

|  |
| --- |
| MEMO CARD  2024 |
|  |
| 25 juillet  ESGI 3 IW  Créé par : GAILLET Mathys GitHub : https://github.com/ThysmaJS/memoCard |

# Sommaire

[Sommaire 2](#_Toc172802601)

[Présentation 3](#_Toc172802602)

[Inspiration 4](#_Toc172802603)

[Base de données 5](#_Toc172802604)

[Fonctionnalités / A. Authentification 7](#_Toc172802605)

[Fonctionnalités / B. Gestion CRUD 10](#_Toc172802606)

[Fonctionnalités / C. Révision 16](#_Toc172802607)

[Fonctionnalités / D. PWA 21](#_Toc172802608)

[Fonctionnalités / E. PublicPrivate 23](#_Toc172802609)

[Fonctionnalités / F. FactoriesSeeders 24](#_Toc172802610)

[Fonctionnalités / G. Promesses 26](#_Toc172802611)

[Fonctionnalités / H. Composants 28](#_Toc172802612)

[Annexe 30](#_Toc172802613)

1. Présentation

MemoCard est une application de révision conçue pour aider les utilisateurs à gérer et suivre leurs sessions de révision sur divers sujets en utilisant la technique de la répétition espacée, permettant une expérience de révision organisée et efficace.

L’application est entièrement sous Docker et elle est développé en VueJS (3) pour le Frontend et Laravel (10) pour le Backend.

Une image contenant motif, art, Symétrie, conception

Description générée automatiquementUne image contenant Graphique, graphisme, texte, clipart

Description générée automatiquementUne image contenant Graphique, capture d’écran, Caractère coloré, graphisme

Description générée automatiquement

1. Inspiration

Quizlet

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

Description générée automatiquement

1. Base de données

Dans l’objectif de la réalisation de mon application et en fonction du cahier des charges fourni, je suis arrivé à cette conclusion pour ma base de données.

Une image contenant texte, capture d’écran, nombre, diagramme

Description générée automatiquement

On peut voir que ma BDD contient 5 tables (Review, User, Theme, Category et Card).

Les relations misent en place sont les suivantes :

* ****User**** et ****Category**** : Une relation one-to-many où un utilisateur peut créer plusieurs catégories (user\_id dans la table Category).
* ****User**** et ****Theme**** : Une relation one-to-many où un utilisateur peut créer plusieurs thèmes (user\_id dans la table Theme).
* ****Category**** et ****Theme**** : Une relation one-to-many où une catégorie peut contenir plusieurs thèmes (category\_id dans la table Theme).
* ****Theme**** et ****Card**** : Une relation one-to-many où un thème peut contenir plusieurs cartes (theme\_id dans la table Card).
* ****User**** et ****Review**** : Une relation one-to-many où un utilisateur peut effectuer plusieurs révisions (user\_id dans la table Review).
* ****Theme**** et ****Review**** : Une relation one-to-many où un thème peut être révisé plusieurs fois (theme\_id dans la table Review).

Cette structure de base de données permet de gérer plusieurs fonctionnalités clés pour une application de révision de cartes mémoire :

1. Gestion des Utilisateurs : Création, mise à jour, et suppression d'utilisateurs avec des informations personnelles et des préférences de révision.
2. Gestion des Catégories : Les utilisateurs peuvent créer et organiser des thèmes dans des catégories.
3. Gestion des Thèmes : Les utilisateurs peuvent créer des thèmes, les rendre publics ou privés, et les organiser par catégorie.
4. Gestion des Cartes : Les utilisateurs peuvent créer des cartes à l'intérieur des thèmes pour les réviser.
5. Révisions : Suivi des révisions effectuées par les utilisateurs sur les différents thèmes, incluant les niveaux de révision et les dates de révision.
6. Fonctionnalités / A. Authentification

Ici nous allons voir le fonctionnement de l’inscription et de la connexion à l’application

#### Backend

##### Contrôleur Utilisateur (UserController)

**Créer un Utilisateur 🡪(createUser)**

* ****Description**** : Cette méthode gère la création d'un nouvel utilisateur.
* ****Validation**** :
  + name : requis
  + email : requis, doit être un email valide et unique dans la table users
  + password : requis
* ****Traitement**** :
  + Si la validation échoue, une réponse JSON avec les erreurs de validation est renvoyée.
  + Sinon, un nouvel utilisateur est créé avec un mot de passe haché.
  + Un token d'API est généré et renvoyé dans la réponse JSON.
* ****Réponse**** :
  + Succès : status (booléen), message (string), token (string)
  + Erreur : status (booléen), message (string)

**Connecter un Utilisateur 🡪 (loginUser)**

* ****Description**** : Cette méthode gère la connexion des utilisateurs.
* ****Validation**** :
  + email : requis, doit être un email valide
  + password : requis
* ****Traitement**** :
  + Si la validation échoue, une réponse JSON avec les erreurs de validation est renvoyée.
  + Si les informations d'identification sont incorrectes, une réponse JSON avec un message d'erreur est renvoyée.
  + Sinon, un token d'API est généré et renvoyé dans la réponse JSON.
* ****Réponse**** :
  + Succès : status (booléen), message (string), token (string)
  + Erreur : status (booléen), message (string)

##### Modèle Utilisateur (User)

Le modèle User utilise le trait HasApiTokens pour gérer les tokens d'API avec Laravel Sanctum.

* **Champs remplaçables (**$fillable**)** :
  + Name
  + email
  + password
* **Champs cachés (**$hidden**)** :
  + password
  + remember\_token
* **Casts (**$casts**)** :
  + email\_verified\_at : datetime
  + password : hashed

#### Routes

* **Route d'inscription** : POST /auth/register
* **Route de connexion** : POST /auth/login

#### Frontend

##### Vue Login

Cette vue permet à l'utilisateur de se connecter.

* ****Template**** : Formulaire de connexion avec champs pour l'email et le mot de passe.
* ****Méthode**** login : Envoie une requête POST à l'API de connexion. Si la connexion réussit, un token est stocké dans le localStorage et l'utilisateur est redirigé vers la page d'accueil.

##### Vue Register

Cette vue permet à l'utilisateur de s'inscrire.

* ****Template**** : Formulaire d'inscription avec champs pour le nom, l'email et le mot de passe.
* ****Méthode**** saveData : Envoie une requête POST à l'API d'inscription. Si l'inscription réussit, un message de succès est affiché.

##### Store d'Authentification (auth.js)

Le store gère l'état d'authentification de l'utilisateur.

* ****État**** : isAuthenticated (booléen)
* ****Actions**** :
  + initialize : Initialise l'état d'authentification à partir du localStorage.
  + login : Met à jour l'état d'authentification et stocke l'information dans le localStorage.
  + logout : Réinitialise l'état d'authentification et supprime l'information du localStorage.

1. Fonctionnalités / B. Gestion CRUD

Cette section de la documentation décrit les fonctionnalités CRUD (Create, Read, Update, Delete) pour les thèmes, catégories et cartes dans l’application.

L’exemple choisi ici sera les catégories, mais c’est le exactement le même principe pour les thèmes et les catégories.

#### Backend

##### Contrôleur Catégories (CategoryController)

##### Récupérer les catégories **🡪 (**allCategories**)**

* ****Description**** : Cette méthode récupère toutes les catégories.
* ****Accessibilité**** : Accessible publiquement.
* ****Traitement**** :
  + Récupère toutes les catégories de la base de données et les renvoie sous forme de réponse JSON.
* ****Réponse**** :
  + Succès : Liste des catégories (array d'objets JSON).

##### Récupérer les catégories en fonction du User **🡪 (**userCategories**)**

* ****Description**** : Cette méthode récupère les catégories de l'utilisateur connecté.
* ****Accessibilité**** : Nécessite l'authentification.
* ****Traitement**** :
  + Récupère l'utilisateur connecté via l'authentification.
  + Récupère les catégories associées à cet utilisateur.
  + Renvoie les catégories sous forme de réponse JSON.
* ****Réponse**** :
  + Succès : Liste des catégories de l'utilisateur (array d'objets JSON).

##### Création d’une catégorie en fonction du User **🡪 (**store**)**

* ****Description**** : Cette méthode crée une nouvelle catégorie associée à l'utilisateur connecté.
* ****Accessibilité**** : Nécessite l'authentification.
* ****Validation**** :
  + name: requis, string, maximum 255 caractères.
  + Description : optionnel, string.
* ****Traitement**** :
  + Si la validation échoue, une réponse JSON avec les erreurs de validation est renvoyée.
  + Sinon, une nouvelle catégorie est créée et associée à l'utilisateur connecté.
  + La catégorie est sauvegardée dans la base de données.
  + La nouvelle catégorie est renvoyée sous forme de réponse JSON.
* ****Réponse**** :
  + Succès : La nouvelle catégorie (objet JSON).
  + Erreur : status (booléen), message (string), errors (objet JSON contenant les erreurs de validation).

##### Affichage d’une catégorie choisie **🡪 (**show**)**

* ****Description**** : Cette méthode affiche une catégorie spécifique.
* ****Accessibilité**** : Nécessite l'authentification et l'autorisation.
* ****Traitement**** :
  + Vérifie si l'utilisateur a le droit de voir la catégorie spécifique.
  + Renvoie la catégorie sous forme de réponse JSON si l'autorisation est accordée.
  + Si l'autorisation échoue, une réponse JSON avec un message d'erreur est renvoyée.
* ****Réponse**** :
  + Succès : La catégorie (objet JSON).
  + Erreur : status (booléen), message (string).

##### Mise à jour d’une catégorie choisie **🡪 (**update**)**

* ****Description**** : Cette méthode met à jour une catégorie spécifique.
* ****Accessibilité**** : Nécessite l'authentification et l'autorisation.
* ****Validation**** :
  + name: requis, string, maximum 255 caractères.
  + Description: optionnel, string.
* ****Traitement**** :
  + Vérifie si l'utilisateur a le droit de mettre à jour la catégorie spécifique.
  + Si la validation échoue, une réponse JSON avec les erreurs de validation est renvoyée.
  + Sinon, la catégorie est mise à jour avec les nouvelles données.
  + La catégorie mise à jour est renvoyée sous forme de réponse JSON.
  + Si l'autorisation échoue, une réponse JSON avec un message d'erreur est renvoyée.
* ****Réponse**** :
  + Succès : La catégorie mise à jour (objet JSON).
  + Erreur : status (booléen), message (string), errors (objet JSON contenant les erreurs de validation).

##### Suppression d’une catégorie choisie **🡪 (**destroy**)**

* ****Description**** : Cette méthode supprime une catégorie spécifique.
* ****Accessibilité**** : Nécessite l'authentification et l'autorisation.
* ****Traitement**** :
  + Vérifie si l'utilisateur a le droit de supprimer la catégorie spécifique.
  + Si l'autorisation est accordée, la catégorie est supprimée de la base de données.
  + Renvoie une réponse JSON avec un statut de succès.
  + Si l'autorisation échoue, une réponse JSON avec un message d'erreur est renvoyée.
* ****Réponse**** :
  + Succès : null, avec un statut HTTP 204 (No Content).
  + Erreur : status (booléen), message (string).

1. Frontend

* ****Description**** : Cette vue permet à l'utilisateur de gérer ses catégories, thèmes et cartes.
* ****Structure**** :
  + Affiche les informations de profil de l'utilisateur (nom, email).
  + Contient des onglets pour basculer entre les catégories, thèmes et cartes.

**Onglet Catégories**

1. **Récupérer les Catégories de l'Utilisateur**
   * ****Méthode**** : fetchCategories
   * ****Description**** : Récupère toutes les catégories associées à l'utilisateur connecté.
   * ****Traitement**** :
     + Envoie une requête GET à l'API via axios pour récupérer les catégories.
     + Met à jour l'état categories dans le store avec les données reçues.
   * ****Réponse**** :
     + Succès : Met à jour l'état categories avec la liste des catégories de l'utilisateur.
2. **Créer une Nouvelle Catégorie**
   * ****Méthode**** : saveCategory
   * ****Description**** : Crée une nouvelle catégorie pour l'utilisateur connecté.
   * ****Traitement**** :
     + Envoie une requête POST à l'API avec les détails de la nouvelle catégorie.
     + Ajoute la nouvelle catégorie à l'état categories dans le store si la création est réussie.
     + Réinitialise le formulaire de création de catégorie.
   * ****Réponse**** :
     + Succès : Ajoute la nouvelle catégorie à l'état categories et réinitialise le formulaire.
     + Erreur : Logue l'erreur et affiche un message d'erreur dans la console.
3. **Modifier une Catégorie**
   * ****Méthode**** : editCategory
   * ****Description**** : Pré-remplit le formulaire avec les données de la catégorie à modifier.
   * ****Traitement**** :
     + Récupère les détails de la catégorie sélectionnée et les attribue à currentCategory.
     + Affiche le formulaire de modification de catégorie.
4. **Mettre à Jour une Catégorie**
   * ****Méthode**** : updateCategory
   * ****Description**** : Met à jour une catégorie spécifique pour l'utilisateur connecté.
   * ****Traitement**** :
     + Envoie une requête PUT à l'API avec les nouvelles données de la catégorie.
     + Met à jour l'état categories dans le store avec les données mises à jour.
     + Réinitialise le formulaire de modification de catégorie.
   * ****Réponse**** :
     + Succès : Met à jour la catégorie dans l'état categories et réinitialise le formulaire.
     + Erreur : Logue l'erreur et affiche un message d'erreur dans la console.
5. **Supprimer une Catégorie**
   * ****Méthode**** : deleteCategory
   * ****Description**** : Supprime une catégorie spécifique pour l'utilisateur connecté.
   * ****Traitement**** :
     + Envoie une requête DELETE à l'API avec l'identifiant de la catégorie à supprimer.
     + Supprime la catégorie de l'état categories dans le store si la suppression est réussie.
   * ****Réponse**** :
     + Succès : Supprime la catégorie de l'état categories.
     + Erreur : Logue l'erreur et affiche un message d'erreur dans la console.
6. **Annuler l'Action sur une Catégorie**
   * ****Méthode**** : cancelCategoryAction
   * ****Description**** : Réinitialise le formulaire et les états associés à la création ou modification de catégorie.
   * **Traitement** :
     + Réinitialise currentCategory à ses valeurs par défaut.
     + Cache les formulaires de création et modification de catégorie.

#### Store : Gestion des Catégories

**State**

* categories : Tableau des catégories de l'utilisateur.

**Actions**

1. **fetchCategories**
   * ****Description**** : Récupère les catégories de l'utilisateur connecté depuis l'API.
   * ****Traitement**** :
     + Envoie une requête GET à l'API pour récupérer les catégories.
     + Met à jour l'état categories avec les données reçues.
2. **addCategory**
   * ****Description**** : Ajoute une nouvelle catégorie à l'état categories.
   * ****Traitement**** :
     + Ajoute la catégorie reçue au tableau categories.
3. **updateCategory**
   * ****Description**** : Met à jour une catégorie existante dans l'état categories.
   * ****Traitement**** :
     + Remplace la catégorie existante par la catégorie mise à jour dans le tableau categories.
4. **removeCategory**
   * ****Description**** : Supprime une catégorie de l'état categories.
   * ****Traitement**** :
     + Filtre le tableau categories pour retirer la catégorie avec l'identifiant spécifié.
5. Fonctionnalités / C. Révision

Cette section de la documentation décrit les fonctionnalités du système de révision espacé.

#### Backend

**Contrôleur des Révisions (CategoryController)**

##### Récupération des révisions de l’utilisateur **🡪 **(****userReviews****)****

* + ****Description**** : Cette méthode récupère toutes les révisions associées à l'utilisateur connecté.
  + ****Accessibilité**** : Nécessite l'authentification.
  + ****Traitement**** :
    - Récupère l'ID de l'utilisateur connecté via l'authentification.
    - Récupère toutes les révisions associées à cet utilisateur, en incluant les informations du thème associé.
    - Renvoie les révisions sous forme de réponse JSON.
  + ****Réponse**** :
    - ****Succès**** : Liste des révisions de l'utilisateur (array d'objets JSON).
    - ****Erreur**** : Message d'erreur (string) en cas d'absence d'utilisateur authentifié.

**Création d’une révision 🡪 (**store**)**

* + ****Description**** : Cette méthode crée une nouvelle révision pour l'utilisateur connecté.
  + ****Accessibilité**** : Nécessite l'authentification et l'autorisation.
  + ****Validation**** :
    - review\_date : requis, date.
    - max\_level : requis, entier.
    - user\_id : requis, doit exister dans la table users.
    - theme\_id : requis, doit exister dans la table themes.
    - level : requis, entier.
  + ****Traitement**** :
    - Valide les données de la requête.
    - Si la validation échoue, renvoie une réponse JSON avec les erreurs de validation.
    - Crée une nouvelle révision avec les données validées.
    - Sauvegarde la révision dans la base de données.
  + ****Réponse**** :
    - ****Succès**** : Message de succès et l'objet révision créé (objet JSON).
    - ****Erreur**** : Message d'erreur (string) en cas de problème de création ou d'erreur de serveur.

**Terminer une révision 🡪 (**finishReview)

* + ****Description**** : Cette méthode marque une révision comme terminée et met à jour son niveau et la date de prochaine révision.
  + ****Accessibilité**** : Nécessite l'authentification et l'autorisation.
  + ****Traitement**** :
    - Récupère l'ID du thème et de l'utilisateur depuis la requête.
    - Récupère la dernière révision correspondant à ces IDs.
    - Vérifie si l'utilisateur a le droit de mettre à jour la révision.
    - Met à jour le niveau de la révision et calcule la date de prochaine révision.
  + ****Réponse**** :
    - ****Succès**** : Message de succès.
    - ****Erreur**** : Message d'erreur (string) en cas de révision non trouvée ou d'erreur de serveur.

**Vérifier l’existence de la révision 🡪 (**checkReview)

* + ****Description**** : Cette méthode vérifie si une révision existe pour un utilisateur et un thème donné.
  + ****Accessibilité**** : Nécessite l'authentification.
  + ****Traitement**** :
    - Récupère l'ID du thème et de l'utilisateur depuis la requête.
    - Cherche une révision correspondant à ces IDs.
  + ****Réponse**** :
    - ****Succès**** : Objet révision (objet JSON) ou null si aucune révision n'est trouvée.
    - ****Erreur**** : Message d'erreur (string) en cas de problème de recherche.

**Modèle révision (Review)**

* ****Attributs**** :
  + review\_date : Date de la révision.
  + max\_level : Niveau maximal de révision.
  + user\_id : ID de l'utilisateur.
  + theme\_id : ID du thème.
  + level : Niveau actuel de la révision.
* ****Méthodes**** :
  + ****calculateNextReviewDate**** : Met à jour la date de la prochaine révision en fonction du niveau actuel.
  + ****updateLevel**** : Met à jour le niveau de la révision et calcule la date de la prochaine révision.

1. Frontend

Vue de Révision

* Description : Cette vue permet à l'utilisateur de réviser les cartes d'un thème.
* Structure :
  + Affiche les cartes d'un thème une par une, permettant à l'utilisateur de cliquer pour retourner la carte et voir la réponse.
  + Boutons pour passer à la carte suivante ou terminer la révision.

Composants et Méthodes

1. Récupérer les Cartes (fetchCards)
   * Description : Récupère toutes les cartes associées au thème sélectionné pour la révision.
   * Traitement :
     + Envoie une requête GET à l'API pour récupérer les cartes du thème.
     + Met à jour l'état cards avec les données reçues.
   * Réponse :
     + Succès : Met à jour l'état cards avec la liste des cartes du thème.
     + Erreur : Logue l'erreur et affiche un message d'erreur dans la console.
2. Vérifier l'Existence d'une Révision *(*checkReview*)*
   * Description : Vérifie si une révision existe pour l'utilisateur et le thème sélectionné.
   * Traitement :
     + Envoie une requête GET à l'API pour vérifier l'existence d'une révision.
     + Met à jour l'état review avec les données reçues.
   * Réponse :
     + Succès : Met à jour l'état review avec l'objet révision existant ou null.
     + Erreur : Logue l'erreur et affiche un message d'erreur dans la console.
3. Retourner une Carte *(*flipCard*)*
   * Description : Retourne la carte actuellement affichée pour montrer la réponse.
   * Traitement :
     + Inverse l'état flipped pour montrer l'autre face de la carte.
4. Passer à la Carte Suivante *(nextCard)*
   * Description : Passe à la carte suivante dans la liste.
   * Traitement :
     + Incrémente l'index de la carte actuelle et réinitialise l'état flipped.
5. Terminer une Révision *(finishReview)*
   * Description : Marque la révision comme terminée et met à jour la base de données.
   * Traitement :
     + Envoie une requête POST à l'API pour marquer la révision comme terminée.
     + Met à jour l'état isReviewFinished pour indiquer que la révision est terminée.
   * Réponse :
     + Succès : Logue le succès de la mise à jour ou de la création de la révision.
     + Erreur : Logue l'erreur et affiche un message d'erreur dans la console.

Page des Révisions

* Description : Cette page affiche toutes les révisions de l'utilisateur avec leurs niveaux actuels et dates de prochaine révision.
* Structure :
  + Affiche une liste de révisions avec le nom du thème, le niveau actuel et la description du thème.
  + Bouton pour revoir le thème si la date de révision est arrivée.

Composants et Méthodes

1. Récupérer les Révisions de l'Utilisateur *(*fetchReviews*)*
   * Description : Récupère toutes les révisions associées à l'utilisateur connecté.
   * Traitement :
     + Envoie une requête GET à l'API pour récupérer les révisions de l'utilisateur.
     + Met à jour l'état reviews avec les données reçues.
   * Réponse :
     + Succès : Met à jour l'état reviews avec la liste des révisions de l'utilisateur.
     + Erreur : Logue l'erreur et affiche un message d'erreur dans la console.
2. Vérifier si une Révision est due *(*isReviewDue*)*
   * Description : Vérifie si la date de révision est arrivée pour afficher le bouton de révision.
   * Traitement :
     + Compare la date de révision avec la date actuelle pour déterminer si la révision est due.
3. Formater la Date *(*formatDate*)*
   * Description : Formate une date en chaîne de caractères pour affichage.
   * Traitement :
     + Convertit la date en un format lisible (JJ/MM/AAAA).
4. Fonctionnalités / D. PWA

#### Qu'est-ce qu'une PWA ?

Une **P**rogressive **W**eb **A**pp (PWA) est une application web conçue pour offrir une expérience utilisateur similaire à celle des applications natives

#### Intégration dans MemoCard

Pour transformer une application en PWA, deux éléments principaux doivent être mis en place : un fichier manifest et un service worker.

##### Manifest

Le fichier manifest contient des métadonnées sur l'application telles que le nom, les icônes, et les couleurs du thème. Ce fichier permet également à l'application d'être installée sur l'écran d'accueil de l'utilisateur.

1. ****Nom et court nom**** : Définissent le nom complet et le nom court de l'application.
2. ****Description**** : Fournit une brève description de l'application.
3. ****URL de démarrage**** : Spécifie l'URL à partir de laquelle l'application doit démarrer.
4. ****Mode d'affichage**** : Configure l'apparence de l'application lorsqu'elle est lancée (par exemple, plein écran ou en mode standalone).
5. ****Couleurs du thème et de l'arrière-plan**** : Définissent les couleurs utilisées dans l'application pour l'écran de lancement et la barre d'adresse du navigateur.
6. ****Icônes**** : Spécifient les différentes tailles d'icônes à utiliser pour l'application sur différents appareils.

##### Service Worker

Le service worker est un script qui s'exécute en arrière-plan et gère les fonctionnalités hors ligne, la mise en cache et les notifications push. Il permet de :

1. ****Mise en cache des ressources**** : Stocke les fichiers de l'application dans un cache pour permettre un chargement rapide et une utilisation hors ligne.
2. ****Gestion des requêtes réseau**** : Intercepte les requêtes réseau et peut fournir des réponses à partir du cache si le réseau est indisponible.
3. ****Activation et mise à jour**** : Gère l'installation, l'activation et la mise à jour des caches de l'application pour s'assurer que les utilisateurs disposent toujours de la version la plus récente de l'application.

En mettant en place ces deux fichiers, MemoCard peut offrir une expérience utilisateur améliorée, similaire à celle des applications natives, tout en bénéficiant des avantages des technologies web modernes.

1. Fonctionnalités / E. PublicPrivate

Les thèmes dans l'application peuvent être définis comme publics ou privés, offrant ainsi une flexibilité quant à leur accessibilité et leur visibilité pour les utilisateurs.

* ****Public**** : Un thème public est accessible à tous les utilisateurs de l'application. Tous les thèmes publics d'une catégorie peuvent être récupérés sans filtrer par utilisateur.
* ****Privé**** : Un thème privé est uniquement accessible à l'utilisateur qui l'a créé. Seul l'utilisateur propriétaire peut visualiser, modifier ou supprimer ce thème.

#### Implémentation

##### Frontend

1. **Gestion des Thèmes dans l'Interface Utilisateur** :
   * ****Création et Modification de Thèmes**** : Lors de la création ou de la modification d'un thème, une case à cocher permet à l'utilisateur de spécifier si le thème doit être public. Si la case est cochée, le thème sera accessible à tous les utilisateurs de l'application.
2. **Affichage des Thèmes** :
   * Les thèmes sont affichés dans différentes sections en fonction de leur catégorie et de leur statut (public ou privé). Les utilisateurs peuvent naviguer et gérer leurs thèmes privés, ainsi que consulter les thèmes publics créés par d'autres utilisateurs.

##### Backend

1. **Contrôleur Thème (ThemeController)** :
   * **Création et Mise à Jour des Thèmes** : Les méthodes de création (store) et de mise à jour (update) des thèmes incluent un champ public pour déterminer si le thème est public ou privé. Les validations appropriées sont appliquées pour ce champ.
   * **Récupération des Thèmes** : La méthode allThemesByCategory récupère tous les thèmes publics d'une catégorie, tandis que les méthodes comme index et allThemes récupèrent les thèmes privés associés à l'utilisateur connecté.
2. Fonctionnalités / F. FactoriesSeeders

Dans notre projet, nous utilisons des factories et des seeders pour générer des données fictives dans notre base de données. Les factories sont des modèles qui définissent comment les données fictives doivent être générées pour chaque modèle Eloquent. Les seeders insèrent ces données générées dans la base de données.

Factories

Les factories sont situées dans le répertoire database/factories. Elles permettent de définir des modèles de données fictives pour chaque modèle Eloquent de l'application. Une factory est associée à un modèle spécifique et utilise Faker pour générer des valeurs réalistes.

Fonctionnement

* ****UserFactory**** : Définit les données fictives pour les utilisateurs. Elle génère des utilisateurs avec des noms, des emails uniques, des mots de passe hashés, des niveaux de révision et des préférences de rappel quotidien.
* ****CategoryFactory**** : Définit les données fictives pour les catégories. Elle génère des catégories avec des noms et des descriptions.
* ****ThemeFactory**** : Définit les données fictives pour les thèmes. Elle génère des thèmes spécifiques à chaque catégorie avec des noms et des descriptions.
* ****CardFactory**** : Définit les données fictives pour les cartes. Elle génère des cartes avec des questions et des réponses spécifiques à chaque thème.

Seeders

Les seeders sont situés dans le répertoire database/seeders. Ils insèrent les données générées par les factories dans la base de données. Les seeders sont exécutés en utilisant la commande php artisan db:seed.

Fonctionnement

* ****UserSeeder**** : Insère des utilisateurs fictifs dans la base de données. En plus de générer des utilisateurs aléatoires, il crée un utilisateur spécifique avec le pseudo "Gary", l'email "gary@gary.com" et le mot de passe "gary".
* ****CategorySeeder**** : Insère 10 catégories spécifiques dans la base de données.
* ****ThemeSeeder**** : Insère 3 thèmes pour chaque catégorie, avec des noms et des descriptions spécifiques à chaque thème.
* ****CardSeeder**** : Insère 5 cartes pour chaque thème, avec des questions et des réponses spécifiques à chaque thème.

Utilisation dans l'application

1. **Configuration des Factories** : Les factories sont définies pour chaque modèle dans database/factories. Elles utilisent Faker pour générer des valeurs fictives réalistes.
2. **Exécution des Seeders** : Les seeders sont définis dans database/seeders. Vous pouvez exécuter tous les seeders en utilisant la commande :

php artisan db:seed

Pour réinitialiser la base de données et exécuter les seeders, utilisez :

php artisan migrate:fresh --seed

1. **Exemple d'utilisation** :
   * Lors de l'exécution du UserSeeder, un utilisateur spécifique nommé Gary est créé en plus de 10 utilisateurs aléatoires.
   * Le CategorySeeder crée exactement 10 catégories prédéfinies.
   * Le ThemeSeeder ajoute 3 thèmes prédéfinis pour chaque catégorie.
   * Le CardSeeder génère 5 cartes pour chaque thème, avec des questions et des réponses adaptées au thème.

Cette approche permet de rapidement peupler la base de données avec des données réalistes et cohérentes, facilitant ainsi le développement et les tests de l'application.

1. Fonctionnalités / G. Promesses

Les promesses sont une fonctionnalité essentielle de JavaScript pour gérer les opérations asynchrones, telles que les appels réseau ou les temporisations. Dans Vue.js, les promesses sont couramment utilisées pour manipuler les données et les interactions réseau au sein des composants et des stores.

Utilisation des Promesses dans Vue.js avec Pinia

Dans l'exemple suivant, nous utilisons Pinia, une bibliothèque de gestion d'état pour Vue.js, pour démontrer l'utilisation des promesses. Les actions asynchrones dans Pinia, telles que les appels API, sont souvent gérées à l'aide de la syntaxe async/await, qui est une manière moderne et propre de travailler avec des promesses.

Exemple : Store de Révisions (Review Store)

#### Déclaration du Store

Le store de révisions gère l'état et les actions associées aux révisions de thèmes. Voici une description des principales actions asynchrones dans ce store.

#### Actions Asynchrones

1. **fetchUserReviews**
   * **Description** : Récupère les révisions de l'utilisateur connecté à partir du serveur.
   * ****Traitement**** :
     + Effectue un appel API pour obtenir les révisions.
     + Stocke les révisions récupérées dans l'état du store.
   * ****Gestion des erreurs**** : Si l'appel API échoue, une erreur est loggée dans la console.
2. **finishReview**
   * ****Description**** : Termine une révision spécifique pour un thème donné.
   * ****Traitement**** :
     + Effectue un appel API pour marquer la révision comme terminée.
     + Met à jour l'état du store avec la révision mise à jour.
   * ****Gestion des erreurs**** : Si l'appel API échoue, une erreur est loggée dans la console.

#### Méthodes de Mise à Jour de l'État

* ****addReview**** : Ajoute une nouvelle révision à la liste des révisions.
* ****updateReview**** : Met à jour une révision existante dans la liste des révisions.
* ****removeReview**** : Supprime une révision de la liste des révisions en fonction de son identifiant.

Exemple de Fonctionnement Asynchrone

Voici comment ces méthodes asynchrones fonctionnent :

* ****Appel API**** : Utilisation de axios pour effectuer des requêtes HTTP vers le serveur.
* ****Syntaxe**** async/await : Permet une gestion plus simple et plus lisible des promesses.
* ****Gestion des Réponses**** : Les données obtenues sont directement stockées dans l'état du store via des mutations d'état.

Gestion des Erreurs

Il est crucial de gérer les erreurs lors des appels API pour assurer la robustesse de l'application. Dans les exemples, les erreurs sont capturées et loggées, mais dans une application réelle, vous pourriez également afficher des messages d'erreur à l'utilisateur ou effectuer d'autres actions correctives.

Conclusion

L'utilisation des promesses dans Vue.js, en particulier avec la syntaxe async/await, permet de gérer les opérations asynchrones de manière propre et efficace. En combinant cela avec un store d'état comme Pinia, il est possible de créer des applications réactives et maintenables qui interagissent de manière fluide avec des API externes.

1. Fonctionnalités / H. Composants

Les composants sont les éléments fondamentaux de Vue.js, permettant de construire des interfaces utilisateur modulaires, réutilisables et maintenables. Dans cette application, les composants sont utilisés pour structurer l'interface utilisateur et gérer diverses fonctionnalités de manière organisée.

Exemple de Composant : AppNavBar.vue

Le composant AppNavBar.vue est utilisé pour la barre de navigation de l'application, permettant aux utilisateurs de naviguer entre différentes sections de l'application.

#### Description

AppNavBar.vue est conçu pour fournir une navigation intuitive et accessible. Il inclut des éléments interactifs tels que des liens vers les pages principales et des boutons pour les actions courantes comme la connexion et l'inscription.

#### Fonctionnalités Principales

1. **Navigation entre les pages** :
   * Liens vers les différentes sections de l'application.
   * Gestion des états actifs pour indiquer la page courante.
2. **Actions utilisateur** :
   * Boutons pour se connecter et s'inscrire.
   * Gestion des états de l'utilisateur (connecté ou non).
3. **Réactivité et accessibilité** :
   * Adaptation de l'interface en fonction de l'état de l'utilisateur.
   * Utilisation de classes CSS pour les transitions et les animations.

#### Événements

Le composant peut émettre des événements pour notifier les composants parents des actions utilisateur, comme un clic sur un bouton de déconnexion. Cela permet une communication efficace entre les composants et une gestion centralisée des actions.

#### Gestion d'État

AppNavBar.vue utilise les stores Pinia pour gérer l'état global de l'application. Par exemple, l'état de l'utilisateur (connecté ou non) est géré via le store utilisateur, permettant au composant de réagir aux changements d'état en temps réel.

Exemple de Fonctionnalité dans AppNavBar.vue

1. **Connexion et Inscription** :
   * Affiche des boutons de connexion et d'inscription si l'utilisateur n'est pas connecté.
   * Émet des événements pour initier les processus de connexion et d'inscription.
2. **Navigation Dynamique** :
   * Affiche des liens de navigation en fonction de l'état de l'application et des permissions de l'utilisateur.
   * Utilise des classes dynamiques pour indiquer la page active.

Conclusion

Les composants dans Vue.js, tels que AppNavBar.vue, permettent de structurer l'interface utilisateur de manière modulaire et réactive. En utilisant les événements, et les stores Pinia, AppNavBar.vue offre une barre de navigation flexible et réactive, essentielle pour une navigation efficace dans l'application.

1. Annexe

Routes Backend :

Une image contenant texte, capture d’écran, affichage

Description générée automatiquement

Routes Backend :

Une image contenant texte, capture d’écran

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement