Database: Migrations

Introduction

Migrations are like version control for your database, allowing a team to easily modify and share the application's database schema. Migrations are typically paired with Laravel's schema builder to easily build your application's database schema.

The Laravel `Schema` facade provides database agnostic support for creating and manipulating tables. It shares the same expressive, fluent API across all of Laravel's supported database systems.

Generating Migrations

To create a migration, use the `make:migration` Artisan command:

```
php artisan make:migration create_users_table
```

The new migration will be placed in your `database/migrations` directory. Each migration file name contains a timestamp which allows Laravel to determine the order of the migrations.

The `--table` and `--create` options may also be used to indicate the name of the table and whether the migration will be creating a new table. These options simply pre-fill the generated migration stub file with the specified table:

```
php artisan make:migration add_votes_to_users_table --table=users

php artisan make:migration create_users_table --create=users
```

If you would like to specify a custom output path for the generated migration, you may use the `--path` option when executing the `make:migration` command. The provided path should be relative to your application's base path.

Migration Structure

A migration class contains two methods: `up` and `down`. The `up` method is used to add new tables, columns, or indexes to your database, while the `down` method should simply reverse the operations performed by the `up` method.

Within both of these methods you may use the Laravel schema builder to expressively create and modify tables. To learn about all of the methods available on the `Schema` builder, check out its documentation. For example, let's look at a sample migration that creates a `flights` table:

```
<?php

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

class CreateFlightsTable extends Migration
```

```
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}
```

Running Migrations

To run all outstanding migrations for your application, use the `migrate` Artisan command. If you are using the Homestead virtual machine, you should run this command from within your VM:

```
php artisan migrate
```

If you receive a "class not found" error when running migrations, try running the `composer dump-autoload` command and re-issuing the migrate command.

Forcing Migrations To Run In Production

Some migration operations are destructive, meaning they may cause you to lose data. In order to protect you from running these commands against your production database, you will be prompted for confirmation before these commands are executed. To force the commands to run without a prompt, use the `--force` flag:

```
php artisan migrate --force
```

Rolling Back Migrations

To rollback the latest migration "operation", you may use the `rollback` command. Note that this rolls back the last "batch" of migrations that ran, which may include multiple migration files:

```
php artisan migrate:rollback
```

The `migrate:reset` command will roll back all of your application's migrations:

```
php artisan migrate:reset
```

Rollback / Migrate In Single Command

The `migrate:refresh` command will first roll back all of your database migrations, and then run the `migrate` command. This command effectively re-creates your entire database:

```
php artisan migrate:refresh

php artisan migrate:refresh --seed
```

Writing Migrations

Creating Tables

To create a new database table, use the `create` method on the `Schema` facade. The `create` method accepts two arguments. The first is the name of the table, while the second is a `Closure` which receives a `Blueprint` object used to define the new table:

```
Schema::create('users', function (Blueprint $table) {
    $table->increments('id');
});
```

Of course, when creating the table, you may use any of the schema builder's column methods to define the table's columns.

Checking For Table / Column Existence

You may easily check for the existence of a table or column using the `hasTable` and `hasColumn` methods:

```
if (Schema::hasTable('users')) {
    //
}

if (Schema::hasColumn('users', 'email')) {
    //
}
```

Connection & Storage Engine

If you want to perform a schema operation on a database connection that is not your default connection, use the `connection` method:

```
Schema::connection('foo')->create('users', function ($table) {
    $table->increments('id');
});
```

To set the storage engine for a table, set the `engine` property on the schema builder:

```
Schema::create('users', function ($table) {
    $table->engine = 'InnoDB';

    $table->increments('id');
});
```

Renaming / Dropping Tables

To rename an existing database table, use the `rename` method:

```
Schema::rename($from, $to);
```

To drop an existing table, you may use the `drop` or `dropIfExists` methods:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

Renaming Tables With Foreign Keys

Before renaming a table, you should verify that any foreign key constraints on the table have an explicit name in your migration files instead of letting Laravel assign a convention based name. Otherwise, the foreign key constraint name will refer to the old table name.

Creating Columns

To update an existing table, we will use the `table` method on the `Schema` facade. Like the `create` method, the `table` method accepts two arguments: the name of the table and a `Closure` that receives a `Blueprint` instance we can use to add columns to the table:

```
Schema::table('users', function ($table) {
    $table->string('email');
});
```

Available Column Types

Of course, the schema builder contains a variety of column types that you may use when building your tables:

Command	Description
`$table->bigIncrements('id');`	Incrementing ID (primary key) using a "UNSIGNED BIG INTEGER" equivalent.
`$table->bigInteger('votes');`	BIGINT equivalent for the database.
`$table->binary('data');`	BLOB equivalent for the database.

`$table->boolean('confirmed');`	BOOLEAN equivalent for the database.
`$table->char('name', 4);`	CHAR equivalent with a length.
`$table->date('created_at');`	DATE equivalent for the database.
`$table->dateTime('created_at');`	DATETIME equivalent for the database.
`$table->dateTimeTz('created_at');`	DATETIME (with timezone) equivalent for the database.
`$table->decimal('amount', 5, 2);`	DECIMAL equivalent with a precision and scale.
`$table->double('column', 15, 8);`	DOUBLE equivalent with precision, 15 digits in total and 8 after the decimal point.
`$table->enum('choices', ['foo', 'bar']);`	ENUM equivalent for the database.
`$table->float('amount');`	FLOAT equivalent for the database.
`$table->increments('id');`	Incrementing ID (primary key) using a "UNSIGNED INTEGER" equivalent.
`$table->integer('votes');`	INTEGER equivalent for the database.
`$table->ipAddress('visitor');`	IP address equivalent for the database.
`$table->json('options');`	JSON equivalent for the database.
`$table->jsonb('options');`	JSONB equivalent for the database.
`$table->longText('description');`	LONGTEXT equivalent for the database.
`$table->macAddress('device');`	MAC address equivalent for the database.
`$table->mediumInteger('numbers');`	MEDIUMINT equivalent for the database.
`$table->mediumText('description');`	MEDIUMTEXT equivalent for the database.
`$table->morphs('taggable');`	Adds INTEGER `taggable_id` and STRING `taggable_type`.
`$table->nullableTimestamps();`	Same as `timestamps()`, except allows NULLs.
`$table->rememberToken();`	Adds `remember_token` as VARCHAR(100) NULL.
`$table->smallInteger('votes');`	SMALLINT equivalent for the database.

`$table->softDeletes();`	Adds `deleted_at` column for soft deletes.
`$table->string('email');`	VARCHAR equivalent column.
`$table->string('name', 100);`	VARCHAR equivalent with a length.
`$table->text('description');`	TEXT equivalent for the database.
`$table->time('sunrise');`	TIME equivalent for the database.
`$table->timeTz('sunrise');`	TIME (with timezone) equivalent for the database.
`$table->tinyInteger('numbers');`	TINYINT equivalent for the database.
`$table->timestamp('added_on');`	TIMESTAMP equivalent for the database.
`$table->timestampTz('added_on');`	TIMESTAMP (with timezone) equivalent for the database.
`$table->timestamps();`	Adds `created_at` and `updated_at` columns.
`$table->uuid('id');`	UUID equivalent for the database.

Column Modifiers

In addition to the column types listed above, there are several other column "modifiers" which you may use while adding the column. For example, to make the column "nullable", you may use the `nullable` method:

```
Schema::table('users', function ($table) {
    $table->string('email')->nullable();
});
```

Below is a list of all the available column modifiers. This list does not include the index modifiers:

Modifier	Description
`->first()`	Place the column "first" in the table (MySQL Only)
`->after('column')`	Place the column "after" another column (MySQL Only)
`->nullable()`	Allow NULL values to be inserted into the column
`->default($value)`	Specify a "default" value for the column

`->unsigned()`	Set `integer` columns to `UNSIGNED`
`->comment('my comment')`	Add a comment to a column

Modifying Columns

Prerequisites

Before modifying a column, be sure to add the `doctrine/dbal` dependency to your `composer.json` file. The Doctrine DBAL library is used to determine the current state of the column and create the SQL queries needed to make the specified adjustments to the column.

Updating Column Attributes

The `change` method allows you to modify an existing column to a new type, or modify the column's attributes. For example, you may wish to increase the size of a string column. To see the `change` method in action, let's increase the size of the `name` column from 25 to 50:

```
Schema::table('users', function ($table) {
    $table->string('name', 50)->change();
});
```

We could also modify a column to be nullable:

```
Schema::table('users', function ($table) {
    $table->string('name', 50)->nullable()->change();
});
```

> **Note:** Modifying any column in a table that also has a column of type `enum`, `json` or `jsonb` is not currently supported.

Renaming Columns

To rename a column, you may use the `renameColumn` method on the Schema builder. Before renaming a column, be sure to add the `doctrine/dbal` dependency to your `composer.json` file:

```
Schema::table('users', function ($table) {
    $table->renameColumn('from', 'to');
});
```

Note: Renaming any column in a table that also has a column of type `enum`, `json` or `jsonb` is not currently supported.

Dropping Columns

To drop a column, use the `dropColumn` method on the Schema builder:

```
Schema::table('users', function ($table) {
    $table->dropColumn('votes');
});
```

You may drop multiple columns from a table by passing an array of column names to the `dropColumn` method:

```
Schema::table('users', function ($table) {
    $table->dropColumn(['votes', 'avatar', 'location']);
});
```

Note: Before dropping columns from a SQLite database, you will need to add the `doctrine/dbal` dependency to your `composer.json` file and run the `composer update` command in your terminal to install the library.

Note: Dropping or modifying multiple columns within a single migration while using a SQLite database is not supported.

Creating Indexes

The schema builder supports several types of indexes. First, let's look at an example that specifies a column's values should be unique. To create the index, we can simply chain the `unique` method onto the column definition:

```
$table->string('email')->unique();
```

Alternatively, you may create the index after defining the column. For example:

```
$table->unique('email');
```

You may even pass an array of columns to an index method to create a compound index:

```
$table->index(['account_id', 'created_at']);
```

Laravel will automatically generate a reasonable index name, but you may pass a second argument to the method to specify the name yourself:

```
$table->index('email', 'my_index_name');
```

Available Index Types

Command	Description
`$table->primary('id');`	Add a primary key.
`$table->primary(['first', 'last']);`	Add composite keys.
`$table->unique('email');`	Add a unique index.
`$table->unique('state', 'my_index_name');`	Add a custom index name.
`$table->index('state');`	Add a basic index.

Dropping Indexes

To drop an index, you must specify the index's name. By default, Laravel automatically assigns a reasonable name to the indexes. Simply concatenate the table name, the name of the indexed column, and the index type. Here are some examples:

Command	Description
`$table->dropPrimary('users_id_primary');`	Drop a primary key from the "users" table.

`$table->dropUnique('users_email_unique');`	Drop a unique index from the "users" table.
`$table->dropIndex('geo_state_index');`	Drop a basic index from the "geo" table.

If you pass an array of columns into a method that drops indexes, the conventional index name will be generated based on the table name, columns and key type.

```
Schema::table('geo', function ($table) {
    $table->dropIndex(['state']); // Drops index 'geo_state_index'
});
```

Foreign Key Constraints

Laravel also provides support for creating foreign key constraints, which are used to force referential integrity at the database level. For example, let's define a `user_id` column on the `posts` table that references the `id` column on a `users` table:

```
Schema::table('posts', function ($table) {
    $table->integer('user_id')->unsigned();

    $table->foreign('user_id')->references('id')->on('users');
});
```

You may also specify the desired action for the "on delete" and "on update" properties of the constraint:

```
$table->foreign('user_id')
      ->references('id')->on('users')
      ->onDelete('cascade');
```

To drop a foreign key, you may use the `dropForeign` method. Foreign key constraints use the same naming convention as indexes. So, we will concatenate the table name and the columns in the constraint then suffix the name with "_foreign":

```
$table->dropForeign('posts_user_id_foreign');
```

Or you may pass an array value which will automatically use the conventional constraint name when dropping:

```
$table->dropForeign(['user_id']);
```

You may enable or disable foreign key constraints within your migrations by using the following methods:

```
Schema::enableForeignKeyConstraints();

Schema::disableForeignKeyConstraints();
```