Plateforme participative de partage de recettes en rapport avec l'univers du cinéma et des séries.

Un projet créé dans le cadre de la présentation au :

Titre Professionnel Développeur Web et Web Mobile Accessible

Réalisée en collaboration avec :

Ngyuen Kim

Coelho Kévin

Ajdigue Youssef

Hannoteaux Anthony

Présenté par :

Hannoteaux Anthony

Sommaire

Introduction

Ayant toujours possédé un attrait prononcé pour le domaine technologique et informatique, l'opportunité de pouvoir me présenter au passage du titre professionnel développeur web et web mobile, c'est présenté comme une chance de pouvoir exercer une profession dans laquelle je pourrai faire valoir mes compétences et m'épanouir sur le plan professionnel.

Bénéficiant dernièrement d'une expérience de deux ans en tant que technicien support IT, j'ai pu mettre à profit mes qualifications et assouvir en partie ma demande de connaissances.
Malheureusement, je me suis retrouvé très vite limité dans mon périmètre d'action et surtout dans l'acquisition de nouvelles compétences.

D'un naturel curieux, j'aime comprendre et n'hésite pas à chercher comment résoudre des problématiques qui seraient amenées à se présenter.
Le domaine du développement informatique, étant en perpétuelle évolution, nécessite un apprentissage constant et continu qui, à mon sens, permettra de répondre à ma soif d'apprentissage.

C'est dans cette optique que je me suis orienté vers ce cursus et avec plaisir que j'ai pu rejoindre la promotion BEHEMOTH 2024-2025 du centre de formation O'Clock en distanciel, où j'ai pu apprendre les rudimentaires du métier de développeur web et web mobile accessible, les compétences attendues et avoir une première approche du travail en collaboratif.

Conscient des nombreuses perspectives que peuvent offrir le domaine, l'obtention du titre professionnel développeur web et web mobile est la première étape qui me permettra de réaliser mes objectifs.

Ces objectifs qui sont, dans un premier temps approfondir mes connaissances dans un contexte de travail réel, les consolider et pouvoir ensuite décider d'une future spécialisation dans un langage et/ou technologie en fonction de mes préférences à venir.

Compétences du référentiel couvertes par le projet

I. Développer la partie front-end d'une application web ou web mobile sécurisée

1. Organisation : Délimiter une vision commune du projet, recueil des besoins

Avant même de pouvoir parler de conception et de développement, notre équipe c'est décidé de savoir comment pouvoir partager et organiser ses idées.
Pour cela, nous avons utilisé des outils collaboratifs comme *Notion (https://www.notion.com/)* pour le partage de prises de notes, *GoogleDrive* permettant de pouvoir mettre en ligne nos fichiers de conception à venir, ou encore *Trello (https://trello.com/)*, outil de gestion de projet.

2. Maquetter des interfaces utilisateur web ou web mobile

Toujours dans l'optique de vouloir répondre au mieux à la demande de notre client, il fut important de réaliser en premier lieu nos documents de conception.
Pour rendre de façon compréhensible les attentes d'un utilisateur lors de l'utilisation de nos fonctionnalités, des *users stories* ont été mises en place.
Suite à cela, les *wireframes* et maquettes graphiques furent réalisées en version mobile et desktop, dans le but d'avoir pour l'un, une vue schématique de la composition de nos pages à venir et d'autre, d'avoir une représentation graphique de notre application.

3. Réaliser des interfaces utilisateur statiques web ou web mobile

Après avoir constitué nos premiers documents de conception et tout en prenant en compte le fait que la majeure partie des utilisateurs naviguent, aujourd'hui, sur le web depuis leur mobile, il était important de mettre en place une application web adaptative en *mobile-first*. Sans oublier de la rendre accessible au plus grand nombre, notamment aux personnes souffrant de handicaps.

4. Développer la partie dynamique des interfaces utilisateur web

Au vu des besoins utilisateur, il était primordial de mettre en place une plateforme à l'interface dynamique, nous avons donc fait le choix de nous orienter vers la bibliothèque *ReactJS* et sa gestion des composants.

II. Développer la partie back-end d'une application web ou web mobile sécurisée

1. Mettre en place une base de données relationnelle

Pour nous aider à concevoir notre base de données, nous nous sommes appuyés sur la méthode *Merise* de conception pour l'élaboration de nos documents. La création d'un Modèle Conceptuel de Données (*MCD)*, d'un Modèle Logique de Donnée *(MLD)*, d'un Modèle Physique de Donnée *(MPD)* ainsi qu'un dictionnaire de données, nous a aidés à mieux entrevoir la construction de notre base de données.

2. Développer des composants d'accès aux données SQL et NoSQL

Faisant suite à la conception de notre MPD, nous avons décidés d'utiliser le Relational DataBase Management System *(RDBMS) PostgreSQL* pour créer notre base de données.

3. Développer des composants métier coté serveur

Pour la gestion de nos routes back-end, nous nous sommes tournés vers le framework *Express*. La mise en place de contrôleurs, de middlewares et la création d'un jeton d'authentification, nous a permis d'organiser nos routes contrôlées. Tout en s'appuyant sur le design pattern *Active Record*, des modèles ont été prévu pour établir nos méthodes CRUD (*Create, Read, Update, Delete).*

Contexte de réalisation du projet

I. Résumé du projet : présentation de Ciné-Délices

1. Organisation de l'équipe : méthodes et outils utilisés

Avec l'évolution de la consommation des contenus audiovisuels, les plateformes de streaming s'imposent comme un acteur incontournable. De ce fait, le temps passé dessus est plus long que par le biais de diffusions traditionnelles qui offrent moins de choix et de flexibilité. Désormais, tout le confort de retrouver le cinéma chez soi donne la possibilité de profiter de plusieurs activités/loisirs en même temps, ou bien de les associer plus facilement. La cuisine fait partie de ces bienfaits qui occupent une place de plus en plus importante chez la population cinéphile. Alors pourquoi ne pas combiner ces deux passions ?

C'est la question que l'entreprise *O'Flix*, industriel du divertissement, c'est posé et c'est ici qu'intervient *Ciné-Délices*.
Amateurs de cuisine et de cinéma, curieux gourmands, fans de films et de séries, *Ciné-Délices* propose à ses utilisateurs de pouvoir partager des recettes de cuisine inspirées de leurs histoires préférées.

II. Environnement et équipe de conception

1. Cadre de développement du projet

Ciné-Délices est une application réalisée dans le cadre d'un projet de groupe en fin de cursus de formation en tant que développeur web et web mobile accessible. Le but étant de pouvoir livrer une application à la fois moderne, conforme aux réglementations sécuritaires et accessible, le tout en répondant aux besoins de notre client.
Notre groupe fut composé de :

- Nguyen Kim : Lead Developers Back-End
- Cohelho Kévin : Scrum Master
- Ajdigue Youssef : Product Owner
- Hannoteaux Anthony : Lead Developers Front-End

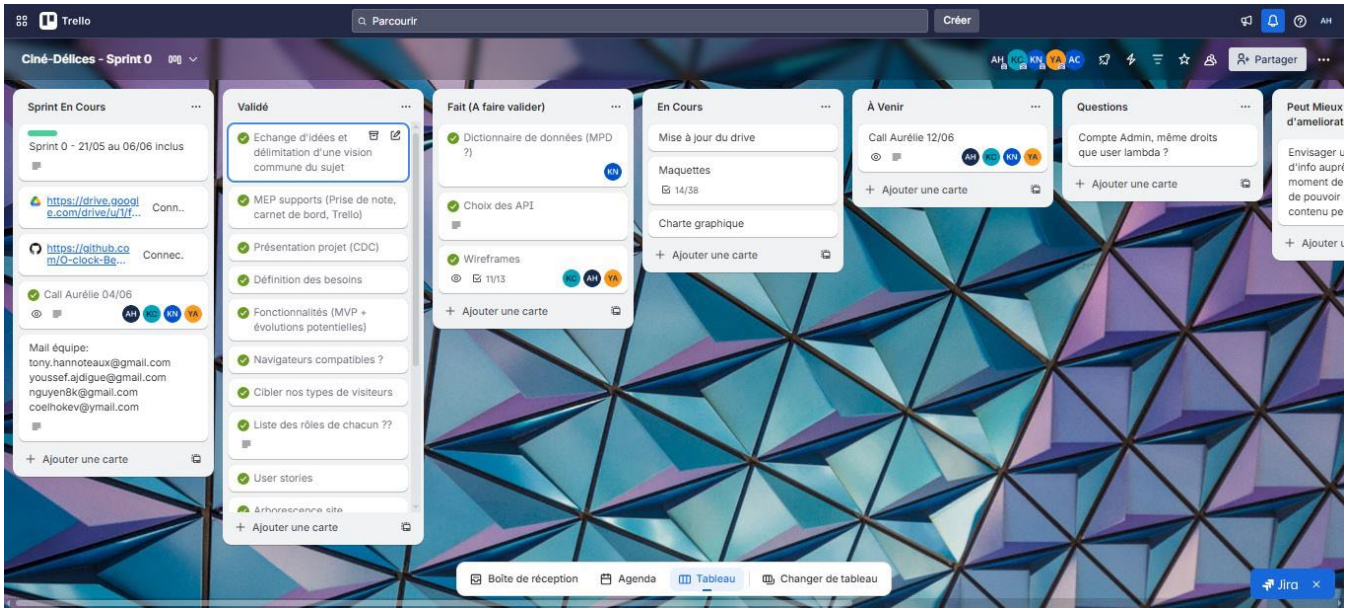
Ces rôles ayant avant tout pour but, de pouvoir nous permettre de trancher nos décisions en cas de différents concernant la direction du projet en fonction du secteur.

2. Organisation de l'équipe : méthodes et outils utilisés

Notre équipe s'est orientée vers une méthode de développement *Agile*, cette méthode de conception étant parfaitement adaptée à notre projet, nous permettant de pouvoir revenir sur des étapes déjà réalisées tout en leur offrant des perspectives d'évolution et ce même en cours de route.
La méthode *Kanban* fut appliquée via l'intermédiaire de l'utilisation de la plateforme du site *Trello*, méthode qui nous a permise de facilement mettre en place le déroulement de nos différents *Sprint*. En plus de nous permettre de déléguer et de partager les tâches de

façon aisée entre chaque membre de notre équipe, celle-ci nous a également accordé la possibilité de suivre en temps réel, l'avancée de tout à chacun, donnant lieu à une réorganisation simple et rapide de la répartition des équipes en fonction des besoins du moment.

- *Exemple : extrait du tableau kanban du sprint 0*



Cahier des charges

I. Besoins et objectifs Cibles

1. Besoins

Dans une ère où la consommation de contenu audiovisuel se fait de façon de plus en plus abondante, par le biais de plateformes facilitant l'accès en streaming à une grande diversité de contenu, les séries et le cinéma sont aujourd'hui des œuvres de divertissement qui font partie intégrante de notre quotidien. De plus, la population tendant vers une certaine indépendance souhaite au fur et à mesure apprendre à réaliser eux-mêmes, des plats simples, rapides et délicieux.
Nous pouvons constater, après quelques recherches sur le web, qu'il n'est pas si aisé de trouver des sites donnant accès à des recettes de cuisine inspirées de médias des plus consommés.

- Manque d'interaction entre cuisine et culture populaire :
  Les amateurs de cinéma et de cuisine ont rarement une plateforme combinant ces deux passions.
- Absence d'une communauté dédiée :
  Pas de lieu centralisé pour partager et découvrir des recettes inspirées du cinéma.
- Problèmes d'accessibilité :
  Les recettes sont souvent éparpillées sur divers sites sans filtrage avancé.

2. Objectifs

Ciné-Délices se veut donc être une plateforme participative de recettes de cuisine en rapport avec le cinéma et les séries télévisées. Pour répondre à cette demande, elle se doit avant tout d'être capable de proposer un catalogue varié de recettes mais également de films et séries, tout en permettant à ses utilisateurs de pouvoir soumettre de nouvelles recettes de façon sécurisée et individuelle.

- Créer une passerelle entre gastronomie et cinéma :
  Un site dédié regroupant recettes inspirées de films et séries avec anecdotes et références.
- Faciliter la recherche et la découverte :
  Mise en place d'un catalogue organisé avec filtres avancés (titre du film, catégorie culinaire, etc.).
- Favoriser l'interaction et la créativité :
  Les utilisateurs pourront soumettre leurs propres recettes et avoir des appréciations (note) dessus.
- Garantir une expérience utilisateur fluide et immersive :
  Interface intuitive, responsive, accessible et optimisée pour le référencement (SEO).
- Développement du site dans une démarche d'éco-conception :
  Réduction des requêtes serveur et utilisation de pratiques de développement éco-responsables (minification CSS/JS, suppression du code inutile).

II. Cibles

- Amateur de série/film
- Amateur de cuisine
- Entre 20 et 50 ans [Cœur de cible 25-35 ans] :

  - Autonomes et actifs : généralement en pleine vie active, souvent indépendants, ce qui favorise l'envie de cuisiner chez soi.

  - Curieux et connectés : utilisateurs réguliers d'internet et réseaux sociaux qui consomment beaucoup de contenu en streaming, friands de contenus originaux et concepts innovants (mélange de recettes et cinéma) à la recherche d'une nouvelle expérience.

  - Culture populaire : Ils ont grandi avec les films cultes et apprécient les références culturelles et culinaires.

III. Minimum Viable Product (MVP)

Notre MVP se basera donc sur la possibilité à nos utilisateurs de pouvoir consulter un catalogue de recette en rapport avec les séries ou le cinéma, tout en lui octroyant la capacité de pouvoir créer de nouvelle recette en rapport avec les œuvres disponibles et proposé par notre plateforme.
Il est donc naturel qu'il puisse créer un compte personnel et sécurisé lui permettant de s'authentifier pour qu'il puisse soumettre ses propres recettes à la communauté.

1. Front-Office (Visiteurs)

- Création de compte
- Consultation du catalogue de recettes disponibles
- Recherche par mot-clé
- Consultation des fiches recettes
- Consultation du catalogue de film & séries
- Consultation des fiches film/série

2. Front-Office (Adhérents)

- Connexion, Déconnexion
- Création, modification, suppression de recettes de l'adhérent
- Notation des recettes (note sur 5)
- Gestion du profil utilisateur

3. Back-Office (Administrateur)

- Modération des recettes (édition, suppression)
- Gestion des utilisateurs

IV. Users Stories

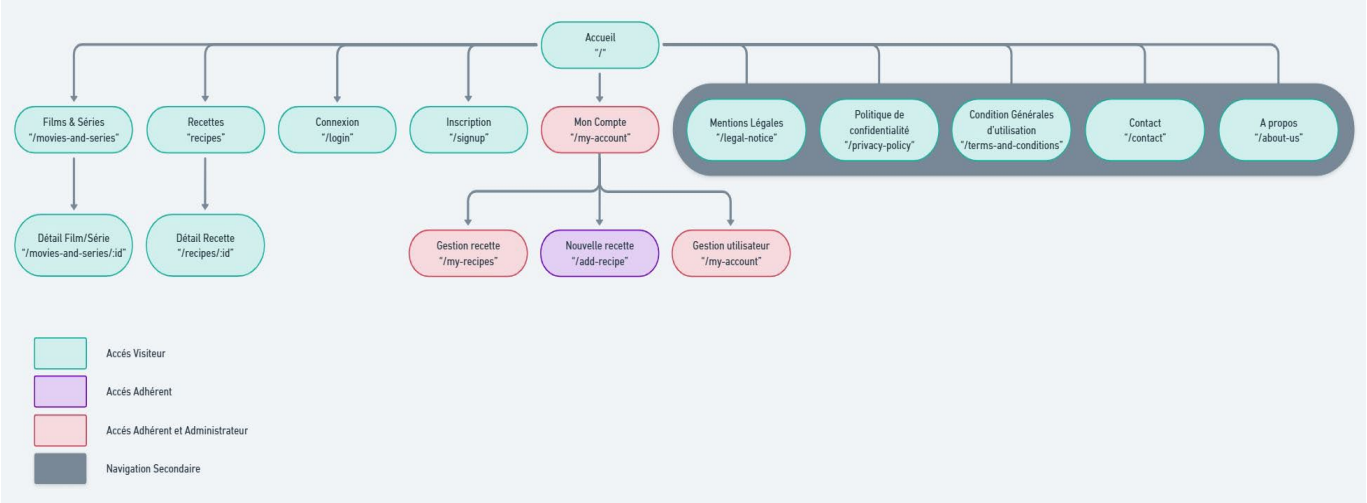
(Voir Annexes page 1)

V. Dictionnaire de données

(Voir Annexes page 3 et 4)

VI. Arborescence, Routes front-end et back-end, Endpoint API

1. Arborescence front-end



2. Routes back-end

(Voir Annexes page 2 et 3)

3. Enpoints API

➢ The MealDB :
- Liste des ingrédients complètes : www.themealdb.com/api/json/v1/1/list.php?i=list

VII. Spécifications techniques

- **Langage Front :** HTML, CSS, JavaScript
- **Préprocesseur :** Sass (syntaxe SCSS)
- **Langage Back :** JavaScript, SQL
- **Environnement d'exécution [Back] :** Node.js
- **Framework Front :** React.js
- **Framework Back :** Express.js
- **Bundler :**
  - *Parcel* :
    - Utilisation d'un serveur de développement
    - Utilisation du préprocesseur Sass ⇒ (transformation des fichiers sources en fichiers de production)
- **Bibliothèque/Package :**
  - *pg* : Bibliothèque permettant la connexion avec notre BD
  - *dotenv* : Gestion des variables d'environnement
  - *bcrypt* : Hachage, sécurisation de données sensibles (mot de passe)
  - *validator* : Permet de valider des formats de données

    - *slugify* : Permet de transformer des chaînes de caractère (lors des définitions de nos URL)
    - *react-helmet* : Gestion des composants REACT
        - Optimisation du SEO
        - Amélioration de l'accessibilité
    - *cors* : Définir la politique CORS dans Express ⇒ Accès à des ressources situées sur un autre domaine que celui de l'application web (APIs)
    - *react-slick :* bibliothèque de carrousel
    - *slick-carousel :* bibliothèque populaire pour créer des carrousels et des diaporamas réactifs
    - *multer* : middleware pour Express et Node.js qui facilite la gestion des données multipart/form, utilisées pour le téléchargement de fichiers.
- Base de données : PostgreSQL
- Authentification : JWT (JSON Web Token)
- Responsive : Notre application sera développée en mobile first et responsive
- Accessibilité :
    - Nous avons prévu de respecter les normes d'accessibilité web WCAG
    - Utilisation de *react-helmet*
- RGPD et mentions légales : Nous avons prévu mettre en place des mentions légales liées au règlement général sur la protection des données (RGPD)
- SEO : Utilisation, mise en place des techniques de référencement naturel avec *react-helmet*
- Versioning : Git, GitHub

VIII. Evolutions potentielles

- Possibilité de mettre les recettes en favoris
- Fonctionnalités sociales (commenter, liker et partager les recettes)
- Ajout d'un mode clair / sombre
- Possibilité d'accéder à un cours de cuisine collectif en ligne (1/mois par exemple)
- Système de recommandation (produits, ustensiles, partenariat…)
- Mise en place d'un système de notation utilisateur en fonction de ses likes (trophée : recette la plus originale par exemple)
- Utilisation d'une API de traduction pour les ingrédients (voire films également si synopsis en anglais)
- Ajout manuel par l'utilisateur de nouveau ingrédient si ce dernier n'est pas déjà disponible dans l'API (BD)
- Afficher les ingrédients nécessaires à la réalisation d'une étape ⇒ relation entre ingrédients et étapes

IX. Navigateurs compatibles

Notre application devra être testée pour être compatible avec les versions les plus récentes des navigateurs suivants :

- Chrome
- Firefox
- Brave

Déroulement du projet

I. Développer la partie front-end de notre application web

a. Introduction

Après avoir cerné la demande principale du client et défini une direction commune à notre projet, notre équipe avait besoin d'une vision globale de celui-ci. Pour ce faire, la réalisation de *wireframe* nous a octroyé une image schématique des fonctionnalités attendues par notre application. Suite à cela, les maquettes graphiques purent être créées, nous laissant entrevoir une représentation visuelle de nos pages.

Ayant fixé au préalable les technologies utilisées pour l'élaboration de notre projet, la mise en place et la configuration de notre environnement de développement furent les étapes naturelles qui s'en est suivie.

Nous avons pu enchaîner par la création de nos premières pages statiques, responsives et accessibles pour pouvoir finaliser ensuite sur une version dynamique de ces dernières.

b. Wireframes [Annexe p. 5 à 12]

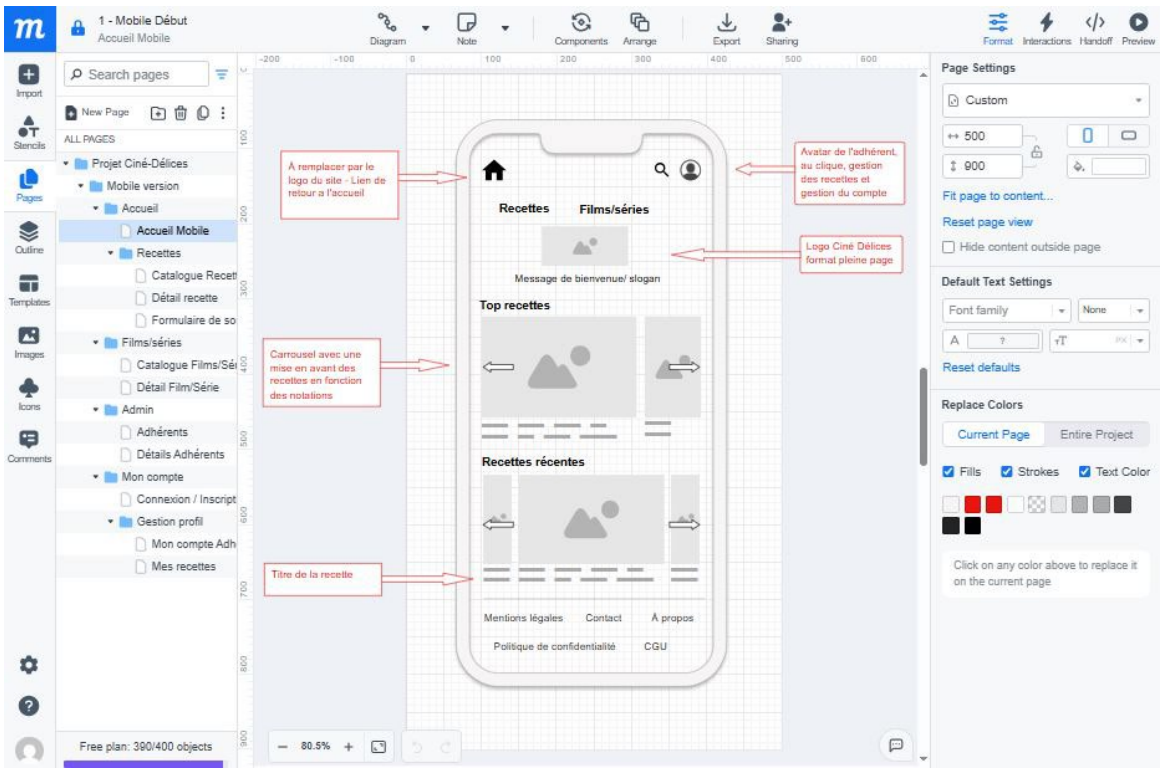
1. Outils de conception

Pour nous aider au mieux à concevoir nos interfaces utilisateur, l'élaboration des *wireframes* fut la phase primordiale à effectuer. Pour cela, nous avons eu recours à l'utilisation de la plateforme *Moqups* (https://moqups.com/).

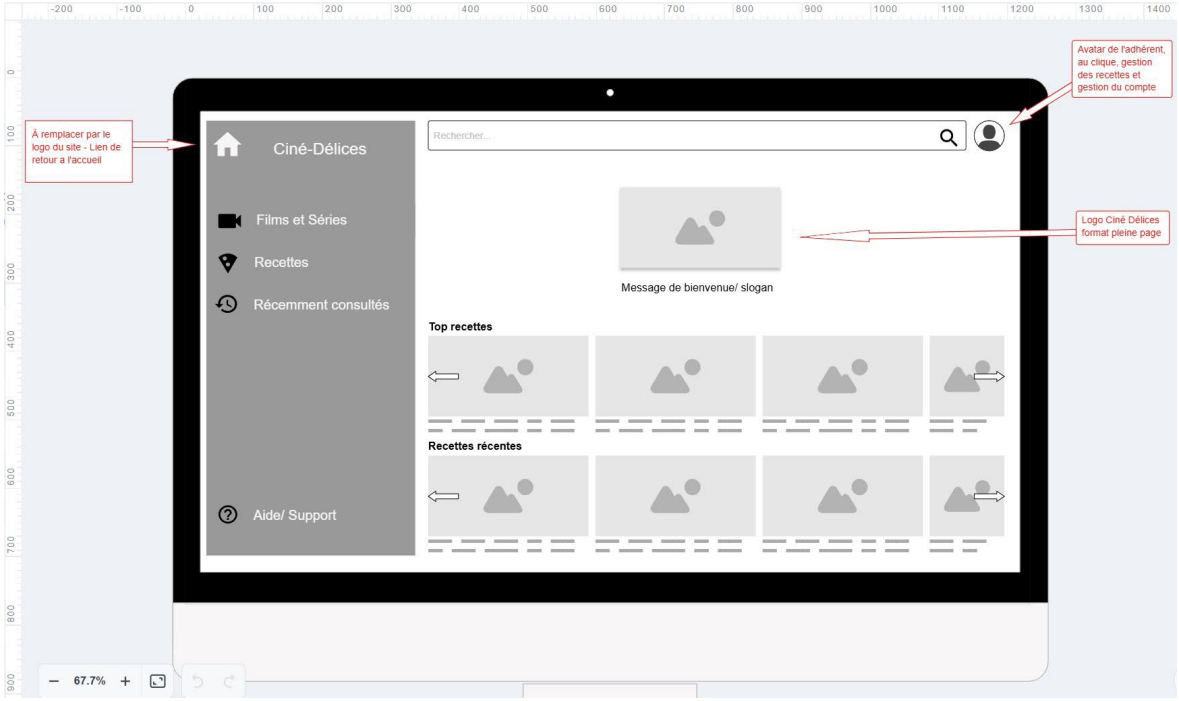
Moqups est un espace collaboratif, facile d'utilisation et qui nous a accordé la capacité de pouvoir partager et réaliser nos travaux de façon synchrone. L'échange et le partage d'idées étant grandement simplifiés, nous a permis de pouvoir avancer de façon efficace sur nos *wireframes*.

A l'instar d'autres outils de conception comparables (on peut penser au logiciel de retouche *Photoshop*, ou encore l'éditeur de graphiques vectoriels *Figma* sur lequel nous reviendrons par la suite), *Moqups* fonctionne grâce à l'utilisation de calques, que nous pouvons gérer de façon indépendante, modifier ou même dupliquer. De plus, le regroupement de plusieurs éléments sous forme de composant, permet d'éviter les actions redondantes comme la reproduction de certains ensembles d'éléments susceptibles de se répéter d'une page à une autre.
Par ailleurs, la possibilité de pouvoir organiser nos fichiers dans différents dossiers a favorisé le visuel sur la future arborescence de notre application.

- *Interface Moqups – Accueil version mobile :*



- *Exemple de wireframe édité sur Moqups – Accueil version desktop :*



2. Application : Page de détails du film/série

Prenons le cas de la réalisation de la page permettant d'afficher le détail d'un film ou d'une série (voir Annexes page 9). Ici le but étant de pouvoir donner un aperçu facilement compréhensible de la structure générale de la page tout en donnant des précisions sur des fonctionnalités prévues, qui mériteraient certaines informations complémentaires absentes du cahier des charges.

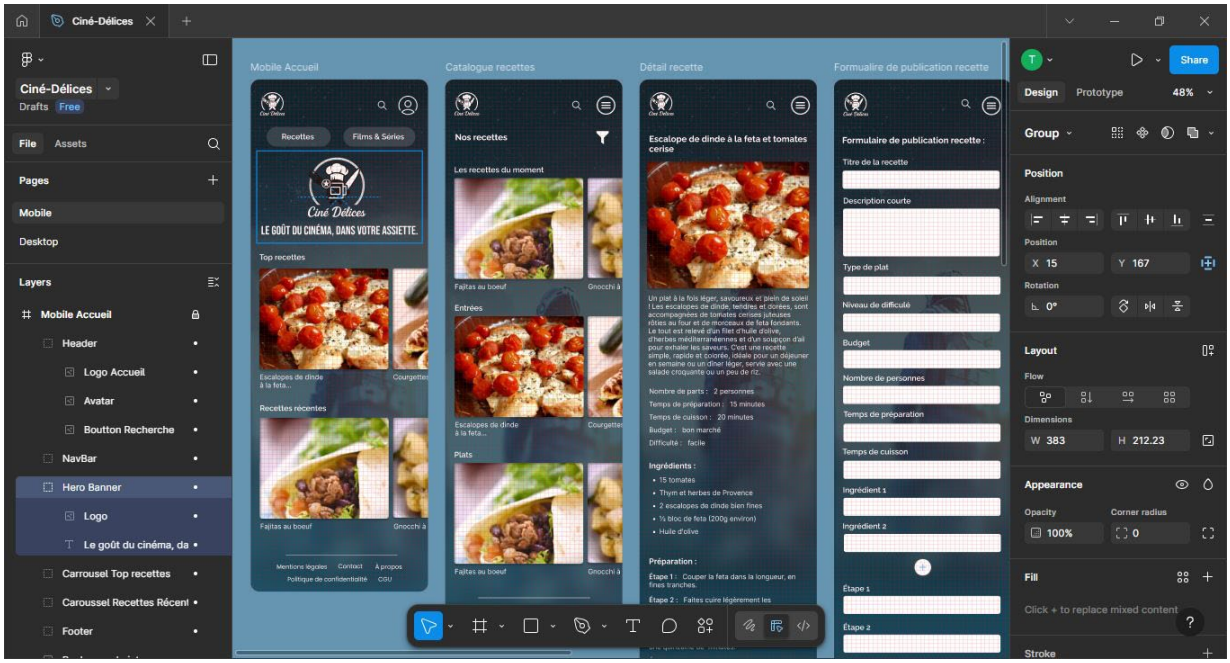
On peut y indiquer par exemple le type d'information attendu ou décrire les comportements de certains boutons d'action lors d'un clique.

c. Maquettes graphiques [Annexe p. 14 à 16]

1. Outils de conception

Etant en possession d'une meilleure perception de l'architecture générale de nos pages et des fonctionnalités attendues sur chacune d'entre-elles, nous pûmes entamer l'élaboration de nos maquettes graphiques. Comme mentionné plus tôt, lors de la phase de création de nos *wireframes*, nous nous sommes dirigés vers l'éditeur de graphiques vectoriels *Figma*.

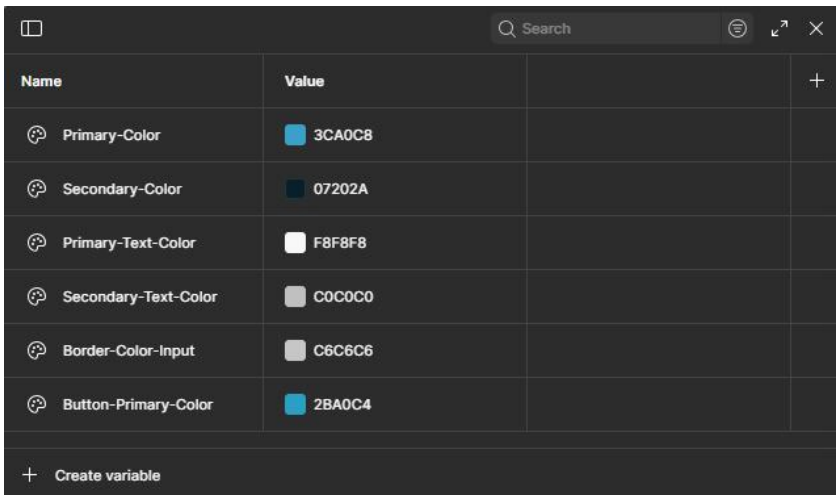
- *Interface Figma :*

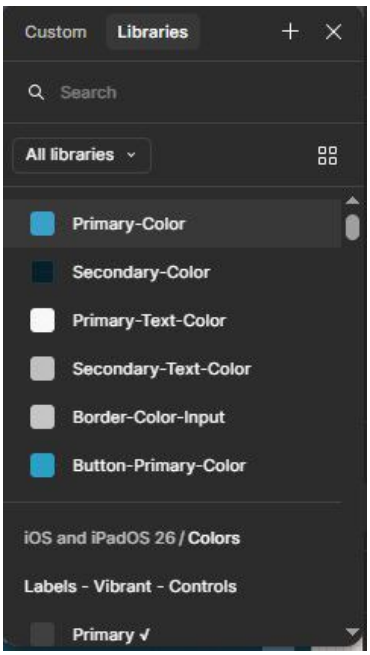


Tout comme *Moqups*, *Figma* se gère également par l'emploi de différentes couches appelées calques et a la possibilité de pouvoir nous proposer de regrouper plusieurs éléments ensemble.

Figma propose tout de même des fonctions plus avancées, comme la capacité de créer ses propres variables et d'y stocker les valeurs souhaitées dans l'optique de pouvoir les réutiliser par la suite.

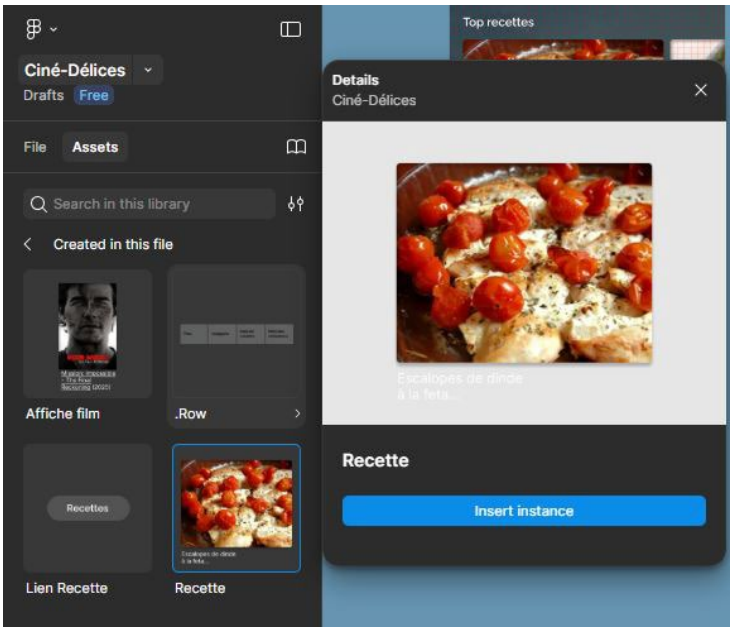
- *Exemple de création et d'utilisation de variables avec Figma :*





Figma est un outil de prototypage, il nous offre le choix de pouvoir créer nos propres composants réutilisables, l'un des plus grands avantages est qu'en plus de faciliter la duplication, ils fonctionnent avec une mécanique de « parent/enfant », ce qui veut dire que les modifications apportées au composant d'origine, se répercuteront de façon héréditaire à tout composant issu de ce dernier.

- *Visuel de composants créés sous Figma :*



2. Application : Page du catalogue de recettes (Voir Annexes page 15)

Ciné-Délices est une plateforme participative de partage de recettes de cuisine, mais il ne faut pas oublier que celles-ci sont inspirées de films ou de séries. Dans la

perspective de vouloir rassembler les deux univers, nous avons pris la décision de nous orienter sur un visuel rappelant les services en ligne de streaming.
C'est sur des tonalités froides et métalliques que nous avons dirigés notre choix concernant le panel de couleurs et un visuel proche des carrousels rencontrés sur des sites de partage de vidéos.
Les carrousels présents sur cette page sont des instances d'un composant d'origine.

d. Conception de l'interface utilisateur

1. Démarrage du projet : configuration de base

Conformément aux spécifications techniques de notre projet, notre équipe démarra par la mise en place de l'environnement de développement de notre application. Pour rappel, nous avons décidé d'utiliser la librairie *ReactJS* pour le rendu de notre interface visuelle.
Pour ce faire, nous utiliserons l'environnement d'exécution *Node.js (https://nodejs.org/fr)* pour l'installation de nos dépendances. Il est important de noter qu'on commencera par initialiser un fichier `package.json` directement depuis notre terminal avec la commande suivante :

```
npm init -y
```

Ce fichier `package.json` nous permettra de gérer nos dépendances (packages) et d'y inscrire nos scripts d'automatisation.
Une fois fait, toujours depuis notre terminal, nous pourrons installer toutes les dépendances utiles à la réalisation de notre projet comme dans l'exemple ci-dessous :

```
npm i react react-dom
```

Etant un package disponible sur *npm* (gestionnaire de paquets de notre environnement *node*), nous pourrons trouver cette fameuse ligne de commande d'installation ainsi que de plus amples détails sur la documentation du package *(https://www.npmjs.com/package/react)*. Il en va de même pour toutes les autres dépendances de notre projet.
On notera également la présence de l'installation de la dépendance de *react-dom* dans l'extrait ci-dessus, ce dernier permettant d'effectuer le processus de « réconciliation » entre le *DOM* (Document Object Model) virtuel et le *DOM* « réel ». Voir documentation pour plus de détails *(https://fr.legacy.reactjs.org/docs/faq-internals.html)*.

2. Création de nos premiers composants *React* :

ReactJS nous permet de concevoir une application sous forme de composant. C'est-à-dire qu'il nous est possible de pouvoir fractionner notre code.
Nous commencerons dans un premier temps, par créer un fichier `index.html` qui sera le fichier racine de notre projet et surtout qui contiendra l'élément représentant le composant parent de notre application.

index.html :

```html
<!DOCTYPE html>
<html lang="fr">
    <head>
        <title></title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link href="styles/index.scss" rel="stylesheet">
        <script src="index.js" type="module" defer></script>
    </head>
    <body>
        <!-- Elément racine de notre application -->
        <div id="root"></div>
    </body>
</html>
```

L'élément html : * * possédant l'identifiant « root » sera donc ce fameux composant racine.

index.js :

```js
// Import des styles pour react-slick
import "slick-carousel/slick/slick.css";
import "slick-carousel/slick/slick-theme.css";

// Création de notre composant rooter
import { createRoot } from "react-dom/client";

// Import du composant App, qui s'occupera de renvoyer nos rendus
import App from "./Components/App";

// Import de notre contexte
import { AuthProvider } from "./Authentication";

// Nous ciblons notre élément racine
const rootContainer = createRoot(document.getElementById('root'));

// Nous renvoyons le rendu dans notre élément racine
rootContainer.render(
    // Tous les composants enfants hériterons des valeurs du contexte
    <AuthProvider>
        <App />
    </AuthProvider>
);
```

Nous nous concentrerons ici pour l'instant sur la récupération de l'élément racine *« rootContainer »* dans lequel non imbriquerons le composant contenant toute notre application *« App »* (Nous reviendrons plus en détail sur le composant *AuthProvider* lorsque nous parlerons de l'authentification).

Composant *App* :

```
import AppRouter from "../Router";

export default function App() {
    return <AppRouter />;
}
```

Composant *AppRouter* :

```
/**
 * Import des méthodes de react-router-dom:
 * BrowserRouter que l'on renommera Router
 * Routes et Route
 */
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
// Import des composants renvoyant nos rendues
import Home from "../pages/Home";
import RecipeDetail from "../pages/Food/RecipeDetail";
import Recipes from "../pages/Food/Recipes";
```

BrowserRouter est un composant de la dépendance *react-router-dom*, il sert de composant principal à la navigation, c'est lui qui permettra entre autres, les modifications d'URLs lors des changements de pages, il fut renommé ici *Router*.
La configuration du composant qui s'occupera de la gestion des routes se fera donc ainsi :

```
export default function AppRouter() {
  return (
    // Composant principale (BrowserRouter)
    <Router>
      {/* Conteneur de toutes les routes */}
      <Routes>
        {/* Pages principales */}
        {/* Route permet d'associer une url à un composant  */}
        <Route path="/" element={<Home />} />
        <Route path="/recipes" element={<Recipes />} />
        <Route path="/recipes/:recipeId" element={<RecipeDetail />} />
```

Le composant *BrowserRouter* imbrique *Routes* qui sert de conteneur pour nos routes, on y imbrique par la suite nos composants *Route* qui s'occuperont de renvoyer les composants s'occupant d'afficher les rendus *(element)* en fonction de l'URL affichée *(path)*.

3. Rendu d'un composant React (statique) :

a. Structure HTML : Valeur sémantique, accessibilité et SEO

Nous utiliserons la syntaxe *JSX (JavaScrip Syntax Extension)* qui nous permettra de faciliter la création de notre interface utilisateur. Celle-ci permet d'écrire des « balises *HTML* » directement dans nos scripts *JS*.

(Voir Annexes page 17 pour l'extrait du Composant de formulaire de connexion)

Nous pouvons constater via ce bout de code, que notre composant est donc, l'initialisation d'une fonction *JavaScript*, qui retourne du code *HTML*. Du moins en apparence, mais en réalité le *JSX* permet d'écrire du code qui ressemble à du *HTML* mais qui n'en est pas. C'est pour cela qu'il reste indispensable d'utiliser un transpileur, comme *Babel* par exemple. Dans notre cas, il s'agira du bundler *Parcel (Babel y étant intégré)* qui nous offrira la capacité de pouvoir transformer notre code *JSX* en code *JavaScript* compréhensible par les navigateurs.

Il n'empêche que, lors de la transformation de notre code, il s'agit tout de même bien de balises *HTML* qui seront affichées par notre navigateur, il reste donc intéressant de pouvoir analyser la sémantique de nos balises.

Examinons donc nos balises, pour cela nous repartirons d'une version simplifiée du code de notre formulaire d'inscription :

```
<form
    className="account__form"
    method="POST"
>
    <fieldset>
        <h1>Créer votre compte</h1>
        <legend className="sr-only">
            Formulaire de création de compte :
        </legend>
        <div className="account__form__group">
            <label htmlFor="email" className=`account__form__label>
                <span>Email</span>
                {/* Indication sur le format attendu accessible */}
                <span className="sr-only">Exemple : nom@domaine.extension</span>
            </label>
            <input
                className="account__form__input"
                type="email"
                id="email"
                name="email"
                required
            />
        </div>
```

HTML est un langage de balisage, chacune de ces balises servent une fonction et ont une valeur sémantique. Cela a son importance pour le référencement naturel *(SEO)* mais également pour l'accessibilité, les outils d'aide à la navigation se basent en partie sur le sens qu'elles apportent.

Si nous décomposons un peu plus le code :

- *<form>* : est la balise représentant notre formulaire, elle permet d'indiquer une section de notre page contenant des contrôles interactifs
- *<fieldset>* : quant à elle, permet de regrouper nos champs ensemble
- *<h1>* : est un titre de niveau 1, important pour le référencement, il a également de la valeur pour l'accessibilité puisqu'il sera indiqué par un lecteur d'écran par exemple.
- *<legend>* : N'est pas obligatoire à proprement parlé pour le fonctionnement du formulaire mais reste important car il donne des informations concernant la section du formulaire *<fieldset>*

- *<label>* : indique la légende associée à son champ d'interaction *<input>* grâce à la valeur de son attribut *for* qu'il partage avec la valeur de l'attribut *id* de son *<input>*

Nous conclurons cette partie avec encore un peu d'accessibilité en mentionnant la présence d'une classe *« sr-only »*

```css
.sr-only {
    position: absolute;
    width: 1px;
    height: 1px;
    padding: 0;
    margin: -1px;
    overflow: hidden;
    clip-path: rect(0,0,0,0);
    white-space: nowrap;
    border: 0;
}
```

Cette classe a pour but de masquer visuellement le texte, tout en rendant possible la lecture de son contenu accessible aux liseuses d'écran.

b. Stylisation : la syntaxe SCSS et fonctionnalités *Sass*

Maintenant que nous avons initié une première approche avec la stylisation de notre application par l'intermédiaire de notre classe *« sr-only »*, approfondissons cela encore un peu plus.
Notre équipe a décidée d'utiliser la syntaxe *SCSS*, une des syntaxes du langage du préprocesseur *Sass* (*https ://sass-lang.com/*), tout comme pour la syntaxe *JSX*, *SCSS* ne peut être lu directement par le navigateur, c'est pour cela que notre bundler *Parcel* s'occupera une fois de plus de la transpilation et de la compilation du code.
Afin d'organiser au mieux notre code, nous avons décidé de séparer nos feuilles *SCSS*, le préprocesseur *Sass* rend possible la préparation de *modules SCSS*, favorisant une meilleure maintenabilité du code.
Il est donc possible de s'occuper d'une stylisation plus globale de notre application, allant jusqu'à la réalisation de feuilles plus spécifiques en fonction des cas.

Amenant à être compilé dans un seul et même fichier, nous pourrons lier nos différents modules ensemble grâce à l'utilisation du mot-clé « *@use* ».

`_global.scss` :

```scss
@use "./vars";
@use "sass:color";

body {
  margin: 0;
  padding: 0;
  font-family: vars.$font-main;
  background: vars.$background;
  color: vars.$text-light;
  line-height: 1.5;
}
```

Ceci est un extrait du fichier `_global.scss`, cette feuille servira notamment à styliser les éléments de manière générale de notre application.

Remarquons l'importation d'un module nommé `_vars.scss`, *Sass* nous offre l'opportunité de pouvoir créer des variables, ce seront des valeurs que nous stockerons dans un même fichier, augmentant encore une fois, la maintenabilité de notre code.

`_vars.scss` :

```scss
// Couleurs principales
$primary: #3ca0c8;
$bg-dark: #07202A;
$bg-light: #18576E;
$background: linear-gradient(to bottom, $bg-dark, $bg-light);
$bg-section: #2a2a2a;

// Couleurs des textes
$text-light: #f8f8f8;
$text-secondary: #c0c0c0;

// Typographie
$font-main: 'Raleway', sans-serif;
$font-secondary: 'Inter', sans-serif;
```

Voici à quoi peut ressembler un fichier de variable écrit avec la syntaxe *SCSS*.
On commence par initialiser notre variable avec le symbole « *$* » suivi du nom de notre variable et de la valeur en CSS.
Nous n'aurons plus qu'à importer notre module dans les fichiers où l'on en aura besoin. Une fois lié, il ne nous restera plus qu'a y faire appel en respectant la syntaxe suivante :

```scss
@use "./vars";

body {
  color: vars.$text-light;
}
```

Il nous est également permis d'utiliser des fonctions disponibles dans des modules déjà créé par *Sass* comme dans notre extrait de code du fichier `_global.scss` :
« *@use « sass :color » ;* » est l'un de ces fameux modules en question *(https ://sass-lang.com/documentation/modules/color/)*.
Il ne nous restera plus qu'à rassembler toutes nos feuilles de style dans un même fichier `index.scss`, qui aura pour vocation d'être compilé et transformé pour rassembler l'ensemble de nos styles et être lisible par le navigateur.

`index.scss` :

```scss
// Fichier qui s'occupe de la gestion générale du style de notre application
@use './global.scss';

// Feuilles de styles de nos différents composants et pages
@use '../Components/Footer/footer';
@use '../Components/NavLink/navlink';
@use '../Components/Header/NavBar/navbar';
```

C'est en suivant donc ce processus que nous avons réalisés pour chacun des composants, nos feuilles de styles.
Repartons donc sur la stylisation de notre composant de formulaire de connexion (cf. code complet : Annexes p.18).

Extrait de code du fichier `AccountForm.scss` :

```scss
@use "../../vars";

.main__form {
    display: flex;
    justify-content: center;

    .account__form__wrapper {
        background-color: rgba(0, 0, 0, 0.1);
        width: 80%;
        padding: 1.5rem;
        border-radius: 10px;

        // Boutton de formulaire
        .account__form__button {
            position: relative;
            margin-top: 4rem;
            padding: 1.4rem 0;
            border: 2px solid vars.$border-color-input;
            border-radius: 40px;
            background-color: vars.$button-primary-color;
            transition: .3s;

            &:hover,
            &:focus
            {
                background-color: vars.$button-secondary-color;
                outline: none;
            }
        }
    }
}
```

Nous nous focaliserons sur cette partie du code pour évoquer certaines fonctionnalités exploitables avec *Sass* et la syntaxe *SCSS*.
Parlons donc de la notion de *Nesting* qui nous permettra d'éviter la répétition de sélecteurs lors de notre stylisation, il nous suffira d'imbriquer nos sélecteurs enfants, directement dans leur parent, ce que l'on pourrait rapprocher d'une structure *HTML*. De plus, l'utilisation du symbole « *&* » permet de répéter un sélecteur complet, parfait pour l'utilisation de pseudo-classe comme « *:hover* » par exemple.

Extrait de code du fichier `AccountForm.scss` sans utilisation du nesting*:*

```
.main__form {
  display: flex;
  justify-content: center;
}

.main__form .account__form__wrapper {
  background-color: rgba(0, 0, 0, 0.1);
  width: 80%;
  padding: 1.5rem;
  border-radius: 10px;
}

/* Bouton de formulaire */
.main__form .account__form__wrapper .account__form__button {
  position: relative;
  margin-top: 4rem;
  padding: 1.4rem 0;
  border: 2px solid vars.$border-color-input;
  border-radius: 40px;
  background-color: vars.$button-primary-color;
  transition: 0.3s;
}

.main__form .account__form__wrapper .account__form__button:hover,
.main__form .account__form__wrapper .account__form__button:focus {
  background-color: vars.$button-secondary-color;
  outline: none;
}
```

Nous terminerons ce point avec l'utilisation d'un dernier concept qui est celui des *mixins*, il reprendra légèrement les fonctionnalités d'une fonction *JavaScript* par exemple, surtout qu'ils partageront un but commun qui est celui d'éviter de répéter du code au maximum. Nous pourrons séparer nos *mixins* dans un module et comme pour les variables, les appeler en cas de besoin après les avoir importées.
En plus de certaines propriétés susceptibles d'être répété sur des éléments similaires entre eux comme des boutons d'actions, des *flags*, etc… on peut y configurer aussi les différents breakpoints de nos *media queries.*

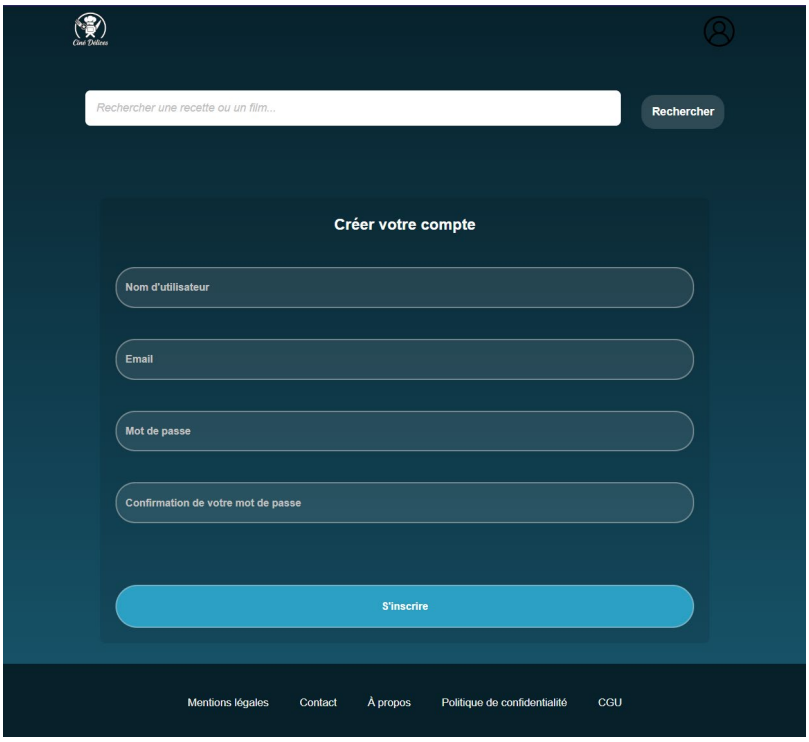
`_mixins.scss` :

```
@mixin small {
    @media screen and (min-width: 480px) {
        @content;
    }
}

@mixin medium {
    @media screen and (min-width: 768px) {
        @content;
    }
}

@mixin large {
    @media screen and (min-width: 1024px) {
        @content;
    }
}
```

Exemple d'utilisation des media queries (Voir Annexes page 19):

C'est grâce à ces différentes méthodes qu'il nous a été possible de construire, de manière statique, la multitude de composants responsables du rendu de notre application, tout en respectant les conventions d'écriture HTML, en optimisant le référencement naturel et en veillant à garantir l'accessibilité pour tous les utilisateurs, quelle que soit la taille de leur écran.

Résultat statique du rendu de notre formulaire d'inscription :



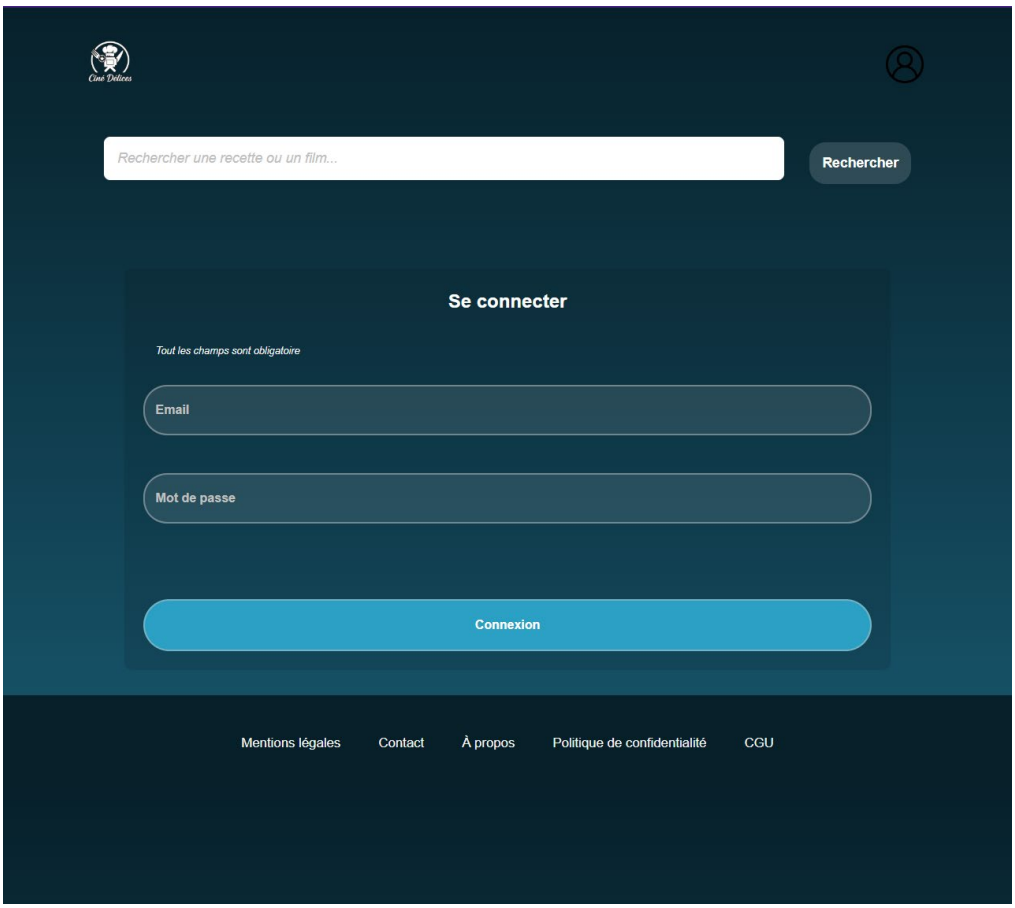
4. Dynamisation de nos composants React

Dorénavant en possession de composants statiques, il est néanmoins essentiel de mettre en place la dynamisation de ces derniers. Le tout ayant pour but d'optimiser les interactions avec l'utilisateur. Que ce soit d'un point de vue visuel ou dans le cadre de traitement de données.

Nous avons mentionné plus tôt, la notion de *DOM* virtuel et de *DOM* « réel ». Le *DOM* est le document que le navigateur va créer pour représenter nos éléments *HTML* sous forme de nœud *(Node)*. C'est le *DOM* que nos scripts *JS* vont manipuler pour pouvoir dynamiser nos éléments et les mettre à jour.

Dans le cas de *React*, la manipulation passe par ce fameux *DOM* virtuel, qui est une représentation du *DOM* « réel » qu'il garde en mémoire. Pour chaque changement, il comparera donc le *DOM* virtuel avec le *DOM* « réel » et modifiera uniquement les valeurs à mettre à jour.
Pour pouvoir illustrer au mieux, la manipulation des variables de nos composants, prenons cette fois-ci notre formulaire d'inscription.

Affichage de notre formulaire de connexion :



Nous visons, d'une part à récupérer les informations de notre formulaire pour être dans la capacité de pouvoir les traiter en base de données et d'autre part pouvoir manipuler le *DOM* pour y ajouter certains effets visuels. Il nous a fallu définir les valeurs qui seront donc amenées à évoluer.

ReactJS met à notre disposition plusieurs outils pour pouvoir interagir avec nos composants. Des fonctions natives à *React* (telles que les *Hooks,* ou les *Handlers*) sont prévues à cet effet. Notre objectif est donc d'utiliser ces fameux *Handlers* afin de gérer nos événements en fonction d'un résultat obtenu par la modification d'un état initial.

Tout d'abord, occupons-nous de récupérer la valeur de nos champs de formulaire, ces valeurs sont des données qui seront amenées à évoluer.
Par exemple, lorsque l'utilisateur entrera son adresse e-mail pour se connecter, la valeur devra correspondre à la donnée saisie.
React nous permet également de créer nos propres *Hooks*. Dans un souci de clarté et pour mieux organiser notre code, nous avons décidé de séparer la logique qui traitera les données dans un *Hook personnalisé,* tandis que le composant principal s'occupera du rendu visuel.

Pour illustrer tout ça, nous nous baserons sur un code simplifié de ces deux différents fichiers, le code complet étant disponible en annexes (Voir Annexes page 20 et 21).

Hook personnalisé :

```
import { useState } from "react";

export default function useLogin() {
    // Définition des variables d'état pour chaque champs, etat initial = chaîne de caractère vide
    const [email, setEmail] = useState("");
    /**
     * Handler (événement) lors de la soumission du formulaire
     * Ici, nous nous contentons d'empêcher le rechargement de la page et d'afficher les valeurs saisies
     */
    const handleSubmit = (e) => {
        e.preventDefault();
        console.log("Email :", email);
    };

    return {
        email,
        setEmail,
        handleSubmit
    };
}
```

Ci-dessus, nous commençons par importer le *Hook React useState*, c'est lui qui fera en sorte que nous puissions paramétrer et surtout récupérer la valeur de nos champs, il sera composé de deux valeurs, la première étant la valeur de l'état actuel et la seconde qui nous permet de la mettre à jour. Nous passerons en argument de *useState()* une valeur initiale, ici une chaîne de caractères vide.
handleSubmit est notre *Handler,* il correspond à l'événement qui sera exécuté lorsqu'il sera appelé. Et pour finir nous n'oublions pas de retourner nos variables pour pouvoir les utiliser dans notre composant principal.

Composant principal :

```
import useLogin from "../../../Hook/useLogin";

export default function LoginForm() {
    // On importe nos variables d'état ainsi que nos handler de notre hook personnalisé
    const { email, setEmail, handleSubmit } = useLogin();

    return (
        <form
            className="account__form"
            method="POST"
            onSubmit={handleSubmit}
        >
            {/* Ajout d'un fieldset, pour mieux structurer sémantiquement le formulaire */}
            <fieldset>
                {/* Ajout d'une légende, qui sera "cachée" visuellement, mais accessible via une liseuse d'écran */}
                <legend className="sr-only">Formulaire de connexion :</legend>
                <span className="account__form__info">Tous les champs sont obligatoires</span>

                <div className="account__form__group">
                    <label htmlFor="email" className="account__form__label">
                        Email
                        {/* Indication sur le format attendu accessible */}
                        <span className="sr-only">Exemple : nom@domaine.extension</span>
                    </label>
                    <input
                        className="account__form__input"
                        type="email"
                        id="email"
                        name="email"
                        // prise en compte de la valeur du champ une fois rempli
                        onChange={(e) => setEmail(e.target.value)}
                        value={email}
                        required
                    />
                </div>
                <button type="submit" className="account__form__button">
                    Connexion
                </button>
            </fieldset>
        </form>
    );
}
```

On commence par importer notre *hook personnalisé*, puis en utilisant le *destructuring* (*https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring*) on fait appel à nos variables contenues dans notre *hook*.

Ensuite, nous n'avons plus qu'à les paramétrer pour qu'elle puisse récupérer les valeurs en fonction des événements associés. Dans ce code simplifié, nous affectons la valeur de nos champs *inputs* à celle de nos variables d'état lors de l'appel d'un événement de soumission de formulaire. A chaque changement de cet état, la valeur est bien mise à jour.

Nous pourrons nous attarder brièvement sur la gestion dynamique de nos *labels* étant donné qu'elle se basera sur le même principe.

Comme nous pouvons le constater sur notre visuel (Voir page 26 de ce dossier), ce sont bien nos *labels* qui se trouvent dans nos champs de formulaire. Ayant pris la décision de ne pas user d'attribut *placeholder* pour nos champs, notre équipe s'orienta donc sur la mise en place de « *floating-label* ».
Cette fois-ci, nous créerons une variable d'état, *focusState* qui permettra de déterminer si un champ est focus par notre utilisateur ou non. Elle sera composée d'un objet contenant des propriétés correspondantes à nos différents champs, nous leur affecterons à tous, une valeur initiale à *false (booléen)* qui indiquera bien une absence de focus.

```
/**
 * Initialisation de variable d'état
 * Gestion via un booléen
 * true au focus de l'input / false défocus
 */
const [focusState, setFocusState] = useState({
    username: false,
    email: false,
    password: false,
    confirmPassword: false
})
```

Nous utiliserons le *Handler*, *handleFocus* pour passer nos valeurs à *true* en utilisant ce coup-ci, la syntaxe de *décomposition*.

```
/**
 * Handler permettant la gestion des variables d'état au focus d'un champs
 * @param {string} field - Chaîne de caractère représentant la clé de notre objet stocké dans notre variable d'état
 */
const handleFocus = (field) => {
    /**
     * ``prev`` représente notre objet stocké dans notre variable d'état
     * Donc nous récupèrons l'objet, ses clés et valeurs associées
     */
    setFocusState((prev) => ({
        /**
         * Utilisation du spread operator pour récupèrer cet objet
         * Et mettre à jour une première fois notre variable d'état
         * Nous indiquerons ensuite qu'il faudra asigner la valeur ``true``
         * A la clé [field] de notre objet
         */
        ...prev, [field]: true
    }))
}
```

Et les valeurs à *false*, lorsque nos champs perdent le focus avec l'aide du *Handler handleBlur.*

```
// Même chose que pour le handler précédent mais cette fois ci pour gérer la perte de focus
const handleBlur = (field) => {
    setFocusState((prev) => ({
        ...prev, [field]: false
    }))
}
```

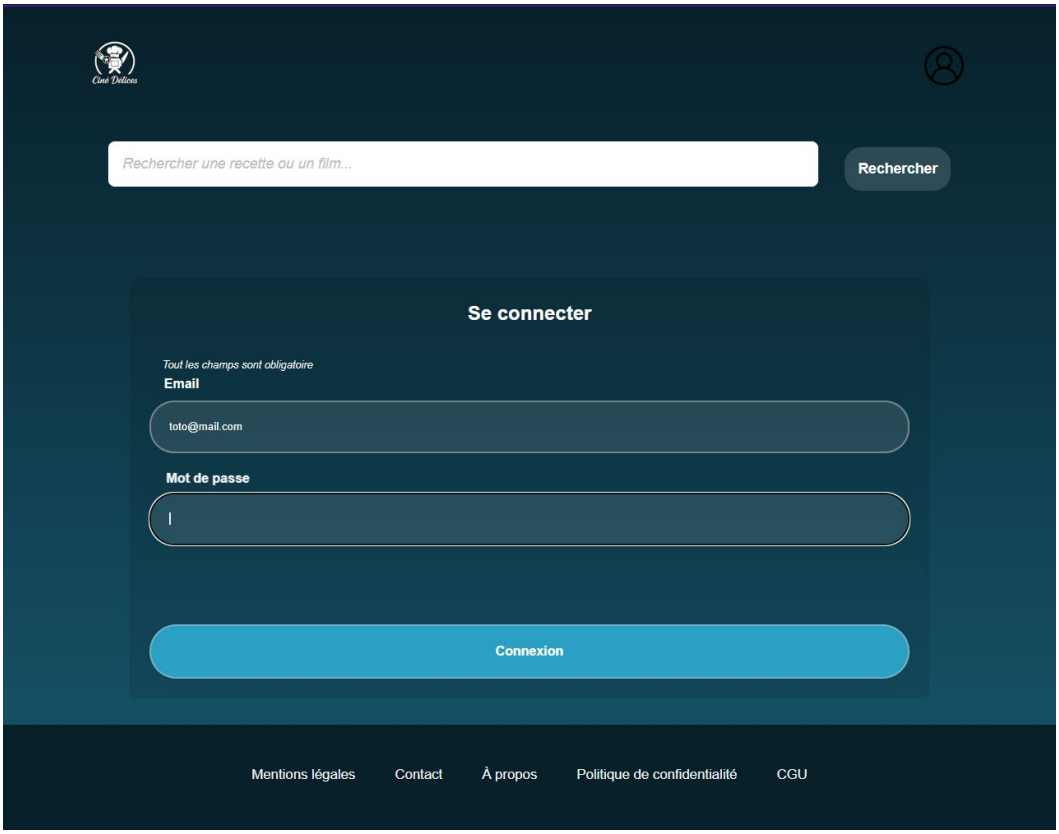
On récupère la valeur de nos champs de formulaire.

```
/**
 * On récupère séparément la valeur de chaque clé de notre objet stocké dans notre variable d'état
 * Elle nous servirons pour gérer la condition pour ajouter une classe au focus ou la retirer
 */
const isUsernameActive = focusState.username;
const isEmailActive = focusState.email;
const isPasswordActive = focusState.password;
const isConfirmedPasswordActive = focusState.confirmPassword;
```

Il ne nous restera plus qu'à conditionner directement notre vue pour nos différents *inputs*.

```
{/*
Concernant la gestion des classes CSS :
Utilisation de la synthaxe ternaire
SI isUsernameActive est vraie (donc si le focus est actif)
OU si le champs possède une valeur
ALORS ajout de la classe ``is-active``
SINON => Chaîne de caractère vide
*/}
<label htmlFor="username" className={`account__form__label ${isUsernameActive || username ? "is-active" : ""}`}>
    Nom d'utilisateur
</label>
<input
    className="account__form__input"
    type="text"
    id="username"
    name="username"
    // prise en compte de la valeur du champ une fois rempli, ici spécifique au nom d'utilisateur
    // puis par la suite adapté en fonction de la valeur des autres champs
    onChange={(e) => setUsername(e.target.value)}
    // Passe la valeur de username (variable d'état à true, lors du focus)
    onFocus={() => handleFocus("username")}
    // Passe la valeur de username (variable d'état à false, lors de la perte du focus)
    onBlur={() => handleBlur("username")}
    value={username}
    required
/>
```

Ce qui nous permet d'obtenir le résultat suivant :



5. Authentification : Mise en place d'un contexte React

L'un des objectifs principaux de *Ciné-Délices* est de pouvoir proposer à un utilisateur la capacité de créer de nouvelles recettes de cuisine. Il pourra également les modifier ou les supprimer. Mais il serait regrettable qu'un autre utilisateur puisse le faire à sa place. C'est pour ça qu'il est important de mettre en place un système d'autorisation par authentification. Pour cela, nous nous sommes orientés sur une gestion de l'authentification grâce à la génération d'un jeton sécurisé *(token)*. Notre choix c'est dirigé vers *JWT (JSON Web Token) (https://www.npmjs.com/package/jsonwebtoken)* qui propose, par l'intermédiaire de sa dépendance et ses fonctions, de permettre de générer un *token*.
Pour pouvoir paramétrer et gérer ses routes, il est essentiel que notre *token* puisse être lu par l'ensemble de nos composants principaux. Le concept de *Context* en *React* répond à cette demande, grâce à cela nous pourrons centraliser la logique d'authentification sans avoir à les transmettre manuellement *(https://fr.react.dev/reference/react/createContext)*.

Fonctionnant comme un composant, il sera le parent global de notre application (encore une fois on souhaite qu'il puisse être applicable sur tout nos composants) nous avons donc initialiser ce *Context* comme un composant *React* (Voir Annexes page 22 et 23).
Mais il est important tout de même de souligner que des *hooks* plus spécifiques à son utilisation seront indispensables à sa conception.

```jsx
//Import des Hooks essentielles à la création d'un contexte d'environnement

import { createContext, useContext } from "react";

/**
 * On initialise un nouveau contexte qu'on exportera
 * @link https://fr.react.dev/reference/react/createContext#importing-and-exporting-context-from-a-file
*/
export const AuthContext = createContext(null);
// On crée un hook personnalisé pour accéder au contexte d'authentification
// Ce hook permet d'utiliser le contexte d'authentification dans n'importe quel composant
export function useAuth() {
    return useContext(AuthContext);
}
```

Une fois notre nouveau contexte initialisé, ça sera par l'intermédiaire du *Provider* que nous pourrons communiquer les valeurs de nos variables d'état à nos composants, le *Provider* ayant pour fonction d'envelopper nos composants qui auront besoin de ces résultats.
On y initialise donc nos variables d'état qui correspondront au *token* et d'autres essentielles à l'authentification.

```jsx
export function AuthProvider({ children }) {
    const [token, setToken] = useState(null);
    const [isAuthenticated, setIsAuthenticated] = useState(false);
    const [isLoadingUser, setIsLoadingUser] = useState(true);
    const [username, setUsername] = useState(null);
    const [userData, setUserData] = useState(null);
```

Notons que nous utiliserons la fonction *useEffect* qui nous permettra de réaliser via une fonction anonyme appelée qu'une seule fois après le chargement du composant la vérification de l'existence du *token*, et si oui la vérification de sa validité côté back de notre application.
Il ne nous restera plus qu'à retourner la fonction et ses variables à tout composant enfant de notre *Provider*.

```
/**
     * Le contexte fonctionne comme un composant React
     */
    return (
        <AuthContext.Provider value={{ token, isAuthenticated, isLoadingUser, username, userData, login, logout }}>
            {children}
        </AuthContext.Provider>
    );
```

index.js :

```
// Import de notre contexte
import { AuthProvider } from "./Authentication";

// Nous ciblons notre élément racine
const rootContainer = createRoot(document.getElementById('root'));

// Nous renvoyons le rendu dans notre élément racine
rootContainer.render(
    // Tous les composants enfants hériterons des valeurs du contexte
    <AuthProvider>
        <App />
    </AuthProvider>
);
```

Le code ci-dessus est un extrait du fichier racine de notre application *React*, on y remarque donc que notre contexte d'environnement est bien appliqué à l'ensemble de nos composants.

II. Développer la partie back-end de notre application web

a. Introduction

La partie front-end de notre application est l'interface visible et interactive destinée à nos utilisateurs, quant à la partie back-end, elle constituera la couche logique et fonctionnelle de notre projet. C'est ici que nous ferons le lien avec notre base de données et donc potentiellement des données confidentielles de nos utilisateurs. C'est pour cela qu'il sera important de proposer un environnement sécurisé pour le traitement de nos ressources.

A l'image de la conception de la partie front-end, les premières étapes d'élaboration de notre back-end furent les documents de conception, support sur lequel nous avons pu nous appuyer pour la construction de notre base de données.
Nous nous sommes dirigés vers l'utilisation du *RDBMS* PostgreSQL pour la mise en place de celle-ci.

Après configuration de notre client *pg (dépendance Node)* qui nous a permis de mettre en relation notre environnement d'exécution avec notre base *PostgreSQL*, nous avons pu paramétrer les routes de notre API tout en suivant l'architecture *MVC (Modèles, Vues, Contrôleurs)*.
Les modèles ont été conçus selon le *design pattern Active Record*, grâce auquel nous avons défini nos méthodes du *CRUD (Create Read Update Delete)* permettant de faire le lien entre notre BD et notre back-end.

b. Documents de conception

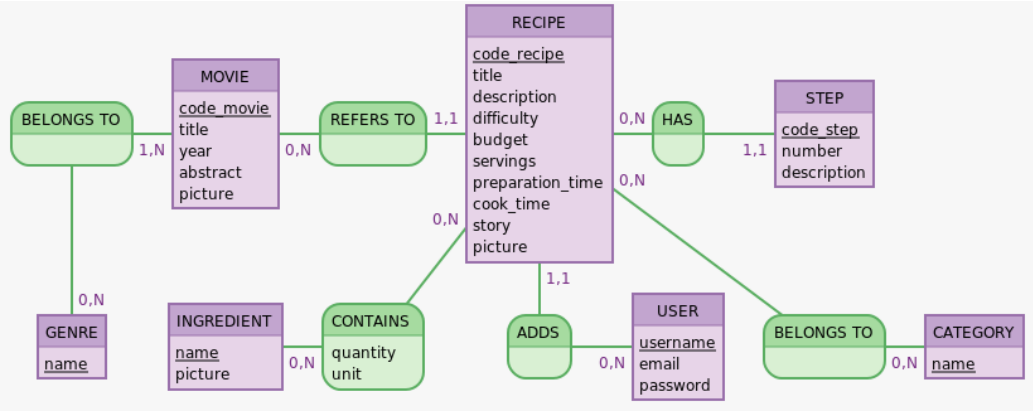
Pour pouvoir préparer au mieux notre base de données, la création de nos documents de conception fut une étape importante pour nous permettre de pouvoir entrevoir de façon claire nos objectifs. S'inspirant de la méthode de conception *Merise*, notre équipe commença par l'élaboration de trois documents clés ayant pour but de minimiser au maximum les erreurs de conception.

1. Modèle de Conception de Données :

Commençons donc par évoquer notre *MCD* (Modèle de Conception de Données), ce qu'il représente et le parcours que nous avons emprunté pour sa réalisation.

Son but est de nous permettre d'identifier nos différentes entités. Pour chacune d'entre elles, nous pourrons lister les attributs qui les composeront. De plus elle nous donne la capacité de visualiser les liens entre chaque entité. Pour finir, nous pourrons déterminer quel type d'association lie nos différentes entités grâce aux cardinalités.

MCD de notre projet (réaliser avec MOCODO (https://www.mocodo.net/)):



Pour illustrer nos dires, prenons l'entité RECIPE. RECIPE possèdent des attributs : un titre, une description, une difficulté, etc… ce seront ces derniers qui composeront nos futures tables et qui définiront chaque enregistrement. On peut remarquer que « *code_recipe* » est souligné, il s'agit de notre discriminant et il représentera un attribut qui sera unique à chaque recette. Une recette (RECIPE) fera référence à un film ou série au minimum et au maximum, cela est déterminé par nos cardinalités. Mais un film ou série pourra faire référence à aucune recette (donc pourra « exister » en base de données sans forcément être lié à une recette) ou plusieurs au maximum. On parlera donc de relation « Many To One » pour l'association entre MOVIE et RECIPE et de relation « One To Many » pour celle entre RECIPE et MOVIE.

En suivant ce schéma, nous pouvons donc déterminer comment nos entités seront construites et quel lien elles entretiendront entre elles.
Après avoir modélisé notre MCD, nous avons pu enchaîner sur la création de notre *MLD* (Modèle Logique de données)

2. Modèle Logique de Données

Seconde étape de notre méthode *Merise*, le MLD quant à lui nous permettra de traduire nos associations, déterminer nos clés primaires et étrangères ainsi que nos tables de liaison.

MLD de notre projet :

MLD :

recipe (id, title, description, difficulty, budget, servings, preparation_time, cook_time, story, picture #user_id, #movie_id)

category (id, name)

recipe_has_category (#recipe_id, #category_id)

ingredient (id, name, picture)

recipe_has_ingredient (id, quantity, unit, #recipe_id, #ingredient_id)

user (id, username, email, password)

step (id, number, description, #recipe_id)

movie (id, title, year, abstract, picture)

genre (id, name)

movie_has_genre (#movie_id, #genre_id)

Reprenons une nouvelle fois notre entité *RECIPE* comme exemple. Composé des mêmes attributs (futures colonnes de notre table), le discriminant est devenu un identifiant de plus, tout en interprétant nos associations, nous avons déterminé nos clés étrangères. Si on reprend l'association « *Many To One* » entre *RECIPE* et *MOVIE*, notre clé étrangère se trouvera dans notre table *RECIPE*. L'enregistrement d'une recette contiendra donc bien l'identifiant d'un film, « elle fera référence à ». Nous pourrons également évoquer la présence de *« recipe_has_ingredient »* ceci sera donc une table de liaison, découlant de la traduction de la relation « Many To Many » entre *RECIPE* et *INGREDIENT*.

3. Modèle Physique de Données

La dernière étape sera la conception de notre MPD. Il se veut être le plus proche au niveau syntaxique de notre script de création de notre base de données. On y ajoutera donc nos contraintes et le type de données pour chacune de nos colonnes.

MPD de notre projet :

```
"user" (
"id" INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
"username" TEXT UNIQUE NOT NULL,
"email" TEXT UNIQUE NOT NULL,
"password" TEXT NOT NULL
)

"recipe" (
"id" INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
"title" TEXT NOT NULL,
"description" TEXT UNIQUE NOT NULL,
"difficulty" TEXT NOT NULL,
"budget" TEXT NOT NULL,
"servings" INTEGER NOT NULL,
"preparation_time" INTEGER NOT NULL,
"cook_time" INTEGER NOT NULL,
"story" TEXT UNIQUE,
"picture" TEXT UNIQUE,
"user_id" INTEGER REFERENCES "user"("id") NOT NULL
)
```

```
"category" (
"id" INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
"name" TEXT NOT NULL
)

"recipe_has_category" (
"id" INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
"recipe_id" INTEGER REFERENCES "recipe"("id"),
"category_id" INTEGER REFERENCES "category"("id")
)

"step" (
"id" INTEGER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
"number" INTEGER NOT NULL,
"description" TEXT NOT NULL,
"recipe_id" INTEGER REFERENCES "recipe"("id")
)
```

Nous continuerons d'utiliser la table *RECIPE* comme référence.
Rappelons que nous avions décidé d'utiliser le *RDBMS PostgreSQL*, prenons quelques colonnes de notre table *RECIPE :*

- « id » est de type INTEGER, il est un identifiant et sera représenté par un nombre, il sera toujours généré de façon automatique à chaque nouvel enregistrement comme clé primaire
- « title » sera de type TEXT, donc une chaîne de caractère et devra être obligatoire
- « description » sera également de type TEXT et devra en plus être unique à chaque enregistrement.
- Etc…

Suite à cela, nous avions quasiment notre script de migration qui fut créé. Mais avant de pouvoir le mettre en place, il nous faut avant tout créer notre base de données à proprement dit et son utilisateur.

c. Mise en place : Création de notre base de données et configuration

1. Création de notre base de données avec PostgreSQL

Après avoir conçu tous nos documents de conception, notre équipe fut en possession des moyens nécessaires pour concevoir la base de données de notre projet. Nous avons donc commencé par créer cette dernière via l'utilisation de notre outil de gestion de bases de données qui est *PostgreSQL.*
Depuis notre terminal directement, il est possible de se connecter en tant que *superUser*, compte administrateur possédant tous les droits, avec la ligne de commande suivante :

```
psql -U postgres
Mot de passe pour l'utilisateur postgres :

postgres=#
```

Ci-dessus, nous indiquons vouloir nous connecter à notre *RDBMS,* l'option ***-U*** suivi de ***postgres,*** précise que nous utiliserons le compte ***postgres***, compte créé lors de l'installation de notre outil. Si après avoir saisi notre mot de passe *superUser* la ligne ***postgres=#*** apparaît, c'est que notre connexion est un succès.

Nous énumérerons par la suite, les commandes essentielles à la création de notre BD, puis nous les commenterons :

```
# Création d'un nouvel utilisateur suivie de son mot de passe depuis notre compte superUser
postgres=# CREATE USER cinedelices WITH PASSWORD 'motDePasseDeNotreBD';

# Indique que notre utilisateur fut créé avec succès
CREATE ROLE

# Création de la base de données, nous lui indiquerons son nom ainsi que l'utilisateur associé
postgres=# CREATE DATABASE cinedelicesdb WITH OWNER cinedelices;

# Indique que notre BD fut créé avec succès
CREATE DATABASE

# Depuis notre terminal directement
psql -d cinedelicesdb -U cinedelices
Mot de passe pour l'utilisateur cinedelices :

# Nous sommes bien connecté en tant qu'utilisateur cinedelices
# Si on le souhaite nous pouvons vérifier la liste de nos bases de données avec la commande suivante
cinedelices=> \l

        Nom        |    PropriÛtaire    | Encodage | Fournisseur de locale | Collationnement | Type caract. | Locale | RÞgles ICU : |
Droits d'acc
-------------------+--------------------+----------+-----------------------+-----------------+--------------+--------+--------------+------
 cinedelices       | cinedelicesdb      | UTF8     | libc                  | fr              | fr           |        |              |
```

2. Création de nos tables : Migration

Suite à la création de notre base de données et de notre utilisateur depuis notre *RDBMS*, nous avons décidé de réaliser la migration de nos tables. Cela consiste tout simplement à retranscrire notre MPD sous format *.SQL* (Voir Annexes page 24).
Il nous suffit seulement donc d'ajouter quelques instructions supplémentaires pour s'assurer du bon déroulement de notre script.

Apportons quelques précisions sur notre fichier. Nous utiliserons le mot clé *BEGIN* pour démarrer notre script et le mot clé *COMMIT* pour le terminer. Cela permettra de lancer notre *transaction*, notre base de données simulera toutes les instructions se trouvant entre ces mots clés.
De plus nous souhaiterons pour éviter toutes erreurs, supprimer nos tables si elles existent avant de lancer la création de nos tables.
Et pour finir il nous suffira d'ajouter l'instruction *CREATE TABLE* devant chacune de nos tables pour pouvoir les créer.
Suite au bon déroulement du script, nous avons fait le choix d'éditer des fichiers de seeding, nous permettant d'alimenter notre base de données avec quelques enregistrements pour d'éventuels tests de manipulation à venir.

Maintenant que nos tables sont finalement créées et alimentées avec des informations, l'étape suivante consiste à mettre en relation notre environnement d'exécution et notre *RDBMS* pour pouvoir écrire nos requêtes *SQL* directement depuis notre back-end.

Pour ce faire, nous avons besoin avant tout, d'installer la dépendance *pg* sur notre projet back. Une fois fait, nous sommes passé à la configuration de notre client pour qu'il puisse être capable de se mettre en relation avec la bonne base de données.
Etant des données confidentielles, nous stockerons nos informations de compte dans des variables d'environnements. Nous aurons donc besoin de la dépendance *dotenv* et d'éditer un fichier `.env` directement à la racine de notre projet, voilà à quoi pourrait ressembler notre fichier `.env` avec les informations tirées de notre exemple :

```
# Nom de l'utilisateur de notre BD
PGUSER=cinedelices
# Domaine : localhost étant en développement
PGHOST=localhost
# Mot de passe de notre compte utilisateur
PGPASSWORD=motDePasseDeNotreBD
# Nom de notre base de données
PGDATABASE=cinedelicesdb
# Port d'écoute de posgreSQL : 5432 par défaut
PGPORT=5432
# Port d'écoute de notre serveur HTTP back
PORT=3000
# Chaîne de caractère ajouter pour la signature de notre token JWT
SECRET=
```

Pour finir notre configuration, il ne nous restera plus qu'à initialiser notre fichier `index.js` (Voir Annexes page 25). Retenons pour l'instant que seule la configuration de notre serveur http est importante à cette étape de développement. Notre équipe ayant opté pour l'utilisation du framework *Express* pour la gestion de notre serveur, l'import de nos variables d'environnement et de nos dépendances *Express* sont donc nécessaires.

d. Mettre en relation notre serveur back avec notre base de données

1. Connection avec l'aide de *pg*

Nous possédons dorénavant une base de données alimentée grâce à notre seeding, notre objectif est maintenant de pouvoir faire communiquer notre serveur back-end avec notre base de données *PostgreSQL*. Pour cela, nous avons mis en place via l'architecture MVC (Modèles Vues Contrôleurs) des modèles (classe) qui assureront le lien avec une table définie. Pour la création de nos modèles nous utiliserons le *design pattern Active Record*, le but étant de lier chaque méthode à une action du *CRUD* (Create Read Update Delete). Ce qui nous permettra, lors de chaque appel de celle-ci, d'effectuer une action sur nos tables et enregistrements. Nous associerons ensuite ces méthodes à des contrôleurs pour pouvoir paramétrer les routes de notre serveur back-end.

Mais en premier lieu, pour pouvoir réaliser des requêtes SQL depuis notre back, il nous faut terminer la configuration de notre client *pg*. Ce dernier mettant à disposition un objet prévu à cet effet, qui aura pour intention de nous connecter à notre BD et via ce même objet de pouvoir lancer nos requêtes SQL.

Création de notre fichier database.js :

```
import pg from "pg";
// import * as dotenv from "dotenv";
import dotenv from "dotenv";

// Lecture de nos variables d'environnement
dotenv.config();

// Création de notre objet client
const client = new pg.Client();

// Connexion à notre DB
client.connect();

// Export pour pouvoir utiliser les méthodes associées
export default client;
```

Dans le code ci-dessus, nous éditons un fichier qui aura pour fonction de faire communiquer par l'intermédiaire de notre dépendance *pg* notre serveur et notre BD. La connexion s'effectuera grâce à la lecture de nos variables d'environnement initialisées plus tôt.

2. Interaction avec notre base de données : création de modèle

Nous utiliserons notre modèle *User.js* pour expliquer la relation avec notre table du même nom (Voir Annexes page 26 et 27).
Nous commencerons à initialiser une nouvelle classe *JS*, pour que ce soit plus explicite, nous lui donnerons le même nom que la table avec laquelle elle sera mise en relation. Pour la création et la construction de nos objets de type *User* nous passerons par des propriétés dites privées. Ce qui empêchera toutes éventuelles modifications en dehors de cette classe.

Si nous souhaitons manipuler les valeurs nous passerons donc par l'initialisation de *getteurs* pour pouvoir récupérer nos valeurs et de *setteurs* pour pouvoir les modifier. C'est dans nos *setteurs* que nous pourrons réaliser les premières vérifications de conformité.

Prenons l'exemple de notre propriété *email* :

Extrait de code User.js :

```
import validator from "validator";

class User {
    // initialisation des propriétés privées
    #id;
    #username;
    #email;
    #password;
    #is_active;

    constructor(config) {
        this.id = config.id;
        this.username = config.username;
        this.email = config.email;
        this.password = config.password;
        this.is_active = config.is_active;
    }
    // Mise en place du getteur permettant d'afficher notre valeur
    get email() {
        return this.#email;
    }
    // Mise en place d'une vérification grâce à la dépendance validator
    set email(value) {
        if (!validator.isEmail(value)) {
            throw new Error("Le format d'email n'est pas valide");
        }
        this.#email = value;
    }
```

Cet extrait contient l'initialisation des propriétés privées de notre modèle, pour la lecture de ces valeurs, il nous faut passer par son *getteur* associé, ensuite nous pouvons noter l'utilisation d'une autre dépendance : *validator* (*https://www.npmjs.com/package/validator/v/10.10.0*), grâce à sa méthode *isEmail()* nous pouvons donc vérifier le bon format attendu pour une adresse mail, nous conditionnerons une erreur pour les cas contraires.

C'est dans notre modèle que nous mettrons également en place nos méthodes du CRUD, grâce à celles-ci, nous pourrons donc, par l'intermédiaire de requêtes SQL et réalisées avec l'utilisation de notre client *pg*, interagir avec notre table *user* de notre base de données.

```
    // mise en place du CRUD via le design pattern active record
    async create() {
        const result = await client.query(`INSERT INTO "user"
            (username, email, password)
            VALUES ($1, $2, $3)`, [
            this.#username,
            this.#email,
            this.#password
        ])
        return result.rowCount
    }
    // affiche l'ensemble des enregistrements de la table user
    static async findAll() {
        const result = await client.query(`SELECT * FROM "user";`)
        return result.rows
    }
```

Soulignons plusieurs points importants dans cet extrait de code.
Tout d'abord, nous réalisons notre requête SQL dans une fonction asynchrone, étant une opération qui prend du temps (quelques millisecondes) il est primordiale que le reste de notre script puisse continuer à s'exécuter, de plus nous remarquerons également l'utilisation de notre objet *client* créé avec notre dépendance *pg* qui fait bien le lien avec notre DB (étape précédente de configuration).

On utilisera la méthode *query* pour exécuter notre requête. Dans le cas de la méthode *create()* de notre objet *User* il s'agit bien de l'instruction *INSERT* qui nous permet d'ajouter un nouvel enregistrement.

Nous pouvons observer que l'on fait bien appel à nos valeurs de notre objet instancié pour construire notre requête. Ce qui veut dire que si un objet *User* à bien était créé, nous ajouterons bien un nouvel enregistrement avec les valeurs associées à ses propriétés.
Autre point très important, soulignons l'utilisation de variables intermédiaires (« $1, $2,… »). Il est indispensable, pour des raisons de sécurité de ne pas concaténer directement nos valeurs à nos requêtes SQL, les raisons sont simples, éviter l'injection de code malicieux (injections SQL) pour ce faire, *pg* met à notre disposition ces fameuses variables intermédiaires que s'occuperons d'assainir nos valeurs, permettant de sécuriser l'appel de notre méthode.

Observons maintenant la méthode *findAll()* de notre class *User*. Notons l'utilisation du mot clé *static*, le but de cette méthode est de réaliser une requête permettant d'afficher l'ensemble de nos enregistrements contenu dans notre table *user*. N'ayant aucunement besoin de valeur pour être effectué, nous passerons donc par une méthode dite de *class* pour faire appel à cette dernière (pas besoin de créer un nouvel objet de type *User* pour faire appel à cette méthode).

3. Maintenir un code stable : test unitaire

Depuis le début de notre projet, nous avons pris l'habitude de décomposer notre code via la création de modules, de composants ou même de *hook* personnalisé avec *React*, mais comme nous avons pu le constater avec notre modèle, il peut arriver parfois que notre code devienne très vite volumineux, ce qui peut compliquer sa maintenabilité, notamment si, il est amené à évoluer. De plus, il pourrait être regrettable si une nouvelle fonctionnalité venait à venir « casser » notre code déjà existant.

Pour pouvoir sécuriser ce genre de cas, il peut être intéressant de mettre en place des tests. Nous nous sommes tournés vers le framework *Vitest* qui propose plusieurs fonctionnalités, aidant à mettre en place facilement des tests unitaires (*https://vitest.dev/guide/*). C'est ce que nous allons voir avec notre modèle *User* et sa méthode de création d'un nouvel enregistrement.

Comme nous avons pu le voir précédemment, cette méthode réalise plusieurs actions comme le fait de récupérer les données de notre utilisateur, d'envoyer une requête SQL à notre BD et de retourner le nombre de lignes créées dans la table *user*.
Dans le cas ou nous souhaiterions pouvoir tester si notre méthode fonctionne sans avoir à modifier notre base de données, nous pourrons mettre en place un test unitaire.

Après avoir installé notre dépendance et créé notre script dans notre fichier `package.json`, nous avons dans un fichier `user.test.js` initialisé notre test, celui-ci nous permettra de vérifier que nous avons bien envoyé une requête *SQL INSERT INTO* et que nous récupérions également « 1 » comme valeur de retour, étant donné que nous testons l'enregistrement d'un seul utilisateur. Ci-dessous donc, vous verrez le code de notre test unitaire et le résultat en cas de réussite sur notre terminal après avoir lancé notre script d'automatisation.

Code du test unitaire :

```
// Import des modules de vitest permettant de créer notre test
import { describe, it, expect, vi } from "vitest";
// Import de notre modèle User
import User from "../src/app/models/User";
// Import de notre client pg
import client from "../src/app/database.js";

/**
 * Nous permet de simuler l'appel de notre module
 * @link https://vitest.dev/api/vi.html#vi-mock
 * Ce qui permet de pouvoir tester le résultat de notre requête SQL
 * Sans modifier notre véritable base de données
 */
vi.mock("../src/app/database.js", () => ({
    // default : Doit être indiqué lorsque l'on simule un module exporté par défaut
    // query : représente notre fonction simulé
    default: { query: vi.fn() },
}));

// Annotation permettant de décrire le test englobé
describe("Test notre méthode de création de notre modèle User", () => {
    it("Devrait enregistrer un nouvel utilisateur en BD", async () => {
        // Valeur simulé attendu en cas de réussite
        client.query.mockResolvedValueOnce({ rowCount: 1 });
        // On instancie un nouvel utilisateur
        const user = new User({
            username: "Toto",
            email: "toto@mail.com",
            password: "superMdpSecure@1234",
        });
        // On souhaite tester la méthode create de notre modèle
        const result = await user.create();
        // On vérifie que c'est bien la bonne requête envoyé
        expect(client.query).toHaveBeenCalledWith(
            expect.stringContaining("INSERT INTO"),
            [user.username, user.email, user.password]
        );
        // On vérifie qu'on récupère "1", étant la valeur
        expect(result).toBe(1);
    });
});
```

Résultat du test unitaire en cas de réussite :

```
RERUN tests Vitest/user.test.js x14

✓ tests Vitest/user.test.js (1 test) 24ms
  ✓ Test notre méthode de création de notre modèle User (1)
    ✓ Devrait enregistrer un nouvel utilisateur en BD 18ms

 Test Files  1 passed (1)
      Tests  1 passed (1)
   Start at  14:41:12
   Duration  2.65s

PASS Waiting for file changes...
     press h to show help, press q to quit
```

Dans notre modèle, rappelons-nous que, nous avons utilisé la dépendance *validator* pour vérifier la conformité de l'adresse email récupérée.
Vérifions donc le résultat de notre test si la valeur de notre email n'est pas valide, retirons l'arobase par exemple.

```
RERUN tests Vitest/user.test.js x15

❯ tests Vitest/user.test.js (1 test | 1 failed) 23ms
  ❯ Test notre méthode de création de notre modèle User (1)
    × Devrait enregistrer un nouvel utilisateur en BD 20ms

                                Failed Tests 1

FAIL tests Vitest/user.test.js > Test notre méthode de création de notre modèle User > Devrait enregistrer un nouvel utilisateur en BD
Error: Le format d'email n'est pas valide
 ❯ User.set email [as email] src/app/models/User.js:56:19
    54|     set email(value) {
    55|         if (!validator.isEmail(value)) {
    56|             throw new Error("Le format d'email n'est pas valide");
      |                   ^
    57|         }
    58|         this.#email = value;
 ❯ new User src/app/models/User.js:15:20
 ❯ tests Vitest/user.test.js:26:22

 Test Files  1 failed (1)
      Tests  1 failed (1)
   Start at  15:00:42
   Duration  1.18s

FAIL Tests failed. Watching for file changes...
     press h to show help, press q to quit
```

Notre test est donc bien fonctionnel, il nous permet de tester facilement notre code tout en restant compréhensible pour nous permettre d'identifier rapidement le souci.

4. Fournir une réponse http adéquate : création de contrôleur

Notre modèle est maintenant créé (Voir Annexes page 26 et 27) et testé, c'est lui qui réalisera grâce à ces méthodes associées aux requêtes SQL correspondantes, le lien avec notre BD. Suite à cette étape, nous nous occuperons donc de créer les requêtes http permettant de faire appel à chacune de nos fonctions. Pour cela nous initialiserons un contrôleur prévu à cet effet (Voir Annexes page 27 à 29).
Attardons-nous un peu sur quelques méthodes de ce contrôleur.

Méthode appelé pour la création d'un nouvel utilisateur :

```
    // Route POST /api/users
    createNewUser: async (req, res) => {
        try {
            // On replace par notre méthode findByEmail, plus approprié
            const userByEmail = await User.findByEmail(req.body.email)
            if (userByEmail) {
                return res.status(409).json({ message: "Email déjà utilisé pour la création d'un compte" })
            }
            /**
             *  On définit nos options qui représenterons nos critères de validation de mot de passe
             *  Au moins 12 caractères et 4 types différents
             *  Pour plus d'informations :
             *  @link https://www.cnil.fr/fr/mots-de-passe
             */
            const options = { minLength: 12, minLowercase: 1, minUppercase: 1, minNumbers: 1, minSymbols: 1 }
            if (!validator.isStrongPassword(req.body.password, options)) {
                return res.status(409).json({ message: "Le mot de passe doit comporter au moins 12 caractères et au moins 1 majuscule,
1 minuscule, 1 chiffre et 1 caractère spécial" })
            }
            // Si mot de passe valide on le hache
            const hash = await bcrypt.hash(req.body.password, 10);
            const newUser = new User({
                username: req.body.username,
                password: hash,
                email: req.body.email,
                created_at: new Date()
            })
            await newUser.create()
            return res.status(201).json("Nouvel utilisateur enregistré avec succès")
        }
        catch (error) {
            return res.status(500).json({ message: 'Erreur serveur' })
        }
    },
```

Si nous regardons un peu plus en détail ce bout de code, nous pouvons remarquer que nous faisons appel à plusieurs méthodes de notre classe *User*. Gardons en mémoire qu'il s'agit d'opérations qui ne sont pas instantanées, on se doit d'utiliser le mot clé *async*. Dans le corps de notre fonction, nous réaliserons une autre série de tests conditionnés. Pour cela, nous engloberons le tout dans une instruction *try and catch*.

Nous récupérons tout d'abord l'enregistrement grâce à la valeur récupérée (dans notre formulaire), du coup si un utilisateur est déjà enregistré sous cette adresse email, alors nous retournons en réponse dans le corps de notre requête http, la chaîne de caractère «Email déjà utilisé pour la création d'un compte » accompagné du code status 409.

Conformément aux préconisations du CNIL (Commission National de l'Informatique et des Libertées) il est recommandé qu'un mot de passe doit être composé d'au moins 12 caractères, d'une minuscule, d'une majuscule, d'un chiffre et d'un symbole pour être considéré un minimum sécurisé.
Cette vérification peut être réalisée une fois de plus avec *validator*. Si le mot de passe est conforme alors pour plus de sécurité, il sera « haché », pour cela une autre dépendance est utilisée, *bcrypt*. Nous pourrons ensuite instancier un nouvel objet *User* et appliquerons la méthode *create()* de notre modèle pour ajouter un nouvel enregistrement en base de données. On pourra conclure avec le fait que si, une erreur survient, elle sera « attrapée » dans notre *catch*.

Voila ce qu'il en est pour l'analyse d'une méthode de notre contrôleur, permettant de conditionner une réponse http en fonction de son résultat.

5. Configurer les routes de nos réponses : création d'un router

Il nous restera une dernière étape, qui est d'affecter cette méthode de contrôleur à une route. Pour cela nous passerons par notre fichier *rooter.js.*

```
// Import de notre dépendance Express, utile pour paramétrer notre router
import express from "express";
// Import de notre contrôleur userController, qui s'occupera de la gestion de nos réponse http
import userController from "./controllers/userController.js"

// Création de notre routeur express
const router = express.Router();

// Configuration de nos routes faisant appel à ce même contrôleur
// Créer un utilisateur
router.post("/api/users/", userController.createNewUser);
// Récupérer tous les utilisateurs
router.get("/api/users", userController.getAllUser);
// Récupérer un utilisateur par ID
router.get("/api/users/id/:id", userController.getUserById);
// Récupérer un utilisateur par email
router.get("/api/users/email/:email", userController.getUserByEmail);
// Récupérer l'utilisateur authentifié
router.get("/api/users/me", authenticateUser, userController.getMe);
// Modifier un utilisateur
router.patch("/api/users/:id", userController.updateUser);
// Supprimer un utilisateur
router.delete("/api/users/:id", userController.delete);
// Récupérer la note moyenne des publications d'un utilisateur
router.get('/api/users/:id/average-rating', userController.getAverageRatingByUser);
```

Nous retrouvons bien la route paramétrée « /api/users/ » en méthode POST, qui fait donc appel à notre méthode de contrôleurs permettant de pouvoir créer un nouvel enregistrement sur notre table *user*.
On n'oubliera pas d'importer notre router dans notre fichier `index.js`. (Voir Annexes page 25).

En suivant ce procédé pour chacune de nos tables, nous avons donc été capables de paramétrer les routes de notre serveur http back, pour qu'elle puisse appliquer, nos différentes actions du CRUD dans un environnement sécurisé.

e. Communication entre le back-end de notre projet et son front-end

Après avoir été dans la capacité de créer, des composants *React*, permettant de réaliser le rendu et d'interagir avec nos utilisateurs de façon dynamique depuis notre front et suite à la conception d'un environnement back sécurisé, capable de communiquer avec une base de données, la dernière étape fut donc de pouvoir faire dialoguer les deux environnements de notre projet ensemble.

Reprenons notre formulaire d'inscription et plus particulièrement son *hook* personnalisé. Si nous vérifions cette partie du code :

Handler permettant la gestion lors de la soumission du formulaire de connexion :

```
// Création de compte
try {
    // On vérifie la correspondance de nos mots de passe
    if (password === confirmPassword) {
        /**
         * On créé un objet avec les valeurs récupérées de nos variables d'état
         * C'est cet objet que nous passerons dans le corps de notre requêtes
         */
        const payload = {
            username,
            email,
            password
        }
        // On réalise notre requête sur la route associée côté back
        const response = await fetch("http://localhost:3000/api/users/", {
            // Méthode
            method: "POST",
            // Header de la requête
            headers: {
                "Content-type": "application/json",
                "charset": "utf-8"
            },
            // Body de la requête
            body: JSON.stringify(payload)
        })
        /**
         * Nous souhaitons récupérer en cas d'erreur le message associé
         * Par exemple si un email est déjà utilisé pour la création d'un compte
         * Impossibilité de créer un nouvel enregistrement donc un nouveau compte
         *
         * Utilisation de la propriété en lecture seule ``.ok``
         * @link https://developer.mozilla.org/fr/docs/Web/API/Response/ok
         * Si le status code n'est pas compris entre 200 et 299
         * Alors renvoie false
         */
        if (!response.ok) {
            // On récupère notre réponse json
            const error = await response.json()
            // On jette l'erreur associée
            throw new Error(error.message)
        } else {
            navigate("/login")
        }
    } else {
        throw new Error("Les deux mots de passes ne correspondent pas.")
    }
    // On réinitialise le message si un est déjà affiché
    setMessage('')
} catch (error) {
    setMessage(error.message)
}
```

Si on analyse un peu le code, on constate que nous récupérons de l'objet *payload,* les informations communiquées par notre utilisateur par le biais du formulaire.
Ces données sont envoyées sous forme de requête vers notre back à l'url défini par notre router back. On y indique sous forme d'objet, la méthode, ici POST étant donné qu'il s'agit d'une nouvelle entrée, des éléments du headers de notre requête, ici le type de données envoyées ainsi que le type d'encodage et pour finir, dans le corps de notre requête, les informations récupérées de notre formulaire en chaîne JSON.

Mais comme notre environnement front ne partage pas le même domaine que celui de notre back, nous serons donc dans l'impossibilité actuellement de faire communiquer nos deux environnements. C'est tout à fait normal, puisque pour des raisons de sécurité, la communication *cross-origin* est par défaut limitée au même domaine. Pour cela il nous faut donc paramétrer nos *CORS (Cross-Origin Resource Sharing)* depuis notre back. (Voir Annexes page 25). La dépendance *cors (https://www.npmjs.com/package/cors)* permet un paramétrage de celles-ci, il nous suffit donc de faire appel au middleware avant l'appel de nos routes tout en prenant soin de régler les *options* vers le domaine de notre front :

Extrait du fichier index.js côté back :

```js
// Import de nos dépendances
// Import de notre Framework express
import express from "express";
// Import de notre routeur
import router from "./src/app/router.js";
// Import du module cors pour permettre les echanges entre front et back
import cors from "cors";

// Création de notre application express
const app = express();

// Initialisation des options des cors
const optionsCORS = {
    origin: "http://localhost:1234"
}

// Middleware permettant la gestion des CORS
app.use(cors(optionsCORS))

// Appel du router de notre application
app.use(router);
```

Ce qui nous permet d'obtenir une communication entre les deux environnements de notre projet.

Exemple de rendu final de notre formulaire de connexion :

Nous avons délibérément indiqué un mauvais mot de passe dans le but d'afficher le message d'erreur récupéré depuis la réponse à notre requête.
Le message se veut être générique pour éviter de donner des précisions sur la donnée erronée.

Conclusion

C'est ainsi que nous pourrons conclure la présentation de notre projet, *Ciné-Délices.* Il nous aura permis de mettre en pratique l'ensemble des compétences abordées tout au long de notre formation.

A travers ses différentes étapes de conception, allant de la réalisation de la documentation conceptuelle et contractuelle, à la création d'une interface fonctionnelle et accessible, jusqu'à la mise en place d'un serveur assurant un traitement sécurisé des données, ce projet s'est révélé être une véritable mise en situation, proche de ce que l'on pourrait rencontrer en situation professionnelle.

De par sa proximité avec un projet réel, il nous a confrontés aux problématiques que tout développeur peut rencontrer au cours de sa carrière. Tant sur le plan technique que méthodologique, il nous a surtout appris à cerner les attentes d'un client, à répondre à un besoin concret, et à collaborer efficacement en équipe.

Enfin, cette expérience nous a permis de développer une vision commune du travail collectif, en avançant de manière coordonnée et optimale vers un objectif partagé.

Ressources

Documentation :

- https://developer.mozilla.org/fr/
- https://www.w3.org/WAI/tutorials/
- https://www.npmjs.com/
- https://sass-lang.com/documentation/
- https://fr.legacy.reactjs.org/
- https://reactrouter.com/
- https://ejs.co/
- https://fr.javascript.info/
- https://www.postgresql.org/
- https://vitest.dev/
- https://caninclude.onrender.com/
- https://checklists.opquast.com/fr/assurance-qualite-web/
- https://accessibilite.numerique.gouv.fr/
- https://www.cnil.fr/fr/reglement-europeen-protection-donnees
- https://entreprendre.service-public.gouv.fr/vosdroits/F31228

Outils :

- https://www.mocodo.net/
- https://validator.w3.org/
- https://trello.com/