterminer quel public serait intéressé par l'application et les événements qu'elle proposerait. Sans surprise, avec des manifestations de type concerts, festivals, food-court, compétitions sportives, expositions etc... c'est le **public des 20-35 ans** qui s'est imposé.

Deux **types d'utilisateurs** ont également été naturellement identifiés : Les simples visiteurs, et les utilisateurs authentifiés.

Comme indiqué plus haut dans la section MVP, le **visiteur** n'aura d'accès qu'à 3 **Pages** de l'application. La page d'accueil l'invitant à se connecter si il s'est au préalable

enregistré, la page d'**inscription** et la page “**A propos**” qui présente notre équipe de développement.

Un **utilisateur authentifié** a, quant à lui, accès aux **pages d'un visiteur** mais également à l'**ensemble des fonctionnalités** de l'application (le Feed d'actualité, la page de recherche d'évènements, son profil, ses favoris etc...)

2. User stories :

Tous les scénarios d'interactions possibles pour un utilisateur ont été écrits dans un **user stories**, il s'agit d'un **tableau** dont les entêtes de colonnes permettent d'**identifier** l'utilisateur, son **besoin** et l'**action** qu'il souhaite réaliser.

Voici le user stories réalisé durant la phase de conception théorique de l'application pour un visiteur :

“En tant que...”	“J'ai besoin de...”	“Afin de...”
Visiteur	M'inscrire	D'utiliser l'application
Visiteur	Me connecter	D'utiliser l'application & accéder à mon compte utilisateur

Ci-dessous un extrait des user stories d'un utilisateurs authentifié :

“En tant que...”	“J'ai besoin de...”	“Afin de...”
utilisateur connecté	visualiser les évènements du fil d'actualité	m'inspirer et trouver une activité
utilisateur connecté	accéder à la page de recherche des évènements	rechercher un événement précis par son nom ou sa catégorie
utilisateur connecté	accéder à mes évènements favoris	pour connaître leur actualité
utilisateur connecté	mentionner ma participation à un évènement	informer la communauté de Sonowistes
utilisateur connecté	accéder à mon profil	le modifier/me déconnecter
...

D. Road Map de conception de l'application

L'application a été réalisée en 4 sprints d'une semaine chacun. Les sprints permettent de définir l'avancement progressif de chaque partie de l'application suivant un planning. Ils permettent de garantir un équilibre entre le front et le back.

Ci dessous la roadMap des sprints planifiés :

Equipes	Sprint 0	Sprint 1	Sprint 2	Sprint 3
Frontend	Documentation, Roles, User stories, Wireframes, Workflow, MCD, MLD, Routes.	Architecture Front + authentication (team complète), pages profil, inscription, login, Feed, recherche, Mise en place de la logique d'état dans les composants, Mise en responsive design	Dynamisation champs de recherche, routes dynamiques pages évènements, connexion API (récupérer évènement), Page Amis, Favoris, déconnexion du compte	Finalisation derniers éléments front.
Backend	(Réalisé team complète)	Architecture Back (routes et méthodes), MPD base de données & déploiement Heroku + authentication (team complète), CRUD user, event, tag	avancement du CRUD sur tags et events (Recherche par catégories), finalisation CRUD User (social methods), recherche par nickname d'un user, recherche d'un événement par nom et par catégorie, gestion des erreurs	Finalisation derniers éléments back et demandes supplémentaires du front.

V. SPÉCIFICATIONS TECHNIQUES DE SONOW

A. Gestion du versionning de projet :

1. Le versionning avec Git :

L'équipe étant composée de plusieurs équipiers et aucun d'entre eux n'habitant dans la même région, il était nécessaire de **travailler de façon synchronisée**. L'idée est que chacun puisse développer le code de l'application dont il a la charge puis que l'ensemble soit **centralisé dans un “lieu” unique**. De plus, la perte de données à la suite d'une panne matérielle n'étant pas à écarter, il est important de **sauvegarder régulièrement son travail**. L'outil **open source Git** propose de **répondre à ces deux besoins**. L'idée est que chaque **développeur** de l'application puisse **déposer** sur un **serveur** le **code** qu'il **produit** et **récupérer** le code produit par ses **coéquipiers**, tout cela de **façon sécurisée** avec des **clés de certification**. Mais Git va au-delà même de ces deux aspects. Il permet aussi de procéder à du **versioning** c'est à dire **consigner** de façon **chronologique** les différents états d'un projet.

Aussi pour **imager le concept**, on peut comparer **Git** à un **immense entrepôt** proposant **d'occuper** un **emplacement** pour y stocker le code de son projet. Chaque emplacement dispose de **meubles tiroirs** dans lesquels on viendrait **déposer** ou **consulter** le **code** en le **clonant** comme on peut faire une **photocopie** en fonction du besoin de chacun au sein de l'équipe. On peut ainsi cloner les 3 premiers tiroirs sur les 10 disponibles par exemple.

Il existe **plusieurs outils logiciels** permettant d'utiliser **Git**. Notre équipe s'est servi de **Github** qui est en quelque sorte un **prestashop** permettant d'utiliser les **concepts** de Git et met à **disposition ses serveurs** pour les **dépôts** des différents projets. Github offre également une **interface graphique** sur le **web** pour **piloter ses projets**. Elle permet de gérer via une liste des tâches typées **kanban** le **planning** de développement. Chaque tâche peut être **convertie** en **branche** sur laquelle on peut **travailler sans risquer d'endommager** le reste du code validé tant que l'on a pas **fusionné (Merge)** l'ensemble.

Nous avons décidé de **travailler** selon le système du **mono-repo**, le code **front-end** et **back-end** étant alors accessible et **stocké** au **même** emplacement. La partie serveur API à la racine du projet (pour satisfaire aux contraintes de l'hébergeur du projet déployé), la partie client dans un répertoire dédié sur le repo.

Chaque **équipier créait** une **branche** au moyen de **github project** avec pour titre le nom de la fonctionnalité développée, puis se **positionnait** dessus **celle - ci** au moyen de la **commande git fetch origin** (pour rafraîchir les branches disponibles) puis **git switch nom_de_la_branche** pour se **déplacer** sur la branche choisie.

Une fois le **code testé**, on **merge la branche de travail sur la branche principale dev** en faisant au préalable une **pull request** vérifiée par le **git master**.

2. Versionning de la base de données :

Un autre aspect de versionning, géré uniquement en backend, est celui de la **base de données**. En effet, il est possible de créer un **historique des versions** d'une **BDD** afin de remplir les objectifs exposés plus haut. Qui plus est, la gestion d'une BDD étant sensible, il est primordial de **pouvoir**, à la manière "d'une machine à voyager dans le temps", **naviguer** au sein des **différents états** de celle-ci.

Pour y parvenir, nous avons **utilisé** le logiciel **Sqitch** qui est **dédié** à ces opérations.

Sqitch fonctionne via la **lecture** de **scripts** en extensions **.sh**. Ces fichiers font référence au langage **Shell d'Unix** (une interface système).

On commence tout d'abord par **initialiser** le **projet** sous **sqitch** avec la création d'un **fichier** dédié **init.sh** dans lequel on paramètre les **variable d'environnement** de la BDD **stockée** dans **postgres** (notre SGBD).

```
### VARIABLE D'ENV #####
export PGUSER=kjjoanbdqvjdqd          # User de la DB
export PGPORT=5432                      # Port de la DB

export PGHOST=ec2-54-228-218-84.eu-west-1.compute.amazonaws.com # Adresse serveur
export ENGINE=pg                           # Nom du service utilisé
export SQITCHTARGET=db:pg:$SQITCHNAME      # Target pour sqitch
#####
#
```

Ceci fait, on lance l'**initialisation** en définissant un **nom** pour la base de données (SQITCHNAME) ainsi qu'un **dossier** (SQITCHDIR) dans lequel on souhaite **travailler** au moyen de questions qui s'affichent à l'écran **read -p "...":**

```
## Init de Sqitch
read -p "Quelle est la cible de votre base de donnée (juste le nom de la base pour une BDD locale) ?: "
" SQITCHNAME
read -p "Dans quel dossier voulez vous initialiser Sqitch ?: " SQITCHDIR
sqitch init $SQITCHNAME --engine $ENGINE --top-dir $SQITCHDIR

## Config de Sqitch
sqitch config --bool deploy.verify true
sqitch config revert.no_prompt true

## Ajout d'une cible
sqitch target add origin $SQITCHTARGET
sqitch engine add $ENGINE origin

## Ajout d'une migration
read -p "Quel est le nom de la 1ière migration ?: " NAMEMIGRAT
read -p "Quelle note voulez vous inscrire pour '$NAMEMIGRAT' ?: " NAMENOTE
sqitch add $NAMEMIGRAT -n "$NAMENOTE"
```

- Avec la commande **sqitch init \$SQITCHNAME --engine \$ENGINE --top-dir \$SQITCHDIR**
- Dans la partie config de sqitch on paramètre la vérification de changements opérés après un déploiement avec **sqitch config --bool deploy.verify true**, et on désactive l'affichage d'un prompt avant l'exécution d'un revert (un rétropédalage) avec **sqitch config revert.no_prompt true**.
- L'ajout de la cible permet de définir la base de données Postgresql à manipuler avec **sqitch target add origin \$SQITCHTARGET**, et on sélectionne également le moteur à utiliser **sqitch engine add \$ENGINE origin**.
- Enfin on lance une première migration, avec deux questions pour définir le nom et un descriptif de la migration, on lance ensuite la commande **sqitch add \$NAMEMIGRAT -n "\$NAMENOTE"**

Sqitch **crée son arborescence de travail** : des **répertoires** permettant le déploiement (**deploy**), le rétropédalage (**revert**) et la vérification (**verify**). On **positionne** dans ces répertoires et plus particulièrement **dans les fichiers de révision** qu'il crée pour chaque migration le **script SQL** exécutant des **manipulations** de la **base de données**. (Par exemple le script du MPD). On peut **créer** une nouvelle **migration** avec la commande **sqitch add nom_migration -n "Descriptif_dela_migration"** dans un fichier bash review.sh

Ci-dessous le code du fichier :

```
● ● ●

#### Création d'une nouvelle révision de la DB avec sqitch ####

##### VARIABLE D'ENV #####
export PGUSER=kjjoanbdqvjdqd

#####
# Demande à l'user le nom de la révision
read -p "Nom de la révision (ex: *****_r2): " INPUTNAMEREV

# Demande à l'user la note de la révision
read -p "Note de la révision : " INPUTNOTEREV

# Création de la révision
sqitch add $INPUTNAMEREV -n "$INPUTNOTEREV"
```

Pour le **déploiement** on utilise un **script deploy.sh** avec la commande **sqitch deploy** suivi des éléments nécessaires pour établir une connexion.

B. Les technologies back-end :

L'ensemble de la partie back-end repose sur le **développement d'une API REST** et une **base de données SQL** que nous avons conçues avec **différentes technologies**.

Node.js est un **environnement d'exécution Javascript open source** permettant de développer des applications **côté serveur**. Il **fonctionne avec le moteur Javascript V8**. Son architecture d'entrée / sortie **repose sur le concept d'événements**. Il est tout indiqué pour les **applications dynamiques** et permet de **gérer plusieurs clients** à la fois de **façon plus optimisée** car il utilise moins de threads (fils d'exécution, tâches partagées par plusieurs processeurs) et sa boucle d'événement (Single Threaded Event Loop) attend les requêtes indéfiniment mais procède à un tri suivant le caractère bloquant ou pas de celles-ci. Node.js **utilise donc moins de ressources** et de **mémoire**, cette optimisation **garantit un traitement plus rapide des tâches** y compris celles à **forte intensité de données**.

Express.js est un **framework**, c'est -à -dire un **ensemble d'outils logiciels organisés** permettant de **concevoir des architectures d'applications basées sur node.js**. Il offre par exemple la **possibilité de créer un serveur**, comme celui de notre application, de façon efficace et peut être **complété par l'ajout de librairies** installables via **npm**.

Voici les **librairies** utilisées dans **notre projet** :

- **Bcrypt** pour **sécuriser les mots de passe** via un processus de **hachage** de ce dernier,
- **Joi** pour la **validation des schémas de données** attendus par l'api (Mot de passe, email, adresse, téléphone etc...) cela permet **d'éviter les erreurs** de saisie par un **utilisateur**.
- **Cors** pour la **sécurité** dans le cadre de la **gestion du partage de ressources** avec des **domaines externes**. C'est le cas lorsqu'une **api** est **interrogée** par un **client externe** ce dernier doit être **reconnu** comme **légitime**. Aussi un **client** disposant des **identifiants** et **mot de passe** pour accéder à une API peut être **rejeté si son domaine n'est pas validé** par le cors de l'API.
- **Jsonwebtoken** est également une **librairie** qui assure la **mise en place** de **jetons signés** permettant **d'affirmer l'authenticité** d'un **utilisateur** lors de sa **tentative de connexion**. L'échange est alors entièrement **sécurisé** entre les parties **client et serveur**.
- **Dotenv**, quant à elle, sert à la **gestion** des **variables d'environnement** nécessaires comme par **exemple** les **variables de connexions à la base de données** (nom d'utilisateur et mot de passe, port d'écoute etc...).
- Enfin, **Pg** permet la **connexion** à la **BDD** via un **pool** de connexion mais apporte en plus le **support** de **fonctionnalités PostgreSQL** telles que les **requêtes paramétrées**.

PostgreSQL est un **système de gestion de bases de données** relationnelles **utilisé** dans le cadre de la **programmation orientée objet** (POO). Il permet de **gérer le CRUD** (Lecture, Création, Mise à jour, Suppression) d'une BDD de **façon sécurisée**. Seuls

les utilisateurs ayant les rôles adéquats peuvent accéder et manipuler les données stockées grâce au langage SQL (Structured Query Language). Nous avons développé la base de données de SoNow au moyen des nombreux documents techniques exposés dans les sections précédentes. (MCD, MLD, MPD etc...)

Dans SoNow, tous ces composants permettent au serveur, qui est de type dynamique, de retourner une réponse lors de chaque requête HTTP émise par le client.

C. Les technologies front-end :

L'ensemble de la partie front-end de l'application a été codée avec la librairie open source REACT. Elle est légère et plus modulable qu'un framework. Elle a été développée par Méta en 2013. Son fonctionnement repose sur le concept de DOM virtuel (Un DOM plus rapide que le DOM du navigateur) et de composants (components) interconnectés qui disposent chacun d'un état (state). REACT gère l'affichage de l'application considéré comme la vue dans l'architecture MVC. Lors d'un changement de données ou d'état, la page HTML générée est modulable et le rafraîchissement ne s'effectue que sur l'état du composant ayant changé, ce changement ayant provoqué une réaction (d'où le nom porté par cette librairie). L'avantage est de pouvoir faire des transitions d'affichages sans changer de page et sans la rafraîchir intégralement, l'expérience de l'utilisateur reste fluide. Les composants étant interconnectés peuvent partager des données et de ce fait interagir. Pour résumer, la philosophie de REACT consiste à découper la page en composants qui sont des fonctions JS, indépendantes et réutilisables dans le but de structurer l'application.

Dans SoNow, tous ces concepts ont permis de développer par exemple la page de création d'un compte (createAccount), que vous pourrez trouver en totalité en annexes.

Dans celui-ci sont importées les packages, styles css et modules nécessaires, puis on détaille la fonction : une constante en destructuring des différents champs du formulaire avec le state initial. Puis on retrouve les différentes actions applicables sur les champs et enfin le renvoi du rendu de la page contenu dans un return. Pour finir on exporte le composant pour qu'il puisse être appelé par son parent (App).

Redux, quant à elle, est un centralisateur d'états, elle permet de gérer ceux-ci de façon globale pour toute l'application. Grâce à cette bibliothèque on a plus besoin de modifier les états directement mais on développe les changements qui leur sont apportés. Ces changements sont appelés des actions, elles peuvent être stockées et appliquées à différents états. L'utilisation de Redux dans notre application a été décidée en prévision des évolutions en termes de volumes d'états à gérer dans le futur de SoNow. L'utilisation d'un store permet de diffuser un état à tous les composants concernés plus rapidement que par l'utilisation d'états locaux à chaque composant qu'ils communiquent aux autres. Avec Redux tous les composants peuvent ainsi accéder aux données de l'état de façon plus aisée. Elle apporte ainsi un gain de performance non négligeable dans le cas de gros volumes de composants à modifier.

La communication de la partie front-end a été faite au moyen de middlewares implémentés dans le store de Redux combinés à la librairie Axios pour les requêtes

HTTP auprès de notre **API**. Ainsi elle **récupère** des **données** sous la **forme de tableau d'objets JSON** pour ensuite les **exploiter** dans les **composants react**.

La partie **graphique** a été réalisée au moyen du **framework Semantic UI**. L'avantage étant un **gain de temps** considérable grâce aux **outils proposés et immédiatement utilisables**.

D. La sécurité de l'application

L'**expérience proposée** par l'application est **importante** pour **séduire l'utilisateur**, néanmoins un **autre aspect primordial** aujourd'hui pour gagner la confiance de ce dernier est celui des moyens de **sécurisation** des **données personnelles collectées**. De plus, depuis la création en 2018 du règlement général sur la protection des données (**RGPD**), **toute application** se doit d'être **conforme** aux **attentes** en matière de **sécurité**. Pour cela **plusieurs méthodes** peuvent être **combinées**. Pour **SoNow** nous **avons mis** en place **certaines** d'entre elles côté **client et serveur** :

- Client (JWT) :

Afin d'accéder à l'intégralité des fonctionnalités proposées et notamment celles ayant accès aux données gérées par le **SGBD**, il est **obligatoire** de s'authentifier au préalable au moyen du **formulaire** mis à disposition.

Afin de **sécuriser** cette **opération**, nous avons utilisé les **jetons JWT (Json Web Token)**. Développée dans un **middleware** dédié au **login**, celui-ci utilise la **librairie Axios** et **plusieurs actions** de **Redux** et son **store**.

```
import axios from 'axios';
import { getEvents, SUBMIT_LOGIN, submitLoginSuccess, submitLoginError, LOGOUT } from '../actions';

const loginMiddleware = (store) => (next) => (action) => {

  if (action.type === SUBMIT_LOGIN) {
    next(action);
    const state = store.getState();
    let url = 'https://sonow.herokuapp.com/api/user/login/';

    const config = {
      method: 'post',
      url: url + '?nocache=' + new Date().getTime(),
      headers: {
        'content-type': 'application/json; charset=utf-8',
        'Access-Control-Allow-Origin': 'https://sonow.herokuapp.com/api'
      },
      data: {
        email: state.user.login.emailInput,
        password: state.user.login.passwordInput
      }
    }

    axios(config)
      .then((response) => {
        store.dispatch(submitLoginSuccess(response.data.accessToken, response.data.refreshToken, response.data.user));
        localStorage.setItem('accessToken', `${response.data.accessToken}`);
        localStorage.setItem('refreshToken', `${response.data.refreshToken}`);
        localStorage.setItem('id', `${response.data.user.id}`);
        store.dispatch(getEvents());
      })
      .catch(() => {
        store.dispatch(submitLoginError());
      });
  }
}
```

Si le **type d'action** est celui de soumettre le login (**SUBMIT_LOGIN**) alors on récupère l'état dans une constante avec la méthode **getState** du **store**, puis on configure les paramètres pour la soumission du formulaire (la méthode **HTTP** de type **POST**, l'url d'appel à l'api, le **headers** dans lequel on définit l'adresse du site ayant l'autorisation d'accès et de contrôle pour stocker le token, les données envoyées qui sont l'**identifiant** (email) et le **mot de passe** de l'utilisateur. On lance ensuite **axios** avec pour paramètres la variable **config**, si la réponse est correcte alors on stocke le token en local storage

sinon on récupère l'erreur par le biais du **catch** qui la **signale**.

```
    } else if (action.type === LOGOUT) {
      next(action);

      let url = 'https://sonow.herokuapp.com/api/user/logout/'

      const config = {
        method: 'get',
        url: url + '?nocache=' + new Date().getTime(),
        headers: {
          'content-type': 'application/json; charset=utf-8',
          'Access-Control-Allow-Origin': 'https://sonow.herokuapp.com/api'
        },
      }

      axios(config)
        .then(() => {
          localStorage.removeItem('accessToken');
          localStorage.removeItem('refreshToken');
          localStorage.removeItem('id');
        })
        .catch((error) => {
          console.log(error.message);
        });
    } else {
      next(action);
    }
  };

export default loginMiddleware;
```

Dans le cas où l'**action** n'est pas de type **submit**, elle peut alors être de type **LOGOUT** (la déconnexion). On paramètre l'url d'appel à l'API dédiée à cette fonctionnalité côté **serveur** dans une **variable** puis les **paramètres**, et on lance à nouveau **axios** qui **supprime** du **local storage** le **token** via la méthode **removeItem()**, si l'**action** ne peut être menée à bien le **catch** renvoie un message d'erreur via un **console.log**.

- Serveur (CORS, JWT, Prepared Queries, validation & regex) :

Pour commencer, nous n'avons **autorisé que le serveur front à opérer des requêtes avec le CORS en paramétrant l'origine sur son URL dans le point d'entrée de notre API index.js.**

Aucun appel ne peut **aboutir** si l'utilisateur n'est pas authentifié ce qui génère un **JWT** (Service authToken.js) **récupéré** par le **client** comme expliqué plus haut.

```
require('dotenv').config();
const jwt = require('jsonwebtoken');
const SECRET_KEY = process.env.ACCESSION_SECRET_KEY;

const authMiddleware = (req, res, next) => {

  let accessToken = req.headers['x-access-token'] || req.headers['authorization'];
  if (!accessToken && accessToken.startsWith('Bearer ')) {
    accessToken = accessToken.slice(7, accessToken.length);
  }

  if (accessToken) {
    jwt.verify(accessToken, SECRET_KEY, (err, decoded) => {
      if (err) {
        return res.status(401).json('token_not_valid');
      }
    });
  }
};

export default authMiddleware;
```

On appelle les **variables d'environnement** inscrites dans le fichier **.env** avec **require('dotenv').config()**;

Idem avec le **module jsonwebtoken** que l'on stocke dans une **constante jwt**.

AccessToken détient les **infos contenues** dans le **headers** du client front.

Si les **jetons correspondent** alors `jwt.verify` permet le **décodage** Si il y a une **erreur** alors on **revoit un json avec le message d'un token non valide**, sinon l'**authentification aboutit** et la **session est active**.

```
    } else {
      req.decoded = decoded;

      const expiresIn = 24 * 60 * 60;
      const newToken = jwt.sign({
        user: decoded.user
      },
      SECRET_KEY,
      {
        expiresIn: expiresIn
      });
    }

    res.header('Authorization', 'Bearer ' + newToken);
    next();
  }
};

} else {
  return res.status(401).json('token_required');
}

};

module.exports = authTokenMiddleware;
```

Si il n'y a pas de jeton, alors on **renvoie un message** informant que l'**authentification en requiert un**.

Dans **toutes les transactions opérées entre l'API et la BDD** nous avons mis en place au niveau des **modèles** des **requêtes préparées** et l'utilisation de **jetons SQL**. Très **sommaire mais** il s'agit d'une **bonne pratique conseillée**.

Ci-dessous un exemple de **requête préparée** dans le **modèle event.js** permettant la recherche d'un **événement** par son **Id** :

```
... ...

//Rechercher un évènement par son ID.
async findByPk(eventId) {
  //Je prépare une requête sql séparément pour éviter les injections.
  //J'utilise les jetons sql également par souci de sécurité.
  const preparedQuery = {
    text: `
      SELECT *
      FROM public.event
      WHERE id = $1
    `,
    values: [eventId],
  };
  const result = await client.query(preparedQuery);
  if (result.rowCount === 0) {
    return null;
  };

  return result.rows[0];
},
```

Dans la **méthode asynchrone** `findByPk` ayant pour **paramètre** `eventId`, on crée dans la **constante preparedQuery** la requête, objet ayant **deux propriétés**, la **première** est le **texte** de la requête **SQL** qui comprend un **jetons (\$1)**, la valeur de ce **jeton est fournies** dans la seconde propriété **“value”** et correspond à l'**ID de l'événement** demandé. On **interroge la BDD** et on **stocke le résultat** dans la **constante result**. Si la **BDD** ne **retourne rien** qui correspond à l'id alors on **renvoie null** au **controller** sinon on **retourne la ligne correspondante**.

Afin d'ajouter une couche de sécurité sur la page d'inscription, et pour prévenir des éventuelles erreurs de saisie d'un utilisateur dans un champ du formulaire, nous avons tout de même implémenté avec la librairie "Joi" un validateur de schéma sous forme de middleware avec l'utilisation de regex (expression régulière décrivant la logique structurelle d'information attendue de la part de l'utilisateur par l'API) pour les champs du formulaire définis.

```
const { ApiError } = require("../services/errorHandler");

module.exports = (prop, schema) => async (request, _, next) => {
    try {
        await schema.validateAsync(request[prop]);
        next();
    } catch (error) {
        next(new ApiError(error.details[0].message, { statusCode: 400 }));
    }
};
```

a une erreur alors le catch intervient et renvoie une erreur et un statut 400 (syntaxe de requête erronée).

Le schéma permettant cette validation est le suivant :

```
const Joi = require('joi');

module.exports = Joi.object({
    email: Joi.string().required().regex(/^[^\w-\.]+@[^\w-]+\.\w+[\w-]{2,4}$/,),
    password: Joi.string().required().regex(/^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)(?=.*[-!@#$%^&_])[^\w]{8,15}$/,),
    firstname: Joi.string().required(),
    lastname: Joi.string().required(),
    nickname: Joi.string().required(),
}).required();
```

On exporte l'objet dont Joi va se servir pour valider les données entrées via la commande module.exports. Toutes les propriétés de l'objet sont requises par la méthode .required(), cela sous-entend que si l'utilisateur laisse un champ du formulaire d'inscription vide ce dernier ne pourra pas être soumis car Joi ne validera pas.

Avec .string() on demande à Joi de générer un objet qui permet de comparer et valider un type de données au format chaîne de caractère. En d'autres termes si le type n'est pas respecté alors la validation ne sera pas opérée.

On récupère les propriétés et les schémas (regex) puis on lance dans un "try" la fonction qui valide la propriété testée en fonction du schéma défini. Si tout est correct on passe à la suite du code avec le next(); si il y

Enfin la **méthode .regex()** permet de **définir des contraintes de structure à respecter** par les **données envoyées**. Ici on **applique les expressions régulières** sur la **structure** que doit **respecter un email et un mot de passe** :

- Email : mots séparés de tiret ou de points + @ + des mots séparés de tirets + . + mot de 2 à 4 caractères
- Mot de passe : composé minimum de 8 caractères, maximum 15 caractères, avec 1 majuscule 1 minuscule 1 chiffre et un caractère spécial minimum obligatoires.

VI. GESTION DE PROJET

A. Présentation de l'équipe

L'équipe du projet SoNow se compose de **5 personnes** : **3 en front** spécialité **REACT** et **2 en back** spécialité **Data**. Chaque projet a été présenté et les personnes ont émis des vœux pour se positionner sur l'un d'entre eux. Un **tirage au sort** a permis de **définir les équipes en fonction des vœux émis**.

Front-end :



Raphaël



Elinor



Romain

Back-end :



Sébastien



Frédéric

B. Organisation du travail

Mode d'organisation :

Chaque **fonctionnalité** étant **définie** par une **route** dans le **cahier des charges** nous nous sommes **appuyé** sur ce **tableau** pour **créer** sur le **Github** **projet** des **tâches** avec les **rubriques correspondantes** (Front, Back, BDD, VS Code, PostgreSQL, Heroku App, Vercel etc...). Un **développeur** qui voulait **prendre en charge** et **travailler** sur une **fonctionnalité**, n'avait plus qu'à **s'ajouter**, **créer l'issue** puis la **convertir** en **branche**. Aussi il était **simple** de **savoir** qui **faisait quoi** et ne pas se retrouver à travailler sur la même fonctionnalité par erreur, hors cas de **pair programming** bien entendu. D'ailleurs ce dernier a **été pratiqué**, il **consiste à utiliser** une **extension** de **VS Code** appelée **liveshare** permettant le **partage du code** à un ou plusieurs **équipiers** et leur **permettre d'intervenir**. Nous avons **travaillé ainsi** sur les **grosses fonctionnalités** comme l'**authentification** par exemple.

Déroulement d'une journée de travail :

Durant un mois nous avons **travaillé** minimum **35h/semaine**. Les **horaires** choisis étaient **classiques à ceux de la formation** à savoir **9h - 17h**. Si un développeur souhaitait travailler en plus de ces horaires, il le faisait savoir aux autres par souci d'organisation.

Tous les matins nous avons **commencé** par une **réunion vocale** dite "**kickoff**" à laquelle était **souvent convié** notre **product owner externe** pour éventuellement **traiter** de **question avec lui**. Celle - ci permettait de traiter en outre les points suivants :

- **Rappel des réalisations accomplies** de la veille et de celles **restant à œuvrer** avec l'aide de **GitHub project**.
- **Définition des tâches prévues de chacun** durant la journée par une prise de parole individuelle et **exposé des éventuelles difficultés rencontrées** à la mise en œuvre du développement d'une fonctionnalité, accompagné d'une **recherche de solution opérable en groupe**.

A la fin de cette réunion, les **sections front et back se séparaient** pour au choix partir sur un **travail individuel ou en pair programming**.

Nous avions pour habitude de faire un **point écrit sur le groupe privé Discord à la pause déjeuner** de l'avancement de la matinée.

En **fin de journée**, nous nous **retrouvions de nouveau quelques minutes** pour débriefer des réalisations accomplies.

C. Outil de gestion utilisés

Afin de coller aux contraintes géographiques, nous avons utilisé des outils permettant une organisation adaptée.

Slack et Discord sont deux **logiciels de communication** accès **VOIP et conversation** typées **messagerie instantanée**. Nous avons créé **différents groupes de conversation**, un **groupe intégral SoNow** avec les **5 développeurs** et le **product owner externe**, une **conversation** pour l'équipe **frontend**, une **conversation** pour l'équipe **backend**. Nous avons de ce fait pu **échanger en vocal** ou à l'**écrit** et **partager des documents** de manière aisée. Ces groupes sont **privés** et permettent de **conserver un historique**.

Pour la **création de la documentation** et sa **centralisation** nous avons utilisé les **outils suivants** : Notion qui permet de **prendre et organiser tous types de notes** en **simultanée** avec **insertions de documents** et **gestion** du format **markdown** (celui utilisé pour la rédaction des readme.md des repos GitHub), ce dernier nous a notamment permis de **gérer nos carnets de bord respectifs** et celui des **sprints du groupe**. Figma et Whimsical pour la **création des wireframes, MCD, MLD etc...** Google drive & google sheet pour le **dictionnaire des données**.

Pour la **gestion du projet**, nous avons utilisé **GitHub Project**. Il s'agit d'une **fonctionnalité de GitHub intégrée à chaque repos** créé. **Présenté** sous la forme d'un **kanban** (Todo, work in progress, done) on **crée une tâche** qui peut être **convertie en issue puis en branche**. La branche est ensuite **récupérable** dans le **logiciel de développement**. Une fois le **code produit** un émet un **commit** et un **push** que l'on **merge** sur le code déjà existant. D'ailleurs la **totalité du versionning** du projet a été gérée avec **Git et Github**, outils open source que nous avons utilisés durant toute la formation.

Pour le **développement**, le logiciel **VS Code**, éditeur de **code combiné à un panel d'extensions** gratuites de son store intégré. Pour le **test** des **endpoints API**, nous avons utilisé le logiciel **postman**.

Enfin **même si** ce n'était **pas obligatoire** nous avons **fait un déploiement**. L'**API** sur les **serveurs**, encore pour le moment gratuits, de l'hébergeur **Heroku**. Le **front-end** sur ceux de **Vercel**.

D. Répartition et détails des rôle de chacun

Lors de la première journée de travail nous avons **démarré** par une **réunion visant à définir les rôles sur le projet**, la **répartition** s'est faite de **façon** très aisée après **concertation des préférences de chacun en fonction des domaines** dans **lesquels** nous nous **sentions** plus ou moins à l'**aise**. Le poste de **product Owner principal** a toutefois été **naturellement octroyé** à **Jérémy Pyronnet** qui a **contacté Sébastien** pour lui **présenter l'idée** de l'application à **développer**.

Répartition des rôles :

Elinor

Scrum master, elle a supervisé le planning du projet et son avancement en consignant ce dernier dans le carnet de bord de l'équipe à travers l'application Notion.

Front Développeuse

Raphaël

Git & GitHub master, il a apporté son aide à tous les coéquipiers pour la gestion des branches et les merging.

Front Développeur

Romain

Lead Dev Front, il a supervisé toute la partie front de l'application.

Référent REACT, chaque équipier a pu le solliciter en cas de question sur cette technologie.

Sébastien

Product Owner associé, contacté par le product owner principal Jérémy Pyronnet

Back Développeur

Frédéric

Lead Dev Back, il a supervisé la partie back du développement et les choix techniques.

E. Programme des sprints

Le travail a été planifié en **4 sprints d'une semaine** suivant la **méthode AGILE SCRUM**. À chaque **Mi-sprint** nous retrouvions dans un des **salons du serveur Discord** de l'**école O'clock** nos tuteurs **Jordan BRULL** et **Virginie LEMAIRE** pour faire un **point** avec eux, évoquer les **difficultés** et la **méthode de travail utilisée**, mais nous avions également des **réunions** avec **Jérémy PYRONNET** pour lui faire part de notre **avancée**. Chaque **transition entre deux sprints** était effectuée par une **présentation orale** de l'un des membres de l'équipe à l'ensemble des autres groupes, un groupe était désigné pour poser des questions auxquelles nous nous entraînions à répondre en vue de la présentation finale en live sur Youtube. (Présentation à laquelle Jérémy nous a fait l'honneur d'assister)

Durant le **sprint 0** nous avons réalisé l'**ensemble des documents** du **cahier des charges** au fil des réunions avec mes coéquipiers. Aussi nous avons produit des documents supports tels que **le MCD, le MLD, le dictionnaire des données** pour la partie liée à la BDD, mais également **les routes d'appels à l'API, l'arborescence de l'application** et **les wireframes**. **Chacun** a pu **participer** sur la création des différents documents car nous avons utilisé **figma** qui permet de **collaborer** sur une **réalisation en simultanée**. Seuls les **wireframes** ont été **produits individuellement**.

Une fois **validés** par nos **responsables pédagogiques** nous sommes **passés** à l'étape suivante, le **sprint 1** en démarrant le **développement** dans **VS CODE** :

- **Clonage** du repo vierge
- **Initialisation** du **projet REACT** par l'équipe **front** et **création** dans un **répertoire data** de **fausses données** pour **pouvoir** amorcer dès que possible la **dynamisation**
- **Initialisation** du projet au **niveau** de L'API, **installation** des **dépendances** par **npm** et **lancement** de la création de l'**architecture globale** avec **dossiers** et **principaux fichiers**

Le **sprint 2** a été celui au cours duquel nous avons réalisé, pour l'API, le **CRUD** en développant des **méthodes** dans les **contrôleurs** permettant la **circulation de la donnée** et les **requêtes SQL** dans les **Modèles faites à la BDD**. Côté **client**, mise en **dynamisation** avec les **fausses données créées** lors du **sprint 1** puis via les **Json après appels aux endpoints**. À noter que faute de temps suffisant, **certaines fonctionnalités prévues** n'ont pas pu être **raccordées** (mise en favori, participation par exemple) **d'autres** pages sont en **attente de création** (ajouter/modifier supprimer un évènement, modifier/supprimer le profil utilisateur) **Tout est fonctionnel** côté **API**.

Le **sprint 3** est celui de la **finalisation** des **fonctionnalités en cours de développement** lors du **sprint 2**, il est en effet **trop risqué** de continuer à coder à ce **stade** et causer une **erreur fatale** à l'**intégralité** de l'**application**. En **parallèle** nous avons **préparé** la **présentation Youtube** auprès de l'équipe pédagogique et notre product Owner externe.

VII. PRODUCTION

A. Mes réalisations personnelles

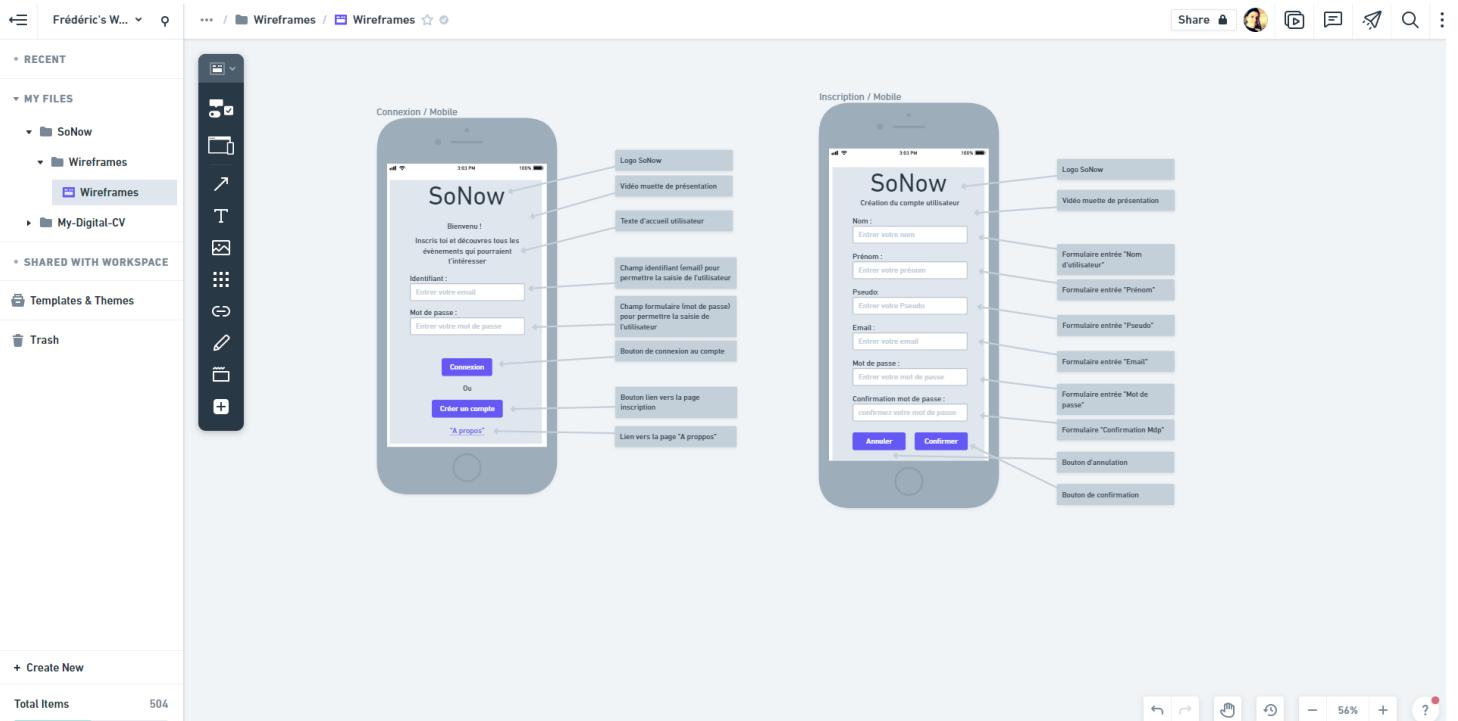
1. Création des wireframes des pages d'inscription et de connexion :

Lors des premières réunions du **sprint 0**, l'équipe a défini les grandes lignes concernant l'**interface** du projet. J'ai pris en charge, suite à un **tirage au sort**, la **réalisation** des **wireframes** des **pages d'inscription** et de **connexion**.

Pour ce faire j'ai utilisé l'outil en ligne gratuit **Whimsical** qui dispose de nombreux outils intuitifs et les rendus, tout en restant dans les standards attendus d'un wireframe, sont **agréables à regarder**.

Dans l'**arborescence** des projets réalisables j'ai créé un **répertoire dédié** et choisi un **modèle type vierge** de **wireframe**. J'ai commencé par l'**interface** au **format mobile first** car c'est **celui - ci** qui est **priorisé** par le **product owner** qui à termes souhaiterait lancer l'**application** sur les **stores Android et Apple**.

Ci-dessous une capture d'écran de Whimsical :



À gauche on remarque la barre d'outils intuitive qui permet de sélectionner différents composants. J'ai démarré par le "Add Frame" pour la page de connexion offrant la possibilité de rajouter un modèle d'appareil, et j'ai choisi "Phone". Pour connaître les interactions possibles et les informations à afficher dans la page, j'ai consulté les user stories et le dictionnaire des données. Au moyen de la commande "Add Element" j'ai ajouté le nombre de "Text input" nécessaire à la construction du formulaire à savoir l'identifiant défini par l'email et le mot de passe, les labels pour les présenter, et enfin les boutons pour lancer la connexion ou la création d'un compte. Au-dessus un court texte d'accueil invitant l'utilisateur à s'inscrire.

Le logo est positionné en Header et un lien vers la page "À propos" en footer qui présente l'équipe de développement.

Les wireframes ont pour objectifs de présenter visuellement une ébauche de l'interface et le positionnement des fonctionnalités attendues de chaque élément. Aussi on détaille au moyen de légendes chacune d'elles.

Une fois ce wireframe terminé en mobile first, je transpose dans un nouveau "frame" que j'ai choisi type window et qui offre la forme d'une fenêtre de navigateur en Desktop. (Cf annexes pour le rendu)

J'ai revu légèrement les marges entre chaque élément (la structure de la page étant simple il n'y a pas besoin de trop de modifications).

2. Création de l'architecture back-end et du router :

Le **sprint 1** a été celui durant lequel l'équipe a **démarré le développement** dans l'**éditeur de code**. Pendant que mon binôme se documentait sur les méthodes d'hébergement pour la BDD et notre API, J'ai **pris en charge la création de l'architecture**.

Après avoir **installé** les **dépendances** via **npm**, j'ai **créé l'arborescence de dossiers et fichiers vides de code** que nous avons **agrémentés au fil des semaines**. Dans le **package.json** sont mentionnés les différents **scripts** permettant de lancer le serveur. Le **point d'entrée** de l'**API** est **index.js** à la racine du projet.

J'ai plus particulièrement **traité le code des routeurs et contrôleurs**. La phase de mise en route du serveur et vérification de son fonctionnement a été faite en pair programming.

Le **routeur** a pour **utilité de gérer les endpoints** en fonction des **appels de la partie client**. Nous avions **au départ** décidé lors du sprint 0 de n'en faire **qu'un seul**. Puis en constatant la **quantité de routes**, il nous était plus aisés de les **différencier par composant**.

```
...  
  
const express = require('express');  
const router = express.Router();  
const userRouter = require('./user');  
const eventRouter = require('./event');  
const tagRouter = require('./tag');  
  
router.all('/', async function(_, res) {  
    res.send("Welcome on SoNow API !");  
});  
  
router.use('/user', userRouter);  
router.use('/event', eventRouter);  
router.use('/tag', tagRouter);  
  
module.exports = router;
```

Nous avons donc un **gestionnaire de routage index.js** composé de **4 routes** :

'/' permet de voir dans le navigateur si celui-ci fonctionne en renvoyant un **message via res.send**.

'/user' appelle la constante userRouter pour toutes les **routes** concernant les méthodes liées à **l'utilisateur**.

'/event' appelle la constante eventRouter pour toutes les **routes** concernant les méthodes liées aux **événements**.

Enfin '/tag' appelle la constante tagRouter pour toutes les **routes** concernant les méthodes liées aux **tag/catégories**.

Dans ces trois constantes se trouve les code respectifs des fichiers user.js, event.js et tag.js les routeurs secondaires.

La **structure de ces 3 composants** étant **similaire** je vais vous **présenter** celle de **l'utilisateur**. L'ensemble étant **réutilisable** après quelques **ajustements** pour **event et tag**.

Extrait du code du userRouter (Totalité consultable en annexe) :

```
...  
  
const express = require('express');  
  
const validate = require('../validation/validator');  
const createSchema = require('../validation/schemas/userCreateSchema');  
const updateSchema = require('../validation/schemas/userUpdateSchema');  
  
const router = express.Router();  
  
//Importation du controller des utilisateurs.  
const { UserController: controller } = require('../controllers');  
  
const controllerHandler = require('../services/controllerHandler');  
  
//Les différentes routes de l'API pour l'utilisateur.  
  
//Route pour récupérer tous les utilisateurs.  
router  
  .route('/')  
  .get(controllerHandler(controller.getAllUsers));  
  
//Routes pour créer ou (dé)connecter l'utilisateur.  
router  
  .route('/login')  
  .post(controllerHandler(controller.loginUser));  
  
router  
  .route('/signup')  
  .post(validate('body', createSchema), controllerHandler(controller.createUser));  
  
router  
  .route('/logout')  
  .get(controllerHandler(controller.logoutUser));
```

Je commence par **appeler express** dans une constante ainsi que sa méthode native Router().

Le **routeur** étant celui de **l'utilisateur** certaines fonctionnalités sont concernées par l'utilisation du **validateur**, j'importe alors le **middleware** ainsi que les **deux schémas**.

J'importe le **contrôleur** référent aux **méthodes** pour l'utilisateur.

J'énumère les **routes** permettant d'accéder à chaque **endpoint**. La structure d'appel est toujours la même :

On appelle la **constante router**, on définit la **route** avec **.route('/chemin')**, puis la **méthode HTTP** avec son **verbe**, le **controller** et enfin la **méthode**.

À noter que l'utilisation de **.route()** permet d'organiser les différents **endpoints** de façon plus synthétique en enchaînant **.get**, **.post**, **.patch** et **.delete**.

Ainsi pour créer par exemple un compte utilisateur on appelle la méthode `createUser` dans son controller avec la route '/signup' via l'adresse <https://sonow.herokuapp.com/api/user/signup>.

3. Circulation de la donnée au travers des contrôleurs

Intéressons-nous de plus près au **contrôleur**. Son rôle est **primordial** car celui - ci est **consacré à la gestion de la circulation de la donnée** entre le **client front** (assimilé à la vue ici) et le **modèle** dans une **architecture MVC**.

Pour SoNow, je suis intervenu sur le **userController** et ai eu à développer plusieurs méthodes.

Extrait du code du UserController (Totalité consultable en annexe je n'ai conservée qu'une méthode createUser) :

```
require('dotenv').config();
const userDataMapper = require('../models/user');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const SECRET_KEY = process.env.ACCESS_SECRET_KEY;
const REFRESH_SECRET_KEY = process.env.REFRESH_SECRET_KEY;

const { ApiError } = require("../services/errorHandler");

module.exports = {

    //Méthode qui permet à l'utilisateur de se créer un compte sur l'application.
    async createUser(req, res) {
        const userDb = await userDataMapper.findByEmail(req.body.email);

        if (userDb) {
            throw new ApiError('User already exists', { statusCode: 400 });
        } else {
            //On crypte le mot de passe.
            const encryptedPwd = await bcrypt.hash(req.body.password, 10);
            const newUser = {
                firstname: req.body.firstname,
                lastname: req.body.lastname,
                nickname: req.body.nickname,
                email: req.body.email,
                password: encryptedPwd
            };
            const insertUser = await userDataMapper.insert(newUser);

            res.json(insertUser);
        };
    },
};


```

Je commence par importer les variables d'environnement grâce à la méthode native `.config()` de la librairie `dotenv`.

Je fais également appel aux différentes dépendances pour la bonne exécution des méthodes développées :

- `bcrypt` pour le hachage des mots de passe.
- `jsonwebtoken` pour la sécurité lors de l'authentification.
- les variables d'accès et du rafraîchissement du token avec `process.env` dans deux constantes distinctes.
- Le modèle `user` dans une constante `userDataMapper` (qui sera exposé dans la prochain point).

- Enfin pour gérer les erreurs, j'importe le service `errorHandler` dans une constante `ApiError` qui pourra être utilisée pour envoyer un message.

Le contrôleur est composé de plusieurs méthodes faisant office de endpoint dans le routeur. Ces méthodes peuvent faire appel à des ressources externes pour obtenir des données et les circulariser. Comme je ne sais pas si et quand la source externe répondra, j'utilise les promesses avec `async/await`. C'est le cas de la méthode `createUser` utilisée lors de l'inscription d'un utilisateur.

Je commence par interroger la BDD pour savoir si l'email entré dans le formulaire et envoyé par le client existe déjà. (Je détaillerai le `findByEmail` du `UserDataMapper` dans le point suivant).

À l'aide d'une **condition if/else** je développe les **réponses** du **contrôleur** en fonction de la **valeur** de **userDb**.

Si l'email entré dans le **formulaire existe** en **BDD** alors on **renvoie ApiError** avec un **message** et un **statut 400**. (Cela veut dire que l'utilisateur existe déjà et que je ne peux le créer de nouveau)

Sinon je **sécurise** le **mot de passe** avec **méthode native .hash** de **bcrypt**

Je **crée** un **nouvel utilisateur newUser** avec les **propriétés** en **provenance** du **formulaire front-end** reçu

Je termine par un **envoi** vers le **userDataMapper** avec la **méthode .insert avec** comme **argument newUser**.

Enfin le **contrôleur renvoie** un **tableau d'objet json** qui **pourra être utilisé** par le client au besoin.

4. Création du modèle dataMapper User

Afin de **pouvoir accéder à la donnée** demandée par le **client**, il est **nécessaire** de **développer des composants** permettant d'émettre des **requêtes** en **langage SQL**. Ces **composants** sont appelés les **Modèles**.

Durant l'apothéose j'ai bien évidemment **créé l'un** d'entre **eux** et certaines **méthodes** pour **obtenir les données sécurisées** dans la **BDD**.

Afin de suivre un certain fil d'Ariane, je vais vous **présenter** un extrait du **User data Mapper** et les **méthodes findByEmail et Insert** évoquées dans la **partie précédente**.

Bien entendu l'intégralité du **code** est **disponible** en **annexe** mais je ne traiterai que la logique de ces deux fonctions ici.

```
const client = require("../config/db");

module.exports = {

    //Retrouver un user par son email pour l'authentification
    async findByEmail(reqEmail){
        //Je prépare une requête sql séparément pour éviter les injections.
        //J'utilise les jetons sql également par souci de sécurité.
        const preparedQuery = {
            text: `
                SELECT *
                FROM public.user
                WHERE email = $1
            `,
            values: [reqEmail],
        };

        const result = await client.query(preparedQuery);

        if (result.rowCount === 0) {
            return null;
        };

        return result.rows[0];
    },
}
```

Ci-contre l'extrait de code du modèle user :

J'importe via un require dans une constante client le code du pool de connexion à la BDD du fichier db.js .

Puis j'exporte les méthodes dans un objet avec la commande module.exports.

La méthode **asynchrone** `findByEmail` récupère le paramètre `reqEmail`, celui-ci correspond au `req.body.email` de la méthode `createUser` détaillée plus haut dans le contrôleur.

Afin d'éviter les injections SQL de code malveillant on utilise une requête préparée combinée à l'utilisation d'un jeton dont la valeur est externalisée. C'est ici dans **values** que la valeur du paramètre `reqEmail` est passée en argument sous forme de tableau. Dans le **texte** de la requête je demande de sélectionner (SELECT) toutes les données (*) depuis (FROM) la table user de la BDD où (WHERE) la colonne email a pour valeur le jeton (\$1).

Dans une autre constante je lance la requête au moyen de `client.query` à laquelle on passe en argument la constante `preparedQuery`.

Si il n'y a pas de résultat alors la méthode renvoie null, dans le cas contraire elle retourne la ligne demandée avec `result.rows[0]`.

```
//Insérer un nouvel utilisateur dans la BDD.
async insert(newUser) {
    //Je prépare une requête sql séparément pour éviter les injections.
    //J'utilise les jetons sql également par souci de sécurité.
    const preparedQuery = {
        text: `
            INSERT INTO public.user
            (firstname, lastname, nickname, email, password) VALUES
            ($1, $2, $3, $4, $5) RETURNING *
        `,
        values: [newUser.firstname, newUser.lastname, newUser.nickname, newUser.email,
        newUser.password],
    };

    const result = await client.query(preparedQuery);

    return result.rows[0];
},
};
```

valeurs des jetons `firstname`, `lastname` etc...

Le **texte** de la **requête** utilise des **commandes SQL différentes**. On demande à insérer (**INSERT**) dans (**INTO**) la table **user** on précise les **colonnes** dans lesquelles on souhaite insérer les **valeurs** (`firstname`, `lastname`...) avec pour valeur les **jetons**, une fois fait on demande à **récupérer le tout**.

Attention l'ordre dans lequel les **colonnes** et les **valeurs** de **jetons** sont positionnés dans les **parenthèses** du texte et le **tableau** de **valeurs** est très **important** pour éviter une **tentative d'insertion d'email** dans un **nom** qui **risquerait** de **causer une erreur** lors de l'opération.

Toujours à l'aide de `client.query` et l'argument `preparedQuery` on lance la requête et on retourne la ligne générée par le **RETURNING ***.

Pour la méthode **insert** également utilisée dans la méthode `createUser` du contrôleur, on garde la même bonne pratique de la requête préparée et des Jeton SQL.

Ici on combine plusieurs jetons séparés par des virgules car on récupère en paramètre `newUser` qui permettra de composer le tableau de

B. Test et jeu d'essai

• Test de routes back-end avec postman

Avant de pusher le code de la fonctionnalité développé dans la branche isolée au sein de la branche dev de SoNow, il est nécessaire de tester la route et l'obtention d'une réponse au format Json pour permettre son exploitation côté front-end.

J'ai utilisé le logiciel **postman** qui permet d'émettre des appels à l'API et ainsi comparer la réponse obtenue avec celle attendue.

The screenshot shows the Postman application interface. On the left, there's a sidebar with sections for Collections, APIs, Environments, Mock Servers, Monitors, Flows, and History. The main area displays a collection named 'SONOW' with several sections: '1.Events', '2.Users', '3.Tags', and '4.TEST-LOCAL'. Under '1.Events', there are several GET requests for events, such as 'Get All Events', 'Get Events By TagId', 'Get One Event By Id', 'Get One Event By Slug', and 'Get Events By Title'. A specific request for 'Get All Events' is selected, showing its details. The 'Body' tab of the request details shows a JSON response with a single event object. The response body is as follows:

```
1  {
2   "id": 2,
3   "title": "Angele",
4   "metadescription": "Lorem ipsum dolor sit amet, consectetur adipiscing elit",
5   "description": "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque tempor nisi ligula, a varius elit viverra ac. Aenean et libero eget quam euismod fringilla quis eu ipsum. In sagittis orci id euismod ornare. Ut ac erat nec metus imperdiet porta. Pellentesque vitae aliquet purus, ac sagittis nisl. Fusce elementum lobortis odio in scelerisque. Mauris massa leo, auctor non sem sed, tempus pharetra massa. In urna nulla, viverra quis sollicitudin non, bibendum non orci.",
6   "start": "2022-12-03T19:00:00+00:00",
7   "stop": "2022-12-03T23:00:00+00:00",
8   "location": "La Défense Arena",
9   "address": "8 Rue des Sorins",
10  "zipcode": "92000",
11  "city": "Nanterre",
12  "media": "https://res.cloudinary.com/sonow/image/upload/v1659513253/ANGELE_j2o1qp.jpg",
13  "price_ttc": null,
14  "url": null,
15  "latitude": 48.8964484,
16  "longitude": 2.338756,
17  "slug": "angele",
18  "phone_number": null,
19  "tag": [
20    {
21      "id": 1,
22      "name": "concert"
23    }
  ]
```

Ci-dessus vous pouvez avoir un aperçu de l'interface de postman. L'avantage de cet outil est que l'on peut créer un projet d'équipe, chaque membre peut alors créer un test et le lancer mais tous les autres équipiers y ont également accès.

Après initialisation du projet, des sections ont été implémentées. L'équipe a choisi une organisation en fonction des routeurs pour une meilleure lisibilité, il y a donc Root (pour les points d'entrées), Events, Users et Tags.

Dans chaque section sont détaillées les routes à tester portant le nom des endpoints. Elles sont également identifiées par un logo suivant la méthode HTTP paramétrée dans les détails.

Le processus pour valider un test est le suivant :

- Entrer les paramètres tests de la requête qui sera envoyée par le client
- Lancer l'appel à l'API
- Obtention d'une réponse
- Comparaison des données obtenues avec celles attendues
- Validation du test de la route

Exemple d'un test sur la route Login du routeur User :

L'objectif de ce test est de lancer un appel à l'API simulant l'envoie du formulaire de connexion d'un utilisateur. La réponse obtenue doit être un tableau d'objet JSON du jeton JWT.

2.Users / Login

POST http://sonow.herokuapp.com/api/user/login

Params Authorization Headers (11) **Body** Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL

KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/> email	julie.durand@fakemail.fr			
<input checked="" type="checkbox"/> password	Durand13*			

Body Cookies (1) Headers (12) Test Results

Pretty Raw Preview Visualize JSON

1 "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cIiKpXVCJ9.",
2 "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cIiKpXVCJ9.",
3 "user": {
4 "id": 15,
5 "firstname": "Julie",
6 "lastname": "Durand",
7 "nickname": "JulieDurand",
8 "gender": null,
9 "email": "julie.durand@fakemail.fr",
10 "biography": null,
11 "birthday": null,
12 "profile_picture": "https://res.cloudinary.com/sonow/image/upload/v1659539845/default_profile_user_ldkwmd.png",
13 "language": "FR",
14 "darkmode": true,
15 "phone_number": null,
16 "address": null,
17 "zipcode": null,
18 "city": null,
19 "latitude": null,
20 "longitude": null,
21 "created_at": "2022-09-20T18:24:55.679Z",
22 "update_at": null
23 }
24 }

Je commence par définir la méthode HTTP utilisée. Dans le routeur et le dictionnaire des routes il s'agit de **POST** car nous envoyons de la données à partir d'un formulaire.

J'entre ensuite l'URL d'appel à l'API et je paramètre le body. Le format du body est **x-www-form-urlencoded** (ceci a été défini dans les paramètres du serveur de SoNow avec la commande `app.use(express.urlencoded({ extended: true }));`)

Je crée le formulaire avec le nom des champs (**email** et **password**) et les valeurs correspondantes.

L'utilisatrice que je tente de connecter est Julie Durand, son email est julie.durand@fakemail.fr son mot de passe Durand13* (données et identité inventées de toutes pièces)

Après envoi de la requête au moyen de la commande SEND, j'obtiens bien un jeton JWT dans lequel les données correspondent à l'utilisatrice Julie DURAND ainsi que l'access Token et le refreshToken.

Je valide la route "Login", le test est réussi.

VIII. VEILLE ET TROUBLESHOOTING

A. Veille technologique mise en place

- Implémentation d'un middleware de validation de l'email et mot de passe de l'utilisateur

Comme je l'ai exposé précédemment, dans le but d'ajouter une couche de sécurité au niveau de l'API quand l'utilisateur remplit le formulaire d'inscription, j'ai développé un service de validation à l'aide de la librairie Joi composé d'un validateur et de schémas, un pour le formulaire de création, le second pour celui de mise à jour du profil (quand la fonctionnalité sera disponible).

Celle-ci est installable via NPM. Afin de réaliser le code de façon plus aisée j'ai en amont effectué une veille dans la documentation mise à disposition à l'adresse <https://joi.dev/api/?v=17.6.1>. J'avais eu bien évidemment une démonstration en cours lors de la spécialisation Data mais n'avais pu m'exercer suffisamment lors du challenge quotidien.

Mon objectif étant d'avoir un objet remplissant les contraintes imposées par la logique de validation pouvant être comparé aux données entrées.

La méthode .object() permet de réaliser cela : 

Generates a schema object that matches an object data type.
Defaults to allowing any child key.

Supports the same methods of the any() type.

```
const object = Joi.object({
  a: Joi.number().min(1).max(10).integer(),
  b: 'some string'
});

await object.validateAsync({ a: 5 });
```

Je souhaitais également rendre les champs requis du formulaire d'inscription obligatoires. Joi avec sa méthode .required() le permet :

any.required() - aliases: exist 

Marks a key as required which will not allow undefined as value.

All keys are optional by default.

```
const schema = Joi.any().required();
```

Possible validation errors: any.required

Pour que **chaque propriété de l'objet** soit **composé de chaîne de caractère** j'utilise la méthode `.string()` :

string

Generates a schema object that matches a string data type.

Note that the empty string is not allowed by default and must be enabled with `allow('')`. Don't over think, just remember that the empty string is not a valid string by default. Also, don't ask to change it or argue why it doesn't make sense. This topic is closed.

To specify a default value in case of the empty string use:

```
Joi.string()  
.empty('')  
.default('default value');
```

If the `convert` preference is `true` (the default value), a string will be converted using the specified modifiers for `string.lowercase()`, `string.uppercase()`, `string.trim()`, and each replacement specified with `string.replace()`.

Supports the same methods of the `any()` type.

```
const schema = Joi.string().min(1).max(10);  
await schema.validateAsync('12345');
```

Possible validation errors: `string.base`, `string.empty`

Enfin `.regex()` permet de **contraindre l'objet à respecter le schéma d'expression régulière choisi** (détallé plus haut dans le chapitre sécurité) :

object.regex()

Requires the object to be a `RegExp` object.

```
const schema = Joi.object().regex();
```

Possible validation errors: `object.regex`

Concernant la **force de sécurité du schéma de mot de passe choisi** j'ai consulté **plusieurs sites** après recherche avec la requête suivante : "recommandation en matière de choix de mot de passe".

Mon **attention** s'est arrêtée sur l'article du "**blog du modérateur**"

<https://www.blogdumoderateur.com/comment-mettre-en-place-politique-mot-de-passe-efficace/>

Celui-ci **définit ce qu'est une politique de mot de passe**, à savoir, "des mesures mises en place pour renforcer la sécurité d'accès aux données et aux outils à travers la création et l'utilisation de mot de passe complexes".

L'auteur rappelle également le **cadre de l'application des articles concernés de la RGPD** (5 & 32) : "organiser et cadrer l'authentification des utilisateurs qui est un facteur essentiel pour garantir la sécurité des traitements des données personnelles".

Enfin, il **énumère les éléments indispensables** à une **bonne politique de mot de passe** : longueur, complexité, délais d'expiration, mécanisme de contrôle de robustesse, limitation d'essai, solution de secours en cas de perte ou de vol.

Combiné aux **conseils** prodigué par la **cnil** , j'ai **proposé** à l'équipe la **règle suivante** pour la **regex** liée au **mot de passe de l'utilisateur** :

- *Entre 8 et 15 caractères, une majuscule et une minuscule obligatoire, un chiffre et un caractère spécial obligatoire*

B. Résolution des problèmes rencontrés

- **Problème de permission refusée pour le lancement de fichier .sh dans Linux**

Comme je vous l'ai **expliqué** dans le **chapitre consacré au versionning**, l'utilisation de **sqitch nécessite le lancement de scripts** dans des **fichiers shell** à l'extension de fichier **.sh**. Durant l'apothéose j'ai **rencontré** un **message d'erreur** au **lancement du fichier init.sh**. Travaillant sous un **environnement Linux**, je rentrais la **ligne de commande** demandée dans le terminal, à savoir : **./init.sh**

Le **message renvoyé** par le **terminal** était le **suivant** :

bash: ./init.sh: Permission denied

Après une **recherche sur google** au moyen de l'expression "**permission denied .sh**", j'ai finalement trouvé une **solution** sur le **premier résultat** fourni par le **forum de discussion**.

<https://askubuntu.com/questions/409025/permission-denied-when-running-sh-scripts>

Il faut au **préalable accorder** les **droits** pour **autoriser** l'**exécution** du **fichier avec** la **commande** : **chmod +x nom_du_fichier** puis **relancer ./nom_du_fichier** .

Je me suis **aperçu** que **plusieurs variantes** de la **commande étaient mentionnées**, j'ai ainsi pris un peu de temps et lu quelques autres sources dont une bien détaillée en **anglais** que je vais vous traduire dans la **section suivante**. Il s'agit du site **shells.com** qui propose une **solution** de **cloud computing**.

C. Traduction d'une ressource anglophone

Le site shells.com a attiré mon attention comme indiqué dans la section précédente.

Voici le lien que j'ai consulté :

<https://www.shells.com/l/fr-FR/tutorial/How-to-Fix-Shell-Script-Permission-Denied-Error-in-Linux>

En voici mon **interprétation** :

"Comment réparer les erreurs de permissions refusées au script shell dans linux.

L'article explique que lorsqu'on cherche à lancer un fichier bash dans un terminal linux on peut rencontrer des erreurs de type "bash: ./program_name: permission denied" couramment appelée erreur de permission refusée.

Ce tutoriel explique ce type d'erreur, les raisons qui font qu'elles se produisent et comment les régler.

Ce type d'erreur se produit quand un script shell n'a pas la permission de s'exécuter. Linux renvoie le problème en affichant le terminal le message énoncé plus haut car les systèmes d'exploitations se soucient de leur sécurité. Seuls les utilisateurs avec le statut "Sudo" (super user do) ont un accès total à tous les fichiers et répertoires pour faire les changements requis. C'est la raison pour laquelle les scripts s'interrompent pour les utilisateurs "normaux". Pour résoudre cette erreur dans linux, il est obligatoire de changer l'autorisation du fichier avec la commande "chmod" pour "Change Mode". On peut vérifier la permission du fichier au préalable avec la commande `lls -l nom_du_fichier.sh`

La commande `chmod` doit être suivie du mode de permission ajouté au fichier à savoir : `x` pour exécuter, `w` pour écrire, `r` pour lire.

On peut ainsi octroyer au fichier `nom_du_fichier.sh` l'apossibilité d'être exécuté avec la commande :
`sudo chmod +x nom_du_fichier.sh`

L'article parle également des utilisateurs ayant le droit de lancer la commande :

-`g` pour ceux qui font partis du groupe du fichier
-`u` pour les propriétaires du fichier
-`a` pour tous les utilisateur
-`o` pour les autres utilisateurs hormis le propriétaire et membre du groupe du fichier.

La commande "cat" permet de voir le contenu du script.

Enfin on peut lancer la commande de lancement du fichier dans le terminal qui s'opère correctement."

IX. CONCLUSION

A. Difficultés rencontrées, accomplissements durant le projet, pistes d'amélioration

Ce mois de projet d'apothéose m'a permis de faire la **synthèse** de toutes les heures **d'apprentissage théorique** et **mise en pratique** lors des challenges et parcours de fin de saison. J'ai consolidé mes **connaissances** et aussi **découvert** des **technologies** qui m'étaient **inconnues** auparavant, mais j'ai également fait la **connaissance** de **personnes** nouvelles. En effet, la **mise en situation** du projet d'apothéose est l'occasion d'être en **conditions quasi-professionnelles**. Ce fut le **cas** pour ma part avec **SoNow** car la demande provenait d'un **project owner externe** à l'école et je ne **connaissais** que très peu les autres personnes de l'équipe de **développement**. Pour ma part, l'**essai** est transformé en **réussite** sur **tous les tableaux**.

Bien évidemment il y a eu des **moments de doutes** lors de **difficultés rencontrées** quand un **message d'erreur** dans le **terminal** s'affiche vous annonçant que le code de la fonctionnalité fraîchement développée ne peut aboutir et renvoyer le fichier json tant attendu par le front-end, mais c'est aussi l'occasion de prendre du **recul** sur la **situation**, faire le **point** sur la **logique élaborée**, chercher dans les **documentations**, échanger avec le reste de l'équipe sur le **problème rencontré** et avancer jusqu'à trouver la **solution**.

L'application, nous en sommes tous **fiers**, et nous **savons** également qu'elle est **perfectible** avec par **exemple** au **niveau** de la **sécurité** et plus particulièrement **l'inscription/authentification** de la mise en place d'un **envoi par mail** d'un **lien validant** l'opération ou encore la **gestion** du **cas** d'un **mot de passe oublié**. L'implémentation d'un **rôle administrateur** de l'application lui permettant de gérer tous les **événements** et

utilisateurs qui ne respectent pas la politique de bonne conduite demandée est également une piste évoquée.

Sur un **plan plus technique**, nous **pourrions** aussi éviter les failles de type **XSS** avec un **middleware** dédié et la **librairie** **npm sanitize** qui **nettoie** les **formulaires envoyés** par l'utilisateur si celui - ci tente d'entrer du code malveillant. **Ceci** avait été **envisagé** lors du **sprint 0** mais **hélas** par **manque de temps** nous n'avons **pu réaliser** la **totalité** de nos **prévisions**. En effet **certaines fonctionnalités** de **l'API** sont **opérationnelles** mais le **raccordement à l'interface** n'a pas **pu être finalisé**. Pour autant, je considère **SoNow** comme le premier vrai projet auquel j'ai pu participer.

B. Projets personnels à courts, moyens, longs termes

Sur le plan de mes **projets personnels** après cette soutenance et plus généralement après ces 6 derniers mois, je **compte** à court terme démarrer **l'apprentissage** de **REACT** qui a sincèrement **éveillé ma curiosité** lors du développement de SoNow et de la préparation de ce dossier. **Ayant fait une spécialité data**, je n'ai pu **qu'avoir un aperçu des possibilités offertes** par cette librairie et les **explications** qui m'ont été **apportées** par mes **coéquipiers front-end**.

Bien évidemment cette **reconversion** a été réalisée avec pour **objectif de trouver** un **emploi** dans ce secteur d'activité. **Start up, ESN, agence de communication**, mon **choix** n'est pas encore **fixé** mais j'espère intégrer une **équipe** très prochainement et pouvoir **exploiter** tout ce que j'ai déjà **découvert** ainsi que me **former** à de **nouveaux outils** car le monde de l'informatique étant en perpétuelle évolution, les **technologies** de **demain peuvent très rapidement devenir celles d'aujourd'hui**. **Rencontrer** avec ce nouveau départ professionnel un **développeur expérimenté** qui souhaite m'aider à **monter en compétences** grâce à son **expérience** et devenir mon **mentor** serait un **privilège**.

C. Remerciements

Je tiens à **remercier** pour commencer l'ensemble de **l'équipe pédagogique** de l'école **O'clock** qui m'a accompagné durant toutes les étapes de ma formation mais également après par leurs encouragements avec les nombreux moyens de communications dont nous disposons aujourd'hui, et plus **particulièrement** **Jordan BRULL** et **Virginie LEMAIRE** pour leur **accompagnement quotidien** en tant que **tuteurs**, **Benjamin NOUGADERE**, **Laurent NEVEUX**, **Enzo TESTA**, **Fabien TAVERNIER** pour leur **enseignement**, leur **patience**, leurs **conseils** et la **sympathie** manifestée encore aujourd'hui.

Merci également à tous les **apprenants** de la promotion **CASSINI** qui ont chacun **apporté** au **quotidien** de **l'énergie**, de **l'entraide**, de **la détermination** quand c'était nécessaire. Mention spéciale à mes coéquipiers de SoNow sans qui ce projet n'existerait pas et avec qui j'ai eu grand plaisir à travailler.

Je voudrais également remercier les **personnes** qui, **avant même le démarrage** de cette **reconversion professionnelle**, **m'accompagnaient** dans la **vraie vie** en tant que **proches** et durant ont **su être présents** tout en **restant discrets** pour me permettre de **réaliser** cette **transformation** et **apprendre** les **nombreux concepts** de ce **métier**.

Enfin pour terminer, je **vous remercie** vous **jury** pour la **lecture** de ce **dossier** et le **temps** que **vous** m'avez **accordé** à **m'écouter** durant cette **présentation**.

C'est **grâce à vous tous** que je peux enfin aujourd'hui me **sentir prêt à vivre** ce rêve **d'enfant** et exerce le **métier de développeur web** pour lequel je me sens **passionné**.

X. ANNEXES

Wireframes Desktop Connexion/Inscription :

Connexion / Desktop

Logo SoNow

Vidéo muette de présentation

Texte d'accueil utilisateur

Champ identifiant (email) pour permettre la saisie de l'utilisateur

Champ formulaire (mot de passe) pour permettre la saisie de l'utilisateur

Bouton de connexion au compte

Bienvenu !
Inscrис toi et découvre tous les événements qui pourraient t'intéresser

Identifiant : Entrer votre email

Mot de passe : Entrer votre mot de passe

Connexion

Ou

Créer un compte

A propos

Inscription / Desktop

Logo SoNow

Vidéo muette de présentation

Formulaire entrée "Nom d'utilisateur"

Formulaire entrée "Prénom"

Formulaire entrée "Pseudo"

Formulaire entrée "Email"

Formulaire entrée "Mot de passe"

Formulaire "Confirmation Mdp"

Création du compte utilisateur

Nom : Entrer votre nom

Prénom : Entrer votre prénom

Pseudo : Entrer votre Pseudo

Email : Entrer votre email

Mot de passe : Entrer votre mot de passe

Confirmation mot de passe : confirmez votre mot de passe

Annuler

Confirmer

Routeur User destiné à la gestion des endpoints liés à l'utilisateur :

```
const express = require('express');

const validate = require('../validation/validator');
const createSchema = require('../validation/schemas/userCreateSchema');
const updateSchema = require('../validation/schemas/userUpdateSchema');

const router = express.Router();

//Importation du controller des utilisateurs.
const { userController: controller } = require('../controllers');

const controllerHandler = require('../services/controllerHandler');

//Les différentes routes de l'API pour l'utilisateur.

//Route pour récupérer tous les utilisateurs.
router
  .route('/')
  .get(controllerHandler(controller.getAllUsers));

//Routes pour créer ou (dé)connecter l'utilisateur.
router
  .route('/login')
  .post(controllerHandler(controller.loginUser));

router
  .route('/signup')
  .post(validate('body', createSchema), controllerHandler(controller.createUser));

router
  .route('/logout')
  .get(controllerHandler(controller.logoutUser));

//Routes pour qu'un utilisateur recherche un autre utilisateur par son surnom.
router
  .route('/search')
  .post(controllerHandler(controller.getOneUserByNickname));

//Routes pour récupérer, modifier, supprimer, s'abonner et se désabonner d'un utilisateur.
router
  .route('/follow')
  .post(controllerHandler(controller.followUser))
  .delete(controllerHandler(controller.unfollowUser));

//Routes pour récupérer, modifier, supprimer un utilisateur par son ID.
router
  .route('/:user_id')
  .get(controllerHandler(controller.getOneUserById))
  .patch(validate('body', updateSchema), controllerHandler(controller.updateUser))
  .delete(controllerHandler(controller.deleteUser));

//Route pour récupérer la liste des abonnés d'un utilisateur par son ID.
router
  .route('/:user_id/followers')
  .get(controllerHandler(controller.getFollowers))

//Route pour récupérer la liste des abonnements d'un utilisateurs à d'autres comptes par son ID.
router
  .route('/:user_id/followed')
  .get(controllerHandler(controller.getFollowed));

module.exports = router;
```

Contrôleur User destiné à la circulation de la donnée pour l'utilisateur :

```
...
require('dotenv').config();
const userDataMapper = require('../models/user');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const SECRET_KEY = process.env.ACCESS_SECRET_KEY;
const REFRESH_SECRET_KEY = process.env.REFRESH_SECRET_KEY;

const { ApiError } = require("../services/errorHandler");

module.exports = {

    //Méthode qui permet à l'utilisateur de se connecter.
    async loginUser(req, res) {
        const { email, password } = req.body;

        let user = await userDataMapper.findByEmailForLogin(email);

        if (user) {
            bcrypt.compare(password, user.password, function (err, result){
                if (result) {
                    req.session.user = user

                    const expiresIn = 24 * 60 * 60;
                    const accessToken = jwt.sign({ user },SECRET_KEY,{ expiresIn: expiresIn });
                    const refreshToken = jwt.sign({ user },REFRESH_SECRET_KEY,{ expiresIn: expiresIn });

                    res.header('Authorization', 'Bearer ' + accessToken);
                    res.header('RefreshToken', 'Bearer ' + refreshToken);

                    delete req.session.user.password;
                    return res.status(200).json({accessToken, refreshToken ,user: req.session.user});

                } else {
                    return res.status(403).json({status : 'error', statusCode :403, message:'Wrong email or password'});
                }
            });
        }
    },

    //Méthode qui permet à l'utilisateur de se déconnecter de sa session.
    async logoutUser (req, res) {
        //On détruit la session utilisateur.
        req.session.destroy();

        return res.status(200).json('Successful disconnected');
    },
...
}
```

```

...
//Méthode qui permet de s'abonner à l'activité d'un ami.
async followUser (req, res) {
    const follow = await userDataMapper.pinFollowUser(req.body.user_follower,
req.body.user_followed);

    if(!follow) {
        throw new ApiError('User already followed', {statusCode: 400 });
    } else {

        return res.status(200).json({follow, "message": "User followed succesfully"});
    }
},

//Méthode qui permet de se désabonner de l'activité d'un ami.
async unfollowUser (req, res) {
    const unfollow = await userDataMapper.unpinFollowUser(req.body.user_follower,
req.body.user_followed);

    if(!unfollow) {
        throw new ApiError('Users not found', {statusCode: 404 });
    } else{

        return res.status(200).json({unfollow, "message": "User unfollowed succesfully"});
    }
},

//Méthode qui permet de récupérer tous les utilisateurs.
async getAllUsers(_, res) {
    const userDb = await userDataMapper.findAll();

    if(!userDb) {
        throw new ApiError('No any user in database', { statusCode: 404 });
    };

    return res.json(userDb);
},

//Méthode qui permet de récupérer un utilisateur par son Id.
async getOneUserById(req, res) {

    const userDb = await userDataMapper.findByPk(req.params.user_id);

    if(!userDb) {
        throw new ApiError('User not found', { statusCode: 404 });
    }

    return res.json(userDb);
},

//Méthode qui permet de rechercher les utilisateurs par leur surnom, leur nom ou prénom.
async getOneUserByNickname(req, res) {
    const userDb = await userDataMapper.findByNickname(req.body.nickname);
    if(!userDb) {
        throw new ApiError('User not found', { statusCode: 404 });
    };

    return res.json(userDb);
},
...

```

```

...
//Méthode qui permet à l'utilisateur de se créer un compte sur l'application.
async createUser(req, res) {
    const userDb = await userDataMapper.findByEmail(req.body.email);

    if (userDb) {
        throw new ApiError('User already exists', { statusCode: 400 });
    } else {
        //On crypte le mot de passe.
        const encryptedPwd = await bcrypt.hash(req.body.password, 10);
        const newUser = {
            firstname: req.body.firstname,
            lastname: req.body.lastname,
            nickname: req.body.nickname,
            email: req.body.email,
            password: encryptedPwd
        };
        const insertUser = await userDataMapper.insert(newUser);

        res.json(insertUser);
    };
}

//Méthode qui permet à l'utilisateur de mettre à jour son profil.
async updateUser(req, res) {

    const userDb = await userDataMapper.findByPk(req.params.user_id);

    if (!userDb) {
        throw new ApiError('User not found', { statusCode: 404 });
    };

    const savedUser = await userDataMapper.update(req.params.user_id, req.body);

    return res.json(savedUser);
};

//Méthode qui permet à l'utilisateur de supprimer son compte.
async deleteUser(req, res) {
    const userDb = await userDataMapper.findByPk(req.params.user_id);

    if (!userDb) {
        throw new ApiError('User not found', { statusCode: 404 });
    };

    await userDataMapper.delete(req.params.user_id);

    return res.status(200).json({code: 200, message: "User has been deleted"});
};

//Méthode qui permet de récupérer la liste des abonnés d'un utilisateur.
async getFollowers(req, res) {
    const result = await userDataMapper.findFollowersByUserId(req.params.user_id);

    if(!result) {
        throw new ApiError('User not found', { statusCode: 404 });
    };

    return res.json(result);
};

//Méthode qui permet de récupérer la liste des abonnements d'un utilisateur.
async getFollowed(req, res) {
    const result = await userDataMapper.findFollowedByUserId(req.params.user_id);

    if(!result) {
        throw new ApiError('User not found', { statusCode: 404 });
    };

    return res.json(result);
},
}

```

Modèle User destiné à émettre des requêtes SQL à la base de données :

```
● ● ●

const client = require("../config/db");

module.exports = {

    //Rechercher tous les utilisateurs de la BDD.
    async findAll() {
        //Je prépare une requête sql séparément pour éviter les injections.
        //J'utilise les jetons sql également par souci de sécurité.
        const preparedQuery = {
            text: `
                SELECT *
                FROM public.user
            `,
            values: []
        };

        const result = await client.query(preparedQuery);

        if (result.rowCount === 0) {
            return null;
        };

        return result.rows;
    },

    //Recherche un utilisateur en fonction de sa clé primaire ID.
    async findByPk(userId) {
        //Je prépare une requête sql séparément pour éviter les injections.
        //J'utilise les jetons sql également par souci de sécurité.
        const preparedQuery = {
            text: `
                SELECT *
                FROM public.user
                WHERE id = $1
            `,
            values: [userId],
        };
        const result = await client.query(preparedQuery);

        if (result.rowCount === 0) {
            return null;
        };

        return result.rows[0];
    },

    //Retrouver un user par son email pour l'authentification
    async findByEmail(reqEmail){
        //Je prépare une requête sql séparément pour éviter les injections.
        //J'utilise les jetons sql également par souci de sécurité.
        const preparedQuery = {
            text: `
                SELECT *
                FROM public.user
                WHERE email = $1
            `,
            values: [reqEmail],
        };

        const result = await client.query(preparedQuery);

        if (result.rowCount === 0) {
            return null;
        };

        return result.rows[0];
    }
};
```

```

● ● ●

async findByEmailForLogin(reqEmail){
    //Je prépare une requête sql séparément pour éviter les injections.
    //J'utilise les jetons sql également par souci de sécurité.
    const preparedQuery = {
        text: `
            SELECT *
            FROM public.user
            WHERE email = $1
        `,
        values: [reqEmail],
    };

    const result = await client.query(preparedQuery);

    if (result.rowCount === 0) {
        return null;
    }

    return result.rows[0];
},

//Rechercher un utilisateur par son surnom, son nom ou son prénom.
async findByNickname(nickname){
    //Je prépare une requête sql séparément pour éviter les injections.
    //J'utilise les jetons sql également par souci de sécurité.
    const preparedQuery = {
        text: `
            SELECT *
            FROM public.user
            WHERE nickname ILIKE $1
        `,
        values: [`%${nickname}%`],
    };

    const result = await client.query(preparedQuery);

    if (result.rowCount === 0) {
        return null;
    }

    return result.rows;
},

//Insérer un nouvel utilisateur dans la BDD.
async insert(newUser) {
    //Je prépare une requête sql séparément pour éviter les injections.
    //J'utilise les jetons sql également par souci de sécurité.
    const preparedQuery = {
        text: `
            INSERT INTO public.user
            (firstname, lastname, nickname, email, password) VALUES
            ($1, $2, $3, $4, $5) RETURNING *
        `,
        values: [newUser.firstname, newUser.lastname, newUser.nickname, newUser.email,
newUser.password],
    };

    const result = await client.query(preparedQuery);

    return result.rows[0];
},

```

```

//Mettre à jours les infos d'un utilisateur en BDD.
async update(id, user) {
    const fields = Object.keys(user).map((prop, index) => `${prop} = ${index + 1}`);
    const values = Object.values(user);
    const savedUser = await client.query(`

        UPDATE public.user SET
            ${fields}
        WHERE id = ${fields.length + 1}
        RETURNING *

    `,
    [...values, id],
);
    return savedUser.rows[0];
},

//Supprimer un utilisateur de la BDD.
async delete(id) {
    const preparedQuery = {
        text: `
            DELETE
            FROM public.user
            WHERE id = $1
        `,
        values: [id],
    };
    const result = await client.query(preparedQuery);
    return !!result.rowCount;
},

async findFollowersByUserId(id) {
    const preparedQuery = {
        text: `
            SELECT u1.id as id_followed, u1.nickname as user_followed, u1.profile_picture as
            profile_picture_followed, u2.id as id_follower, u2.nickname as user_follower, u2.profile_picture as
            profile_picture_follower FROM user_follow_user
            JOIN public.user u1 ON user_follow_user.code_user = u1.id
            JOIN public.user u2 ON user_follow_user.code_user2 = u2.id
            WHERE code_user2 = $1
        `,
        values: [id],
    };
    const result = await client.query(preparedQuery);
    if (result.rowCount === 0) {
        return null;
    };
    return result.rows;
},

```

```

● ● ●

    async findFollowedByUserId(id) {
      const preparedQuery = {
        text: `

          SELECT u1.id as id_follower, u1.nickname as user_follower, u1.profile_picture as
          profile_picture_follower, u2.id as id_followed, u2.nickname as user_followed, u2.profile_picture as
          profile_picture_followed FROM user_follow_user
            JOIN public.user u1 ON user_follow_user.code_user = u1.id
            JOIN public.user u2 ON user_follow_user.code_user2 = u2.id
            WHERE code_user = $1
          `,
        values: [id],
      };

      const result = await client.query(preparedQuery);

      if (result.rowCount === 0) {
        return null;
      };

      return result.rows;
    },

    //Suivre un autre utilisateur.
    async pinFollowUser(userId1, userId2) {

      const pinFollowUser = await client.query(
        `

          INSERT INTO public.user_follow_user (code_user, code_user2) VALUES ($1, $2)
          RETURNING *
        `,
        [userId1, userId2],
      );
      return pinFollowUser.rows;
    },

    //Ne plus Suivre un autre utilisateur.
    async unpinFollowUser(userId1, userId2) {

      const unpinFollowUser = await client.query(
        `

          DELETE FROM public.user_follow_user
          WHERE code_user= $1 AND code_user2=$2
        `,
        [userId1, userId2],
      );

      return !!unpinFollowUser.rowCount;
    },
  };
}

```

Détail du code du composant createAccount en REACT :

```
import { Link } from 'react-router-dom';
import { useNavigate } from 'react-router-dom';
import { useDispatch, useSelector } from 'react-redux';
import { changeSignupInputs, submitSignup } from '../store/actions';
import { Form, Header } from 'semantic-ui-react';
import loop from '../images/assets/sonow-bis.mp4';

import "../styles/createAccount.scss";

function CreateAccount() {

  const {
    firstnameInput,
    lastnameInput,
    nicknameInput,
    emailInput,
    passwordInput,
    confirmedPasswordInput,
    isRegistered
  } = useSelector((state) => state.user.signup) || {};

  const dispatch = useDispatch();
  const navigate = useNavigate();

  const handleSubmit=(e)=>{
    dispatch(submitSignup());
  };

  const handleFirstnameChange =(e)=>{
    dispatch(changeSignupInputs('firstnameInput', e.target.value));
  };

  const handleLastnameChange =(e)=>{
    dispatch(changeSignupInputs('lastnameInput', e.target.value));
  };

  const handleNicknameChange =(e)=>{
    dispatch(changeSignupInputs('nicknameInput', e.target.value));
  };

  const handleEmailChange =(e)=>{
    dispatch(changeSignupInputs('emailInput', e.target.value));
  };

  const handlePasswordChange =(e)=>{
    dispatch(changeSignupInputs('passwordInput', e.target.value));
  };

  const handleConfirmedPasswordChange =(e)=>{
    dispatch(changeSignupInputs('confirmedPasswordInput', e.target.value));
  };
}
```

```
return (
  <div className="create-account">
    <div className='create-account__video'>
      <div className='create-account__video__overlay'></div>
      <video className='create-account__video__content' src={loop} autoPlay loop muted />
    </div>
    <div className="create-account__main">
      <div className="account-container">
        <div className="account-container__form">
          <Header inverted as='h1' textAlign='center' style={{marginTop: '15px'}}>
            Créer un compte
          </Header>
          <Form
            inverted
            size='large'
            // style={{
            //   margin: '1rem',
            //   padding: '1rem'
            // }}
            onSubmit={(e) => {handleSubmit(e)}}
          >
            <Form.Input className='create-account__form-input'
              fluid
              required
              name='firstname'
              label='Prénom'
              placeholder='Prénom'
              value={firstnameInput}
              onChange={(e) => {handleFirstnameChange(e)}}
            />
            <Form.Input className='create-account__form-input'
              fluid
              required
              name='lastname'
              label='Nom'
              placeholder='Nom'
              value={lastnameInput} onChange={(e) => {handleLastnameChange(e)}}
            />
            <Form.Input
              fluid
              name='email'
              label='Email'
              placeholder='Email'
              value={emailInput} required onChange={(e) => {handleEmailChange(e)}}
            />
            <Form.Input className='create-account__form-input'
              fluid
              required
              name='nickname'
              label='Pseudo'
              placeholder="Nom d'utilisateur"
              value={nicknameInput}
              onChange={(e) => {handleNicknameChange(e)}}
            />
            <Form.Input className='create-account__form-input'
              fluid
              required
              type='password'
              name='password'
              label='Mot de passe'
              placeholder='Mot de passe'
              value={passwordInput}
              onChange={(e) => {handlePasswordChange(e)}}
            />

```

```

<Form.Input className='create-account__form-input'
    fluid
    required
    type='password'
    name='confirmedPassword'
    label='Confirmation du mot de passe'
    placeholder='Confirmation du mot de passe'
    value={confirmedPasswordInput}
    onChange={(e) => {handleConfirmedPasswordChange(e)}}
/>
<Form.Checkbox className="create-account__form-checkbox"
    defaultChecked
    label="En vous inscrivant sur SoNow vous acceptez nos conditions
d'utilisation
        et notre politique de confidentialité"
/>
<Form.Button className="create-account__form-button"
    style={{
        backgroundColor: '#F30067',
        color: 'white'
    }}
>
    C'est parti !
</Form.Button>
{
    isRegistered && navigate('/')
}
</Form>
<div className="account-container__form_footer">
    <p>Vous avez déjà un compte ?</p>
        <Link className='account-container__form_link' to='/'>Connectez-
vous</Link>
    </div>
</div>
</div>
</div>
</div>
);
}

export default CreateAccount;

```