Database: Migrations

#Introduction

Migrations are like version control for your database, allowing your 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. If you have ever had to tell a teammate to manually add a column to their local database schema, you've faced the problem that database migrations solve.

The Laravel `Schema` facade provides database agnostic support for creating and manipulating tables 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 pre-fill the generated migration stub file with the specified table:

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

php artisan make:migration add_votes_to_users_table --table=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 given 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 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, this migration example creates a `flights` table:

```
<?php

use Illuminate\Support\Facades\Schema;
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 of your outstanding migrations, execute the `migrate` Artisan command:

```
php artisan migrate
```

> If you are using the Homestead virtual machine, you should run this command from within your virtual machine.

Forcing Migrations To Run In Production

Some migration operations are destructive, which means 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 the 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. This command rolls back the last "batch" of migrations, which may include multiple migration files:

```
php artisan migrate:rollback
```

You may rollback a limited number of migrations by providing the `step` option to the `rollback` command. For example, the following command will rollback the last five migrations:

```
php artisan migrate:rollback --step=5
```

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 roll back all of your migrations and then execute the `migrate` command. This command effectively re-creates your entire database:

```
php artisan migrate:refresh

// Refresh the database and run all database seeds...
php artisan migrate:refresh --seed
```

You may rollback & re-migrate a limited number of migrations by providing the `step` option to the `refresh` command. For example, the following command will rollback & re-migrate the last five migrations:

```
php artisan migrate:refresh --step=5
```

Drop All Tables & Migrate

The `migrate:fresh` command will drop all tables from the database and then execute the `migrate` command:

```
php artisan migrate:fresh

php artisan migrate:fresh --seed
```

#Tables

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 that may be 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')) {
    //
}
```

Database Connection & Table Options

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 (Blueprint $table) {
    $table->increments('id');
```

```
});
```

You may use the following commands on the schema builder to define the table's options:

Command	Description
`$table->engine = 'InnoDB';`	Specify the table storage engine (MySQL).
`$table->charset = 'utf8';`	Specify a default character set for the table (MySQL).
`$table->collation = 'utf8_unicode_ci';`	Specify a default collation for the table (MySQL).
`$table->temporary();`	Create a temporary table (except SQL Server).

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.

#Columns

Creating Columns

The `table` method on the `Schema` facade may be used to update existing tables. Like the `create` method, the `table` method accepts two arguments: the name of the table and a `Closure` that receives a `Blueprint` instance you may use to add columns to the table:

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

Available Column Types

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

Command	Description
`$table->bigIncrements('id');`	Auto-incrementing UNSIGNED BIGINT (primary key) equivalent column.
`$table->bigInteger('votes');`	BIGINT equivalent column.
`$table->binary('data');`	BLOB equivalent column.
`$table->boolean('confirmed');`	BOOLEAN equivalent column.
`$table->char('name', 100);`	CHAR equivalent column with an optional length.
`$table->date('created_at');`	DATE equivalent column.
`$table->dateTime('created_at');`	DATETIME equivalent column.
`$table->dateTimeTz('created_at');`	DATETIME (with timezone) equivalent column.
`$table->decimal('amount', 8, 2);`	DECIMAL equivalent column with a precision (total digits) and scale (decimal digits).
`$table->double('amount', 8, 2);`	DOUBLE equivalent column with a precision (total digits) and scale (decimal digits).
`$table->enum('level', ['easy', 'hard']);`	ENUM equivalent column.
`$table->float('amount', 8, 2);`	FLOAT equivalent column with a precision (total digits) and scale (decimal digits).
`$table->geometry('positions');`	GEOMETRY equivalent column.
`$table->geometryCollection('positions');`	GEOMETRYCOLLECTION equivalent column.
`$table->increments('id');`	Auto-incrementing UNSIGNED INTEGER (primary key) equivalent column.
`$table->integer('votes');`	INTEGER equivalent column.

Command	Description
`$table->ipAddress('visitor');`	IP address equivalent column.
`$table->json('options');`	JSON equivalent column.
`$table->jsonb('options');`	JSONB equivalent column.
`$table->lineString('positions');`	LINESTRING equivalent column.
`$table->longText('description');`	LONGTEXT equivalent column.
`$table->macAddress('device');`	MAC address equivalent column.
`$table->mediumIncrements('id');`	Auto-incrementing UNSIGNED MEDIUMINT (primary key) equivalent column.
`$table->mediumInteger('votes');`	MEDIUMINT equivalent column.
`$table->mediumText('description');`	MEDIUMTEXT equivalent column.
`$table->morphs('taggable');`	Adds `taggable_id` UNSIGNED BIGINT and `taggable_type` VARCHAR equivalent columns.
`$table->multiLineString('positions');`	MULTILINESTRING equivalent column.
`$table->multiPoint('positions');`	MULTIPOINT equivalent column.
`$table->multiPolygon('positions');`	MULTIPOLYGON equivalent column.
`$table->nullableMorphs('taggable');`	Adds nullable versions of `morphs()` columns.
`$table->nullableTimestamps();`	Alias of `timestamps()` method.
`$table->point('position');`	POINT equivalent column.
`$table->polygon('positions');`	POLYGON equivalent column.
`$table->rememberToken();`	Adds a nullable `remember_token` VARCHAR(100) equivalent column.
`$table->smallIncrements('id');`	Auto-incrementing UNSIGNED SMALLINT (primary key) equivalent column.
`$table->smallInteger('votes');`	SMALLINT equivalent column.

Command	Description
`$table->softDeletes();`	Adds a nullable `deleted_at` TIMESTAMP equivalent column for soft deletes.
`$table->softDeletesTz();`	Adds a nullable `deleted_at` TIMESTAMP (with timezone) equivalent column for soft deletes.
`$table->string('name', 100);`	VARCHAR equivalent column with a optional length.
`$table->text('description');`	TEXT equivalent column.
`$table->time('sunrise');`	TIME equivalent column.
`$table->timeTz('sunrise');`	TIME (with timezone) equivalent column.
`$table->timestamp('added_on');`	TIMESTAMP equivalent column.
`$table->timestampTz('added_on');`	TIMESTAMP (with timezone) equivalent column.
`$table->timestamps();`	Adds nullable `created_at` and `updated_at` TIMESTAMP equivalent columns.
`$table->timestampsTz();`	Adds nullable `created_at` and `updated_at` TIMESTAMP (with timezone) equivalent columns.
`$table->tinyIncrements('id');`	Auto-incrementing UNSIGNED TINYINT (primary key) equivalent column.
`$table->tinyInteger('votes');`	TINYINT equivalent column.
`$table->unsignedBigInteger('votes');`	UNSIGNED BIGINT equivalent column.
`$table->unsignedDecimal('amount', 8, 2);`	UNSIGNED DECIMAL equivalent column with a precision (total digits) and scale (decimal digits).
`$table->unsignedInteger('votes');`	UNSIGNED INTEGER equivalent column.
`$table->unsignedMediumInteger('votes');`	UNSIGNED MEDIUMINT equivalent column.
`$table->unsignedSmallInteger('votes');`	UNSIGNED SMALLINT equivalent column.
`$table->unsignedTinyInteger('votes');`	UNSIGNED TINYINT equivalent column.
`$table->uuid('id');`	UUID equivalent column.
`$table->year('birth_year');`	YEAR equivalent column.

Column Modifiers

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

```
Schema::table('users', function (Blueprint $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
`->after('column')`	Place the column "after" another column (MySQL)
`->autoIncrement()`	Set INTEGER columns as auto-increment (primary key)
`->charset('utf8')`	Specify a character set for the column (MySQL)
`->collation('utf8_unicode_ci')`	Specify a collation for the column (MySQL/SQL Server)
`->comment('my comment')`	Add a comment to a column (MySQL)
`->default($value)`	Specify a "default" value for the column
`->first()`	Place the column "first" in the table (MySQL)
`->nullable($value = true)`	Allows (by default) NULL values to be inserted into the column
`->storedAs($expression)`	Create a stored generated column (MySQL)
`->unsigned()`	Set INTEGER columns as UNSIGNED (MySQL)
`->useCurrent()`	Set TIMESTAMP columns to use CURRENT_TIMESTAMP as default value
`->virtualAs($expression)`	Create a virtual generated column (MySQL)

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:

```
composer require doctrine/dbal
```

Updating Column Attributes

The `change` method allows you to modify some existing column types 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 (Blueprint $table) {
    $table->string('name', 50)->change();
});
```

We could also modify a column to be nullable:

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

> Only the following column types can be "changed": bigInteger, binary, boolean, date, dateTime, dateTimeTz, decimal, integer, json, longText, mediumText, smallInteger, string, text, time, unsignedBigInteger, unsignedInteger and unsignedSmallInteger.

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 (Blueprint $table) {
    $table->renameColumn('from', 'to');
});
```

> Renaming any column in a table that also has a column of type `enum` is not currently supported.

Dropping Columns

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

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

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

Available Command Aliases

Command	Description
`$table->dropRememberToken();`	Drop the `remember_token` column.
`$table->dropSoftDeletes();`	Drop the `deleted_at` column.
`$table->dropSoftDeletesTz();`	Alias of `dropSoftDeletes()` method.
`$table->dropTimestamps();`	Drop the `created_at` and `updated_at` columns.
`$table->dropTimestampsTz();`	Alias of `dropTimestamps()` method.

#Indexes

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 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 (or composite) 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->unique('email', 'unique_email');
```

Available Index Types

Each index method accepts an optional second argument to specify the name of the index. If omitted, the name will be derived from the names of the table and column(s).

Command	Description
`$table->primary('id');`	Adds a primary key.
`$table->primary(['id', 'parent_id']);`	Adds composite keys.
`$table->unique('email');`	Adds a unique index.
`$table->index('state');`	Adds a plain index.
`$table->spatialIndex('location');`	Adds a spatial index. (except SQLite)

Index Lengths & MySQL / MariaDB

Laravel uses the `utf8mb4` character set by default, which includes support for storing "emojis" in the database. If you are running a version of MySQL older than the 5.7.7 release or MariaDB older than the 10.2.2 release, you may need to manually configure the default string length generated by migrations in

order for MySQL to create indexes for them. You may configure this by calling the `Schema::defaultStringLength` method within your `AppServiceProvider`:

```
use Illuminate\Support\Facades\Schema;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Schema::defaultStringLength(191);
}
```

Alternatively, you may enable the `innodb_large_prefix` option for your database. Refer to your database's documentation for instructions on how to properly enable this option.

Renaming Indexes

To rename an index, you may use the `renameIndex` method. This method accepts the current index name as its first argument and the desired name as its second argument:

```
$table->renameIndex('from', 'to')
```

Dropping Indexes

To drop an index, you must specify the index's name. By default, Laravel automatically assigns a reasonable name to the indexes. 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.

Command	Description
`$table->dropSpatialIndex('geo_location_spatialindex');`	Drop a spatial index from the "geo" table (except SQLite).

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 (Blueprint $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 (Blueprint $table) {
    $table->unsignedInteger('user_id');

    $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();
```