Chapitre 4

Base de données et migrations

Pour créer et gérer la structure de la base de données, on utilise des migrations. Ces dernières sont des fichiers PHP qui décrivent les modifications à apporter à la structure de la base de données. Les migrations sont utilisées pour créer/supprimer des tables ou même modifier leur structure (ajouter ou supprimer des colonnes). Une migration est donc un outil de base de données permettant de gérer les modifications de son schéma. Les migrations sont des fichiers de code PHP qui décrivent les modifications de la structure de la base de données.

Les migrations servent aussi à versionner les schémas de votre base de données, ce qui signifie qu'elles permettent de suivre et de gérer les modifications apportées à la structure de la base de données au fil du temps. De cette façon, il est facile de revenir en arrière en cas de problème ou de revenir à une version antérieure de la base de données si nécessaire.

Laravel fournit une interface de ligne de commande (artisan CLI) qui facilite la création, l'exécution et l'annulation des migrations. En utilisant la CLI, cous pouvez créer des migrations pour gérer les changements de schéma de base de données de manière simple et efficace.

4.1 Créer des migrations

Pour créer un fichier de migration, utiliser la commande suivante :

```
php artisan make:migration <nom_de_migration>
```

Après avoir exécuté cette commande, Laravel créera un fichier de migration dans le répertoire `/database/migrations` de votre projet. Le nom de fichier aura la forme suivante : `yyyy_mm_dd_hhmmss_nom_de_la_migration.php`. Le préfixe de date et d'heure permet de garder la chronolgie suivant laquelle nous avons créé les migrations, ce qui est important pour la gestion des versions de la base de données.

Dans ce fichier, vous allez trouver deux méthodes `up()` et `down`. La méthode `up()` contient les instructions pour effectuer la migration, tandis que la méthode `down()` contient les instructions pour annuler la migration. En d'autres termes, la méthode `down()` doit contenir des instructions qui ont exactement l'effet inverse de la méthode `up`.

4.1.1 Le nom de la migration

Lorsque vous créez une migration dans Laravel, il est important de donner un nom significatif pour que vous puissiez facilement identifier ce qu'elle fait. En général, le nom de la migration doit refléter les changements apportés à la base de données. Mais en plus, laravel est suffisament intelligent pour déduire une partie du code de votre migration à partir de son nom. Par exemple, le code ci-dessous est généré si vous appelé votre migration `test`.

```
<?php
use Illuminate\Database\Migrations\Migration;
return new class extends Migration {
    public function up() {
    }
    public function down() {
    }
```

```
};
```

alors que si vous donnez un nom plus significatif à votre migration, comme par exemple `create_products_table`, le code généré est le suivant :

```
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
return new class extends Migration {
    public function up() {
        Schema::create('products', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
        });
    }
    public function down() {
        Schema::dropIfExists('products');
    }
};
```

Dans le code précédent, la méthode `create` permet de créer une table dans la base de données qu'on a définie dans le fichier `.env` (voir section 1.6). La méthode `create` prend deux paramètres, le nom de la table, et une fonction permettant de définir les colonnes de la table. Dans le même exemple, l'instruction `$table->id();` définie une clé primaire auto-incrémentée. L'instruction `$table->timestamps()` ajoute les colonnes `created_at` et `updated_at` à la table. Lorsqu'une nouvelle ligne est insérée dans cette table, Laravel enregistre automatiquement la date et l'heure de création dans la colonne `created_at`, et la date et l'heure de la dernière modification dans la colonne `updated_at`, cette dernière sera mises à jour automatiquement chaque fois qu'une ligne est modifiée. Si la table ne sera pas créée dans cette migration mais seulement modifiée (ajouter des colonnes par

exemple), vous allez utiliser la méthode `table` au lieu de `create`.

Dans la méthode `down()`, la méthode `dropIfExists` comme son nom l'indique, supprime la table dont le nom est passé comme paramètre si cette dernière existe.

4.1.2 Ajouter des colonnes

Pour ajouter d'autres colonnes à cette table, nous ajoutons des instructions suivant la syntaxe suivante :

```
$table->type-colonne('<nom-colonne>', <autres-paramètres>);
```

tel que le type de colonne peut être `string`, `int`, `decimal`, ou n'importe quel autre type supporté par le système de gestion de base de données. Le nom de la colonne est au choix de l'utilisateur. Il est bien sur recommandé d'utiliser des noms significatifs pour faciliter la compréhension de votre code et même la structure de votre base de données. Certains type exigent des paramètres additionnels, d'autre acceptent des paramètre additionnel mais sont obliger les utilisateurs à les mettre.

Par exemple, pour définir le prix comme type `decimal` avec une précision de dix (10) chiffres et ayant deux (2) chiffres après la virgule, l'instruction suivante est utilisée :

```
$table->decimal('price', 10, 2);
```

Laravel supporte jusqu'à 70 types de données comme le montre la figure 4.1, nous n'allons pas tous les expliquer dans ce cours mais nous recommandons de visiter cette page de la documentation pour plus d'informations [1].

4.1.3 Les clés étrangères

Pour ajouter des clés étrangères dans une migration Laravel, vous pouvez utiliser la méthode `foreignId` suivi par la méthode `constrained`. La méthode `foreignId` crée une colonne de type `unsigned big int` comme clé étrangère alors que la méthode `constrained` utilise les conventions Laravel pour déterminer la table et la colonne référencées par la clé

1. https://laravel.com/docs/10.x/migrations#available-column-types

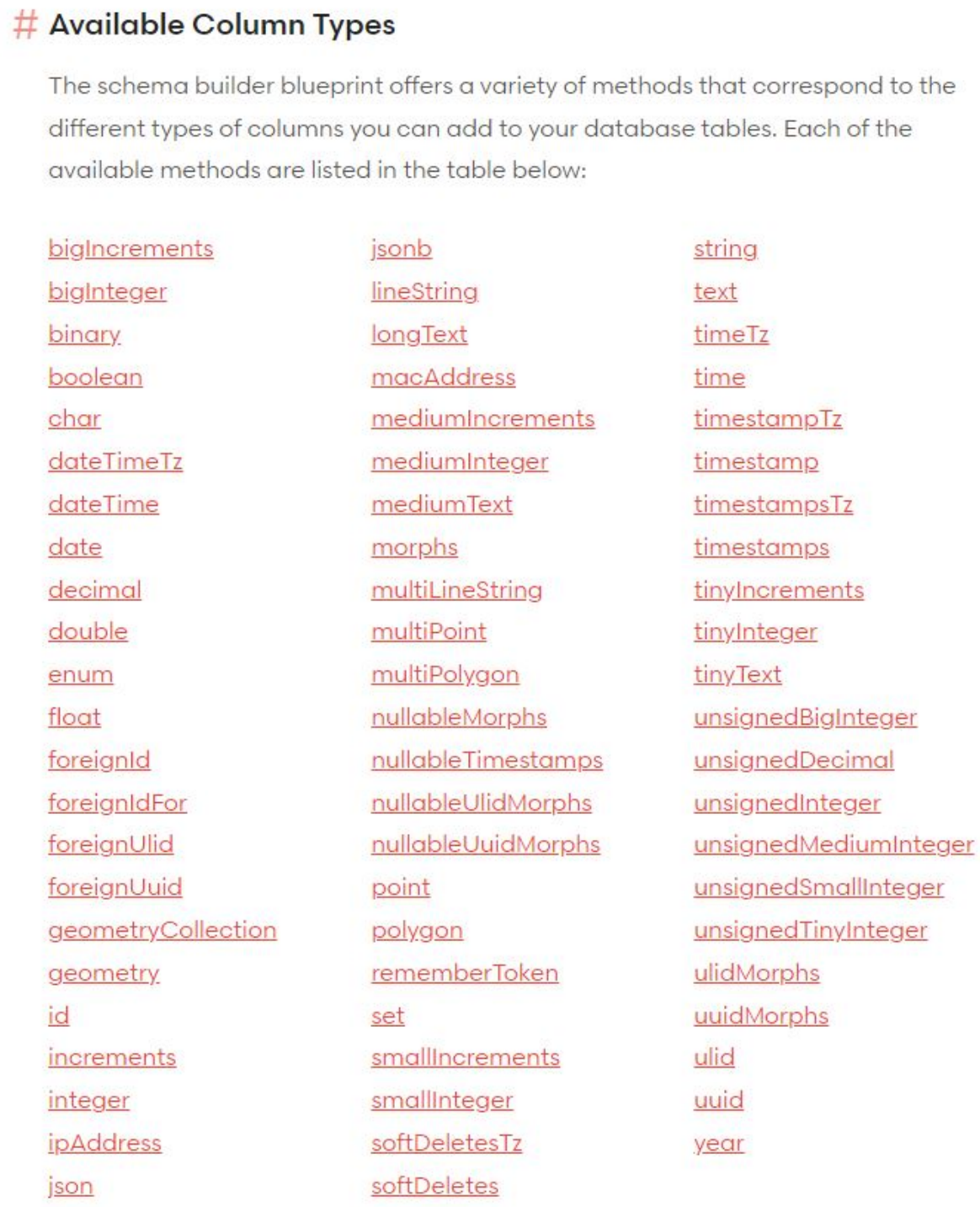

FIGURE 4.1 – Les types de données supportés par laravel

en question. Par exemple, l'instruction suivante permet de créer une clé étrangère vers la colonne `id` de la table `categories`.

```
$table->foreignId('category_id')->constrained();
```

Si le nom de votre colonne ne suit pas exactement les conventions, il est possible de passer

comme paramètre à la méthode `constrained` le nom de la table ainsi que le nom de la colonne contenant l'identifiant comme le montre l'exemple suivant :

```
...
$table->foreignId('class')->constrained('categories', 'id');
...
```

Il est aussi possible de spécifier les actions à effectuer en cas de suppression ou de modification de la clé primaire référencé par cette colonne en utilisant les méthode `onDelete` et `onUpdate`. Les paramètres de ces méthodes correspondent aux mêmes actions qu'on peut utiliser dans le langage `SQL` :

— `cascade` : applique la même action (modification ou suppression) à la table actuelle si une ligne de la table référencée est supprimée ou modifiée. Dans l'exemple précédent, cela signifie que si une catégorie est supprimée, tous les produits qui y appartiennent le sont aussi.

— `restrict` : empêche la suppression (ou la modification) d'une ligne dans la table référencée si elle a des lignes qui la référencent dans la table actuelle. Dans notre exemple, cela signifie qu'il est impossible de supprimer une catégorie s'il y a des produits qui y appartiennent.

— `set null` : si la ligne référencée est supprimée (ou modifiée), les clés étrangères correspondantes prennent la valeur `NULL`.

Si votre clé étrangère n'est pas de type entier, ou que vous voulez définir une colonne déjà créée comme clé étrangère, vous pouvez utiliser la méthode `foreign` au lieu de `foreignId`. Cette méthode peut être enchaînée par les deux méthodes `references` et `on` qui permettent de définir la table et la colonne de la clé étrangère (comme en SQL). Ci-dessous un petit exemple :

```
...
$table->unsignedBigInt('class');
...
```

```
$table->foreign('class')->references('id')->on('categories');
...
```

4.1.4 Autres instructions

Il est possible également d'utiliser les migrations pour ajouter, supprimer ou renommer des colonnes. Renommer des tables, ajouter des `index` ou définir d'autres options sur les colonnes (dire si elle peuvent prendre la valeur `NULL` ou définir une valeur par défaut) est également possible. Pour contrainte de temps, il est impossible d'aborder tous les détails dans ce cours, nous vous invitons donc à consulter la page consacrée aux migrations sur la documentation officielle pour aller plus loin [2].

4.2 Exécuter les migrations

Les migrations que nous avons créés peuvent être exécutées à l'aide de la CLI artisan avec la commande suivante :

```
php artisan migrate
```

Cette commande va exécuter toutes les migrations que vous avez créées ainsi que celles incluses par défaut par Laravel. Si la commande s'exécute correctement, vous allez trouvez toutes les tables correspondantes aux migrations que vous avez créées, et en plus, cinq (5) autres tables. Les trois (3) tables `users`, `personal_access_tokens` et `password_reset_tokens` seront utilisées pour l'authentification des utilisateurs comme nous allons voir dans le chapitre 6. Les deux (2) autres tables `migrations` et `failed_jobs` sont utilisées par Laravel pour garder la trace des migrations exécutées avec succès et celles qui ont échoué.

Il est aussi possible de vérifier à partir de la la CLI le status de chaque fichier de migration en utilisant la commande :

```
php artisan migrate:status
```

Cette commande affiche la liste de tous les fichiers de migration en indiquant s'ils ont été

2. https://laravel.com/docs/10.x/migrations

exécutés avec succès, échoué ou s'ils sont en attente. On dit des migrations qui sont exécutées ensemble dans une même commande qu'elles appartiennent au même `batch`.

4.2.1 Annuler des migration

Pour annuler le dernier batch de migrations exécuté, il est possible d'utiliser la commande :

```
php artisan migrate:rollback
```

Cette commande va exécuter la méthode `down()` dans toutes les migrations qui ont été exécutées dans le dernier `batch`. Pour annuler toutes les migrations au lieu de travailler sur un seul batch, vous pouvez utiliser la commande suivante :

```
php artisan migrate:reset
```

Il est aussi possible d'annuler un nombre spécifique de `batchs` à l'aide de l'option `steps` comme nous le montrons ci-dessous :

```
php artisan migrate:rollback --steps=3
```

La commande précédente indique à Laravel qu'on veut annuler les migrations exécutées dans les trois derniers `batches`. Pour annuler les migrations présentes dans un `batch` spécifique, c'est l'option `batch` qu'il faut utiliser :

```
php artisan migrate:rollback --batch=2
```

La commande précédente va annuler seulement les migrations exécutées dans le deuxième `batch`.

4.2.2 Repartir de zéro

Il est parfois pratique de réinitialiser toute la base de données en utilisant une seule commande, pour cela, nous pouvons utiliser les commandes `migrate:fresh` et `migrate:reset`. LEs deux commandes servent à reinitialiser la base de données, mais elle ont un comportement légèrement différent.

— La commande `migrate:fresh` supprime toutes les tables de la base de données, puis exécute toutes les migrations à nouveau, créant ainsi une nouvelle base de données à partir de zéro. Cette commande est utile lorsqu'on souhaite réinitialiser complètement la base de données, par exemple pour installer une nouvelle version d'une application.

— La commande `migrate:reset`, quant à elle, annule toutes les migrations de la base de données, ramenant la base de données à son état initial avant l'exécution de toute migration. Cette commande est utile lorsqu'on souhaite annuler toutes les modifications apportées par les migrations et revenir à un état de base de données antérieur. Notez que contrairement à la commande `migrate:fresh`, la commande `migrate:reset` par défaut ne supprime pas les table, mais exécute la méthode `down()` dans chaque fichier de migration. Cela revient donc au développeur de contrôler son comportement.

4.3 MyISAM vs InnoDB

Avant d'exécuter vos migrations sur une base de données MySQL, assurez vous que le moteur de base de données utilisé par défaut est `InnoDB` et non pas `MyISAM`. Il y a quelques différences entre les deux moteurs, la plus importante dans notre cas c'est que MyISAM ne supporte pas les clés étrangères et les contrainte contrairement à InnoDB. Pour changer le moteur par défaut dans WampServer, vous cliquer sur l'icône verte de WampServer sur la barre des tâches, puis parcourir `MySQL>Configuration MySQL>default_storage_engine` et choisir l'option `InnoDB` si elle ne l'est pas déjà comme le montre la figure 4.2. Si vous exécuter vos migrations avec le moteur `MyISAM` vous pouvez avoir quelques erreurs et comportements imprédictibles.

Il est aussi possible de spécifier le moteur à utiliser par chaque table à partir des fichiers de migration à l'aide de la commande `$table->engine = 'InnoDB'`. Mais il est recommandé de le mettre comme moteur par défaut pour éviter les erreurs potentielles.

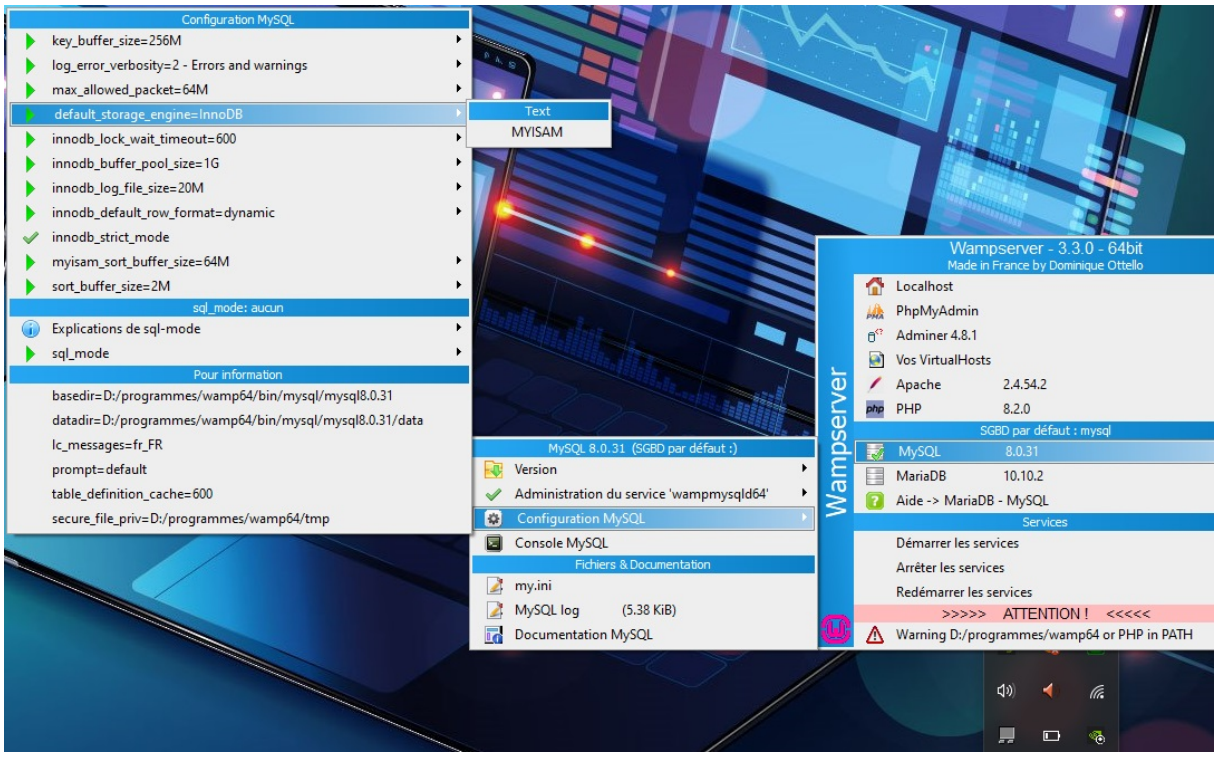

FIGURE 4.2 – Utiliser le moteur InnoDB

4.4 Simuler des migrations

Simuler des migrations peut être utile dans différentes situations, comme lorsqu'on veut vérifier qu'une migration ne va pas causer de problèmes avant de l'exécuter réellement, ou lorsque l'on veut simplement connaître les modifications de base de données que va entraîner une migration particulière. Pour ce faire, on peut utiliser l'option `--pretend` sur les commandes `migrate` Cette option permet de simuler l'exécution d'une migration sans effectuer réellement les modifications de base de données correspondantes. Lorsque cette option est utilisée, Laravel affichera une liste de toutes les requêtes générées par les migrations qui seraient exécutées si l'option n'était pas utilisée. La figure 4.3 montre un exemple d'utilisation de cette option.

```
PS D:\programmes\wamp64\www\dwa1> php artisan migrate --pretend

  INFO  Running migrations.

  2014_10_12_000000_create_users_table ..........................................................................
  ⇂ create table `users` (`id` bigint unsigned not null auto_increment primary key, `name` varchar(100) not null) default character set utf8mb
4 collate 'utf8mb4_unicode_ci' engine = InnoDB
  2014_10_12_100000_create_password_reset_tokens_table ..........................................................
  ⇂ create table `password_reset_tokens` (`email` varchar(255) not null, `token` varchar(255) not null, `created_at` timestamp null) default c
haracter set utf8mb4 collate 'utf8mb4_unicode_ci' engine = InnoDB
  ⇂ alter table `password_reset_tokens` add primary key (`email`)
  2019_08_19_000000_create_failed_jobs_table ....................................................................
  ⇂ create table `failed_jobs` (`id` bigint unsigned not null auto_increment primary key, `uuid` varchar(255) not null, `connection` text not
null, `queue` text not null, `payload` longtext not null, `exception` longtext not null, `failed_at` timestamp not null default CURRENT_TIMEST
AMP) default character set utf8mb4 collate 'utf8mb4_unicode_ci' engine = InnoDB
  ⇂ alter table `failed_jobs` add unique `failed_jobs_uuid_unique`(`uuid`)
  2019_12_14_000001_create_personal_access_tokens_table .........................................................
  ⇂ create table `personal_access_tokens` (`id` bigint unsigned not null auto_increment primary key, `tokenable_type` varchar(255) not null, `
tokenable_id` bigint unsigned not null, `name` varchar(255) not null, `token` varchar(64) not null, `abilities` text null, `last_used_at` time
stamp null, `expires_at` timestamp null, `created_at` timestamp null, `updated_at` timestamp null) default character set utf8mb4 collate 'utf8
mb4_unicode_ci' engine = InnoDB
  ⇂ alter table `personal_access_tokens` add index `personal_access_tokens_tokenable_type_tokenable_id_index`(`tokenable_type`, `tokenable_id`
)
  ⇂ alter table `personal_access_tokens` add unique `personal_access_tokens_token_unique`(`token`)
  2023_03_11_234512_create_products_table .......................................................................
  ⇂ create table `products` (`id` bigint unsigned not null auto_increment primary key, `created_at` timestamp null, `updated_at` timestamp nul
l) default character set utf8mb4 collate 'utf8mb4_unicode_ci'
```

FIGURE 4.3 – Utiliser l'option –pretend