Université Abdelmalek Essaâdi

Ecole Nationale des Sciences Appliquées de Tanger

Année Universitaire 2023/2024

Projet d'Intégration

Application web pour la gestion des Activités pour Enfants

Groupe : GENIUS LAB

RAPPORT TECHNIQUE BACK-END

Réalisé par :
Aymane Arib
Achraf Elaidi Idrissi

Encadré par :
Prof GHAILANI Mohamed

Tuteur :
EL HADDAD Mohammed
ABDELKOUDOUS Rayes

1. Introduction:

1.1. Contexte :

Ce projet consiste à développer une plateforme web pour la gestion des activités pour enfants. La plateforme doit permettre aux parents d'inscrire leurs enfants à diverses activités, de gérer les plannings et les paiements, et d'interagir avec les animateurs. Elle doit également faciliter l'administration des activités et des inscriptions pour les responsables.

Pourquoi ce projet ?

- **Augmentation de la demande pour les activités extra-scolaires :**

Les parents cherchent de plus en plus à inscrire leurs enfants à diverses activités pour favoriser leur développement personnel, social et éducatif. La diversité des activités proposées (sportives, artistiques, culturelles, éducatives) nécessite un outil de gestion complet pour gérer les inscriptions, les horaires et les paiements.

- **Complexité de la gestion :**

Gérer manuellement les inscriptions, les plannings et les paiements peut être fastidieux et sujet à des erreurs.

La nécessité de coordonner les disponibilités des enfants et des animateurs rend la planification encore plus complexe.

Les administrateurs doivent également gérer les avis et les retours des parents pour améliorer constamment les services offerts.

- **Besoin d'interaction et de communication fluide** :

Les parents doivent pouvoir communiquer facilement avec les animateurs pour obtenir des informations sur les activités et le progrès de leurs enfants.
Les notifications en temps réel et les mises à jour automatisées sont essentielles pour tenir les parents informés des changements ou des nouveautés.

❖ **Gestion centralisée et automatisée** :

Une plateforme centralisée permet de regrouper toutes les informations nécessaires à la gestion des activités en un seul endroit. L'automatisation des processus, comme l'inscription en ligne et la génération de factures, réduit la charge administrative et les risques d'erreurs.

1.2. Objectifs du Projet:

1.2.1. Pour les parents :

Offrir une interface intuitive pour inscrire leurs enfants à des activités, gérer les plannings et effectuer les paiements en ligne. Ainsi fournir des outils pour communiquer avec les animateurs et consulter les avis et recommandations sur les activités.

1.2.2. Pour les animateurs :

Faciliter la gestion de leurs plannings et disponibilités et offrir un outil pour suivre les enfants inscrits à leurs activités et gérer les évaluations et retours des parents.

1.2.3. Pour les administrateurs :

Centraliser la gestion des activités, des animateurs et des inscriptions et automatiser la génération des factures et devis, et la gestion des paiements. Par le fait de fournir des outils pour analyser les avis des parents et ajuster les activités en conséquence.

1.2.4. Pour les enfants :

Assurer une planification optimale des activités en fonction de leurs intérêts et disponibilités et permettre une participation enrichissante et structurée aux activités proposées.

1.3. Exigences :

1.3.1. Exigences fonctionnelles :

❖ **Gestion des Utilisateurs**

Authentification : Login et Logout utilisant Sanctum pour la gestion des tokens d'authentification.

CRUD User : Gestion complète des utilisateurs, incluant la création, la lecture, la mise à jour et la suppression des utilisateurs.

❖ Gestion des Activités

CRUD Activité : Création, consultation, modification et suppression des activités proposées.

CRUD Horaire Activité : Définition des horaires des activités par jour et heure, avec début et fin de chaque séance.

CRUD Activité Langue : Définition des langues pour chaque activité.

CRUD Activité Modalité : Définition des modalités des activités.

CRUD Activité Objectif : Définition des objectifs de chaque activité.

CRUD Activité Reviews : Gestion des avis pour chaque activité, incluant le parent qui a laissé chaque avis.

CRUD Offre : Définition des activités incluses dans une offre, avec des remises spécifiques.

❖ Gestion des Animateurs

CRUD Animateur: Gestion des animateurs, incluant leur création, consultation, modification et suppression.

CRUD Animateur Services : Définition des services offerts et maîtrisés par chaque animateur.

CRUD Dispo Animateur : Définition des horaires de disponibilité des animateurs,

CRUD Expérience Animateur : Définition des expériences passées et actuelles des animateurs.

CRUD Langue Animateur : Définition des langues parlées par chaque animateur et leur niveau de maîtrise, facilitant ainsi la sélection par l’administrateur.

❖ Gestion des Parents et des Enfants

CRUD Parent : Gestion des parents, incluant leur création, consultation, modification et suppression.

CRUD Demande : Gestion des demandes des parents. Chaque demande a un statut initial de 0 ; si l'administrateur accepte la demande, le statut passe à 1.

CRUD Enfant : Gestion des enfants de chaque parent, incluant leur création, consultation, modification et suppression.

CRUD Dispo Enfant : Permet aux parents de préciser les horaires de disponibilité de leurs enfants.

CRUD Enfant Intérêt : Définition des intérêts des enfants, aidant à suggérer des activités de manière interactive.

❖ Autres Fonctions

CRUD Catégorie : Affichage des catégories définies par l'administrateur.

CRUD Catégorie Services : Définition des services pour chaque catégorie, définis par l'administrateur.

CRUD Horaire : Définition des horaires des activités, définis par l'administrateur.

1.3.2. Exigences techniques :

❖ Technologies Utilisées :

Backend : Laravel avec Sanctum pour la gestion des tokens d'authentification.

Frontend : JavaScript pour la validation des données côté client.

Recommandation Système : Flask avec Python.

API Testing : Postman.

Notifications en Temps Réel : Pusher.

Login avec Facebook et Google : Intégration des applications Facebook et Google.

Génération des Devis et Factures : TCPDF pour la génération de documents PDF.

2. Conception :

2.1. Modèles de données :

2.1.1. Modèle Conceptuel de Données (MCD) :

2.1.2. Modèle Logique de Données (MLD) :

Le MLD traduit les entités conceptuelles du MCD en tables de base de données

Tables Utilisateurs : utilisateurs, parents, enfants, animateurs

Tables Activités : activités, horaires_activités, activités_langues, activités_modalités, activités_objectifs, activités_reviews

Tables Offres : offres

Tables Animateurs : animateurs, animateurs_services, dispo_animateurs, experiences_animateurs, langues_animateurs

Tables Disponibilités : dispo_enfants, horaires

Tables Devis et Factures : devis, factures

2.1.3. Étapes Clés du Développement du MLD :

Le développement du Modèle Logique de Données (MLD) est une étape cruciale qui transforme les concepts abstraits du Modèle Conceptuel de Données (MCD) en structures de base de données utilisables.

❖ Définition des Tables

Chaque entité définie dans le MCD est convertie en une table dans le MLD. Cette étape implique :

Identification des Entités : Revue des entités définies dans le MCD (par exemple, Utilisateur, Activité, Parent).

Définition des Attributs : Chaque entité est décomposée en attributs spécifiques qui deviennent des colonnes dans la table. Par exemple :

❖ Configuration des Relations

Les relations entre les tables sont définies pour maintenir l'intégrité référentielle et optimiser les performances. Cette étape comprend :

Définition des Clés Primaires : Chaque table doit avoir une clé primaire unique pour identifier de manière unique chaque enregistrement. Par exemple, la colonne `id` est souvent utilisée comme clé primaire.

Définition des Clés Étrangères : Les clés étrangères sont utilisées pour créer des liens entre les tables

Indices et Indexes : Les indices sont créés pour accélérer les requêtes de recherche et de tri. Par exemple, un index peut être créé sur la colonne `email` de la table **Utilisateur** pour accélérer les recherches par email.

❖ Normalisation

La normalisation est le processus de structuration des tables pour minimiser la redondance et dépendance des données. Les principes de normalisation appliqués comprennent :

Première Forme Normale (1NF) : Suppression des groupes de données répétitifs et des colonnes multi-valuées. Chaque table doit

avoir des colonnes atomiques (c'est-à-dire qu'elles ne doivent contenir qu'une seule valeur).

Deuxième Forme Normale (2NF) : Suppression des dépendances partielles. Chaque colonne non-clé doit dépendre entièrement de la clé primaire.

Troisième Forme Normale (3NF) : Suppression des dépendances transitive. Aucune colonne non-clé ne doit dépendre transitivement de la clé primaire.

❖ Établissement des Relations entre les Tables :

Dans le développement d'applications web, établir des relations entre les tables de la base de données est essentiel pour maintenir l'intégrité des données et faciliter les opérations complexes. En Laravel, cela se fait en deux étapes : la création de connexions physiques à l'aide de clés étrangères et la définition de connexions logiques avec les fonctions de relations Eloquent.

BelongsToMany (Many-to-Many) : Cette relation permet de définir une association où plusieurs enregistrements d'une table peuvent être liés à plusieurs enregistrements d'une autre table.

HasMany (One-to-Many) : Cette relation indique qu'un enregistrement dans une table peut avoir plusieurs enregistrements associés dans une autre table.

HasOne (One-to-One) : Cette relation signifie qu'un enregistrement dans une table est associé à un seul enregistrement dans une autre table.

BelongsTo (Inverse One-to-One ou Many-to-One) : Cette relation indique qu'un enregistrement dans une table appartient à un enregistrement dans une autre table.

❖ Exemples Détaillés des Relations entre les Tables :

Dans cette section, nous allons détailler deux exemples spécifiques de relations entre les tables : la relation Catégorie-Service et la relation Animateur - Catégorie - Service.

- **La relation Catégorie-Service**

Connexion Physique :

La table services : contient une clé étrangère categorie_id qui fait référence à l'identifiant de la table catégories. Cette clé étrangère assure l'intégrité référentielle et lie chaque service à une catégorie.

```
$table->foreignIdFor(Categorie::class)->constrained()->onDelete('cascade');
```

La table catégories est plus simple et ne contient pas de référence à une autre table.

Connexion Logique :

Le modèle Catégorie : a plusieurs Service, ce qui est représenté par une relation hasMany. Cela signifie qu'une catégorie peut avoir plusieurs services associés .

```php
class Categorie extends Model
{
    public function services(): HasMany
    {
        return $this->hasMany(Service::class);
    }
}
```

Le modèle Service appartient à une Catégorie, ce qui est représenté par une relation belongsTo. Cela signifie qu'un service est lié à une seule catégorie :

```php
class Service extends Model
{
    public function categorie(): BelongsTo
    {
        return $this->belongsTo(Categorie::class);
    }
}
```

- **La relation Animateur - Catégorie - Service**

Connexion Physique :

Pour représenter la relation entre les tables Animateur, Catégorie, et Service, nous utilisons une table pivot animateur_categorie_services. Cette table contient les clés étrangères référencées pour lier les trois entités. Voici la migration pour créer cette table pivot.

```php
public function up(): void
{
    Schema::create('animateur_categorie_services', function (Blueprint $table) {
        $table->id();
        $table->foreignIdFor(Animateur::class)->constrained()->onDelete('cascade');
        $table->foreignIdFor(Categorie::class)->constrained()->onDelete('cascade');
        $table->foreignIdFor(Service::class)->constrained()->onDelete('cascade');
        $table->timestamps();
    });
}
```

Cette migration crée une table avec les colonnes id, animateur_id, categorie_id, et service_id, chacune étant une clé étrangère qui se réfère aux tables animateurs, categories, et services respectivement. L'option onDelete('cascade') assure que les enregistrements dans la table pivot sont supprimés si les enregistrements correspondants dans les tables référencées sont supprimés.

Connexion Logique :

Pour définir les relations logiques dans les modèles Eloquent, nous utilisons les méthodes belongsToMany pour créer les liens entre les entités Animateur, Catégorie, et Service.

```php
class Animateur extends Model
{
    public function categories(): BelongsToMany
    {
        return $this->belongsToMany(Categorie::class, 'animateur_categorie_services')
                    ->withPivot('service_id');
    }

    public function servicesMaitrises(): BelongsToMany
    {
        return $this->belongsToMany(Service::class, 'animateur_categorie_services')
                    ->withPivot('categorie_id');
    }
}
```

```php
class Service extends Model
{
    public function animateurs(): BelongsToMany
    {
        return $this->belongsToMany(Animateur::class, 'animateur_categorie_services')
                    ->withPivot('categorie_id');
    }
}
```

```php
class Categorie extends Model
{
    public function animateurs(): BelongsToMany
    {
        return $this->belongsToMany(Animateur::class, 'animateur_categorie_services'
                    ->withPivot('service_id');
    }
}
```

La table pivot animateur_categorie_services sert de lien entre les tables animateurs, categories, et services, permettant d'établir une relation many-to-many complexe entre ces entités. Dans les modèles Eloquent, nous définissons des méthodes belongsToMany pour créer ces relations logiques. Chaque méthode belongsToMany spécifie la table pivot à utiliser (animateur_categorie_services) et inclut la colonne de pivot associée (service_id, categorie_id, ou animateur_id). Ces relations permettent de facilement interroger la base de données pour obtenir, par exemple, tous les services maîtrisés par un animateur dans une catégorie spécifique, ou toutes les catégories auxquelles appartient un service via un animateur. Les connexions physiques et logiques sont ainsi combinées pour garantir l'intégrité des données et faciliter les opérations de manipulation des données dans l'application.

3. Implémentation

3.1. Environnement de Développement :

3.1.1. IDE (Environnements de Développement Intégrés) :

Visual Studio Code : Un éditeur de code source polyvalent et largement utilisé, offrant une multitude de extensions pour supporter les langages de programmation, le contrôle de version, le debugging.

PyCharm : Spécifiquement utilisé pour le développement en Python, pour le système de recommandation implémenté avec Flask.

3.1.2. Compilateurs et Interpréteurs :

PHP : Utilisé pour le backend avec Laravel. PHP est interprété directement par le serveur web (par exemple, Apache ou Nginx).

Python : Utilisé pour le système de recommandation avec Flask. Python est interprété par l'interpréteur Python, qui est configuré dans l'environnement de développement.

3.1.3. Gestion des Dépendances :

Composer : Utilisé pour gérer les dépendances PHP du projet Laravel.

3.1.4. Serveurs et Bases de Données :

XAMPP/Adminer: Environnements de serveurs locaux qui regroupent Apache, MySQL, et PHP, permettant le développement et les tests en local.

PostgreSQL/MySQL : Systèmes de gestion de bases de données relationnelles utilisés pour stocker les données de l'application.

3.1.5. Plateformes de Test et Debugging :

Postman : Utilisé pour tester les API et les endpoints, permettant de valider les requêtes et réponses.

3.2. Gestion des Contrôleurs

Dans cette section, nous allons détailler la gestion des contrôleurs utilisés dans notre projet. Les contrôleurs jouent un rôle essentiel en tant que point central pour gérer les requêtes HTTP, les opérations sur les données, et les réponses aux utilisateurs. Voici une liste des contrôleurs et leurs responsabilités dans notre projet de plateforme de gestion des activités pour enfants.

3.2.1. Liste des Contrôleurs et leurs Actions :

```
GET|HEAD   /
POST       _ignition/execute-solution ignition.executeSolution › Spatie\LaravelIgnition › ExecuteSolutionController
GET|HEAD   _ignition/health-check ignition.healthCheck › Spatie\LaravelIgnition › HealthCheckController
POST       _ignition/update-config ignition.updateConfig › Spatie\LaravelIgnition › UpdateConfigController
GET|HEAD   api/activites activites.index › Api\ActiviteController@index
POST       api/activites activites.store › Api\ActiviteController@store
GET|HEAD   api/activites/{activite} activites.show › Api\ActiviteController@show
PUT|PATCH  api/activites/{activite} activites.update › Api\ActiviteController@update
DELETE     api/activites/{activite} activites.destroy › Api\ActiviteController@destroy
GET|HEAD   api/activites/{activite}/horaires activites.horaires.index › Api\ActiviteHoraireController@index
POST       api/activites/{activite}/horaires activites.horaires.store › Api\ActiviteHoraireController@store
GET|HEAD   api/activites/{activite}/horaires/{horaire} activites.horaires.show › Api\ActiviteHoraireController@show
PUT|PATCH  api/activites/{activite}/horaires/{horaire} activites.horaires.update › Api\ActiviteHoraireController@update
DELETE     api/activites/{activite}/horaires/{horaire} activites.horaires.destroy › Api\ActiviteHoraireController@destroy
POST       api/activites/{activite}/langues activites.langues.store › Api\ActiviteLangueController@store
GET|HEAD   api/activites/{activite}/langues/{langue} activites.langues.show › Api\ActiviteLangueController@show
GET|HEAD   api/activites/{activite}/modalites activites.modalites.index › Api\ActiviteModaliteController@index
POST       api/activites/{activite}/modalites activites.modalites.store › Api\ActiviteModaliteController@store
GET|HEAD   api/activites/{activite}/modalites/{modalite} activites.modalites.show › Api\ActiviteModaliteController@show
PUT|PATCH  api/activites/{activite}/modalites/{modalite} activites.modalites.update › Api\ActiviteModaliteController@update
DELETE     api/activites/{activite}/modalites/{modalite} activites.modalites.destroy › Api\ActiviteModaliteController@destroy
GET|HEAD   api/activites/{activite}/objectifs activites.objectifs.index › Api\ObjectifController@index
POST       api/activites/{activite}/objectifs activites.objectifs.store › Api\ObjectifController@store
GET|HEAD   api/activites/{activite}/objectifs/{objectif} activites.objectifs.show › Api\ObjectifController@show
PUT|PATCH  api/activites/{activite}/objectifs/{objectif} activites.objectifs.update › Api\ObjectifController@update
DELETE     api/activites/{activite}/objectifs/{objectif} activites.objectifs.destroy › Api\ObjectifController@destroy
GET|HEAD   api/activites/{id}/image Api\ImageUploadController@getActiviteImage
POST       api/activites/{id}/upload-image Api\ImageUploadController@uploadActiviteImage
GET|HEAD   api/admin/dashboard admin.dashboard › Api\UserController@index
GET|HEAD   api/admin/enable-2fa admin.enable2fa › Api\Admin2FAController@enable2FA
POST       api/admin/verify-2fa admin.verify2fa › Api\Admin2FAController@verify2FA
GET|HEAD   api/animateurs animateurs.index › Api\AnimateurController@index
POST       api/animateurs animateurs.store › Api\AnimateurController@store
GET|HEAD   api/animateurs/{animateur} animateurs.show › Api\AnimateurController@show
PUT|PATCH  api/animateurs/{animateur} animateurs.update › Api\AnimateurController@update
DELETE     api/animateurs/{animateur} animateurs.destroy › Api\AnimateurController@destroy
GET|HEAD   api/animateurs/{animateur}/dispo animateurs.dispo.index › Api\DispoAnimateurController@index
```

Les contrôleurs de l'API sont essentiels pour la plateforme de gestion des activités pour enfants, couvrant divers aspects fonctionnels. Parmi eux, on trouve des contrôleurs pour la gestion des activités (comme la création, la mise à jour et la suppression), la gestion des utilisateurs et des administrateurs, la gestion des services et des catégories, ainsi que la gestion des messages et des notifications. Chaque contrôleur assure des opérations spécifiques telles que la gestion des horaires et des langues associées aux activités, la gestion des disponibilités des animateurs et des enfants, ainsi que la gestion des devis et des factures. Ces contrôleurs permettent une gestion efficace et sécurisée des données et des interactions au sein de la plateforme, assurant une expérience utilisateur optimale.

3.2.2. Exemple et manipulation :

Dans cette section, nous présentons deux exemples de contrôleurs API utilisés dans notre projet de plateforme de gestion des activités pour enfants. Les contrôleurs jouent un rôle central dans la gestion des requêtes HTTP, des opérations sur les données et des réponses aux utilisateurs. Chaque exemple est détaillé ci-dessous avec des informations spécifiques sur les méthodes correspondantes et leurs responsabilités respectives.

❖ **Exemple : Gestion de l'authentification à deux facteurs pour les administrateurs :**

Méthode HTTP	Chemin d'accès	Contrôleur	Méthode	Description
GET / HEAD	api/admin/enable-2fa	Api\Admin2FAController	enable2FA	Activer l'authentification à deux facteurs pour un admin
POST	api/admin/verify-2fa	Api\Admin2FAController	verify2FA	Vérifier le code d'authentification à deux facteurs pour un admin

3.3. Les migrations :

Les migrations dans Laravel sont des scripts PHP utilisés pour gérer la structure de la base de données de manière incrémentielle et reproductible. Elles permettent de créer et de modifier les tables sans avoir à manipuler directement SQL, ce qui facilite la gestion de la base de données au sein de l'application.
Chaque migration correspond généralement à une modification de la base de données, comme la création d'une nouvelle table, l'ajout de colonnes, ou la modification de contraintes. Elles sont utilisées pour maintenir la cohérence de la base de données tout au long du développement et du déploiement de l'application.

3.3.1. Création et Exécution :

Les migrations sont créées à l'aide de la commande : `php artisan make:migration`

Une fois écrites, elles peuvent être exécutées avec, `php artisan migrate`
Revenir en arrière sur les modifications du BD `php artisan migrate:rollback`

3.3.2. Exemple d'utilisation :

Pour voir toutes les migrations crée dans nos projets : `php artisan migrate:status`

```
Migration name ..... Batch / Status
2014_10_12_000000_create_users_table ..... [2] Ran
2014_10_12_100000_create_password_reset_tokens_table ..... [2] Ran
2019_08_19_000000_create_failed_jobs_table ..... [2] Ran
2019_12_14_000001_create_personal_access_tokens_table ..... [2] Ran
2024_05_01_155136_create_animateurs_table ..... [2] Ran
2024_05_01_155334_create_parentals_table ..... [2] Ran
2024_05_01_155417_create_administrateurs_table ..... [2] Ran
2024_05_01_155443_create_enfants_table ..... [2] Ran
2024_05_01_155458_create_horaires_table ..... [2] Ran
2024_05_01_155503_create_activites_table ..... [2] Ran
2024_05_01_155522_create_dispo_animateurs_table ..... [2] Ran
2024_05_01_155625_create_animers_table ..... [2] Ran
2024_05_01_155733_create_offres_table ..... [2] Ran
2024_05_01_155746_create_participers_table ..... [2] Ran
2024_05_01_155826_create_dispo_enfants_table ..... [2] Ran
2024_05_01_170411_create_contient_demandes_table ..... [2] Ran
2024_05_01_170452_create_correspends_table ..... [2] Ran
2024_05_01_170538_create_packs_table ..... [2] Ran
2024_05_01_170559_create_factures_table ..... [2] Ran
2024_05_01_170605_create_devis_table ..... [2] Ran
2024_05_01_170846_create_activite_horaires_table ..... [2] Ran
2024_05_02_092006_create_experiences_table ..... [2] Ran
2024_05_02_092033_create_langues_table ..... [2] Ran
2024_05_02_092041_create_categories_table ..... [2] Ran
2024_05_02_093159_create_services_table ..... [2] Ran
2024_05_02_093334_create_langue_animateurs_table ..... [2] Ran
2024_05_02_103532_create_animateur_categorie_services_table ..... [2] Ran
2024_05_04_133411_create_enfant_interets_table ..... [2] Ran
2024_05_06_151914_create_reviews_table ..... [2] Ran
2024_05_06_151949_create_objectifs_table ..... [2] Ran
2024_05_07_153202_create_activite_langues_table ..... [2] Ran
2024_05_16_171258_create_offre_activites_table ..... [2] Ran
2024_05_17_005907_create_modalites_table ..... [2] Ran
2024_05_17_111155_create_activite_modalites_table ..... [2] Ran
2024_05_18_004823_create_demandes_table ..... [2] Ran
2024_05_18_170543_create_demande_activite_enfants_table ..... [2] Ran
2024_06_10_200544_create_conversations_table ..... [2] Ran
```

❖ Exemple 1 : Migration pour la table recommendation_forms :

La migration suivante illustre la création de la table recommendation_forms dans la base de données, utilisée pour stocker les préférences de recommandation associées aux utilisateurs et aux enfants.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateRecommendationFormsTable extends Migration
{
    public function up()
    {
        Schema::create('recommendation_forms', function (Blueprint $table) {
            $table->id();
            $table->foreignId('user_id')->constrained()->onDelete('cascade');
            $table->foreignId('enfant_id')->constrained()->onDelete('cascade');
            $table->text('preferences');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('recommendation_forms');
    }
}
```

- **Objectif :** Cette migration vise à créer la structure de la table `recommendation_forms` pour gérer les informations de recommandation dans l'application.
- **Méthode `up()` :** Cette méthode est exécutée lors de l'application de la migration. Elle utilise Laravel Schéma Builder (`Schéma::create`) pour définir la structure de la table `recommendation_forms` avec toutes ses colonnes et configurations associées.
- **Méthode `down()` :** Cette méthode est utilisée pour annuler la migration. Si nécessaire, elle supprime la table `recommendation_forms` en utilisant `Schema::dropIfExists`, ce qui permet de revenir à l'état précédent avant l'application de cette migration.

- **Colonnes** :

  id : Clé primaire auto-incrémentée qui identifie de manière unique chaque enregistrement dans la table.

  user_id : Clé étrangère référençant l'identifiant de l'utilisateur associé à la recommandation. La méthode constrained() assure que cette clé étrangère est liée à la table users, et onDelete('cascade') spécifie que si l'utilisateur est supprimé, toutes les recommandations associées seront également supprimées pour maintenir l'intégrité des données.

  enfant_id : Clé étrangère référençant l'identifiant de l'enfant associé à la recommandation, avec une configuration similaire à user_id.

  preferences : Champ de type texte qui stocke les préférences spécifiques à la recommandation.

  timestamps() : Méthode qui ajoute automatiquement deux colonnes created_at et updated_at. Ces colonnes enregistrent les timestamps de création et de dernière mise à jour de chaque enregistrement, ce qui est essentiel pour suivre l'historique des modifications.

Exemple 2 : Migration pour la table modalites

La migration suivante détaille la création de la table modalites dans la base de données de l'application. Cette table est destinée à définir les différents types de modalités utilisées pour les activités gérées par l'application.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('modalites', function (Blueprint $table) {
            $table->id();
            $table->string('type_modalite');
            //$table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('modalites');
    }
};
```

- **Objectif :** Cette migration a pour but de créer la structure de la table modalites, qui stocke les différents types de modalités définies pour les activités de l'application.
- **Méthode `up()` :** Cette méthode utilise Laravel Schema Builder (Schema::create) pour créer la table modalites avec la colonne id et type_modalite.
- **Méthode `down()` :** Cette méthode est utilisée pour annuler la migration. Si nécessaire, elle supprime la table modalites en utilisant Schema::dropIfExists, permettant de revenir à l'état précédent avant l'application de cette migration.
- **Colonnes** :
  id : Clé primaire auto-incrémentée qui identifie de manière unique chaque enregistrement dans la table.
  type_modalite : Champ de type string qui enregistre le nom ou la description du type de modalité. Par exemple, "hebdomadaire", "mensuel", "annuel", etc.enfant_id : Clé étrangère référençant l'identifiant de l'enfant associé à la recommandation, avec une configuration similaire à user_id.

3.4. Gestion des APIs

Dans cette section, nous allons détailler la gestion des APIs utilisées dans notre projet en mettant en lumière leur rôle central dans l'interaction avec les utilisateurs et la manipulation des données.

3.4.1. Liste des APIs et leurs Fonctionnalités :

Les APIs jouent un rôle crucial dans notre plateforme XTRADH, permettant la gestion efficace des données et assurant une expérience utilisateur optimale à travers plusieurs fonctionnalités clés .

```
POST       _ignition/execute-solution
GET|HEAD   _ignition/health-check
POST       _ignition/update-config
GET|HEAD   api/activites
POST       api/activites
GET|HEAD   api/activites/{activite}
PUT|PATCH  api/activites/{activite}
DELETE     api/activites/{activite}
GET|HEAD   api/activites/{activite}/horaires
POST       api/activites/{activite}/horaires
GET|HEAD   api/activites/{activite}/horaires/{horaire}
PUT|PATCH  api/activites/{activite}/horaires/{horaire}
DELETE     api/activites/{activite}/horaires/{horaire}
POST       api/activites/{activite}/langues
GET|HEAD   api/activites/{activite}/langues/{langue}
GET|HEAD   api/activites/{activite}/modalites
POST       api/activites/{activite}/modalites
GET|HEAD   api/activites/{activite}/modalites/{modalite}
PUT|PATCH  api/activites/{activite}/modalites/{modalite}
DELETE     api/activites/{activite}/modalites/{modalite}
GET|HEAD   api/activites/{activite}/objectifs
POST       api/activites/{activite}/objectifs
GET|HEAD   api/activites/{activite}/objectifs/{objectif}
PUT|PATCH  api/activites/{activite}/objectifs/{objectif}
DELETE     api/activites/{activite}/objectifs/{objectif}
GET|HEAD   api/activites/{id}/image
POST       api/activites/{id}/upload-image
GET|HEAD   api/admin/dashboard
GET|HEAD   api/admin/enable-2fa
POST       api/admin/verify-2fa
GET|HEAD   api/animateurs
```

3.4.2. Exemples et Manipulation :

Dans cette section, nous présentons deux exemples illustrant l'utilisation pratique des APIs dans notre projet.

❖ Exemple 1 : Gestion des Factures et Devis :

Description : Ces APIs permettent de générer et de télécharger des factures et des devis pour les transactions effectuées sur la plateforme.

Méthode	Description
POST /generate-devis	Envoie une requête avec les détails nécessaires pour générer un devis. Le devis généré est ensuite stocké et peut être téléchargé par l'utilisateur.
GET /download-devis/{id}	Permet de télécharger un devis spécifique en fournissant son identifiant unique.
POST /generate-facture	Envoie une requête pour générer une facture basée sur les transactions effectuées. La facture générée peut ensuite être téléchargée.
GET /download-facture/{id}	Permet de télécharger une facture spécifique en fournissant son identifiant unique.

❖ Exemple 2 : Gestion des Offres et Activités:

Description : Ces APIs permettent de gérer les offres et les activités associées aux offres sur la plateforme.

Méthode	Description
GET /offres	Récupère la liste de toutes les offres disponibles, permettant aux utilisateurs de visualiser et de choisir des offres.
POST /offres	Crée une nouvelle offre dans la base de données en envoyant les détails nécessaires via une requête POST.
GET /offres/{id}	Récupère les détails d'une offre spécifique en fournissant son identifiant unique, permettant aux utilisateurs de voir les informations détaillées de l'offre.
PUT /offres/{id}	Met à jour les informations d'une offre existante, permettant aux administrateurs de modifier les offres disponibles.

3.5. Gestion de Sanctum et Middleware :

3.5.1. Les Sanctum :

❖ Définition générale :

Sanctum est un package Laravel qui fournit une solution légère pour l'authentification des SPA (Single Page Applications), des applications mobiles et des API simples en utilisant des tokens. Sanctum permet de gérer facilement les tokens d'authentification et de sécuriser les APIs.

❖ Son rôle :

Sanctum permet aux utilisateurs de générer des tokens d'API personnels qui peuvent être utilisés pour authentifier les requêtes API. Il fournit également une authentification sans état pour les API en utilisant les tokens, ce qui est idéal pour les applications SPA et les applications mobiles. De plus, Sanctum peut gérer les sessions de l'utilisateur pour les applications traditionnelles basées sur les sessions.

❖ Usage dans le projet (exemple) :

Dans notre projet, Sanctum est utilisé pour sécuriser les API qui nécessitent une authentification. Voici quelques exemples d'implémentation :

- Authentification de l'utilisateur :

```php
Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});
```

Cette route utilise le middleware `auth:sanctum` pour s'assurer que seules les requêtes authentifiées peuvent accéder à l'information de l'utilisateur. Lorsqu'un utilisateur authentifié envoie une requête, les détails de son profil utilisateur sont renvoyés en réponse.

- Gestion des recommandations :

```php
//Systeme de recomendation
Route::middleware('auth:sanctum')->group(function () {
    Route::post('/recommendations/form', [RecommendationController::class, 'store']);
    Route::get('/recommendations/{enfant}', [RecommendationController::class, 'recommend']);
});
```

Ces routes gèrent les recommandations. Le middleware `auth:sanctum` s'assure que seules les requêtes provenant d'utilisateurs authentifiés peuvent envoyer des recommandations ou récupérer des recommandations pour un enfant spécifique. Par exemple, un parent authentifié peut envoyer des recommandations en remplissant un formulaire, et les recommandations peuvent être récupérées pour un enfant donné.

- Gestion des notifications :

```php
//Systeme de notification
Route::middleware('auth:sanctum')->group(function () {
    Route::get('/notifications', [NotificationController::class, 'index']);
    Route::post('/notifications/{id}/read', [NotificationController::class, 'markAsRead']);
    Route::post('/send-notification', [NotificationController::class, 'sendNotification']);
});
```

Ces routes gèrent le système de notifications. Le middleware `auth:sanctum` s'assure que seules les requêtes provenant d'utilisateurs authentifiés peuvent accéder aux notifications, les marquer comme lues ou envoyer de nouvelles notifications. Cela garantit que chaque utilisateur ne peut accéder qu'à ses propres notifications.

3.5.2. Les Middleware:

❖ Définition générale :

Les middleware sont des composants logiciels qui filtrent les requêtes HTTP entrant dans votre application. Ils permettent de traiter les requêtes avant qu'elles n'atteignent le contrôleur et de manipuler les réponses avant qu'elles ne soient renvoyées au client.

❖ Son rôle :

Les middlewares jouent un rôle crucial dans la sécurisation et la gestion des requêtes. Ils permettent de s'assurer que seules les requêtes authentifiées et autorisées accèdent aux ressources de l'application. Les middlewares peuvent également modifier les requêtes et les réponses.

❖ Usage dans le projet (exemple) :

Dans notre projet, plusieurs middlewares sont utilisés pour sécuriser les routes et gérer les permissions :

- Authentification :

```
Route::middleware('auth:sanctum')->group(function () {
    Route::get('/user', function (Request $request) {
        return $request->user();
    });
});
```

Le middleware auth:api protège les routes liées à l'activation et la vérification de la double authentification (2FA) pour les administrateurs. Cela garantit que seules les requêtes provenant d'administrateurs authentifiés peuvent accéder à ces fonctionnalités.

- 2fa:

```
Route::middleware(['auth:api', '2fa'])->group(function () {
    Route::get('/admin/dashboard', [UserController::class, 'index'])->name('admin.dashboard');
});
```

Le middleware `2fa` est utilisé en conjonction avec `auth:api` pour protéger la route du tableau de bord administratif. Après la vérification de l'authentification via un token API, le middleware `2fa` s'assure que la

deuxième étape de l'authentification (2FA) est également complétée avant d'accorder l'accès au tableau de bord.un formulaire, et les recommandations peuvent être récupérées pour un enfant donné.

4. Tests et Validation avec Postman

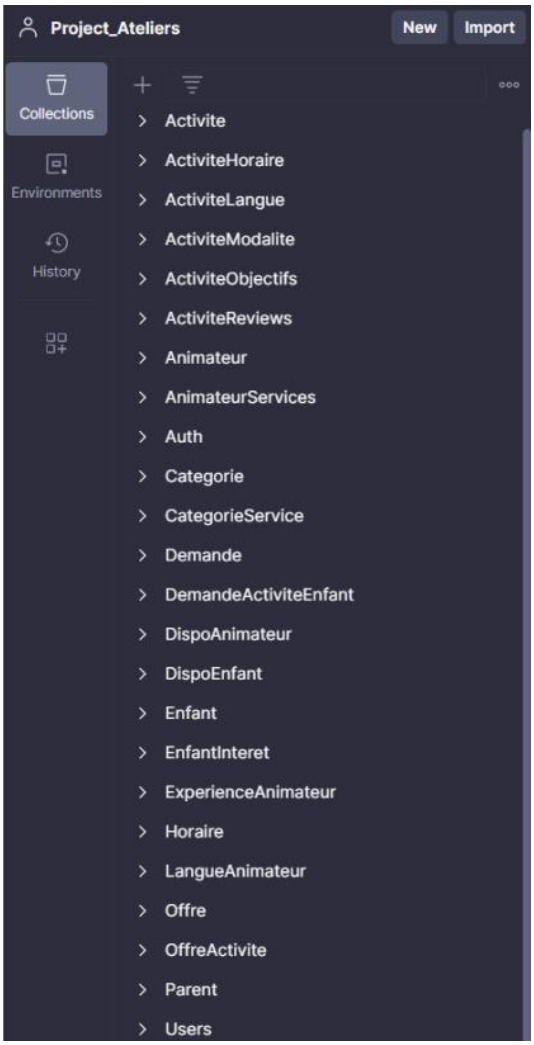
Pour valider les fonctionnalités de l'API développée, Postman est utilisé pour tester les différentes requêtes CRUD. Voici les étapes détaillées pour organiser et effectuer ces tests :

4.1. Organisation des Requêtes dans Postman :

4.1.1. Collections CRUD :

Les requêtes CRUD sont organisées en collections dans Postman

4.1.2. Définition des Variables Locales :

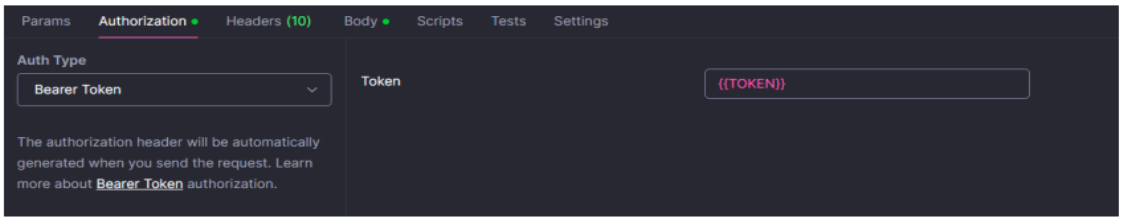


Pour faciliter les tests et éviter la répétition des mêmes valeurs, des variables locales sont définies dans l'environnement Postman. Les variables typiques incluent : base_url : URL de base de l'API. Token : Jeton d'authentification obtenu après la connexion.

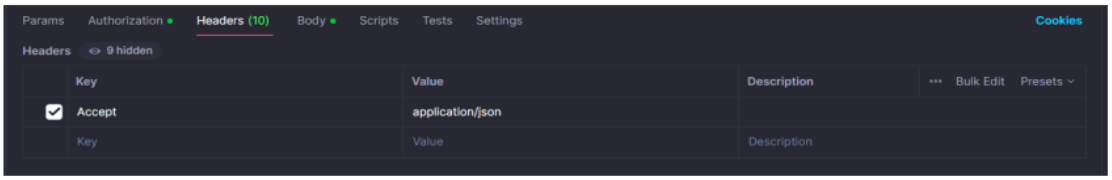
4.1.3. Exemples et Manipulation :

❖ Effectuer une requête POST et créer une nouvelle activité :

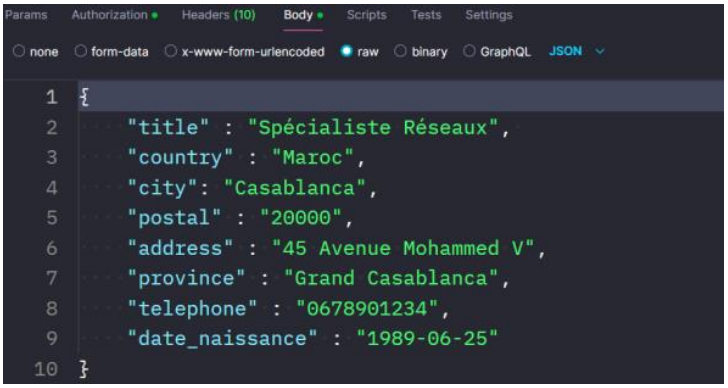
Dans l'onglet "Authorization", sélectionnez "Bearer Token" et utilisez la variable token pour insérer le jeton de connexion.



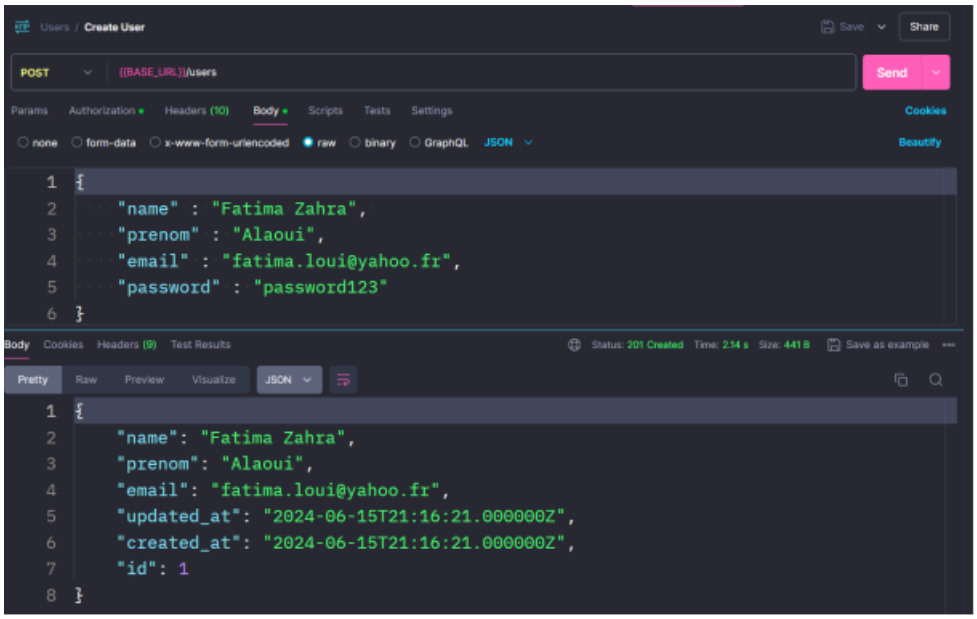
Dans l'onglet "Headers", ajoutez un en-tête pour s'assurer que le retour de la requête est au format JSON



Dans l'onglet "Body", sélectionnez "raw" et choisissez "JSON" dans le menu déroulant. Ajoutez les données du formulaire en format JSON.



Envoyer la Requête (Clique sur send)



On aura cette réponse si l'animateur n'est pas authentifie