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. The schema file's name will correspond to the database connection. Now, when you attempt to migrate your database and no other migrations have been executed, Laravel will first execute the SQL statements in the schema file of the database connection you are using. After executing the schema file's SQL statements, Laravel will execute any remaining migrations that were not part of the schema dump.

If your application's tests use a different database connection than the one you typically use during local development, you should ensure you have dumped a schema file using that database connection so that your tests are able to build your database. You may wish to do this after dumping the database connection you typically use during local development:

```
php artisan schema:dump
```

```
php artisan schema:dump --database=testing --prune
```

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 MariaDB, MySQL, PostgreSQL, and SQLite databases and utilizes the database's command-line client.

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;



return new class extends Migration

{

    /**

     * Run the migrations.

     */

    public function up(): void

    {

        Schema::create('flights', function (Blueprint $table) {

            $table->id();

            $table->string('name');

            $table->string('airline');
```

```
            $table->timestamps();
        });
    }


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

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.

*/

public function up(): void

{

    // ...

}
```

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
```

If you would like to see the SQL statements that will be executed by the migrations without actually running them, you may provide the `--pretend` flag to the `migrate` command:

```
php artisan migrate --pretend
```

Isolating Migration Execution

If you are deploying your application across multiple servers and running migrations as part of your deployment process, you likely do not want two servers attempting to migrate the database at the same time. To avoid this, you may use the `isolated` option when invoking the `migrate` command.

When the `isolated` option is provided, Laravel will acquire an atomic lock using your application's cache driver before attempting to run your migrations. All other attempts to run the `migrate` command while that lock is held will not execute; however, the command will still exit with a successful exit status code:

```
php artisan migrate --isolated
```

> To utilize this feature, your application must be using the `memcached`, `redis`, `dynamodb`, `database`, `file`, or `array` cache driver as your application's default cache driver. In addition, all servers must be communicating with the same central cache server.

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
```

You may roll back a specific "batch" of migrations by providing the `batch` option to the `rollback` command, where the `batch` option corresponds to a batch value within your application's `migrations` database table. For example, the following command will roll back all migrations in batch three:

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

If you would like to see the SQL statements that will be executed by the migrations without actually running them, you may provide the `--pretend` flag to the `migrate:rollback` command:

```
php artisan migrate:rollback --pretend
```

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

```
php artisan migrate:reset
```

Roll Back and 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 and 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
```

By default, the `migrate:fresh` command only drops tables from the default database connection. However, you may use the `--database` option to specify the database connection that should be migrated. The database connection name should correspond to a connection defined in your application's `database` configuration file:

```
php artisan migrate:fresh --database=admin
```

> 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.

Determining Table / Column Existence

You may determine the existence of a table, column, or index using the `hasTable`, `hasColumn`, and `hasIndex` methods:

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

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

```
}
```

```
if (Schema::hasIndex('users', ['email'], 'unique')) {
    // The "users" table exists and has a unique index on the "email" column...
}
```

Database Connection and 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 MariaDB or 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 MariaDB or 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();

    // ...
```

```
});
```

If you would like to add a "comment" to a database table, you may invoke the `comment` method on the table instance. Table comments are currently only supported by MariaDB, MySQL, and PostgreSQL:

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

    // ...
});
```

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

foreignUlid

foreignUuid

geography

geometry

id

increments

integer

ipAddress

json

jsonb

longText

macAddress

mediumIncrements

mediumInteger

mediumText

morphs

nullableMorphs

nullableTimestamps

nullableUlidMorphs

nullableUuidMorphs

rememberToken

set

smallIncrements

smallInteger

softDeletesTz

softDeletes

string

text

timeTz

time

timestampTz

timestamp

timestampsTz

timestamps

tinyIncrements

tinyInteger

tinyText

unsignedBigInteger

unsignedInteger

unsignedMediumInteger

unsignedSmallInteger

unsignedTinyInteger

ulidMorphs

uuidMorphs

ulid

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');
```

When utilizing MySQL, MariaDB, or SQL Server, you may pass `length` and `fixed` arguments to create `VARBINARY` or `BINARY` equivalent column:

```
$table->binary('data', length: 16); // VARBINARY(16)
```

```
$table->binary('data', length: 16, fixed: true); // BINARY(16)
```

`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', length: 100);
```

`dateTimeTz()`

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

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

`dateTime()`

The `dateTime` method creates a `DATETIME` equivalent column with an optional fractional seconds precision:

```
$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', total: 8, places: 2);
```

`double()`

The `double` method creates a `DOUBLE` equivalent column:

```
$table->double('amount');
```

`enum()`

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

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

`float()`

The `float` method creates a `FLOAT` equivalent column with the given precision:

```php
$table->float('amount', precision: 53);
```

`foreignId()`

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

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

`foreignIdFor()`

The `foreignIdFor` method adds a `{column}_id` equivalent column for a given model class. The column type will be `UNSIGNED BIGINT`, `CHAR(36)`, or `CHAR(26)` depending on the model key type:

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

`foreignUlid()`

The `foreignUlid` method creates a `ULID` equivalent column:

```php
$table->foreignUlid('user_id');
```

`foreignUuid()`

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

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

`geography()`

The `geography` method creates a `GEOGRAPHY` equivalent column with the given spatial type and SRID (Spatial Reference System Identifier):

```
$table->geography('coordinates', subtype: 'point', srid: 4326);
```

> Support for spatial types depends on your database driver. Please refer to your database's documentation. If your application is utilizing a PostgreSQL database, you must install the PostGIS extension before the `geography` method may be used.

`geometry()`

The `geometry` method creates a `GEOMETRY` equivalent column with the given spatial type and SRID (Spatial Reference System Identifier):

```
$table->geometry('positions', subtype: 'point', srid: 0);
```

> Support for spatial types depends on your database driver. Please refer to your database's documentation. If your application is utilizing a PostgreSQL database, you must install the PostGIS extension before the `geometry` method may be used.

`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');
```

When using PostgreSQL, an `INET` column will be created.

`json()`

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

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

`jsonb()`

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

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

`longText()`

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

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

When utilizing MySQL or MariaDB, you may apply a `binary` character set to the column in order to create a `LONGBLOB` equivalent column:

```
$table->longText('data')->charset('binary'); // LONGBLOB
```

`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');
```

When utilizing MySQL or MariaDB, you may apply a `binary` character set to the column in order to create a `MEDIUMBLOB` equivalent column:

```
$table->mediumText('data')->charset('binary'); // MEDIUMBLOB
```

`morphs()`

The `morphs` method is a convenience method that adds a `{column}_id` equivalent column and a `{column}_type` `VARCHAR` equivalent column. The column type for the `{column}_id` will be `UNSIGNED BIGINT`, `CHAR(36)`, or `CHAR(26)` depending on the model key type.

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');
```

`nullableTimestamps()`

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

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

`nullableMorphs()`

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

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

`nullableUlidMorphs()`

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

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

`nullableUuidMorphs()`

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

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

`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 fractional seconds precision. This column is intended to store the `deleted_at` timestamp needed for Eloquent's "soft delete" functionality:

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

`softDeletes()`

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

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

`string()`

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

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

`text()`

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

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

When utilizing MySQL or MariaDB, you may apply a `binary` character set to the column in order to create a `BLOB` equivalent column:

```
$table->text('data')->charset('binary'); // BLOB
```

`timeTz()`

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

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

`time()`

The `time` method creates a `TIME` equivalent column with an optional fractional seconds precision:

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

`timestampTz()`

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

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

`timestamp()`

The `timestamp` method creates a `TIMESTAMP` equivalent column with an optional fractional seconds precision:

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

`timestampsTz()`

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

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

`timestamps()`

The `timestamps` method creates `created_at` and `updated_at` `TIMESTAMP` equivalent columns with an optional fractional seconds precision:

```
$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');
```

When utilizing MySQL or MariaDB, you may apply a `binary` character set to the column in order to create a `TINYBLOB` equivalent column:

```
$table->tinyText('data')->charset('binary'); // TINYBLOB
```

`unsignedBigInteger()`

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

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

`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');
```

`ulidMorphs()`

The `ulidMorphs` method is a convenience method that adds a `{column}_id CHAR(26)` 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 ULID identifiers. In the following example, `taggable_id` and `taggable_type` columns would be created:

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

`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');
```

`ulid()`

The `ulid` method creates a `ULID` equivalent column:

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

`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 (MariaDB / MySQL).
`->autoIncrement()`	Set `INTEGER` columns as auto-incrementing (primary key).
`->charset('utf8mb4')`	Specify a character set for the column (MariaDB / MySQL).
`->collation('utf8mb4_unicode_ci')`	Specify a collation for the column.
`->comment('my comment')`	Add a comment to a column (MariaDB / MySQL / PostgreSQL).
`->default($value)`	Specify a "default" value for the column.
`->first()`	Place the column "first" in the table (MariaDB / MySQL).
`->from($integer)`	Set the starting value of an auto-incrementing field (MariaDB / MySQL / PostgreSQL).
`->invisible()`	Make the column "invisible" to `SELECT *` queries (MariaDB / MySQL).
`->nullable($value = true)`	Allow `NULL` values to be inserted into the column.

`->storedAs($expression)`	Create a stored generated column (MariaDB / MySQL / PostgreSQL / SQLite).
`->unsigned()`	Set `INTEGER` columns as `UNSIGNED` (MariaDB / MySQL).
`->useCurrent()`	Set `TIMESTAMP` columns to use `CURRENT_TIMESTAMP` as default value.
`->useCurrentOnUpdate()`	Set `TIMESTAMP` columns to use `CURRENT_TIMESTAMP` when a record is updated (MariaDB / MySQL).
`->virtualAs($expression)`	Create a virtual generated column (MariaDB / MySQL / SQLite).
`->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).

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
```

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

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Database\Query\Expression;

use Illuminate\Database\Migrations\Migration;



return new class extends Migration

{

    /**

     * Run the migrations.

     */

    public function up(): void

    {

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

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

Modifying Columns

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();
});
```

When modifying a column, you must explicitly include all the modifiers you want to keep on the column definition - any missing attribute will be dropped. For example, to retain the `unsigned`, `default`, and `comment` attributes, you must call each modifier explicitly when changing the column:

```
Schema::table('users', function (Blueprint $table) {
    $table->integer('votes')->unsigned()->default(1)->comment('my comment')->change();
});
```

The `change` method does not change the indexes of the column. Therefore, you may use index modifiers to explicitly add or drop an index when modifying the column:

```
// Add an index...
$table->bigIncrements('id')->primary()->change();

// Drop an index...
$table->char('postal_code', 10)->unique(false)->change();
```

Renaming Columns

To rename a column, you may use the `renameColumn` method provided by the schema builder:

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

Dropping Columns

To drop a column, you may use the `dropColumn` method on the schema builder:

```
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']);
});
```

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

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->dropFullText('posts_body_fulltext');`	Drop a full text index from the "posts" 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 being referenced. If your table name does not match Laravel's conventions, you may manually provide it to the `constrained` method. In addition, the name that should be assigned to the generated index may be specified as well:

```
Schema::table('posts', function (Blueprint $table) {

    $table->foreignId('user_id')->constrained(

        table: 'users', indexName: 'posts_user_id'

    );

});
```

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.
`$table->noActionOnUpdate();`	No action on updates.
`$table->cascadeOnDelete();`	Deletes should cascade.
`$table->restrictOnDelete();`	Deletes should be restricted.
`$table->nullOnDelete();`	Deletes should set the foreign key value to null.

`$table->noActionOnDelete();`	Prevents deletes if child records exist.

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();
```

```
Schema::withoutForeignKeyConstraints(function () {
    // Constraints disabled within this closure...
});
```

> 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.

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.
`Illuminate\Database\Events\NoPendingMigrations`	A migration command found no pending migrations.
`Illuminate\Database\Events\SchemaDumped`	A database schema dump has completed.
`Illuminate\Database\Events\SchemaLoaded`	An existing database schema dump has loaded.

Introduction

Laravel provides several different approaches to validate your application's incoming data. It is most common to use the `validate` method available on all incoming HTTP requests. However, we will discuss other approaches to validation as well.

Laravel includes a wide variety of convenient validation rules that you may apply to data, even providing the ability to validate if values are unique in a given database table. We'll cover each of these validation rules in detail so that you are familiar with all of Laravel's validation features.

Validation Quickstart

To learn about Laravel's powerful validation features, let's look at a complete example of validating a form and displaying the error messages back to the user. By reading this high-level overview, you'll be able to gain a good general understanding of how to validate incoming request data using Laravel:

Defining the Routes

First, let's assume we have the following routes defined in our `routes/web.php` file:

```
use App\Http\Controllers\PostController;

Route::get('/post/create', [PostController::class, 'create']);

Route::post('/post', [PostController::class, 'store']);
```

The `GET` route will display a form for the user to create a new blog post, while the `POST` route will store the new blog post in the database.

Creating the Controller

Next, let's take a look at a simple controller that handles incoming requests to these routes.

We'll leave the `store` method empty for now:

```
<?php



namespace App\Http\Controllers;



use Illuminate\Http\RedirectResponse;

use Illuminate\Http\Request;

use Illuminate\View\View;



class PostController extends Controller

{

    /**

     * Show the form to create a new blog post.

     */

    public function create(): View
```

```
    {
        return view('post.create');
    }

    /**
     * Store a new blog post.
     */
    public function store(Request $request): RedirectResponse
    {
        // Validate and store the blog post...

        $post = /** ... */

        return to_route('post.show', ['post' => $post->id]);
    }
}
```

Writing the Validation Logic

Now we are ready to fill in our `store` method with the logic to validate the new blog post. To do this, we will use the `validate` method provided by the `Illuminate\Http\Request` object. If the validation rules pass, your code will keep executing normally; however, if validation fails, an `Illuminate\Validation\ValidationException` exception will be thrown and the proper error response will automatically be sent back to the user.

If validation fails during a traditional HTTP request, a redirect response to the previous URL will be generated. If the incoming request is an XHR request, a JSON response containing the validation error messages will be returned.

To get a better understanding of the `validate` method, let's jump back into the `store` method:

```
/**

* Store a new blog post.

*/

public function store(Request $request): RedirectResponse

{

    $validated = $request->validate([

        'title' => 'required|unique:posts|max:255',

        'body' => 'required',

    ]);
```

```
    // The blog post is valid...

    return redirect('/posts');
}
```

As you can see, the validation rules are passed into the `validate` method. Don't worry - all available validation rules are documented. Again, if the validation fails, the proper response will automatically be generated. If the validation passes, our controller will continue executing normally.

Alternatively, validation rules may be specified as arrays of rules instead of a single `|` delimited string:

```
$validatedData = $request->validate([
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

In addition, you may use the `validateWithBag` method to validate a request and store any error messages within a named error bag:

```
$validatedData = $request->validateWithBag('post', [
```

```
    'title' => ['required', 'unique:posts', 'max:255'],

    'body' => ['required'],

]);
```

Stopping on First Validation Failure

Sometimes you may wish to stop running validation rules on an attribute after the first validation failure. To do so, assign the `bail` rule to the attribute:

```
$request->validate([

    'title' => 'bail|required|unique:posts|max:255',

    'body' => 'required',

]);
```

In this example, if the `unique` rule on the `title` attribute fails, the `max` rule will not be checked. Rules will be validated in the order they are assigned.

A Note on Nested Attributes

If the incoming HTTP request contains "nested" field data, you may specify these fields in your validation rules using "dot" syntax:

```
$request->validate([

    'title' => 'required|unique:posts|max:255',

    'author.name' => 'required',
```

```
    'author.description' => 'required',

]);
```

On the other hand, if your field name contains a literal period, you can explicitly prevent this from being interpreted as "dot" syntax by escaping the period with a backslash:

```
$request->validate([

    'title' => 'required|unique:posts|max:255',

    'v1\.0' => 'required',

]);
```

Displaying the Validation Errors

So, what if the incoming request fields do not pass the given validation rules? As mentioned previously, Laravel will automatically redirect the user back to their previous location. In addition, all of the validation errors and request input will automatically be flashed to the session.

An `$errors` variable is shared with all of your application's views by the `Illuminate\View\Middleware\ShareErrorsFromSession` middleware, which is provided by the `web` middleware group. When this middleware is applied an `$errors` variable will always be available in your views, allowing you to conveniently assume the `$errors` variable is always defined and can be safely used. The `$errors` variable will be an instance of `Illuminate\Support\MessageBag`. For more information on working with this object, check out its documentation.

So, in our example, the user will be redirected to our controller's `create` method when validation fails, allowing us to display the error messages in the view:

```
<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->
```

Customizing the Error Messages

Laravel's built-in validation rules each have an error message that is located in your application's `lang/en/validation.php` file. If your application does not have a `lang` directory, you may instruct Laravel to create it using the `lang:publish` Artisan command.

Within the `lang/en/validation.php` file, you will find a translation entry for each validation rule. You are free to change or modify these messages based on the needs of your application.

In addition, you may copy this file to another language directory to translate the messages for your application's language. To learn more about Laravel localization, check out the complete localization documentation.

> By default, the Laravel application skeleton does not include the `lang` directory. If you would like to customize Laravel's language files, you may publish them via the `lang:publish` Artisan command.

XHR Requests and Validation

In this example, we used a traditional form to send data to the application. However, many applications receive XHR requests from a JavaScript powered frontend. When using the `validate` method during an XHR request, Laravel will not generate a redirect response. Instead, Laravel generates a JSON response containing all of the validation errors. This JSON response will be sent with a 422 HTTP status code.

The `@error` Directive

You may use the `@error` Blade directive to quickly determine if validation error messages exist for a given attribute. Within an `@error` directive, you may echo the `$message` variable to display the error message:

```
<!-- /resources/views/post/create.blade.php -->
```

```
<label for="title">Post Title</label>

<input id="title"
    type="text"
    name="title"
    class="@error('title') is-invalid @enderror">

@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

If you are using named error bags, you may pass the name of the error bag as the second argument to the `@error` directive:

```
<input ... class="@error('title', 'post') is-invalid @enderror">
```

Repopulating Forms

When Laravel generates a redirect response due to a validation error, the framework will automatically flash all of the request's input to the session. This is done so that you may

conveniently access the input during the next request and repopulate the form that the user attempted to submit.

To retrieve flashed input from the previous request, invoke the `old` method on an instance of `Illuminate\Http\Request`. The `old` method will pull the previously flashed input data from the session:

```
$title = $request->old('title');
```

Laravel also provides a global `old` helper. If you are displaying old input within a Blade template, it is more convenient to use the `old` helper to repopulate the form. If no old input exists for the given field, `null` will be returned:

```
<input type="text" name="title" value="{{ old('title') }}">
```

A Note on Optional Fields

By default, Laravel includes the `TrimStrings` and `ConvertEmptyStringsToNull` middleware in your application's global middleware stack. Because of this, you will often need to mark your "optional" request fields as `nullable` if you do not want the validator to consider `null` values as invalid. For example:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
    'publish_at' => 'nullable|date',
```

```
]);
```

In this example, we are specifying that the `publish_at` field may be either `null` or a valid date representation. If the `nullable` modifier is not added to the rule definition, the validator would consider `null` an invalid date.

Validation Error Response Format

When your application throws a `Illuminate\Validation\ValidationException` exception and the incoming HTTP request is expecting a JSON response, Laravel will automatically format the error messages for you and return a `422 Unprocessable Entity` HTTP response.

Below, you can review an example of the JSON response format for validation errors. Note that nested error keys are flattened into "dot" notation format:

```
{
    "message": "The team name must be a string. (and 4 more errors)",
    "errors": {
        "team_name": [
            "The team name must be a string.",
            "The team name must be at least 1 characters."
        ],
```

```
        "authorization.role": [
            "The selected authorization.role is invalid."
        ],
        "users.0.email": [
            "The users.0.email field is required."
        ],
        "users.2.email": [
            "The users.2.email must be a valid email address."
        ]
    }
}
```

Form Request Validation

Creating Form Requests

For more complex validation scenarios, you may wish to create a "form request". Form requests are custom request classes that encapsulate their own validation and authorization logic. To create a form request class, you may use the `make:request` Artisan CLI command:

```
php artisan make:request StorePostRequest
```

The generated form request class will be placed in the `app/Http/Requests` directory. If this directory does not exist, it will be created when you run the `make:request` command. Each form request generated by Laravel has two methods: `authorize` and `rules`.

As you might have guessed, the `authorize` method is responsible for determining if the currently authenticated user can perform the action represented by the request, while the `rules` method returns the validation rules that should apply to the request's data:

```
/**

* Get the validation rules that apply to the request.

*

* @return array<string,
\Illuminate\Contracts\Validation\ValidationRule|array<mixed>|string>

*/

public function rules(): array

{

    return [
```

```
        'title' => 'required|unique:posts|max:255',

        'body' => 'required',

    ];

}
```

You may type-hint any dependencies you require within the `rules` method's signature. They will automatically be resolved via the Laravel service container.

So, how are the validation rules evaluated? All you need to do is type-hint the request on your controller method. The incoming form request is validated before the controller method is called, meaning you do not need to clutter your controller with any validation logic:

```
/**

* Store a new blog post.

*/

public function store(StorePostRequest $request): RedirectResponse

{

    // The incoming request is valid...



    // Retrieve the validated input data...

    $validated = $request->validated();
```

```
    // Retrieve a portion of the validated input data...
    $validated = $request->safe()->only(['name', 'email']);
    $validated = $request->safe()->except(['name', 'email']);

    // Store the blog post...

    return redirect('/posts');
}
```

If validation fails, a redirect response will be generated to send the user back to their previous location. The errors will also be flashed to the session so they are available for display. If the request was an XHR request, an HTTP response with a 422 status code will be returned to the user including a JSON representation of the validation errors.

> Need to add real-time form request validation to your Inertia powered Laravel frontend? Check out Laravel Precognition.

Performing Additional Validation

Sometimes you need to perform additional validation after your initial validation is complete. You can accomplish this using the form request's `after` method.

The `after` method should return an array of callables or closures which will be invoked after validation is complete. The given callables will receive an `Illuminate\Validation\Validator` instance, allowing you to raise additional error messages if necessary:

```
use Illuminate\Validation\Validator;



/**

* Get the "after" validation callables for the request.

*/

public function after(): array

{

    return [

        function (Validator $validator) {

            if ($this->somethingElseIsInvalid()) {

                $validator->errors()->add(

                    'field',

                    'Something is wrong with this field!'
```

```
                );
            }
        }
    ];
}
```

As noted, the array returned by the `after` method may also contain invokable classes. The `__invoke` method of these classes will receive an `Illuminate\Validation\Validator` instance:

```
use App\Validation\ValidateShippingTime;
use App\Validation\ValidateUserStatus;
use Illuminate\Validation\Validator;

/**
 * Get the "after" validation callables for the request.
 */
public function after(): array
{
```

```
    return [
        new ValidateUserStatus,
        new ValidateShippingTime,
        function (Validator $validator) {
            //
        }
    ];
}
```

Stopping on the First Validation Failure

By adding a `stopOnFirstFailure` property to your request class, you may inform the validator that it should stop validating all attributes once a single validation failure has occurred:

```
/**
* Indicates if the validator should stop on the first rule failure.
*
* @var bool
*/
protected $stopOnFirstFailure = true;
```

Customizing the Redirect Location

As previously discussed, a redirect response will be generated to send the user back to their previous location when form request validation fails. However, you are free to customize this behavior. To do so, define a `$redirect` property on your form request:

```
/**
 * The URI that users should be redirected to if validation fails.
 *
 * @var string
 */
protected $redirect = '/dashboard';
```

Or, if you would like to redirect users to a named route, you may define a `$redirectRoute` property instead:

```
/**
 * The route that users should be redirected to if validation fails.
 *
 * @var string
 */
protected $redirectRoute = 'dashboard';
```

Authorizing Form Requests

The form request class also contains an `authorize` method. Within this method, you may determine if the authenticated user actually has the authority to update a given resource. For example, you may determine if a user actually owns a blog comment they are attempting to update. Most likely, you will interact with your authorization gates and policies within this method:

```
use App\Models\Comment;



/**

* Determine if the user is authorized to make this request.

*/

public function authorize(): bool

{

    $comment = Comment::find($this->route('comment'));



    return $comment && $this->user()->can('update', $comment);

}
```

Since all form requests extend the base Laravel request class, we may use the `user` method to access the currently authenticated user. Also, note the call to the `route` method in the example above. This method grants you access to the URI parameters defined on the route being called, such as the `{comment}` parameter in the example below:

```
Route::post('/comment/{comment}');
```

Therefore, if your application is taking advantage of route model binding, your code may be made even more succinct by accessing the resolved model as a property of the request:

```
return $this->user()->can('update', $this->comment);
```

If the `authorize` method returns `false`, an HTTP response with a 403 status code will automatically be returned and your controller method will not execute.

If you plan to handle authorization logic for the request in another part of your application, you may remove the `authorize` method completely, or simply return `true`:

```
/**

* Determine if the user is authorized to make this request.

*/

public function authorize(): bool

{

    return true;

}
```

You may type-hint any dependencies you need within the `authorize` method's signature. They will automatically be resolved via the Laravel service container.

Customizing the Error Messages

You may customize the error messages used by the form request by overriding the `messages` method. This method should return an array of attribute / rule pairs and their corresponding error messages:

```
/**

* Get the error messages for the defined validation rules.

*

* @return array<string, string>

*/

public function messages(): array

{

    return [

        'title.required' => 'A title is required',

        'body.required' => 'A message is required',

    ];
```

```
}
```

Customizing the Validation Attributes

Many of Laravel's built-in validation rule error messages contain an `:attribute` placeholder. If you would like the `:attribute` placeholder of your validation message to be replaced with a custom attribute name, you may specify the custom names by overriding the `attributes` method. This method should return an array of attribute / name pairs:

```
/**

* Get custom attributes for validator errors.

*

* @return array<string, string>

*/

public function attributes(): array

{

    return [

        'email' => 'email address',

    ];

}
```

Preparing Input for Validation

If you need to prepare or sanitize any data from the request before you apply your validation rules, you may use the `prepareForValidation` method:

```
use Illuminate\Support\Str;



/**

* Prepare the data for validation.

*/

protected function prepareForValidation(): void

{

    $this->merge([

        'slug' => Str::slug($this->slug),

    ]);

}
```

Likewise, if you need to normalize any request data after validation is complete, you may use the `passedValidation` method:

```
/**
```

```
* Handle a passed validation attempt.

*/

protected function passedValidation(): void

{

    $this->replace(['name' => 'Taylor']);

}
```

Manually Creating Validators

If you do not want to use the `validate` method on the request, you may create a validator instance manually using the `Validator` facade. The `make` method on the facade generates a new validator instance:

```
<?php



namespace App\Http\Controllers;



use Illuminate\Http\RedirectResponse;
```

```php
use Illuminate\Http\Request;

use Illuminate\Support\Facades\Validator;



class PostController extends Controller

{

    /**

     * Store a new blog post.

     */

    public function store(Request $request): RedirectResponse

    {

        $validator = Validator::make($request->all(), [

            'title' => 'required|unique:posts|max:255',

            'body' => 'required',

        ]);



        if ($validator->fails()) {
```

```
            return redirect('/post/create')
                        ->withErrors($validator)
                        ->withInput();
        }

        // Retrieve the validated input...
        $validated = $validator->validated();

        // Retrieve a portion of the validated input...
        $validated = $validator->safe()->only(['name', 'email']);
        $validated = $validator->safe()->except(['name', 'email']);

        // Store the blog post...

        return redirect('/posts');
    }
```

```
}
```

The first argument passed to the `make` method is the data under validation. The second argument is an array of the validation rules that should be applied to the data.

After determining whether the request validation failed, you may use the `withErrors` method to flash the error messages to the session. When using this method, the `$errors` variable will automatically be shared with your views after redirection, allowing you to easily display them back to the user. The `withErrors` method accepts a validator, a `MessageBag`, or a PHP `array`.

Stopping on First Validation Failure

The `stopOnFirstFailure` method will inform the validator that it should stop validating all attributes once a single validation failure has occurred:

```
if ($validator->stopOnFirstFailure()->fails()) {

    // ...

}
```

Automatic Redirection

If you would like to create a validator instance manually but still take advantage of the automatic redirection offered by the HTTP request's `validate` method, you may call the `validate` method on an existing validator instance. If validation fails, the user will automatically be redirected or, in the case of an XHR request, a JSON response will be returned:

```
Validator::make($request->all(), [
```

```
    'title' => 'required|unique:posts|max:255',

    'body' => 'required',

])->validate();
```

You may use the `validateWithBag` method to store the error messages in a named error bag if validation fails:

```
Validator::make($request->all(), [

    'title' => 'required|unique:posts|max:255',

    'body' => 'required',

])->validateWithBag('post');
```

Named Error Bags

If you have multiple forms on a single page, you may wish to name the `MessageBag` containing the validation errors, allowing you to retrieve the error messages for a specific form. To achieve this, pass a name as the second argument to `withErrors`:

```
return redirect('/register')->withErrors($validator, 'login');
```

You may then access the named `MessageBag` instance from the `$errors` variable:

```
{{ $errors->login->first('email') }}
```

Customizing the Error Messages

If needed, you may provide custom error messages that a validator instance should use instead of the default error messages provided by Laravel. There are several ways to specify custom messages. First, you may pass the custom messages as the third argument to the `Validator::make` method:

```
$validator = Validator::make($input, $rules, $messages = [

    'required' => 'The :attribute field is required.',

]);
```

In this example, the `:attribute` placeholder will be replaced by the actual name of the field under validation. You may also utilize other placeholders in validation messages. For example:

```
$messages = [

    'same' => 'The :attribute and :other must match.',

    'size' => 'The :attribute must be exactly :size.',

    'between' => 'The :attribute value :input is not between :min -
:max.',

    'in' => 'The :attribute must be one of the following types:
:values',

];
```

Specifying a Custom Message for a Given Attribute

Sometimes you may wish to specify a custom error message only for a specific attribute. You may do so using "dot" notation. Specify the attribute's name first, followed by the rule:

```
$messages = [

    'email.required' => 'We need to know your email address!',

];
```

Specifying Custom Attribute Values

Many of Laravel's built-in error messages include an `:attribute` placeholder that is replaced with the name of the field or attribute under validation. To customize the values used to replace these placeholders for specific fields, you may pass an array of custom attributes as the fourth argument to the `Validator::make` method:

```
$validator = Validator::make($input, $rules, $messages, [

    'email' => 'email address',

]);
```

Performing Additional Validation

Sometimes you need to perform additional validation after your initial validation is complete. You can accomplish this using the validator's `after` method. The `after` method accepts a closure or an array of callables which will be invoked after validation is complete. The given callables will receive an `Illuminate\Validation\Validator` instance, allowing you to raise additional error messages if necessary:

```
use Illuminate\Support\Facades\Validator;
```

```
$validator = Validator::make(/* ... */);



$validator->after(function ($validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add(
            'field', 'Something is wrong with this field!'
        );
    }
});



if ($validator->fails()) {
    // ...
}
```

As noted, the `after` method also accepts an array of callables, which is particularly convenient if your "after validation" logic is encapsulated in invokable classes, which will receive an `Illuminate\Validation\Validator` instance via their `__invoke` method:

```
use App\Validation\ValidateShippingTime;

use App\Validation\ValidateUserStatus;



$validator->after([

    new ValidateUserStatus,

    new ValidateShippingTime,

    function ($validator) {

        // ...

    },

]);
```

Working With Validated Input

After validating incoming request data using a form request or a manually created validator instance, you may wish to retrieve the incoming request data that actually underwent validation. This can be accomplished in several ways. First, you may call the `validated` method on a form request or validator instance. This method returns an array of the data that was validated:

```
$validated = $request->validated();
```

```php
$validated = $validator->validated();
```

Alternatively, you may call the `safe` method on a form request or validator instance. This method returns an instance of `Illuminate\Support\ValidatedInput`. This object exposes `only`, `except`, and `all` methods to retrieve a subset of the validated data or the entire array of validated data:

```php
$validated = $request->safe()->only(['name', 'email']);
```

```php
$validated = $request->safe()->except(['name', 'email']);
```

```php
$validated = $request->safe()->all();
```

In addition, the `Illuminate\Support\ValidatedInput` instance may be iterated over and accessed like an array:

```php
// Validated data may be iterated...

foreach ($request->safe() as $key => $value) {
    // ...
}
```

```
// Validated data may be accessed as an array...

$validated = $request->safe();



$email = $validated['email'];
```

If you would like to add additional fields to the validated data, you may call the `merge` method:

```
$validated = $request->safe()->merge(['name' => 'Taylor Otwell']);
```

If you would like to retrieve the validated data as a collection instance, you may call the `collect` method:

```
$collection = $request->safe()->collect();
```

Working With Error Messages

After calling the `errors` method on a `Validator` instance, you will receive an `Illuminate\Support\MessageBag` instance, which has a variety of convenient methods for working with error messages. The `$errors` variable that is automatically made available to all views is also an instance of the `MessageBag` class.

Retrieving the First Error Message for a Field

To retrieve the first error message for a given field, use the `first` method:

```
$errors = $validator->errors();
```

```
echo $errors->first('email');
```

Retrieving All Error Messages for a Field

If you need to retrieve an array of all the messages for a given field, use the `get` method:

```
foreach ($errors->get('email') as $message) {
    // ...
}
```

If you are validating an array form field, you may retrieve all of the messages for each of the array elements using the `*` character:

```
foreach ($errors->get('attachments.*') as $message) {
    // ...
}
```

Retrieving All Error Messages for All Fields

To retrieve an array of all messages for all fields, use the `all` method:

```
foreach ($errors->all() as $message) {
    // ...
```

```
}
```

Determining if Messages Exist for a Field

The `has` method may be used to determine if any error messages exist for a given field:

```
if ($errors->has('email')) {

    // ...

}
```

Specifying Custom Messages in Language Files

Laravel's built-in validation rules each have an error message that is located in your application's `lang/en/validation.php` file. If your application does not have a `lang` directory, you may instruct Laravel to create it using the `lang:publish` Artisan command.

Within the `lang/en/validation.php` file, you will find a translation entry for each validation rule. You are free to change or modify these messages based on the needs of your application.

In addition, you may copy this file to another language directory to translate the messages for your application's language. To learn more about Laravel localization, check out the complete localization documentation.

> By default, the Laravel application skeleton does not include the `lang` directory. If you would like to customize Laravel's language files, you may publish them via the `lang:publish` Artisan command.

Custom Messages for Specific Attributes

You may customize the error messages used for specified attribute and rule combinations within your application's validation language files. To do so, add your message customizations to the `custom` array of your application's `lang/xx/validation.php` language file:

```
'custom' => [

    'email' => [

        'required' => 'We need to know your email address!',

        'max' => 'Your email address is too long!'

    ],

],
```

Specifying Attributes in Language Files

Many of Laravel's built-in error messages include an `:attribute` placeholder that is replaced with the name of the field or attribute under validation. If you would like the `:attribute` portion of your validation message to be replaced with a custom value, you may specify the custom attribute name in the `attributes` array of your `lang/xx/validation.php` language file:

```
'attributes' => [

    'email' => 'email address',

],
```

By default, the Laravel application skeleton does not include the `lang` directory. If you would like to customize Laravel's language files, you may publish them via the `lang:publish` Artisan command.

Specifying Values in Language Files

Some of Laravel's built-in validation rule error messages contain a `:value` placeholder that is replaced with the current value of the request attribute. However, you may occasionally need the `:value` portion of your validation message to be replaced with a custom representation of the value. For example, consider the following rule that specifies that a credit card number is required if the `payment_type` has a value of `cc`:

```
Validator::make($request->all(), [
    'credit_card_number' => 'required_if:payment_type,cc'
]);
```

If this validation rule fails, it will produce the following error message:

```
The credit card number field is required when payment type is cc.
```

Instead of displaying `cc` as the payment type value, you may specify a more user-friendly value representation in your `lang/xx/validation.php` language file by defining a `values` array:

```
'values' => [
    'payment_type' => [
        'cc' => 'credit card'
```

```
    ],

],
```

> By default, the Laravel application skeleton does not include the `lang` directory. If you would like to customize Laravel's language files, you may publish them via the `lang:publish` Artisan command.

After defining this value, the validation rule will produce the following error message:

```
The credit card number field is required when payment type is credit card.
```

Available Validation Rules

Below is a list of all available validation rules and their function:

Accepted

Accepted If

Active URL

After (Date)

After Or Equal (Date)

Alpha

Alpha Dash

Alpha Numeric

Array

Ascii

Bail

Before (Date)

Before Or Equal (Date)

Between

Boolean

Confirmed

Contains

Current Password

Date

Date Equals

Date Format

Decimal

Declined

Declined If

Different

Digits

Digits Between

Dimensions (Image Files)

Distinct

Doesnt Start With

Doesnt End With

Email

Ends With

Enum

Exclude

Exclude If

Exclude Unless

Exclude With

Exclude Without

Exists (Database)

Extensions

File

Filled

Greater Than

Greater Than Or Equal

Hex Color

Image (File)

In

In Array

Integer

IP Address

JSON

Less Than

Less Than Or Equal

List

Lowercase

MAC Address

Max

Max Digits

MIME Types

MIME Type By File Extension

Min

Min Digits

Missing

Missing If

Missing Unless

Missing With

Missing With All

Multiple Of

Not In

Not Regex

Nullable

Numeric

Present

Present If

Present Unless

Present With

Present With All

Prohibited

Prohibited If

Prohibited Unless

Prohibits

Regular Expression

Required

Required If

Required If Accepted

Required If Declined

Required Unless

Required With

Required With All

Required Without

Required Without All

Required Array Keys

Same

Size

Sometimes

Starts With

String

Timezone

Unique (Database)

Uppercase

URL

ULID

UUID

accepted

The field under validation must be `"yes"`, `"on"`, `1`, `"1"`, `true`, or `"true"`. This is useful for validating "Terms of Service" acceptance or similar fields.

accepted_if:anotherfield,value,...

The field under validation must be `"yes"`, `"on"`, `1`, `"1"`, `true`, or `"true"` if another field under validation is equal to a specified value. This is useful for validating "Terms of Service" acceptance or similar fields.

active_url

The field under validation must have a valid A or AAAA record according to the `dns_get_record` PHP function. The hostname of the provided URL is extracted using the `parse_url` PHP function before being passed to `dns_get_record`.

after:*date*

The field under validation must be a value after a given date. The dates will be passed into the `strtotime` PHP function in order to be converted to a valid `DateTime` instance:

```
'start_date' => 'required|date|after:tomorrow'
```

Instead of passing a date string to be evaluated by `strtotime`, you may specify another field to compare against the date:

```
'finish_date' => 'required|date|after:start_date'
```

after_or_equal:*date*

The field under validation must be a value after or equal to the given date. For more information, see the after rule.

alpha

The field under validation must be entirely Unicode alphabetic characters contained in `\p{L}` and `\p{M}`.

To restrict this validation rule to characters in the ASCII range (`a-z` and `A-Z`), you may provide the `ascii` option to the validation rule:

```
'username' => 'alpha:ascii',
```

alpha_dash

The field under validation must be entirely Unicode alpha-numeric characters contained in `\p{L}`, `\p{M}`, `\p{N}`, as well as ASCII dashes (`-`) and ASCII underscores (`_`).

To restrict this validation rule to characters in the ASCII range (`a-z` and `A-Z`), you may provide the `ascii` option to the validation rule:

```
'username' => 'alpha_dash:ascii',
```

alpha_num

The field under validation must be entirely Unicode alpha-numeric characters contained in `\p{L}`, `\p{M}`, and `\p{N}`.

To restrict this validation rule to characters in the ASCII range (`a-z` and `A-Z`), you may provide the `ascii` option to the validation rule:

```
'username' => 'alpha_num:ascii',
```

array

The field under validation must be a PHP `array`.

When additional values are provided to the `array` rule, each key in the input array must be present within the list of values provided to the rule. In the following example, the `admin` key in the input array is invalid since it is not contained in the list of values provided to the `array` rule:

```
use Illuminate\Support\Facades\Validator;



$input = [

    'user' => [

        'name' => 'Taylor Otwell',

        'username' => 'taylorotwell',

        'admin' => true,

    ],

];



Validator::make($input, [

    'user' => 'array:name,username',
```

```
]);
```

In general, you should always specify the array keys that are allowed to be present within your array.

ascii

The field under validation must be entirely 7-bit ASCII characters.

bail

Stop running validation rules for the field after the first validation failure.

While the `bail` rule will only stop validating a specific field when it encounters a validation failure, the `stopOnFirstFailure` method will inform the validator that it should stop validating all attributes once a single validation failure has occurred:

```
if ($validator->stopOnFirstFailure()->fails()) {
    // ...
}
```

before:*date*

The field under validation must be a value preceding the given date. The dates will be passed into the PHP `strtotime` function in order to be converted into a valid `DateTime` instance. In addition, like the `after` rule, the name of another field under validation may be supplied as the value of `date`.

before_or_equal:*date*

The field under validation must be a value preceding or equal to the given date. The dates will be passed into the PHP `strtotime` function in order to be converted into a valid `DateTime`

instance. In addition, like the `after` rule, the name of another field under validation may be supplied as the value of `date`.

between:*min*,*max*

The field under validation must have a size between the given *min* and *max* (inclusive). Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

boolean

The field under validation must be able to be cast as a boolean. Accepted input are `true`, `false`, `1`, `0`, `"1"`, and `"0"`.

confirmed

The field under validation must have a matching field of `{field}_confirmation`. For example, if the field under validation is `password`, a matching `password_confirmation` field must be present in the input.

contains:*foo*,*bar*,...

The field under validation must be an array that contains all of the given parameter values.

current_password

The field under validation must match the authenticated user's password. You may specify an authentication guard using the rule's first parameter:

```
'password' => 'current_password:api'
```

date

The field under validation must be a valid, non-relative date according to the `strtotime` PHP function.

date_equals:*date*

The field under validation must be equal to the given date. The dates will be passed into the PHP `strtotime` function in order to be converted into a valid `DateTime` instance.

date_format:*format*,...

The field under validation must match one of the given *formats*. You should use either `date` or `date_format` when validating a field, not both. This validation rule supports all formats supported by PHP's DateTime class.

decimal:*min*,*max*

The field under validation must be numeric and must contain the specified number of decimal places:

```
// Must have exactly two decimal places (9.99)...

'price' => 'decimal:2'
```

```
// Must have between 2 and 4 decimal places...

'price' => 'decimal:2,4'
```

declined

The field under validation must be `"no"`, `"off"`, `0`, `"0"`, `false`, or `"false"`.

declined_if:anotherfield,value,...

The field under validation must be `"no"`, `"off"`, `0`, `"0"`, `false`, or `"false"` if another field under validation is equal to a specified value.

different:*field*

The field under validation must have a different value than *field*.

digits:*value*

The integer under validation must have an exact length of *value*.

digits_between:*min*,*max*

The integer validation must have a length between the given *min* and *max*.

dimensions

The file under validation must be an image meeting the dimension constraints as specified by the rule's parameters:

```
'avatar' => 'dimensions:min_width=100,min_height=200'
```

Available constraints are: *min_width*, *max_width*, *min_height*, *max_height*, *width*, *height*, *ratio*.

A *ratio* constraint should be represented as width divided by height. This can be specified either by a fraction like `3/2` or a float like `1.5`:

```
'avatar' => 'dimensions:ratio=3/2'
```

Since this rule requires several arguments, you may use the `Rule::dimensions` method to fluently construct the rule:

```
use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;



Validator::make($data, [
```

```
    'avatar' => [
        'required',
        Rule::dimensions()->maxWidth(1000)->maxHeight(500)->ratio(3 / 2),
    ],
]);
```

distinct

When validating arrays, the field under validation must not have any duplicate values:

```
'foo.*.id' => 'distinct'
```

Distinct uses loose variable comparisons by default. To use strict comparisons, you may add the `strict` parameter to your validation rule definition:

```
'foo.*.id' => 'distinct:strict'
```

You may add `ignore_case` to the validation rule's arguments to make the rule ignore capitalization differences:

```
'foo.*.id' => 'distinct:ignore_case'
```

doesnt_start_with:*foo*,*bar*,...

The field under validation must not start with one of the given values.

doesnt_end_with:*foo*,*bar*,...

The field under validation must not end with one of the given values.

email

The field under validation must be formatted as an email address. This validation rule utilizes the `egulias/email-validator` package for validating the email address. By default, the `RFCValidation` validator is applied, but you can apply other validation styles as well:

```
'email' => 'email:rfc,dns'
```

The example above will apply the `RFCValidation` and `DNSCheckValidation` validations. Here's a full list of validation styles you can apply:

- `rfc`: `RFCValidation`
- `strict`: `NoRFCWarningsValidation`
- `dns`: `DNSCheckValidation`
- `spoof`: `SpoofCheckValidation`
- `filter`: `FilterEmailValidation`
- `filter_unicode`: `FilterEmailValidation::unicode()`

The `filter` validator, which uses PHP's `filter_var` function, ships with Laravel and was Laravel's default email validation behavior prior to Laravel version 5.8.

> The `dns` and `spoof` validators require the PHP `intl` extension.

ends_with:*foo*,*bar*,...

The field under validation must end with one of the given values.

enum

The `Enum` rule is a class based rule that validates whether the field under validation contains a valid enum value. The `Enum` rule accepts the name of the enum as its only constructor

argument. When validating primitive values, a backed Enum should be provided to the `Enum` rule:

```
use App\Enums\ServerStatus;

use Illuminate\Validation\Rule;



$request->validate([

    'status' => [Rule::enum(ServerStatus::class)],

]);
```

The `Enum` rule's `only` and `except` methods may be used to limit which enum cases should be considered valid:

```
Rule::enum(ServerStatus::class)

    ->only([ServerStatus::Pending, ServerStatus::Active]);



Rule::enum(ServerStatus::class)

    ->except([ServerStatus::Pending, ServerStatus::Active]);
```

The `when` method may be used to conditionally modify the `Enum` rule:

```
use Illuminate\Support\Facades\Auth;
```

```
use Illuminate\Validation\Rule;
```

```
Rule::enum(ServerStatus::class)
    ->when(
        Auth::user()->isAdmin(),
        fn ($rule) => $rule->only(...),
        fn ($rule) => $rule->only(...),
    );
```

exclude

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods.

exclude_if:*anotherfield*,*value*

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods if the *anotherfield* field is equal to *value*.

If complex conditional exclusion logic is required, you may utilize the `Rule::excludeIf` method. This method accepts a boolean or a closure. When given a closure, the closure should return `true` or `false` to indicate if the field under validation should be excluded:

```
use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;
```

```
Validator::make($request->all(), [
    'role_id' => Rule::excludeIf($request->user()->is_admin),
]);
```

```
Validator::make($request->all(), [
    'role_id' => Rule::excludeIf(fn () => $request->user()->is_admin),
]);
```

exclude_unless:*anotherfield*,*value*

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods unless *anotherfield*'s field is equal to *value*. If *value* is `null` (`exclude_unless:name,null`), the field under validation will be excluded unless the comparison field is `null` or the comparison field is missing from the request data.

exclude_with:*anotherfield*

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods if the *anotherfield* field is present.

exclude_without:*anotherfield*

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods if the *anotherfield* field is not present.

exists:*table*,*column*

The field under validation must exist in a given database table.

Basic Usage of Exists Rule

```
'state' => 'exists:states'
```

If the `column` option is not specified, the field name will be used. So, in this case, the rule will validate that the `states` database table contains a record with a `state` column value matching the request's `state` attribute value.

Specifying a Custom Column Name

You may explicitly specify the database column name that should be used by the validation rule by placing it after the database table name:

```
'state' => 'exists:states,abbreviation'
```

Occasionally, you may need to specify a specific database connection to be used for the `exists` query. You can accomplish this by prepending the connection name to the table name:

```
'email' => 'exists:connection.staff,email'
```

Instead of specifying the table name directly, you may specify the Eloquent model which should be used to determine the table name:

```
'user_id' => 'exists:App\Models\User,id'
```

If you would like to customize the query executed by the validation rule, you may use the `Rule` class to fluently define the rule. In this example, we'll also specify the validation rules as an array instead of using the `|` character to delimit them:

```
use Illuminate\Database\Query\Builder;

use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;



Validator::make($data, [

    'email' => [

        'required',

        Rule::exists('staff')->where(function (Builder $query) {

            return $query->where('account_id', 1);

        }),

    ],

]);
```

You may explicitly specify the database column name that should be used by the `exists` rule generated by the `Rule::exists` method by providing the column name as the second argument to the `exists` method:

```
'state' => Rule::exists('states', 'abbreviation'),
```

extensions:*foo*,*bar*,...

The file under validation must have a user-assigned extension corresponding to one of the listed extensions:

```
'photo' => ['required', 'extensions:jpg,png'],
```

> You should never rely on validating a file by its user-assigned extension alone. This rule should typically always be used in combination with the `mimes` or `mimetypes` rules.

file

The field under validation must be a successfully uploaded file.

filled

The field under validation must not be empty when it is present.

gt:*field*

The field under validation must be greater than the given *field* or *value*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

gte:*field*

The field under validation must be greater than or equal to the given *field* or *value*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

hex_color

The field under validation must contain a valid color value in hexadecimal format.

image

The file under validation must be an image (jpg, jpeg, png, bmp, gif, svg, or webp).

in:*foo*,*bar*,...

The field under validation must be included in the given list of values. Since this rule often requires you to `implode` an array, the `Rule::in` method may be used to fluently construct the rule:

```
use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;



Validator::make($data, [

    'zones' => [

        'required',

        Rule::in(['first-zone', 'second-zone']),

    ],

]);
```

When the `in` rule is combined with the `array` rule, each value in the input array must be present within the list of values provided to the `in` rule. In the following example, the `LAS` airport code in the input array is invalid since it is not contained in the list of airports provided to the `in` rule:

```
use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;
```

```
$input = [

    'airports' => ['NYC', 'LAS'],

];



Validator::make($input, [

    'airports' => [

        'required',

        'array',

    ],

    'airports.*' => Rule::in(['NYC', 'LIT']),

]);
```

in_array:*anotherfield*.*

The field under validation must exist in *anotherfield*'s values.

integer

The field under validation must be an integer.

This validation rule does not verify that the input is of the "integer" variable type, only that the input is of a type accepted by PHP's `FILTER_VALIDATE_INT` rule. If you need to validate the input as being a number please use this rule in combination with the `numeric` validation rule.

ip

The field under validation must be an IP address.

ipv4

The field under validation must be an IPv4 address.

ipv6

The field under validation must be an IPv6 address.

json

The field under validation must be a valid JSON string.

lt:*field*

The field under validation must be less than the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

lte:*field*

The field under validation must be less than or equal to the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

lowercase

The field under validation must be lowercase.

list

The field under validation must be an array that is a list. An array is considered a list if its keys consist of consecutive numbers from 0 to `count($array) - 1`.

mac_address

The field under validation must be a MAC address.

max:*value*

The field under validation must be less than or equal to a maximum *value*. Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

max_digits:*value*

The integer under validation must have a maximum length of *value*.

mimetypes:*text/plain*,...

The file under validation must match one of the given MIME types:

```
'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'
```

To determine the MIME type of the uploaded file, the file's contents will be read and the framework will attempt to guess the MIME type, which may be different from the client's provided MIME type.

mimes:*foo*,*bar*,...

The file under validation must have a MIME type corresponding to one of the listed extensions:

```
'photo' => 'mimes:jpg,bmp,png'
```

Even though you only need to specify the extensions, this rule actually validates the MIME type of the file by reading the file's contents and guessing its MIME type. A full listing of MIME types and their corresponding extensions may be found at the following location:

https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types

MIME Types and Extensions

This validation rule does not verify agreement between the MIME type and the extension the user assigned to the file. For example, the `mimes:png` validation rule would consider a file containing valid PNG content to be a valid PNG image, even if the file is named `photo.txt`. If you would like to validate the user-assigned extension of the file, you may use the `extensions` rule.

min:*value*

The field under validation must have a minimum *value*. Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

min_digits:*value*

The integer under validation must have a minimum length of *value*.

multiple_of:*value*

The field under validation must be a multiple of *value*.

missing

The field under validation must not be present in the input data.

missing_if:*anotherfield*,*value*,...

The field under validation must not be present if the *anotherfield* field is equal to any *value*.

missing_unless:*anotherfield*,*value*

The field under validation must not be present unless the *anotherfield* field is equal to any *value*.

missing_with:*foo*,*bar*,...

The field under validation must not be present *only if* any of the other specified fields are present.

missing_with_all:*foo*,*bar*,...

The field under validation must not be present *only if* all of the other specified fields are present.

not_in:*foo*,*bar*,...

The field under validation must not be included in the given list of values. The `Rule::notIn` method may be used to fluently construct the rule:

```
use Illuminate\Validation\Rule;



Validator::make($data, [
    'toppings' => [
        'required',
        Rule::notIn(['sprinkles', 'cherries']),
    ],
]);
```

not_regex:*pattern*

The field under validation must not match the given regular expression.

Internally, this rule uses the PHP `preg_match` function. The pattern specified should obey the same formatting required by `preg_match` and thus also include valid delimiters. For example: `'email' => 'not_regex:/^.+$/i'`.

> When using the `regex` / `not_regex` patterns, it may be necessary to specify your validation rules using an array instead of using `|` delimiters, especially if the regular expression contains a `|` character.

nullable

The field under validation may be `null`.

numeric

The field under validation must be numeric.

present

The field under validation must exist in the input data.

present_if:*anotherfield*,*value*,...

The field under validation must be present if the *anotherfield* field is equal to any *value*.

present_unless:*anotherfield*,*value*

The field under validation must be present unless the *anotherfield* field is equal to any *value*.

present_with:*foo*,*bar*,...

The field under validation must be present *only if* any of the other specified fields are present.

present_with_all:*foo*,*bar*,...

The field under validation must be present *only if* all of the other specified fields are present.

prohibited

The field under validation must be missing or empty. A field is "empty" if it meets one of the following criteria:

- The value is `null`.
- The value is an empty string.
- The value is an empty array or empty `Countable` object.
- The value is an uploaded file with an empty path.

prohibited_if:*anotherfield*,*value*,...

The field under validation must be missing or empty if the *anotherfield* field is equal to any *value*. A field is "empty" if it meets one of the following criteria:

- The value is `null`.
- The value is an empty string.
- The value is an empty array or empty `Countable` object.
- The value is an uploaded file with an empty path.

If complex conditional prohibition logic is required, you may utilize the `Rule::prohibitedIf` method. This method accepts a boolean or a closure. When given a closure, the closure should return `true` or `false` to indicate if the field under validation should be prohibited:

```
use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;



Validator::make($request->all(), [

    'role_id' => Rule::prohibitedIf($request->user()->is_admin),
```

```
]);
```

```
Validator::make($request->all(), [

    'role_id' => Rule::prohibitedIf(fn () =>
$request->user()->is_admin),

]);
```

prohibited_unless:*anotherfield*,*value*,...

The field under validation must be missing or empty unless the *anotherfield* field is equal to any *value*. A field is "empty" if it meets one of the following criteria:

- The value is `null`.
- The value is an empty string.
- The value is an empty array or empty `Countable` object.
- The value is an uploaded file with an empty path.

prohibits:*anotherfield*,...

If the field under validation is not missing or empty, all fields in *anotherfield* must be missing or empty. A field is "empty" if it meets one of the following criteria:

- The value is `null`.
- The value is an empty string.
- The value is an empty array or empty `Countable` object.
- The value is an uploaded file with an empty path.

regex:*pattern*

The field under validation must match the given regular expression.

Internally, this rule uses the PHP `preg_match` function. The pattern specified should obey the same formatting required by `preg_match` and thus also include valid delimiters. For example: `'email' => 'regex:/^.+@.+$/i'`.

> When using the `regex` / `not_regex` patterns, it may be necessary to specify rules in an array instead of using `|` delimiters, especially if the regular expression contains a `|` character.

required

The field under validation must be present in the input data and not empty. A field is "empty" if it meets one of the following criteria:

- The value is `null`.
- The value is an empty string.
- The value is an empty array or empty `Countable` object.
- The value is an uploaded file with no path.

required_if:*anotherfield*,*value*,...

The field under validation must be present and not empty if the *anotherfield* field is equal to any *value*.

If you would like to construct a more complex condition for the `required_if` rule, you may use the `Rule::requiredIf` method. This method accepts a boolean or a closure. When passed a closure, the closure should return `true` or `false` to indicate if the field under validation is required:

```
use Illuminate\Support\Facades\Validator;
```

```php
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::requiredIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::requiredIf(fn () =>
$request->user()->is_admin),
]);
```

required_if_accepted:*anotherfield*,...

The field under validation must be present and not empty if the *anotherfield* field is equal to `"yes"`, `"on"`, `1`, `"1"`, `true`, or `"true"`.

required_if_declined:*anotherfield*,...

The field under validation must be present and not empty if the *anotherfield* field is equal to `"no"`, `"off"`, `0`, `"0"`, `false`, or `"false"`.

required_unless:*anotherfield*,*value*,...

The field under validation must be present and not empty unless the *anotherfield* field is equal to any *value*. This also means *anotherfield* must be present in the request data unless *value* is `null`. If *value* is `null` (`required_unless:name,null`), the field under validation will be

required unless the comparison field is `null` or the comparison field is missing from the request data.

required_with:*foo*,*bar*,...

The field under validation must be present and not empty *only if* any of the other specified fields are present and not empty.

required_with_all:*foo*,*bar*,...

The field under validation must be present and not empty *only if* all of the other specified fields are present and not empty.

required_without:*foo*,*bar*,...

The field under validation must be present and not empty *only when* any of the other specified fields are empty or not present.

required_without_all:*foo*,*bar*,...

The field under validation must be present and not empty *only when* all of the other specified fields are empty or not present.

required_array_keys:*foo*,*bar*,...

The field under validation must be an array and must contain at least the specified keys.

same:*field*

The given *field* must match the field under validation.

size:*value*

The field under validation must have a size matching the given *value*. For string data, *value* corresponds to the number of characters. For numeric data, *value* corresponds to a given integer value (the attribute must also have the `numeric` or `integer` rule). For an array, *size*

corresponds to the `count` of the array. For files, *size* corresponds to the file size in kilobytes.

Let's look at some examples:

```
// Validate that a string is exactly 12 characters long...

'title' => 'size:12';
```

```
// Validate that a provided integer equals 10...

'seats' => 'integer|size:10';
```

```
// Validate that an array has exactly 5 elements...

'tags' => 'array|size:5';
```

```
// Validate that an uploaded file is exactly 512 kilobytes...

'image' => 'file|size:512';
```

starts_with:*foo*,*bar*,...

The field under validation must start with one of the given values.

string

The field under validation must be a string. If you would like to allow the field to also be `null`, you should assign the `nullable` rule to the field.

timezone

The field under validation must be a valid timezone identifier according to the `DateTimeZone::listIdentifiers` method.

The arguments accepted by the `DateTimeZone::listIdentifiers` method may also be provided to this validation rule:

```
'timezone' => 'required|timezone:all';
```

```
'timezone' => 'required|timezone:Africa';
```

```
'timezone' => 'required|timezone:per_country,US';
```

unique:*table*,*column*

The field under validation must not exist within the given database table.

Specifying a Custom Table / Column Name:

Instead of specifying the table name directly, you may specify the Eloquent model which should be used to determine the table name:

```
'email' => 'unique:App\Models\User,email_address'
```

The `column` option may be used to specify the field's corresponding database column. If the `column` option is not specified, the name of the field under validation will be used.

```
'email' => 'unique:users,email_address'
```

Specifying a Custom Database Connection

Occasionally, you may need to set a custom connection for database queries made by the Validator. To accomplish this, you may prepend the connection name to the table name:

```
'email' => 'unique:connection.users,email_address'
```

Forcing a Unique Rule to Ignore a Given ID:

Sometimes, you may wish to ignore a given ID during unique validation. For example, consider an "update profile" screen that includes the user's name, email address, and location. You will probably want to verify that the email address is unique. However, if the user only changes the name field and not the email field, you do not want a validation error to be thrown because the user is already the owner of the email address in question.

To instruct the validator to ignore the user's ID, we'll use the `Rule` class to fluently define the rule. In this example, we'll also specify the validation rules as an array instead of using the `|` character to delimit the rules:

```
use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;



Validator::make($data, [

    'email' => [

        'required',
```

```
        Rule::unique('users')->ignore($user->id),
    ],
]);
```

> You should never pass any user controlled request input into the `ignore` method. Instead, you should only pass a system generated unique ID such as an auto-incrementing ID or UUID from an Eloquent model instance. Otherwise, your application will be vulnerable to an SQL injection attack.

Instead of passing the model key's value to the `ignore` method, you may also pass the entire model instance. Laravel will automatically extract the key from the model:

```
Rule::unique('users')->ignore($user)
```

If your table uses a primary key column name other than `id`, you may specify the name of the column when calling the `ignore` method:

```
Rule::unique('users')->ignore($user->id, 'user_id')
```

By default, the `unique` rule will check the uniqueness of the column matching the name of the attribute being validated. However, you may pass a different column name as the second argument to the `unique` method:

```
Rule::unique('users', 'email_address')->ignore($user->id)
```

Adding Additional Where Clauses:

You may specify additional query conditions by customizing the query using the `where` method. For example, let's add a query condition that scopes the query to only search records that have an `account_id` column value of `1`:

```
'email' => Rule::unique('users')->where(fn (Builder $query) =>
$query->where('account_id', 1))
```

uppercase

The field under validation must be uppercase.

url

The field under validation must be a valid URL.

If you would like to specify the URL protocols that should be considered valid, you may pass the protocols as validation rule parameters:

```
'url' => 'url:http,https',
```

```
'game' => 'url:minecraft,steam',
```

ulid

The field under validation must be a valid Universally Unique Lexicographically Sortable Identifier (ULID).

uuid

The field under validation must be a valid RFC 4122 (version 1, 3, 4, or 5) universally unique identifier (UUID).

Conditionally Adding Rules

Skipping Validation When Fields Have Certain Values

You may occasionally wish to not validate a given field if another field has a given value. You may accomplish this using the `exclude_if` validation rule. In this example, the `appointment_date` and `doctor_name` fields will not be validated if the `has_appointment` field has a value of `false`:

```
use Illuminate\Support\Facades\Validator;



$validator = Validator::make($data, [

    'has_appointment' => 'required|boolean',

    'appointment_date' =>
'exclude_if:has_appointment,false|required|date',

    'doctor_name' =>
'exclude_if:has_appointment,false|required|string',

]);
```

Alternatively, you may use the `exclude_unless` rule to not validate a given field unless another field has a given value:

```
$validator = Validator::make($data, [
```

```
    'has_appointment' => 'required|boolean',

    'appointment_date' =>
'exclude_unless:has_appointment,true|required|date',

    'doctor_name' =>
'exclude_unless:has_appointment,true|required|string',

]);
```

Validating When Present

In some situations, you may wish to run validation checks against a field only if that field is present in the data being validated. To quickly accomplish this, add the `sometimes` rule to your rule list:

```
$validator = Validator::make($data, [

    'email' => 'sometimes|required|email',

]);
```

In the example above, the `email` field will only be validated if it is present in the `$data` array.

> If you are attempting to validate a field that should always be present but may be empty, check out this note on optional fields.

Complex Conditional Validation

Sometimes you may wish to add validation rules based on more complex conditional logic. For example, you may wish to require a given field only if another field has a greater value than 100. Or, you may need two fields to have a given value only when another field is present. Adding

these validation rules doesn't have to be a pain. First, create a `Validator` instance with your *static rules* that never change:

```
use Illuminate\Support\Facades\Validator;


$validator = Validator::make($request->all(), [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

Let's assume our web application is for game collectors. If a game collector registers with our application and they own more than 100 games, we want them to explain why they own so many games. For example, perhaps they run a game resale shop, or maybe they just enjoy collecting games. To conditionally add this requirement, we can use the `sometimes` method on the `Validator` instance.

```
use Illuminate\Support\Fluent;


$validator->sometimes('reason', 'required|max:500', function (Fluent $input) {
    return $input->games >= 100;
});
```

The first argument passed to the `sometimes` method is the name of the field we are conditionally validating. The second argument is a list of the rules we want to add. If the closure passed as the third argument returns `true`, the rules will be added. This method makes it a breeze to build complex conditional validations. You may even add conditional validations for several fields at once:

```
$validator->sometimes(['reason', 'cost'], 'required', function
(Fluent $input) {

    return $input->games >= 100;

});
```

> The `$input` parameter passed to your closure will be an instance of `Illuminate\Support\Fluent` and may be used to access your input and files under validation.

Complex Conditional Array Validation

Sometimes you may want to validate a field based on another field in the same nested array whose index you do not know. In these situations, you may allow your closure to receive a second argument which will be the current individual item in the array being validated:

```
$input = [

    'channels' => [

        [

            'type' => 'email',
```

```
            'address' => 'abigail@example.com',
        ],
        [
            'type' => 'url',
            'address' => 'https://example.com',
        ],
    ],
];

$validator->sometimes('channels.*.address', 'email', function (Fluent
$input, Fluent $item) {
    return $item->type === 'email';
});

$validator->sometimes('channels.*.address', 'url', function (Fluent
$input, Fluent $item) {
    return $item->type !== 'email';
```

```
});
```

Like the `$input` parameter passed to the closure, the `$item` parameter is an instance of `Illuminate\Support\Fluent` when the attribute data is an array; otherwise, it is a string.

Validating Arrays

As discussed in the `array` validation rule documentation, the `array` rule accepts a list of allowed array keys. If any additional keys are present within the array, validation will fail:

```
use Illuminate\Support\Facades\Validator;


$input = [
    'user' => [
        'name' => 'Taylor Otwell',
        'username' => 'taylorotwell',
        'admin' => true,
    ],
];
```

```
Validator::make($input, [
    'user' => 'array:name,username',
]);
```

In general, you should always specify the array keys that are allowed to be present within your array. Otherwise, the validator's `validate` and `validated` methods will return all of the validated data, including the array and all of its keys, even if those keys were not validated by other nested array validation rules.

Validating Nested Array Input

Validating nested array based form input fields doesn't have to be a pain. You may use "dot notation" to validate attributes within an array. For example, if the incoming HTTP request contains a `photos[profile]` field, you may validate it like so:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
    'photos.profile' => 'required|image',
]);
```

You may also validate each element of an array. For example, to validate that each email in a given array input field is unique, you may do the following:

```
$validator = Validator::make($request->all(), [
    'person.*.email' => 'email|unique:users',
    'person.*.first_name' => 'required_with:person.*.last_name',
]);
```

Likewise, you may use the `*` character when specifying custom validation messages in your language files, making it a breeze to use a single validation message for array based fields:

```
'custom' => [
    'person.*.email' => [
        'unique' => 'Each person must have a unique email address',
    ]
],
```

Accessing Nested Array Data

Sometimes you may need to access the value for a given nested array element when assigning validation rules to the attribute. You may accomplish this using the `Rule::forEach` method. The `forEach` method accepts a closure that will be invoked for each iteration of the array attribute under validation and will receive the attribute's value and explicit, fully-expanded attribute name. The closure should return an array of rules to assign to the array element:

```
use App\Rules\HasPermission;

use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rule;



$validator = Validator::make($request->all(), [

    'companies.*.id' => Rule::forEach(function (string|null $value,
string $attribute) {

        return [

            Rule::exists(Company::class, 'id'),

            new HasPermission('manage-company', $value),

        ];

    }),

]);
```

Error Message Indexes and Positions

When validating arrays, you may want to reference the index or position of a particular item that failed validation within the error message displayed by your application. To accomplish this, you may include the `:index` (starts from `0`) and `:position` (starts from `1`) placeholders within your custom validation message:

```php
use Illuminate\Support\Facades\Validator;


$input = [
    'photos' => [
        [
            'name' => 'BeachVacation.jpg',
            'description' => 'A photo of my beach vacation!',
        ],
        [
            'name' => 'GrandCanyon.jpg',
            'description' => '',
        ],
    ],
];


Validator::validate($input, [
```

```
    'photos.*.description' => 'required',

], [

    'photos.*.description.required' => 'Please describe photo
#:position.',

]);
```

Given the example above, validation will fail and the user will be presented with the following error of *"Please describe photo #2."*

If necessary, you may reference more deeply nested indexes and positions via `second-index`, `second-position`, `third-index`, `third-position`, etc.

```
'photos.*.attributes.*.string' => 'Invalid attribute for photo
#:second-position.',
```

Validating Files

Laravel provides a variety of validation rules that may be used to validate uploaded files, such as `mimes`, `image`, `min`, and `max`. While you are free to specify these rules individually when validating files, Laravel also offers a fluent file validation rule builder that you may find convenient:

```
use Illuminate\Support\Facades\Validator;
```

```
use Illuminate\Validation\Rules\File;



Validator::validate($input, [
    'attachment' => [
        'required',
        File::types(['mp3', 'wav'])
            ->min(1024)
            ->max(12 * 1024),
    ],
]);
```

If your application accepts images uploaded by your users, you may use the `File` rule's `image` constructor method to indicate that the uploaded file should be an image. In addition, the `dimensions` rule may be used to limit the dimensions of the image:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;
use Illuminate\Validation\Rules\File;
```

```
Validator::validate($input, [
    'photo' => [
        'required',
        File::image()
            ->min(1024)
            ->max(12 * 1024)

->dimensions(Rule::dimensions()->maxWidth(1000)->maxHeight(500)),
    ],
]);
```

> More information regarding validating image dimensions may be found in the dimension rule documentation.

File Sizes

For convenience, minimum and maximum file sizes may be specified as a string with a suffix indicating the file size units. The `kb`, `mb`, `gb`, and `tb` suffixes are supported:

```
File::image()
    ->min('1kb')
    ->max('10mb')
```

File Types

Even though you only need to specify the extensions when invoking the `types` method, this method actually validates the MIME type of the file by reading the file's contents and guessing its MIME type. A full listing of MIME types and their corresponding extensions may be found at the following location:

https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types

Validating Passwords

To ensure that passwords have an adequate level of complexity, you may use Laravel's `Password` rule object:

```
use Illuminate\Support\Facades\Validator;

use Illuminate\Validation\Rules\Password;



$validator = Validator::make($request->all(), [

    'password' => ['required', 'confirmed', Password::min(8)],

]);
```

The `Password` rule object allows you to easily customize the password complexity requirements for your application, such as specifying that passwords require at least one letter, number, symbol, or characters with mixed casing:

```php
// Require at least 8 characters...
Password::min(8)

// Require at least one letter...
Password::min(8)->letters()

// Require at least one uppercase and one lowercase letter...
Password::min(8)->mixedCase()

// Require at least one number...
Password::min(8)->numbers()

// Require at least one symbol...
Password::min(8)->symbols()
```

In addition, you may ensure that a password has not been compromised in a public password data breach leak using the `uncompromised` method:

```php
Password::min(8)->uncompromised()
```

Internally, the `Password` rule object uses the k-Anonymity model to determine if a password has been leaked via the haveibeenpwned.com service without sacrificing the user's privacy or security.

By default, if a password appears at least once in a data leak, it will be considered compromised. You can customize this threshold using the first argument of the `uncompromised` method:

```
// Ensure the password appears less than 3 times in the same data leak...

Password::min(8)->uncompromised(3);
```

Of course, you may chain all the methods in the examples above:

```
Password::min(8)

    ->letters()

    ->mixedCase()

    ->numbers()

    ->symbols()

    ->uncompromised()
```

Defining Default Password Rules

You may find it convenient to specify the default validation rules for passwords in a single location of your application. You can easily accomplish this using the `Password::defaults`

method, which accepts a closure. The closure given to the `defaults` method should return the default configuration of the Password rule. Typically, the `defaults` rule should be called within the `boot` method of one of your application's service providers:

```
use Illuminate\Validation\Rules\Password;



/**

* Bootstrap any application services.

*/

public function boot(): void

{

    Password::defaults(function () {

        $rule = Password::min(8);



        return $this->app->isProduction()

                    ? $rule->mixedCase()->uncompromised()

                    : $rule;

    });
```

```
}
```

Then, when you would like to apply the default rules to a particular password undergoing validation, you may invoke the `defaults` method with no arguments:

```
'password' => ['required', Password::defaults()],
```

Occasionally, you may want to attach additional validation rules to your default password validation rules. You may use the `rules` method to accomplish this:

```
use App\Rules\ZxcvbnRule;


Password::defaults(function () {
    $rule = Password::min(8)->rules([new ZxcvbnRule]);

    // ...
});
```

Custom Validation Rules

Using Rule Objects

Laravel provides a variety of helpful validation rules; however, you may wish to specify some of your own. One method of registering custom validation rules is using rule objects. To generate a new rule object, you may use the `make:rule` Artisan command. Let's use this command to generate a rule that verifies a string is uppercase. Laravel will place the new rule in the `app/Rules` directory. If this directory does not exist, Laravel will create it when you execute the Artisan command to create your rule:

```
php artisan make:rule Uppercase
```

Once the rule has been created, we are ready to define its behavior. A rule object contains a single method: `validate`. This method receives the attribute name, its value, and a callback that should be invoked on failure with the validation error message:

```
<?php



namespace App\Rules;



use Closure;

use Illuminate\Contracts\Validation\ValidationRule;



class Uppercase implements ValidationRule

{
```

```
    /**
     * Run the validation rule.
     */
    public function validate(string $attribute, mixed $value, Closure
$fail): void
    {
        if (strtoupper($value) !== $value) {
            $fail('The :attribute must be uppercase.');
        }
    }
}
```

Once the rule has been defined, you may attach it to a validator by passing an instance of the rule object with your other validation rules:

```
use App\Rules\Uppercase;

$request->validate([
    'name' => ['required', 'string', new Uppercase],
```

```
]);
```

Translating Validation Messages

Instead of providing a literal error message to the `$fail` closure, you may also provide a translation string key and instruct Laravel to translate the error message:

```
if (strtoupper($value) !== $value) {
    $fail('validation.uppercase')->translate();
}
```

If necessary, you may provide placeholder replacements and the preferred language as the first and second arguments to the `translate` method:

```
$fail('validation.location')->translate([
    'value' => $this->value,
], 'fr')
```

Accessing Additional Data

If your custom validation rule class needs to access all of the other data undergoing validation, your rule class may implement the `Illuminate\Contracts\Validation\DataAwareRule` interface. This interface requires your class to define a `setData` method. This method will automatically be invoked by Laravel (before validation proceeds) with all of the data under validation:

```
<?php
```

```php
namespace App\Rules;

use Illuminate\Contracts\Validation\DataAwareRule;
use Illuminate\Contracts\Validation\ValidationRule;

class Uppercase implements DataAwareRule, ValidationRule
{
    /**
     * All of the data under validation.
     *
     * @var array<string, mixed>
     */
    protected $data = [];

    // ...
```

```
    /**
     * Set the data under validation.
     *
     * @param  array<string, mixed>  $data
     */
    public function setData(array $data): static
    {
        $this->data = $data;

        return $this;
    }
}
```

Or, if your validation rule requires access to the validator instance performing the validation, you may implement the `ValidatorAwareRule` interface:

```
<?php
```

```php
namespace App\Rules;

use Illuminate\Contracts\Validation\ValidationRule;
use Illuminate\Contracts\Validation\ValidatorAwareRule;
use Illuminate\Validation\Validator;

class Uppercase implements ValidationRule, ValidatorAwareRule
{
    /**
     * The validator instance.
     *
     * @var \Illuminate\Validation\Validator
     */
    protected $validator;

    // ...
```

```
    /**
     * Set the current validator.
     */
    public function setValidator(Validator $validator): static
    {
        $this->validator = $validator;

        return $this;
    }
}
```

Using Closures

If you only need the functionality of a custom rule once throughout your application, you may use a closure instead of a rule object. The closure receives the attribute's name, the attribute's value, and a `$fail` callback that should be called if validation fails:

```
use Illuminate\Support\Facades\Validator;

use Closure;
```

```
$validator = Validator::make($request->all(), [
    'title' => [
        'required',
        'max:255',
        function (string $attribute, mixed $value, Closure $fail) {
            if ($value === 'foo') {
                $fail("The {$attribute} is invalid.");
            }
        },
    ],
]);
```

Implicit Rules

By default, when an attribute being validated is not present or contains an empty string, normal validation rules, including custom rules, are not run. For example, the `unique` rule will not be run against an empty string:

```
use Illuminate\Support\Facades\Validator;
```

```
$rules = ['name' => 'unique:users,name'];
```

```
$input = ['name' => ''];
```

```
Validator::make($input, $rules)->passes(); // true
```

For a custom rule to run even when an attribute is empty, the rule must imply that the attribute is required. To quickly generate a new implicit rule object, you may use the `make:rule` Artisan command with the `--implicit` option:

```
php artisan make:rule Uppercase --implicit
```

> An "implicit" rule only *implies* that the attribute is required. Whether it actually invalidates a missing or empty attribute is up to you.

Introduction

Laravel includes Eloquent, an object-relational mapper (ORM) that makes it enjoyable to interact with your database. When using Eloquent, each database table has a corresponding

"Model" that is used to interact with that table. In addition to retrieving records from the database table, Eloquent models allow you to insert, update, and delete records from the table as well.

> Before getting started, be sure to configure a database connection in your application's `config/database.php` configuration file. For more information on configuring your database, check out the database configuration documentation.

Laravel Bootcamp

If you're new to Laravel, feel free to jump into the Laravel Bootcamp. The Laravel Bootcamp will walk you through building your first Laravel application using Eloquent. It's a great way to get a tour of everything that Laravel and Eloquent have to offer.

Generating Model Classes

To get started, let's create an Eloquent model. Models typically live in the `app\Models` directory and extend the `Illuminate\Database\Eloquent\Model` class. You may use the `make:model` Artisan command to generate a new model:

```
php artisan make:model Flight
```

If you would like to generate a database migration when you generate the model, you may use the `--migration` or `-m` option:

```
php artisan make:model Flight --migration
```

You may generate various other types of classes when generating a model, such as factories, seeders, policies, controllers, and form requests. In addition, these options may be combined to create multiple classes at once:

```
# Generate a model and a FlightFactory class...

php artisan make:model Flight --factory

php artisan make:model Flight -f



# Generate a model and a FlightSeeder class...

php artisan make:model Flight --seed

php artisan make:model Flight -s



# Generate a model and a FlightController class...

php artisan make:model Flight --controller

php artisan make:model Flight -c



# Generate a model, FlightController resource class, and form request classes...
```

```
php artisan make:model Flight --controller --resource --requests

php artisan make:model Flight -crR

# Generate a model and a FlightPolicy class...

php artisan make:model Flight --policy

# Generate a model and a migration, factory, seeder, and
controller...

php artisan make:model Flight -mfsc

# Shortcut to generate a model, migration, factory, seeder, policy,
controller, and form requests...

php artisan make:model Flight --all

php artisan make:model Flight -a

# Generate a pivot model...

php artisan make:model Member --pivot
```

```
php artisan make:model Member -p
```

Inspecting Models

Sometimes it can be difficult to determine all of a model's available attributes and relationships just by skimming its code. Instead, try the `model:show` Artisan command, which provides a convenient overview of all the model's attributes and relations:

```
php artisan model:show Flight
```

Eloquent Model Conventions

Models generated by the `make:model` command will be placed in the `app/Models` directory. Let's examine a basic model class and discuss some of Eloquent's key conventions:

```
<?php


namespace App\Models;


use Illuminate\Database\Eloquent\Model;


class Flight extends Model
```

```
{
    // ...
}
```

Table Names

After glancing at the example above, you may have noticed that we did not tell Eloquent which database table corresponds to our `Flight` model. By convention, the "snake case", plural name of the class will be used as the table name unless another name is explicitly specified. So, in this case, Eloquent will assume the `Flight` model stores records in the `flights` table, while an `AirTrafficController` model would store records in an `air_traffic_controllers` table.

If your model's corresponding database table does not fit this convention, you may manually specify the model's table name by defining a `table` property on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
```

```
class Flight extends Model

{

    /**

     * The table associated with the model.

     *

     * @var string

     */

    protected $table = 'my_flights';

}
```

Primary Keys

Eloquent will also assume that each model's corresponding database table has a primary key column named `id`. If necessary, you may define a protected `$primaryKey` property on your model to specify a different column that serves as your model's primary key:

```
<?php



namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Model;



class Flight extends Model

{

    /**

     * The primary key associated with the table.

     *

     * @var string

     */

    protected $primaryKey = 'flight_id';

}
```

In addition, Eloquent assumes that the primary key is an incrementing integer value, which means that Eloquent will automatically cast the primary key to an integer. If you wish to use a non-incrementing or a non-numeric primary key you must define a public `$incrementing` property on your model that is set to `false`:

```
<?php
```

```
class Flight extends Model

{

    /**

     * Indicates if the model's ID is auto-incrementing.

     *

     * @var bool

     */

    public $incrementing = false;

}
```

If your model's primary key is not an integer, you should define a protected `$keyType` property on your model. This property should have a value of `string`:

```
<?php



class Flight extends Model

{

    /**
```

```
    * The data type of the primary key ID.
    *
    * @var string
    */
    protected $keyType = 'string';
}
```

"Composite" Primary Keys

Eloquent requires each model to have at least one uniquely identifying "ID" that can serve as its primary key. "Composite" primary keys are not supported by Eloquent models. However, you are free to add additional multi-column, unique indexes to your database tables in addition to the table's uniquely identifying primary key.

UUID and ULID Keys

Instead of using auto-incrementing integers as your Eloquent model's primary keys, you may choose to use UUIDs instead. UUIDs are universally unique alpha-numeric identifiers that are 36 characters long.

If you would like a model to use a UUID key instead of an auto-incrementing integer key, you may use the `Illuminate\Database\Eloquent\Concerns\HasUuids` trait on the model. Of course, you should ensure that the model has a UUID equivalent primary key column:

```
use Illuminate\Database\Eloquent\Concerns\HasUuids;
```

```php
use Illuminate\Database\Eloquent\Model;


class Article extends Model

{

    use HasUuids;


    // ...

}


$article = Article::create(['title' => 'Traveling to Europe']);


$article->id; // "8f8e8478-9035-4d23-b9a7-62f4d2612ce5"
```

By default, The `HasUuids` trait will generate "ordered" UUIDs for your models. These UUIDs are more efficient for indexed database storage because they can be sorted lexicographically.

You can override the UUID generation process for a given model by defining a `newUniqueId` method on the model. In addition, you may specify which columns should receive UUIDs by defining a `uniqueIds` method on the model:

```php
use Ramsey\Uuid\Uuid;



/**

* Generate a new UUID for the model.

*/

public function newUniqueId(): string

{

    return (string) Uuid::uuid4();

}



/**

* Get the columns that should receive a unique identifier.

*

* @return array<int, string>

*/

public function uniqueIds(): array
```

```
{
    return ['id', 'discount_code'];
}
```

If you wish, you may choose to utilize "ULIDs" instead of UUIDs. ULIDs are similar to UUIDs; however, they are only 26 characters in length. Like ordered UUIDs, ULIDs are lexicographically sortable for efficient database indexing. To utilize ULIDs, you should use the `Illuminate\Database\Eloquent\Concerns\HasUlids` trait on your model. You should also ensure that the model has a ULID equivalent primary key column:

```
use Illuminate\Database\Eloquent\Concerns\HasUlids;
use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    use HasUlids;

    // ...
}
```

```php
$article = Article::create(['title' => 'Traveling to Asia']);
```

```php
$article->id; // "01gd4d3tgrrfqeda94gdbtdk5c"
```

Timestamps

By default, Eloquent expects `created_at` and `updated_at` columns to exist on your model's corresponding database table. Eloquent will automatically set these column's values when models are created or updated. If you do not want these columns to be automatically managed by Eloquent, you should define a `$timestamps` property on your model with a value of `false`:

```php
<?php
```

```php
namespace App\Models;
```

```php
use Illuminate\Database\Eloquent\Model;
```

```php
class Flight extends Model
```

```php
{
```

```php
    /**
```

```
    * Indicates if the model should be timestamped.
    *
    * @var bool
    */
    public $timestamps = false;
}
```

If you need to customize the format of your model's timestamps, set the `$dateFormat` property on your model. This property determines how date attributes are stored in the database as well as their format when the model is serialized to an array or JSON:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
```

```
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

If you need to customize the names of the columns used to store the timestamps, you may define `CREATED_AT` and `UPDATED_AT` constants on your model:

```
<?php


class Flight extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'updated_date';
}
```

If you would like to perform model operations without the model having its `updated_at` timestamp modified, you may operate on the model within a closure given to the `withoutTimestamps` method:

```
Model::withoutTimestamps(fn () => $post->increment('reads'));
```

Database Connections

By default, all Eloquent models will use the default database connection that is configured for your application. If you would like to specify a different connection that should be used when interacting with a particular model, you should define a `$connection` property on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
```

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

Default Attribute Values

By default, a newly instantiated model instance will not contain any attribute values. If you would like to define the default values for some of your model's attributes, you may define an `$attributes` property on your model. Attribute values placed in the `$attributes` array should be in their raw, "storable" format as if they were just read from the database:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
```

```
class Flight extends Model
{
    /**
     * The model's default values for attributes.
     *
     * @var array
     */
    protected $attributes = [
        'options' => '[]',
        'delayed' => false,
    ];
}
```

Configuring Eloquent Strictness

Laravel offers several methods that allow you to configure Eloquent's behavior and "strictness" in a variety of situations.

First, the `preventLazyLoading` method accepts an optional boolean argument that indicates if lazy loading should be prevented. For example, you may wish to only disable lazy loading in

non-production environments so that your production environment will continue to function normally even if a lazy loaded relationship is accidentally present in production code. Typically, this method should be invoked in the `boot` method of your application's `AppServiceProvider`:

```
use Illuminate\Database\Eloquent\Model;



/**

* Bootstrap any application services.

*/

public function boot(): void

{

    Model::preventLazyLoading(! $this->app->isProduction());

}
```

Also, you may instruct Laravel to throw an exception when attempting to fill an unfillable attribute by invoking the `preventSilentlyDiscardingAttributes` method. This can help prevent unexpected errors during local development when attempting to set an attribute that has not been added to the model's `fillable` array:

```
Model::preventSilentlyDiscardingAttributes(!
$this->app->isProduction());
```

Retrieving Models

Once you have created a model and its associated database table, you are ready to start retrieving data from your database. You can think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model. The model's `all` method will retrieve all of the records from the model's associated database table:

```
use App\Models\Flight;



foreach (Flight::all() as $flight) {

    echo $flight->name;

}
```

Building Queries

The Eloquent `all` method will return all of the results in the model's table. However, since each Eloquent model serves as a query builder, you may add additional constraints to queries and then invoke the `get` method to retrieve the results:

```
$flights = Flight::where('active', 1)

                ->orderBy('name')

                ->take(10)

                ->get();
```

Since Eloquent models are query builders, you should review all of the methods provided by Laravel's query builder. You may use any of these methods when writing your Eloquent queries.

Refreshing Models

If you already have an instance of an Eloquent model that was retrieved from the database, you can "refresh" the model using the `fresh` and `refresh` methods. The `fresh` method will re-retrieve the model from the database. The existing model instance will not be affected:

```
$flight = Flight::where('number', 'FR 900')->first();

$freshFlight = $flight->fresh();
```

The `refresh` method will re-hydrate the existing model using fresh data from the database. In addition, all of its loaded relationships will be refreshed as well:

```
$flight = Flight::where('number', 'FR 900')->first();

$flight->number = 'FR 456';

$flight->refresh();

$flight->number; // "FR 900"
```

Collections

As we have seen, Eloquent methods like `all` and `get` retrieve multiple records from the database. However, these methods don't return a plain PHP array. Instead, an instance of `Illuminate\Database\Eloquent\Collection` is returned.

The Eloquent `Collection` class extends Laravel's base `Illuminate\Support\Collection` class, which provides a variety of helpful methods for interacting with data collections. For example, the `reject` method may be used to remove models from a collection based on the results of an invoked closure:

```
$flights = Flight::where('destination', 'Paris')->get();
```

```
$flights = $flights->reject(function (Flight $flight) {
    return $flight->cancelled;
});
```

In addition to the methods provided by Laravel's base collection class, the Eloquent collection class provides a few extra methods that are specifically intended for interacting with collections of Eloquent models.

Since all of Laravel's collections implement PHP's iterable interfaces, you may loop over collections as if they were an array:

```
foreach ($flights as $flight) {
    echo $flight->name;
```

```
}
```

Chunking Results

Your application may run out of memory if you attempt to load tens of thousands of Eloquent records via the `all` or `get` methods. Instead of using these methods, the `chunk` method may be used to process large numbers of models more efficiently.

The `chunk` method will retrieve a subset of Eloquent models, passing them to a closure for processing. Since only the current chunk of Eloquent models is retrieved at a time, the `chunk` method will provide significantly reduced memory usage when working with a large number of models:

```
use App\Models\Flight;

use Illuminate\Database\Eloquent\Collection;



Flight::chunk(200, function (Collection $flights) {

    foreach ($flights as $flight) {

        // ...

    }

});
```

The first argument passed to the `chunk` method is the number of records you wish to receive per "chunk". The closure passed as the second argument will be invoked for each chunk that is retrieved from the database. A database query will be executed to retrieve each chunk of records passed to the closure.

If you are filtering the results of the `chunk` method based on a column that you will also be updating while iterating over the results, you should use the `chunkById` method. Using the `chunk` method in these scenarios could lead to unexpected and inconsistent results. Internally, the `chunkById` method will always retrieve models with an `id` column greater than the last model in the previous chunk:

```
Flight::where('departed', true)

    ->chunkById(200, function (Collection $flights) {

        $flights->each->update(['departed' => false]);

    }, $column = 'id');
```

Since the `chunkById` and `lazyById` methods add their own "where" conditions to the query being executed, you should typically logically group your own conditions within a closure:

```
Flight::where(function ($query) {

    $query->where('delayed', true)->orWhere('cancelled', true);

})->chunkById(200, function (Collection $flights) {

    $flights->each->update([
```

```
        'departed' => false,

        'cancelled' => true

    ]);

}, column: 'id');
```

Chunking Using Lazy Collections

The `lazy` method works similarly to the `chunk` method in the sense that, behind the scenes, it executes the query in chunks. However, instead of passing each chunk directly into a callback as is, the `lazy` method returns a flattened `LazyCollection` of Eloquent models, which lets you interact with the results as a single stream:

```
use App\Models\Flight;



foreach (Flight::lazy() as $flight) {

    // ...

}
```

If you are filtering the results of the `lazy` method based on a column that you will also be updating while iterating over the results, you should use the `lazyById` method. Internally, the `lazyById` method will always retrieve models with an `id` column greater than the last model in the previous chunk:

```
Flight::where('departed', true)
    ->lazyById(200, $column = 'id')
    ->each->update(['departed' => false]);
```

You may filter the results based on the descending order of the `id` using the `lazyByIdDesc` method.

Cursors

Similar to the `lazy` method, the `cursor` method may be used to significantly reduce your application's memory consumption when iterating through tens of thousands of Eloquent model records.

The `cursor` method will only execute a single database query; however, the individual Eloquent models will not be hydrated until they are actually iterated over. Therefore, only one Eloquent model is kept in memory at any given time while iterating over the cursor.

> Since the `cursor` method only ever holds a single Eloquent model in memory at a time, it cannot eager load relationships. If you need to eager load relationships, consider using the `lazy` method instead.

Internally, the `cursor` method uses PHP generators to implement this functionality:

```
use App\Models\Flight;

foreach (Flight::where('destination', 'Zurich')->cursor() as $flight)
{
```

```
    // ...
}
```

The `cursor` returns an `Illuminate\Support\LazyCollection` instance. Lazy collections allow you to use many of the collection methods available on typical Laravel collections while only loading a single model into memory at a time:

```
use App\Models\User;

$users = User::cursor()->filter(function (User $user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

Although the `cursor` method uses far less memory than a regular query (by only holding a single Eloquent model in memory at a time), it will still eventually run out of memory. This is due to PHP's PDO driver internally caching all raw query results in its buffer. If you're dealing with a very large number of Eloquent records, consider using the `lazy` method instead.

Advanced Subqueries

Subquery Selects

Eloquent also offers advanced subquery support, which allows you to pull information from related tables in a single query. For example, let's imagine that we have a table of flight `destinations` and a table of `flights` to destinations. The `flights` table contains an `arrived_at` column which indicates when the flight arrived at the destination.

Using the subquery functionality available to the query builder's `select` and `addSelect` methods, we can select all of the `destinations` and the name of the flight that most recently arrived at that destination using a single query:

```
use App\Models\Destination;

use App\Models\Flight;



return Destination::addSelect(['last_flight' =>
Flight::select('name')

    ->whereColumn('destination_id', 'destinations.id')

    ->orderByDesc('arrived_at')

    ->limit(1)

])->get();
```

Subquery Ordering

In addition, the query builder's `orderBy` function supports subqueries. Continuing to use our flight example, we may use this functionality to sort all destinations based on when the last flight arrived at that destination. Again, this may be done while executing a single database query:

```
return Destination::orderByDesc(
    Flight::select('arrived_at')
        ->whereColumn('destination_id', 'destinations.id')
        ->orderByDesc('arrived_at')
        ->limit(1)
)->get();
```

Retrieving Single Models / Aggregates

In addition to retrieving all of the records matching a given query, you may also retrieve single records using the `find`, `first`, or `firstWhere` methods. Instead of returning a collection of models, these methods return a single model instance:

```
use App\Models\Flight;
```

```php
// Retrieve a model by its primary key...

$flight = Flight::find(1);



// Retrieve the first model matching the query constraints...

$flight = Flight::where('active', 1)->first();



// Alternative to retrieving the first model matching the query constraints...

$flight = Flight::firstWhere('active', 1);
```

Sometimes you may wish to perform some other action if no results are found. The `findOr` and `firstOr` methods will return a single model instance or, if no results are found, execute the given closure. The value returned by the closure will be considered the result of the method:

```php
$flight = Flight::findOr(1, function () {

    // ...

});



$flight = Flight::where('legs', '>', 3)->firstOr(function () {
```

```
    // ...
});
```

Not Found Exceptions

Sometimes you may wish to throw an exception if a model is not found. This is particularly useful in routes or controllers. The `findOrFail` and `firstOrFail` methods will retrieve the first result of the query; however, if no result is found, an `Illuminate\Database\Eloquent\ModelNotFoundException` will be thrown:

```
$flight = Flight::findOrFail(1);

$flight = Flight::where('legs', '>', 3)->firstOrFail();
```

If the `ModelNotFoundException` is not caught, a 404 HTTP response is automatically sent back to the client:

```
use App\Models\Flight;

Route::get('/api/flights/{id}', function (string $id) {
    return Flight::findOrFail($id);
});
```

Retrieving or Creating Models

The `firstOrCreate` method will attempt to locate a database record using the given column / value pairs. If the model can not be found in the database, a record will be inserted with the attributes resulting from merging the first array argument with the optional second array argument:

The `firstOrNew` method, like `firstOrCreate`, will attempt to locate a record in the database matching the given attributes. However, if a model is not found, a new model instance will be returned. Note that the model returned by `firstOrNew` has not yet been persisted to the database. You will need to manually call the `save` method to persist it:

```
use App\Models\Flight;



// Retrieve flight by name or create it if it doesn't exist...

$flight = Flight::firstOrCreate([

    'name' => 'London to Paris'

]);



// Retrieve flight by name or create it with the name, delayed, and
arrival_time attributes...

$flight = Flight::firstOrCreate(
```

```
    ['name' => 'London to Paris'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);

// Retrieve flight by name or instantiate a new Flight instance...
$flight = Flight::firstOrNew([
    'name' => 'London to Paris'
]);

// Retrieve flight by name or instantiate with the name, delayed, and
arrival_time attributes...
$flight = Flight::firstOrNew(
    ['name' => 'Tokyo to Sydney'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

Retrieving Aggregates

When interacting with Eloquent models, you may also use the `count`, `sum`, `max`, and other aggregate methods provided by the Laravel query builder. As you might expect, these methods return a scalar value instead of an Eloquent model instance:

```
$count = Flight::where('active', 1)->count();
```

```
$max = Flight::where('active', 1)->max('price');
```

Inserting and Updating Models

Inserts

Of course, when using Eloquent, we don't only need to retrieve models from the database. We also need to insert new records. Thankfully, Eloquent makes it simple. To insert a new record into the database, you should instantiate a new model instance and set attributes on the model. Then, call the `save` method on the model instance:

```
<?php
```

```
namespace App\Http\Controllers;
```

```
use App\Http\Controllers\Controller;
```

```php
use App\Models\Flight;

use Illuminate\Http\RedirectResponse;

use Illuminate\Http\Request;



class FlightController extends Controller

{

    /**

     * Store a new flight in the database.

     */

    public function store(Request $request): RedirectResponse

    {

        // Validate the request...



        $flight = new Flight;



        $flight->name = $request->name;
```

```
        $flight->save();


        return redirect('/flights');

    }
}
```

In this example, we assign the `name` field from the incoming HTTP request to the `name` attribute of the `App\Models\Flight` model instance. When we call the `save` method, a record will be inserted into the database. The model's `created_at` and `updated_at` timestamps will automatically be set when the `save` method is called, so there is no need to set them manually.

Alternatively, you may use the `create` method to "save" a new model using a single PHP statement. The inserted model instance will be returned to you by the `create` method:

```
use App\Models\Flight;


$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

However, before using the `create` method, you will need to specify either a `fillable` or `guarded` property on your model class. These properties are required because all Eloquent models are protected against mass assignment vulnerabilities by default. To learn more about mass assignment, please consult the mass assignment documentation.

Updates

The `save` method may also be used to update models that already exist in the database. To update a model, you should retrieve it and set any attributes you wish to update. Then, you should call the model's `save` method. Again, the `updated_at` timestamp will automatically be updated, so there is no need to manually set its value:

```
use App\Models\Flight;
```

```
$flight = Flight::find(1);
```

```
$flight->name = 'Paris to London';
```

```
$flight->save();
```

Occasionally, you may need to update an existing model or create a new model if no matching model exists. Like the `firstOrCreate` method, the `updateOrCreate` method persists the model, so there's no need to manually call the `save` method.

In the example below, if a flight exists with a `departure` location of `Oakland` and a `destination` location of `San Diego`, its `price` and `discounted` columns will be updated. If no such flight exists, a new flight will be created which has the attributes resulting from merging the first argument array with the second argument array:

```
$flight = Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99, 'discounted' => 1]
);
```

Mass Updates

Updates can also be performed against models that match a given query. In this example, all flights that are `active` and have a `destination` of `San Diego` will be marked as delayed:

```
Flight::where('active', 1)
      ->where('destination', 'San Diego')
      ->update(['delayed' => 1]);
```

The `update` method expects an array of column and value pairs representing the columns that should be updated. The `update` method returns the number of affected rows.

> When issuing a mass update via Eloquent, the `saving`, `saved`, `updating`, and `updated` model events will not be fired for the updated models. This is because the models are never actually retrieved when issuing a mass update.

Examining Attribute Changes

Eloquent provides the `isDirty`, `isClean`, and `wasChanged` methods to examine the internal state of your model and determine how its attributes have changed from when the model was originally retrieved.

The `isDirty` method determines if any of the model's attributes have been changed since the model was retrieved. You may pass a specific attribute name or an array of attributes to the `isDirty` method to determine if any of the attributes are "dirty". The `isClean` method will determine if an attribute has remained unchanged since the model was retrieved. This method also accepts an optional attribute argument:

```
use App\Models\User;



$user = User::create([

    'first_name' => 'Taylor',

    'last_name' => 'Otwell',

    'title' => 'Developer',

]);



$user->title = 'Painter';
```

```php
$user->isDirty(); // true

$user->isDirty('title'); // true

$user->isDirty('first_name'); // false

$user->isDirty(['first_name', 'title']); // true



$user->isClean(); // false

$user->isClean('title'); // false

$user->isClean('first_name'); // true

$user->isClean(['first_name', 'title']); // false



$user->save();



$user->isDirty(); // false

$user->isClean(); // true
```

The `wasChanged` method determines if any attributes were changed when the model was last saved within the current request cycle. If needed, you may pass an attribute name to see if a particular attribute was changed:

```php
$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->save();

$user->wasChanged(); // true
$user->wasChanged('title'); // true
$user->wasChanged(['title', 'slug']); // true
$user->wasChanged('first_name'); // false
$user->wasChanged(['first_name', 'title']); // true
```

The `getOriginal` method returns an array containing the original attributes of the model regardless of any changes to the model since it was retrieved. If needed, you may pass a specific attribute name to get the original value of a particular attribute:

```
$user = User::find(1);
```

```
$user->name; // John
```

```
$user->email; // john@example.com
```

```
$user->name = "Jack";
```

```
$user->name; // Jack
```

```
$user->getOriginal('name'); // John
```

```
$user->getOriginal(); // Array of original attributes...
```

Mass Assignment

You may use the `create` method to "save" a new model using a single PHP statement. The inserted model instance will be returned to you by the method:

```
use App\Models\Flight;
```

```
$flight = Flight::create([

    'name' => 'London to Paris',

]);
```

However, before using the `create` method, you will need to specify either a `fillable` or `guarded` property on your model class. These properties are required because all Eloquent models are protected against mass assignment vulnerabilities by default.

A mass assignment vulnerability occurs when a user passes an unexpected HTTP request field and that field changes a column in your database that you did not expect. For example, a malicious user might send an `is_admin` parameter through an HTTP request, which is then passed to your model's `create` method, allowing the user to escalate themselves to an administrator.

So, to get started, you should define which model attributes you want to make mass assignable. You may do this using the `$fillable` property on the model. For example, let's make the `name` attribute of our `Flight` model mass assignable:

```
<?php


namespace App\Models;
```

```php
use Illuminate\Database\Eloquent\Model;


class Flight extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

Once you have specified which attributes are mass assignable, you may use the `create` method to insert a new record in the database. The `create` method returns the newly created model instance:

```php
$flight = Flight::create(['name' => 'London to Paris']);
```

If you already have a model instance, you may use the `fill` method to populate it with an array of attributes:

```
$flight->fill(['name' => 'Amsterdam to Frankfurt']);
```

Mass Assignment and JSON Columns

When assigning JSON columns, each column's mass assignable key must be specified in your model's `$fillable` array. For security, Laravel does not support updating nested JSON attributes when using the `guarded` property:

```
/**
* The attributes that are mass assignable.
*
* @var array
*/
protected $fillable = [
    'options->enabled',
];
```

Allowing Mass Assignment

If you would like to make all of your attributes mass assignable, you may define your model's `$guarded` property as an empty array. If you choose to unguard your model, you should take special care to always hand-craft the arrays passed to Eloquent's `fill`, `create`, and `update` methods:

```
/**
```

```
* The attributes that aren't mass assignable.

*

* @var array

*/

protected $guarded = [];
```

Mass Assignment Exceptions

By default, attributes that are not included in the `$fillable` array are silently discarded when performing mass-assignment operations. In production, this is expected behavior; however, during local development it can lead to confusion as to why model changes are not taking effect.

If you wish, you may instruct Laravel to throw an exception when attempting to fill an unfillable attribute by invoking the `preventSilentlyDiscardingAttributes` method. Typically, this method should be invoked in the `boot` method of your application's `AppServiceProvider` class:

```
use Illuminate\Database\Eloquent\Model;



/**

* Bootstrap any application services.

*/
```

```
public function boot(): void

{

    Model::preventSilentlyDiscardingAttributes($this->app->isLocal());

}
```

Upserts

Eloquent's `upsert` method may be used to update or create records in a single, atomic operation. The method's first argument consists of the values to insert or update, while the second argument lists the column(s) that uniquely identify records within the associated table. The method's third and final argument is an array of the columns that should be updated if a matching record already exists in the database. The `upsert` method will automatically set the `created_at` and `updated_at` timestamps if timestamps are enabled on the model:

```
Flight::upsert([

    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],

    ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]

], uniqueBy: ['departure', 'destination'], update: ['price']);
```

> All databases except SQL Server require the columns in the second argument of the `upsert` method to have a "primary" or "unique" index. In addition, the MariaDB and MySQL

database drivers ignore the second argument of the `upsert` method and always use the "primary" and "unique" indexes of the table to detect existing records.

Deleting Models

To delete a model, you may call the `delete` method on the model instance:

```
use App\Models\Flight;
```

```
$flight = Flight::find(1);
```

```
$flight->delete();
```

You may call the `truncate` method to delete all of the model's associated database records. The `truncate` operation will also reset any auto-incrementing IDs on the model's associated table:

```
Flight::truncate();
```

Deleting an Existing Model by its Primary Key

In the example above, we are retrieving the model from the database before calling the `delete` method. However, if you know the primary key of the model, you may delete the model without explicitly retrieving it by calling the `destroy` method. In addition to accepting the single primary

key, the `destroy` method will accept multiple primary keys, an array of primary keys, or a collection of primary keys:

```
Flight::destroy(1);
```

```
Flight::destroy(1, 2, 3);
```

```
Flight::destroy([1, 2, 3]);
```

```
Flight::destroy(collect([1, 2, 3]));
```

If you are utilizing soft deleting models, you may permanently delete models via the `forceDestroy` method:

```
Flight::forceDestroy(1);
```

> The `destroy` method loads each model individually and calls the `delete` method so that the `deleting` and `deleted` events are properly dispatched for each model.

Deleting Models Using Queries

Of course, you may build an Eloquent query to delete all models matching your query's criteria. In this example, we will delete all flights that are marked as inactive. Like mass updates, mass deletes will not dispatch model events for the models that are deleted:

```
$deleted = Flight::where('active', 0)->delete();
```

> When executing a mass delete statement via Eloquent, the `deleting` and `deleted` model events will not be dispatched for the deleted models. This is because the models are never actually retrieved when executing the delete statement.

Soft Deleting

In addition to actually removing records from your database, Eloquent can also "soft delete" models. When models are soft deleted, they are not actually removed from your database. Instead, a `deleted_at` attribute is set on the model indicating the date and time at which the model was "deleted". To enable soft deletes for a model, add the `Illuminate\Database\Eloquent\SoftDeletes` trait to the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
```

```
{
    use SoftDeletes;
}
```

The `SoftDeletes` trait will automatically cast the `deleted_at` attribute to a `DateTime` / `Carbon` instance for you.

You should also add the `deleted_at` column to your database table. The Laravel schema builder contains a helper method to create this column:

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

Schema::table('flights', function (Blueprint $table) {
    $table->softDeletes();
});

Schema::table('flights', function (Blueprint $table) {
    $table->dropSoftDeletes();
});
```

Now, when you call the `delete` method on the model, the `deleted_at` column will be set to the current date and time. However, the model's database record will be left in the table. When querying a model that uses soft deletes, the soft deleted models will automatically be excluded from all query results.

To determine if a given model instance has been soft deleted, you may use the `trashed` method:

```
if ($flight->trashed()) {

    // ...

}
```

Restoring Soft Deleted Models

Sometimes you may wish to "un-delete" a soft deleted model. To restore a soft deleted model, you may call the `restore` method on a model instance. The `restore` method will set the model's `deleted_at` column to `null`:

```
$flight->restore();
```

You may also use the `restore` method in a query to restore multiple models. Again, like other "mass" operations, this will not dispatch any model events for the models that are restored:

```
Flight::withTrashed()

        ->where('airline_id', 1)

        ->restore();
```

The `restore` method may also be used when building relationship queries:

```
$flight->history()->restore();
```

Permanently Deleting Models

Sometimes you may need to truly remove a model from your database. You may use the `forceDelete` method to permanently remove a soft deleted model from the database table:

```
$flight->forceDelete();
```

You may also use the `forceDelete` method when building Eloquent relationship queries:

```
$flight->history()->forceDelete();
```

Querying Soft Deleted Models

Including Soft Deleted Models

As noted above, soft deleted models will automatically be excluded from query results. However, you may force soft deleted models to be included in a query's results by calling the `withTrashed` method on the query:

```
use App\Models\Flight;
```

```
$flights = Flight::withTrashed()
```

```
                ->where('account_id', 1)
```

```
                ->get();
```

The `withTrashed` method may also be called when building a relationship query:

```
$flight->history()->withTrashed()->get();
```

Retrieving Only Soft Deleted Models

The `onlyTrashed` method will retrieve only soft deleted models:

```
$flights = Flight::onlyTrashed()

                ->where('airline_id', 1)

                ->get();
```

Pruning Models

Sometimes you may want to periodically delete models that are no longer needed. To accomplish this, you may add the `Illuminate\Database\Eloquent\Prunable` or `Illuminate\Database\Eloquent\MassPrunable` trait to the models you would like to periodically prune. After adding one of the traits to the model, implement a `prunable` method which returns an Eloquent query builder that resolves the models that are no longer needed:

```
<?php
```

```php
namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Prunable;

class Flight extends Model
{
    use Prunable;

    /**
     * Get the prunable model query.
     */
    public function prunable(): Builder
    {
        return static::where('created_at', '<=', now()->subMonth());
```

```
    }
}
```

When marking models as `Prunable`, you may also define a `pruning` method on the model. This method will be called before the model is deleted. This method can be useful for deleting any additional resources associated with the model, such as stored files, before the model is permanently removed from the database:

```
/**
* Prepare the model for pruning.
*/
protected function pruning(): void
{
    // ...
}
```

After configuring your prunable model, you should schedule the `model:prune` Artisan command in your application's `routes/console.php` file. You are free to choose the appropriate interval at which this command should be run:

```
use Illuminate\Support\Facades\Schedule;
```

```
Schedule::command('model:prune')->daily();
```

Behind the scenes, the `model:prune` command will automatically detect "Prunable" models within your application's `app/Models` directory. If your models are in a different location, you may use the `--model` option to specify the model class names:

```
Schedule::command('model:prune', [
    '--model' => [Address::class, Flight::class],
])->daily();
```

If you wish to exclude certain models from being pruned while pruning all other detected models, you may use the `--except` option:

```
Schedule::command('model:prune', [
    '--except' => [Address::class, Flight::class],
])->daily();
```

You may test your `prunable` query by executing the `model:prune` command with the `--pretend` option. When pretending, the `model:prune` command will simply report how many records would be pruned if the command were to actually run:

```
php artisan model:prune --pretend
```

> Soft deleting models will be permanently deleted (`forceDelete`) if they match the prunable query.

Mass Pruning

When models are marked with the `Illuminate\Database\Eloquent\MassPrunable` trait, models are deleted from the database using mass-deletion queries. Therefore, the `pruning` method will not be invoked, nor will the `deleting` and `deleted` model events be dispatched. This is because the models are never actually retrieved before deletion, thus making the pruning process much more efficient:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\MassPrunable;

class Flight extends Model
{
    use MassPrunable;
```

```
    /**
     * Get the prunable model query.
     */
    public function prunable(): Builder
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

Replicating Models

You may create an unsaved copy of an existing model instance using the `replicate` method. This method is particularly useful when you have model instances that share many of the same attributes:

```
use App\Models\Address;

$shipping = Address::create([
```

```
    'type' => 'shipping',

    'line_1' => '123 Example Street',

    'city' => 'Victorville',

    'state' => 'CA',

    'postcode' => '90001',

]);



$billing = $shipping->replicate()->fill([

    'type' => 'billing'

]);



$billing->save();
```

To exclude one or more attributes from being replicated to the new model, you may pass an array to the `replicate` method:

```
$flight = Flight::create([

    'destination' => 'LAX',

    'origin' => 'LHR',
```

```
    'last_flown' => '2020-03-04 11:00:00',

    'last_pilot_id' => 747,

]);
```

```
$flight = $flight->replicate([

    'last_flown',

    'last_pilot_id'

]);
```

Query Scopes

Global Scopes

Global scopes allow you to add constraints to all queries for a given model. Laravel's own soft delete functionality utilizes global scopes to only retrieve "non-deleted" models from the database. Writing your own global scopes can provide a convenient, easy way to make sure every query for a given model receives certain constraints.

Generating Scopes

To generate a new global scope, you may invoke the `make:scope` Artisan command, which will place the generated scope in your application's `app/Models/Scopes` directory:

```
php artisan make:scope AncientScope
```

Writing Global Scopes

Writing a global scope is simple. First, use the `make:scope` command to generate a class that implements the `Illuminate\Database\Eloquent\Scope` interface. The `Scope` interface requires you to implement one method: `apply`. The `apply` method may add `where` constraints or other types of clauses to the query as needed:

```
<?php



namespace App\Models\Scopes;



use Illuminate\Database\Eloquent\Builder;

use Illuminate\Database\Eloquent\Model;

use Illuminate\Database\Eloquent\Scope;



class AncientScope implements Scope

{
```

```
    /**
     * Apply the scope to a given Eloquent query builder.
     */
    public function apply(Builder $builder, Model $model): void
    {
        $builder->where('created_at', '<', now()->subYears(2000));
    }
}
```

If your global scope is adding columns to the select clause of the query, you should use the `addSelect` method instead of `select`. This will prevent the unintentional replacement of the query's existing select clause.

Applying Global Scopes

To assign a global scope to a model, you may simply place the `ScopedBy` attribute on the model:

```
<?php

namespace App\Models;
```

```php
use App\Models\Scopes\AncientScope;

use Illuminate\Database\Eloquent\Attributes\ScopedBy;



#[ScopedBy([AncientScope::class])]

class User extends Model

{

    //

}
```

Or, you may manually register the global scope by overriding the model's `booted` method and invoke the model's `addGlobalScope` method. The `addGlobalScope` method accepts an instance of your scope as its only argument:

```php
<?php



namespace App\Models;



use App\Models\Scopes\AncientScope;

use Illuminate\Database\Eloquent\Model;
```

```
class User extends Model

{

    /**

     * The "booted" method of the model.

     */

    protected static function booted(): void

    {

        static::addGlobalScope(new AncientScope);

    }

}
```

After adding the scope in the example above to the `App\Models\User` model, a call to the `User::all()` method will execute the following SQL query:

```
select * from `users` where `created_at` < 0021-02-18 00:00:00
```

Anonymous Global Scopes

Eloquent also allows you to define global scopes using closures, which is particularly useful for simple scopes that do not warrant a separate class of their own. When defining a global scope

using a closure, you should provide a scope name of your own choosing as the first argument to the `addGlobalScope` method:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booted" method of the model.
     */
    protected static function booted(): void
    {
```

```
        static::addGlobalScope('ancient', function (Builder $builder)
{

            $builder->where('created_at', '<', now()->subYears(2000));

        });

    }

}
```

Removing Global Scopes

If you would like to remove a global scope for a given query, you may use the `withoutGlobalScope` method. This method accepts the class name of the global scope as its only argument:

```
User::withoutGlobalScope(AncientScope::class)->get();
```

Or, if you defined the global scope using a closure, you should pass the string name that you assigned to the global scope:

```
User::withoutGlobalScope('ancient')->get();
```

If you would like to remove several or even all of the query's global scopes, you may use the `withoutGlobalScopes` method:

```
// Remove all of the global scopes...

User::withoutGlobalScopes()->get();
```

```php
// Remove some of the global scopes...

User::withoutGlobalScopes([

    FirstScope::class, SecondScope::class

])->get();
```

Local Scopes

Local scopes allow you to define common sets of query constraints that you may easily re-use throughout your application. For example, you may need to frequently retrieve all users that are considered "popular". To define a scope, prefix an Eloquent model method with `scope`.

Scopes should always return the same query builder instance or `void`:

```php
<?php


namespace App\Models;


use Illuminate\Database\Eloquent\Builder;

use Illuminate\Database\Eloquent\Model;
```

```php
class User extends Model

{

    /**

     * Scope a query to only include popular users.

     */

    public function scopePopular(Builder $query): void

    {

        $query->where('votes', '>', 100);

    }



    /**

     * Scope a query to only include active users.

     */

    public function scopeActive(Builder $query): void

    {

        $query->where('active', 1);
```

```
    }

}
```

Utilizing a Local Scope

Once the scope has been defined, you may call the scope methods when querying the model. However, you should not include the `scope` prefix when calling the method. You can even chain calls to various scopes:

```
use App\Models\User;



$users = User::popular()->active()->orderBy('created_at')->get();
```

Combining multiple Eloquent model scopes via an `or` query operator may require the use of closures to achieve the correct logical grouping:

```
$users = User::popular()->orWhere(function (Builder $query) {

    $query->active();

})->get();
```

However, since this can be cumbersome, Laravel provides a "higher order" `orWhere` method that allows you to fluently chain scopes together without the use of closures:

```
$users = User::popular()->orWhere->active()->get();
```

Dynamic Scopes

Sometimes you may wish to define a scope that accepts parameters. To get started, just add your additional parameters to your scope method's signature. Scope parameters should be defined after the `$query` parameter:

```php
<?php


namespace App\Models;


use Illuminate\Database\Eloquent\Builder;

use Illuminate\Database\Eloquent\Model;


class User extends Model

{

    /**

     * Scope a query to only include users of a given type.

     */

    public function scopeOfType(Builder $query, string $type): void

    {
```

```
        $query->where('type', $type);

    }

}
```

Once the expected arguments have been added to your scope method's signature, you may pass the arguments when calling the scope:

```
$users = User::ofType('admin')->get();
```

Comparing Models

Sometimes you may need to determine if two models are the "same" or not. The `is` and `isNot` methods may be used to quickly verify two models have the same primary key, table, and database connection or not:

```
if ($post->is($anotherPost)) {

    // ...

}
```

```
if ($post->isNot($anotherPost)) {

    // ...
```

```
}
```

The `is` and `isNot` methods are also available when using the `belongsTo`, `hasOne`, `morphTo`, and `morphOne` relationships. This method is particularly helpful when you would like to compare a related model without issuing a query to retrieve that model:

```
if ($post->author()->is($user)) {

    // ...

}
```

Events

> Want to broadcast your Eloquent events directly to your client-side application? Check out Laravel's model event broadcasting.

Eloquent models dispatch several events, allowing you to hook into the following moments in a model's lifecycle: `retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `trashed`, `forceDeleting`, `forceDeleted`, `restoring`, `restored`, and `replicating`.

The `retrieved` event will dispatch when an existing model is retrieved from the database. When a new model is saved for the first time, the `creating` and `created` events will dispatch. The `updating` / `updated` events will dispatch when an existing model is modified and the `save` method is called. The `saving` / `saved` events will dispatch when a model is created or updated - even if the model's attributes have not been changed. Event names ending with `-ing`

are dispatched before any changes to the model are persisted, while events ending with `-ed` are dispatched after the changes to the model are persisted.

To start listening to model events, define a `$dispatchesEvents` property on your Eloquent model. This property maps various points of the Eloquent model's lifecycle to your own event classes. Each model event class should expect to receive an instance of the affected model via its constructor:

```
<?php

namespace App\Models;

use App\Events\UserDeleted;
use App\Events\UserSaved;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;
```

```php
    /**
     * The event map for the model.
     *
     * @var array<string, string>
     */
    protected $dispatchesEvents = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}
```

After defining and mapping your Eloquent events, you may use event listeners to handle the events.

> When issuing a mass update or delete query via Eloquent, the `saved`, `updated`, `deleting`, and `deleted` model events will not be dispatched for the affected models. This is because the models are never actually retrieved when performing mass updates or deletes.

Using Closures

Instead of using custom event classes, you may register closures that execute when various model events are dispatched. Typically, you should register these closures in the `booted` method of your model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booted" method of the model.
     */
    protected static function booted(): void
    {
```

```
        static::created(function (User $user) {
            // ...
        });
    }
}
```

If needed, you may utilize queueable anonymous event listeners when registering model events. This will instruct Laravel to execute the model event listener in the background using your application's queue:

```
use function Illuminate\Events\queueable;

static::created(queueable(function (User $user) {
    // ...
}));
```

Observers

Defining Observers

If you are listening for many events on a given model, you may use observers to group all of your listeners into a single class. Observer classes have method names which reflect the Eloquent events you wish to listen for. Each of these methods receives the affected model as

their only argument. The `make:observer` Artisan command is the easiest way to create a new observer class:

```
php artisan make:observer UserObserver --model=User
```

This command will place the new observer in your `app/Observers` directory. If this directory does not exist, Artisan will create it for you. Your fresh observer will look like the following:

```
<?php



namespace App\Observers;



use App\Models\User;



class UserObserver

{

    /**

     * Handle the User "created" event.

     */

    public function created(User $user): void
```

```
    {
        // ...
    }

    /**
     * Handle the User "updated" event.
     */
    public function updated(User $user): void
    {
        // ...
    }

    /**
     * Handle the User "deleted" event.
     */
    public function deleted(User $user): void
```

```
    {
        // ...
    }

    /**
     * Handle the User "restored" event.
     */
    public function restored(User $user): void
    {
        // ...
    }

    /**
     * Handle the User "forceDeleted" event.
     */
    public function forceDeleted(User $user): void
```

```
    {
        // ...
    }
}
```

To register an observer, you may place the `ObservedBy` attribute on the corresponding model:

```
use App\Observers\UserObserver;
use Illuminate\Database\Eloquent\Attributes\ObservedBy;

#[ObservedBy([UserObserver::class])]
class User extends Authenticatable
{
    //
}
```

Or, you may manually register an observer by invoking the `observe` method on the model you wish to observe. You may register observers in the `boot` method of your application's `AppServiceProvider` class:

```
use App\Models\User;
```

```php
use App\Observers\UserObserver;

/**
* Bootstrap any application services.
*/
public function boot(): void
{
    User::observe(UserObserver::class);
}
```

> There are additional events an observer can listen to, such as `saving` and `retrieved`. These events are described within the events documentation.

Observers and Database Transactions

When models are being created within a database transaction, you may want to instruct an observer to only execute its event handlers after the database transaction is committed. You may accomplish this by implementing the `ShouldHandleEventsAfterCommit` interface on your observer. If a database transaction is not in progress, the event handlers will execute immediately:

```php
<?php
```

```php
namespace App\Observers;

use App\Models\User;

use Illuminate\Contracts\Events\ShouldHandleEventsAfterCommit;

class UserObserver implements ShouldHandleEventsAfterCommit
{
    /**
     * Handle the User "created" event.
     */
    public function created(User $user): void
    {
        // ...
    }
}
```

Muting Events

You may occasionally need to temporarily "mute" all events fired by a model. You may achieve this using the `withoutEvents` method. The `withoutEvents` method accepts a closure as its only argument. Any code executed within this closure will not dispatch model events, and any value returned by the closure will be returned by the `withoutEvents` method:

```
use App\Models\User;



$user = User::withoutEvents(function () {

    User::findOrFail(1)->delete();



    return User::find(2);

});
```

Saving a Single Model Without Events

Sometimes you may wish to "save" a given model without dispatching any events. You may accomplish this using the `saveQuietly` method:

```
$user = User::findOrFail(1);



$user->name = 'Victoria Faith';
```

```
$user->saveQuietly();
```

You may also "update", "delete", "soft delete", "restore", and "replicate" a given model without dispatching any events:

```
$user->deleteQuietly();
```

```
$user->forceDeleteQuietly();
```

```
$user->restoreQuietly();
```