VERSION

8.x

Search

WARNING You're browsing the documentation for an old version of Laravel. Consider upgrading your project to Laravel 9.x.

Database: Migrations

Introduction

Migrations are like version control for your database, allowing your team to define and share the application's database schema definition. If you have ever had to tell a teammate to manually add a column to their local database schema after pulling in your changes from source control, 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. Typically, migrations will use this facade to create and modify database tables and columns.

Generating Migrations

You may use the `make:migration` Artisan command to generate a database migration. The new migration will be placed in your `database/migrations` directory. Each migration filename contains a timestamp that allows Laravel to determine the order of the migrations:

```
php artisan make:migration create_flights_table
```

Laravel will use the name of the migration to attempt to guess the name of the table and whether or not the migration will be creating a new table. If Laravel is

able to determine the table name from the migration name, Laravel will pre-fill the generated migration file with the specified table. Otherwise, you may simply specify the table in the migration file manually.

If you would like to specify a custom 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 stubs may be customized using stub publishing.

Squashing Migrations

As you build your application, you may accumulate more and more migrations over time. This can lead to your `database/migrations` directory becoming bloated with potentially hundreds of migrations. If you would like, you may "squash" your migrations into a single SQL file. To get started, execute the `schema:dump` command:

```
php artisan schema:dump

// Dump the current database schema and prune all existing migrations...
php artisan schema:dump --prune
```

When you execute this command, Laravel will write a "schema" file to your application's `database/schema` directory. Now, when you attempt to migrate your database and no other migrations have been executed, Laravel will execute the schema file's SQL statements first. After executing the schema file's statements, Laravel will execute any remaining migrations that were not part of the schema dump.

You should commit your database schema file to source control so that other new developers on your team may quickly create your application's initial database

structure.

Migration squashing is only available for the MySQL, PostgreSQL, and SQLite databases and utilizes the database's command-line client. Schema dumps may not be restored to in-memory SQLite databases.

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, the following migration creates a `flights` table:

```php
<?php

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

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
```

```
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

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

Anonymous Migrations

As you may have noticed in the example above, Laravel will automatically assign a class name to all of the migrations that you generate using the `make:migration` command. However, if you wish, you may return an anonymous class from your migration file. This is primarily useful if your application accumulates many migrations and two of them have a class name collision:

```
<?php

use Illuminate\Database\Migrations\Migration;

return new class extends Migration
{
    //
};
```

Setting The Migration Connection

If your migration will be interacting with a database connection other than your application's default database connection, you should set the `$connection`

property of your migration:

```
/**
 * The database connection that should be used by the migration.
 *
 * @var string
 */
protected $connection = 'pgsql';

/**
 * Run the migrations.
 *
 * @return void
 */
public function up()
{
    //
}
```

Running Migrations

To run all of your outstanding migrations, execute the `migrate` Artisan command:

```
php artisan migrate
```

If you would like to see which migrations have run thus far, you may use the `migrate:status` Artisan command:

```
php artisan migrate:status
```

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 roll back the latest migration operation, you may use the `rollback` Artisan command. This command rolls back the last "batch" of migrations, which may include multiple migration files:

```
php artisan migrate:rollback
```

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

Roll Back & Migrate Using A 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 roll back and re-migrate a limited number of migrations by providing the `step` option to the `refresh` command. For example, the following command will roll back and 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
```

The `migrate:fresh` command will drop all database tables regardless of their prefix. This command should be used with caution when developing on a database that is shared with other applications.

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:

```
use Illuminate\Database\Schema\Blueprint;
```

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

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

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

```
if (Schema::hasTable('users')) {
    // The "users" table exists...
}

if (Schema::hasColumn('users', 'email')) {
    // The "users" table exists and has an "email" column...
}
```

Database Connection & Table Options

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

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

In addition, a few other properties and methods may be used to define other aspects of the table's creation. The `engine` property may be used to specify the table's storage engine when using MySQL:

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

    // ...
});
```

The `charset` and `collation` properties may be used to specify the character set and collation for the created table when using MySQL:

```
Schema::create('users', function (Blueprint $table) {
    $table->charset = 'utf8mb4';
    $table->collation = 'utf8mb4_unicode_ci';

    // ...
});
```

The `temporary` method may be used to indicate that the table should be "temporary". Temporary tables are only visible to the current connection's database session and are dropped automatically when the connection is closed:

```
Schema::create('calculations', function (Blueprint $table) {
    $table->temporary();

    // ...
});
```

Updating Tables

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 or indexes to the table:

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

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

Renaming / Dropping Tables

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

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

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 an `Illuminate\Database\Schema\Blueprint` instance you may use to add columns to the table:

```
use Illuminate\Database\Schema\Blueprint;
```

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

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

Available Column Types

The schema builder blueprint offers a variety of methods that correspond to the different types of columns you can add to your database tables. Each of the available methods are listed in the table below:

bigIncrements
bigInteger
binary
boolean
char
dateTimeTz
dateTime
date
decimal
double
enum
float
foreignId
foreignIdFor
foreignUuid
geometryCollection
geometry
id
increments
integer
ipAddress
json
jsonb
lineString
longText
macAddress
mediumIncrements
mediumInteger
mediumText
morphs
multiLineString
multiPoint
multiPolygon
nullableMorphs
nullableTimestamps
nullableUuidMorphs
point
polygon
rememberToken
set
smallIncrements
smallInteger
softDeletesTz
softDeletes
string
text
timeTz
time
timestampTz
timestamp
timestampsTz
timestamps
tinyIncrements
tinyInteger
tinyText
unsignedBigInteger
unsignedDecimal
unsignedInteger
unsignedMediumInteger
unsignedSmallInteger
unsignedTinyInteger
uuidMorphs
uuid
year

bigIncrements()

The `bigIncrements` method creates an auto-incrementing `UNSIGNED BIGINT` (primary key) equivalent column:

```
$table->bigIncrements('id');
```

bigInteger()

The `bigInteger` method creates a `BIGINT` equivalent column:

```
$table->bigInteger('votes');
```

binary()

The `binary` method creates a `BLOB` equivalent column:

```
$table->binary('photo');
```

boolean()

The `boolean` method creates a `BOOLEAN` equivalent column:

```
$table->boolean('confirmed');
```

char()

The `char` method creates a `CHAR` equivalent column with of a given length:

```
$table->char('name', 100);
```

dateTimeTz()

The `dateTimeTz` method creates a `DATETIME` (with timezone) equivalent column with an optional precision (total digits):

```
$table->dateTimeTz('created_at', $precision = 0);
```

dateTime()

The `dateTime` method creates a `DATETIME` equivalent column with an optional precision (total digits):

```
$table->dateTime('created_at', $precision = 0);
```

date()

The `date` method creates a `DATE` equivalent column:

```
$table->date('created_at');
```

decimal()

The `decimal` method creates a `DECIMAL` equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->decimal('amount', $precision = 8, $scale = 2);
```

double()

The `double` method creates a `DOUBLE` equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->double('amount', 8, 2);
```

enum()

The `enum` method creates a `ENUM` equivalent column with the given valid values:

```
$table->enum('difficulty', ['easy', 'hard']);
```

float()

The `float` method creates a `FLOAT` equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->float('amount', 8, 2);
```

foreignId()

The `foreignId` method creates an `UNSIGNED BIGINT` equivalent column:

```
$table->foreignId('user_id');
```

foreignIdFor()

The `foreignIdFor` method adds a `{column}_id UNSIGNED BIGINT` equivalent column for a given model class:

```
$table->foreignIdFor(User::class);
```

foreignUuid()

The `foreignUuid` method creates a `UUID` equivalent column:

```
$table->foreignUuid('user_id');
```

geometryCollection()

The `geometryCollection` method creates a `GEOMETRYCOLLECTION` equivalent column:

```
$table->geometryCollection('positions');
```

geometry()

The `geometry` method creates a `GEOMETRY` equivalent column:

```
$table->geometry('positions');
```

id()

The `id` method is an alias of the `bigIncrements` method. By default, the method will create an `id` column; however, you may pass a column name if you would like to assign a different name to the column:

```
$table->id();
```

increments()

The `increments` method creates an auto-incrementing `UNSIGNED INTEGER` equivalent column as a primary key:

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

integer()

The `integer` method creates an `INTEGER` equivalent column:

```
$table->integer('votes');
```

ipAddress()

The `ipAddress` method creates a `VARCHAR` equivalent column:

```
$table->ipAddress('visitor');
```

json()

The `json` method creates a `JSON` equivalent column:

```
$table->json('options');
```

jsonb()

The `jsonb` method creates a `JSONB` equivalent column:

```
$table->jsonb('options');
```

lineString()

The `lineString` method creates a `LINESTRING` equivalent column:

```
$table->lineString('positions');
```

longText()

The `longText` method creates a `LONGTEXT` equivalent column:

```
$table->longText('description');
```

macAddress()

The `macAddress` method creates a column that is intended to hold a MAC address. Some database systems, such as PostgreSQL, have a dedicated column type for this type of data. Other database systems will use a string equivalent column:

```
$table->macAddress('device');
```

mediumIncrements()

The `mediumIncrements` method creates an auto-incrementing `UNSIGNED MEDIUMINT` equivalent column as a primary key:

```
$table->mediumIncrements('id');
```

mediumInteger()

The `mediumInteger` method creates a `MEDIUMINT` equivalent column:

```
$table->mediumInteger('votes');
```

mediumText()

The `mediumText` method creates a `MEDIUMTEXT` equivalent column:

```
$table->mediumText('description');
```

morphs()

The `morphs` method is a convenience method that adds a `{column}_id UNSIGNED BIGINT` equivalent column and a `{column}_type VARCHAR` equivalent column.

This method is intended to be used when defining the columns necessary for a polymorphic Eloquent relationship. In the following example, `taggable_id` and `taggable_type` columns would be created:

```
$table->morphs('taggable');
```

multiLineString()

The `multiLineString` method creates a `MULTILINESTRING` equivalent column:

```
$table->multiLineString('positions');
```

multiPoint()

The `multiPoint` method creates a `MULTIPOINT` equivalent column:

```
$table->multiPoint('positions');
```

multiPolygon()

The `multiPolygon` method creates a `MULTIPOLYGON` equivalent column:

```
$table->multiPolygon('positions');
```

nullableTimestamps()

The `nullableTimestamps` method is an alias of the timestamps method:

```
$table->nullableTimestamps(0);
```

nullableMorphs()

The method is similar to the morphs method; however, the columns that are created will be "nullable":

```
$table->nullableMorphs('taggable');
```

nullableUuidMorphs()

The method is similar to the uuidMorphs method; however, the columns that are created will be "nullable":

```
$table->nullableUuidMorphs('taggable');
```

point()

The `point` method creates a `POINT` equivalent column:

```
$table->point('position');
```

polygon()

The `polygon` method creates a `POLYGON` equivalent column:

```
$table->polygon('position');
```

rememberToken()

The `rememberToken` method creates a nullable, `VARCHAR(100)` equivalent column that is intended to store the current "remember me" authentication token:

```
$table->rememberToken();
```

set()

The `set` method creates a `SET` equivalent column with the given list of valid values:

```
$table->set('flavors', ['strawberry', 'vanilla']);
```

smallIncrements()

The `smallIncrements` method creates an auto-incrementing `UNSIGNED SMALLINT` equivalent column as a primary key:

```
$table->smallIncrements('id');
```

smallInteger()

The `smallInteger` method creates a `SMALLINT` equivalent column:

```
$table->smallInteger('votes');
```

softDeletesTz()

The `softDeletesTz` method adds a nullable `deleted_at TIMESTAMP` (with timezone) equivalent column with an optional precision (total digits). This column is intended to store the `deleted_at` timestamp needed for Eloquent's "soft delete" functionality:

```
$table->softDeletesTz($column = 'deleted_at', $precision = 0);
```

softDeletes()

The `softDeletes` method adds a nullable `deleted_at TIMESTAMP` equivalent column with an optional precision (total digits). This column is intended to store the `deleted_at` timestamp needed for Eloquent's "soft delete" functionality:

```
$table->softDeletes($column = 'deleted_at', $precision = 0);
```

string()

The `string` method creates a `VARCHAR` equivalent column of the given length:

```
$table->string('name', 100);
```

text()

The `text` method creates a `TEXT` equivalent column:

```
$table->text('description');
```

timeTz()

The `timeTz` method creates a `TIME` (with timezone) equivalent column with an optional precision (total digits):

```
$table->timeTz('sunrise', $precision = 0);
```

time()

The `time` method creates a `TIME` equivalent column with an optional precision (total digits):

```
$table->time('sunrise', $precision = 0);
```

timestampTz()

The `timestampTz` method creates a `TIMESTAMP` (with timezone) equivalent column with an optional precision (total digits):

```
$table->timestampTz('added_at', $precision = 0);
```

timestamp()

The `timestamp` method creates a `TIMESTAMP` equivalent column with an optional precision (total digits):

```
$table->timestamp('added_at', $precision = 0);
```

timestampsTz()

The `timestampsTz` method creates `created_at` and `updated_at TIMESTAMP` (with timezone) equivalent columns with an optional precision (total digits):

```
$table->timestampsTz($precision = 0);
```

timestamps()

The `timestamps` method creates `created_at` and `updated_at TIMESTAMP` equivalent columns with an optional precision (total digits):

```
$table->timestamps($precision = 0);
```

tinyIncrements()

The `tinyIncrements` method creates an auto-incrementing `UNSIGNED TINYINT` equivalent column as a primary key:

```
$table->tinyIncrements('id');
```

tinyInteger()

The `tinyInteger` method creates a `TINYINT` equivalent column:

```
$table->tinyInteger('votes');
```

tinyText()

The `tinyText` method creates a `TINYTEXT` equivalent column:

```
$table->tinyText('notes');
```

unsignedBigInteger()

The `unsignedBigInteger` method creates an `UNSIGNED BIGINT` equivalent column:

```
$table->unsignedBigInteger('votes');
```

unsignedDecimal()

The `unsignedDecimal` method creates an `UNSIGNED DECIMAL` equivalent column with an optional precision (total digits) and scale (decimal digits):

```
$table->unsignedDecimal('amount', $precision = 8, $scale = 2);
```

unsignedInteger()

The `unsignedInteger` method creates an `UNSIGNED INTEGER` equivalent column:

```
$table->unsignedInteger('votes');
```

unsignedMediumInteger()

The `unsignedMediumInteger` method creates an `UNSIGNED MEDIUMINT` equivalent column:

```
$table->unsignedMediumInteger('votes');
```

unsignedSmallInteger()

The `unsignedSmallInteger` method creates an `UNSIGNED SMALLINT` equivalent column:

```
$table->unsignedSmallInteger('votes');
```

unsignedTinyInteger()

The `unsignedTinyInteger` method creates an `UNSIGNED TINYINT` equivalent column:

```
$table->unsignedTinyInteger('votes');
```

uuidMorphs()

The `uuidMorphs` method is a convenience method that adds a `{column}_id CHAR(36)` equivalent column and a `{column}_type VARCHAR` equivalent column.

This method is intended to be used when defining the columns necessary for a polymorphic Eloquent relationship that use UUID identifiers. In the following example, `taggable_id` and `taggable_type` columns would be created:

```
$table->uuidMorphs('taggable');
```

uuid()

The `uuid` method creates a `UUID` equivalent column:

```
$table->uuid('id');
```

year()

The `year` method creates a `YEAR` equivalent column:

```
$table->year('birth_year');
```

Column Modifiers

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

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

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

The following table contains all of the available column modifiers. This list does not include index modifiers:

Modifier	Description
`->after('column')`	Place the column "after" another column (MySQL).
`->autoIncrement()`	Set INTEGER columns as auto-incrementing (primary key).
`->charset('utf8mb4')`	Specify a character set for the column (MySQL).
`->collation('utf8mb4_unicode_ci')`	Specify a collation for the column (MySQL/PostgreSQL/SQL Server).
`->comment('my comment')`	Add a comment to a column (MySQL/PostgreSQL).
`->default($value)`	Specify a "default" value for the column.
`->first()`	Place the column "first" in the table (MySQL).
`->from($integer)`	Set the starting value of an auto-incrementing field (MySQL / PostgreSQL).

Modifier	Description
`->invisible()`	Make the column "invisible" to `SELECT *` queries (MySQL).
`->nullable($value = true)`	Allow NULL values to be inserted into the column.
`->storedAs($expression)`	Create a stored generated column (MySQL / PostgreSQL).
`->unsigned()`	Set INTEGER columns as UNSIGNED (MySQL).
`->useCurrent()`	Set TIMESTAMP columns to use CURRENT_TIMESTAMP as default value.
`->useCurrentOnUpdate()`	Set TIMESTAMP columns to use CURRENT_TIMESTAMP when a record is updated.
`->virtualAs($expression)`	Create a virtual generated column (MySQL).
`->generatedAs($expression)`	Create an identity column with specified sequence options (PostgreSQL).
`->always()`	Defines the precedence of sequence values over input for an identity column (PostgreSQL).
`->isGeometry()`	Set spatial column type to `geometry` - the default type is `geography` (PostgreSQL).

Default Expressions

The `default` modifier accepts a value or an `Illuminate\Database\Query\Expression` instance. Using an `Expression` instance will prevent Laravel from wrapping the value in quotes and allow you to use database specific functions. One situation where this is particularly useful is when you need to assign default values to JSON columns:

```
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Query\Expression;
use Illuminate\Database\Migrations\Migration;
```

```
class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->json('movies')->default(new Expression('(JSON_ARRAY())'));
            $table->timestamps();
        });
    }
}
```

Support for default expressions depends on your database driver, database version, and the field type. Please refer to your database's documentation.

Column Order

When using the MySQL database, the `after` method may be used to add columns after an existing column in the schema:

```
$table->after('password', function ($table) {
    $table->string('address_line1');
    $table->string('address_line2');
    $table->string('city');
});
```

Modifying Columns

Prerequisites

Before modifying a column, you must install the `doctrine/dbal` package using the Composer package manager. The Doctrine DBAL library is used to determine the current state of the column and to create the SQL queries needed to make the requested changes to your column:

```
composer require doctrine/dbal
```

If you plan to modify columns created using the `timestamp` method, you must also add the following configuration to your application's `config/database.php` configuration file:

```
use Illuminate\Database\DBAL\TimestampType;

'dbal' => [
    'types' => [
        'timestamp' => TimestampType::class,
    ],
],
```

If your application is using Microsoft SQL Server, please ensure that you install `doctrine/dbal:^3.0`.

Updating Column Attributes

The `change` method allows you to modify the type and attributes of existing columns. 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. To accomplish this, we simply define the new state of the column and then call the `change` method:

```
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();
});
```

The following column types can be modified: `bigInteger`, `binary`, `boolean`, `date`, `dateTime`, `dateTimeTz`, `decimal`, `integer`, `json`, `longText`, `mediumText`, `smallInteger`, `string`, `text`, `time`, `unsignedBigInteger`, `unsignedInteger`, `unsignedSmallInteger`, and `uuid`. To modify a `timestamp` column type a Doctrine type must be registered.

Renaming Columns

To rename a column, you may use the `renameColumn` method provided by the schema builder blueprint. Before renaming a column, ensure that you have installed the `doctrine/dbal` library via the Composer package manager:

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

Renaming an `enum` column is not currently supported.

Dropping Columns

To drop a column, you may use the `dropColumn` method on the schema builder blueprint. If your application is utilizing an SQLite database, you must install the `doctrine/dbal` package via the Composer package manager before the `dropColumn` method may be used:

```
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 an SQLite database is not supported.

Available Command Aliases

Laravel provides several convenient methods related to dropping common types of columns. Each of these methods is described in the table below:

Command	Description
`$table->dropMorphs('morphable');`	Drop the `morphable_id` and `morphable_type` columns.
`$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 Laravel schema builder supports several types of indexes. The following example creates a new `email` column and specifies that its values should be unique. To create the index, we can chain the `unique` method onto the column definition:

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

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

Alternatively, you may create the index after defining the column. To do so, you should call the `unique` method on the schema builder blueprint. This method accepts the name of the column that should receive a unique index:

```
$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']);
```

When creating an index, Laravel will automatically generate an index name based on the table, column names, and the index type, but you may pass a second argument to the method to specify the index name yourself:

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

Available Index Types

Laravel's schema builder blueprint class provides methods for creating each type of index supported by Laravel. 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) used for the index, as well as the index type. Each of the available index methods is described in the table below:

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 an index.
`$table->fulltext('body');`	Adds a fulltext index (MySQL/PostgreSQL).
`$table->fulltext('body')->language('english');`	Adds a fulltext index of the specified language (PostgreSQL).
`$table->spatialIndex('location');`	Adds a spatial index (except SQLite).

Index Lengths & MySQL / MariaDB

By default, Laravel uses the `utf8mb4` character set. 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 the default string length by calling the `Schema::defaultStringLength` method within the `boot` method of your `App\Providers\AppServiceProvider` class:

```
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 provided by the schema builder blueprint. 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 an index name based on 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.
`$table->dropIndex('geo_state_index');`	Drop a basic index from the "geo" table.
`$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 index 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:

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

Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('user_id');

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

Since this syntax is rather verbose, Laravel provides additional, terser methods that use conventions to provide a better developer experience. When using the

`foreignId` method to create your column, the example above can be rewritten like so:

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained();
});
```

The `foreignId` method creates an `UNSIGNED BIGINT` equivalent column, while the `constrained` method will use conventions to determine the table and column name being referenced. If your table name does not match Laravel's conventions, you may specify the table name by passing it as an argument to the `constrained` method:

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained('users');
});
```

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

```
$table->foreignId('user_id')
      ->constrained()
      ->onUpdate('cascade')
      ->onDelete('cascade');
```

An alternative, expressive syntax is also provided for these actions:

Method	Description
`$table->cascadeOnUpdate();`	Updates should cascade.
`$table->restrictOnUpdate();`	Updates should be restricted.
`$table->cascadeOnDelete();`	Deletes should cascade.
`$table->restrictOnDelete();`	Deletes should be restricted.

Method	Description
`$table->nullOnDelete();`	Deletes should set the foreign key value to null.

Any additional column modifiers must be called before the `constrained` method:

```
$table->foreignId('user_id')
      ->nullable()
      ->constrained();
```

Dropping Foreign Keys

To drop a foreign key, you may use the `dropForeign` method, passing the name of the foreign key constraint to be deleted as an argument. Foreign key constraints use the same naming convention as indexes. In other words, the foreign key constraint name is based on the name of the table and the columns in the constraint, followed by a "_foreign" suffix:

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

Alternatively, you may pass an array containing the column name that holds the foreign key to the `dropForeign` method. The array will be converted to a foreign key constraint name using Laravel's constraint naming conventions:

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

Toggling Foreign Key Constraints

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

```
Schema::enableForeignKeyConstraints();

Schema::disableForeignKeyConstraints();
```

SQLite disables foreign key constraints by default. When using SQLite, make sure to enable foreign key support in your database configuration before attempting to create them in your migrations. In addition, SQLite only supports foreign keys upon creation of the table and not when tables are altered.

Events

For convenience, each migration operation will dispatch an event. All of the following events extend the base `Illuminate\Database\Events\MigrationEvent` class:

Class	Description
`Illuminate\Database\Events\MigrationsStarted`	A batch of migrations is about to be executed.
`Illuminate\Database\Events\MigrationsEnded`	A batch of migrations has finished executing.
`Illuminate\Database\Events\MigrationStarted`	A single migration is about to be executed.
`Illuminate\Database\Events\MigrationEnded`	A single migration has finished executing.

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.

HIGHLIGHTS

Our Team
Release Notes
Getting Started
Routing
Blade Templates
Authentication
Authorization
Artisan Console
Database
Eloquent ORM
Testing

RESOURCES

Laracasts
Laravel News
Laracon
Laracon EU
Jobs
Forums

PARTNERS

Vehikl
Tighten
64 Robots
Kirschbaum
Curotec
Jump24
A2 Design
ABOUT YOU
Byte 5
Cubet
Cyber-Duck
DevSquad
Ideil
Romega Software
Worksome
WebReinvent

ECOSYSTEM

Cashier
Dusk
Echo
Envoyer
Forge
Homestead
Horizon
Mix
Nova
Passport
Scout
Socialite
Spark
Telescope
Valet
Vapor

Code highlighting provided by Torchlight