Learning Outcome

## Views in Laravel with complete conditional and looping construct.

## Controllers and its usage.

## Complete database connectivity with DB

## Complete Database connectivity with Eloquent

## Model and its working.

## 

## Views

## Creating Views

## Views contain the HTML served by your application and separate your controller / application logic from your presentation logic. Views are stored in the resources/views directory.

## A view can be called using the view helper function:

## view(string $path, array $data = [])

## The first parameter of the helper is the path to a view file, and the second parameter is an optional array of data to pass to the view.

## Therefore, to call the resources/views/example.php, you would use:

## view('example');A simple view might look something like this:

## <!-- View stored in resources/views/greeting.blade.php -->

## <html>

## <body>

## <h1>Hello, {{ $name }}</h1>

## </body>

## </html>

## Since this view is stored at resources/views/greeting.blade.php, we may return it using the global view helper like so:

## Route::get('/', function () {

## return view('greeting', ['name' => 'James']);

## });

## As you can see, the first argument passed to the view helper corresponds to the name of the view file in the resources/views directory. The second argument is an array of data that should be made available to the view. In this case, we are passing the name variable, which is displayed in the view using Blade syntax.

## Of course, views may also be nested within sub-directories of the resources/views directory. "Dot" notation may be used to reference nested views. For example, if your view is stored at resources/views/admin/profile.blade.php, you may reference it like so:

## return view('admin.profile', $data);

## Determining If A View Exists

## If you need to determine if a view exists, you may use the View facade. The exists method will return true if the view exists:

## use Illuminate\Support\Facades\View;

## if (View::exists('emails.customer')) {

## //

## }

## Passing Data To Views

## As you saw in the previous examples, you may pass an array of data to views:

## return view('greetings', ['name' => 'Victoria']);

## When passing information in this manner, $data should be an array with key/value pairs. Inside your view, you can then access each value using its corresponding key, such as <?php echo $key; ?>. As an alternative to passing a complete array of data to the view helper function, you may use the with method to add individual pieces of data to the view:

## return view('greeting')->with('name', 'Victoria');

## Sharing Data With All Views

## Occasionally, you may need to share a piece of data with all views that are rendered by your application. You may do so using the view facade's share method. Typically, you should place calls to share within a service provider's boot method. You are free to add them to the AppServiceProvider or generate a separate service provider to house them:

## <?php

## namespace App\Providers;

## use Illuminate\Support\Facades\View;

## class AppServiceProvider extends ServiceProvider

## {

## /\*\*

## \* Bootstrap any application services.

## \*

## \* @return void

## \*/

## public function boot()

## {

## View::share('key', 'value');

## }

## /\*\*

## \* Register the service provider.

## \*

## \* @return void

## \*/

## public function register()

## {

## //

## }

## }

## View Composers

## View composers are callbacks or class methods that are called when a view is rendered. If you have data that you want to be bound to a view each time that view is rendered, a view composer can help you organize that logic into a single location.

## For this example, let's register the view composers within a service provider. We'll use the View facade to access the underlying Illuminate\Contracts\View\Factory contract implementation. Remember, Laravel does not include a default directory for view composers. You are free to organize them however you wish. For example, you could create an App\Http\ViewComposers directory:

## <?php

## namespace App\Providers;

## use Illuminate\Support\Facades\View;

## use Illuminate\Support\ServiceProvider;

## class ComposerServiceProvider extends ServiceProvider

## {

## /\*\*

## \* Register bindings in the container.

## \*

## \* @return void

## \*/

## public function boot()

## {

## // Using class based composers...

## View::composer(

## 'profile', 'App\Http\ViewComposers\ProfileComposer'

## );

## // Using Closure based composers...

## View::composer('dashboard', function ($view) {

## //

## });

## }

## /\*\*

## \* Register the service provider.

## \*

## \* @return void

## \*/

## public function register()

## {

## //

## }

## }

## {note} Remember, if you create a new service provider to contain your view composer registrations, you will need to add the service provider to the providers array in the config/app.php configuration file.

## Now that we have registered the composer, the ProfileComposer@compose method will be executed each time the profile view is being rendered. So, let's define the composer class:

## <?php

## namespace App\Http\ViewComposers;

## use Illuminate\View\View;

## use App\Repositories\UserRepository;

## class ProfileComposer

## {

## /\*\*

## \* The user repository implementation.

## \*

## \* @var UserRepository

## \*/

## protected $users;

## /\*\*

## \* Create a new profile composer.

## \*

## \* @param UserRepository $users

## \* @return void

## \*/

## public function \_\_construct(UserRepository $users)

## {

## // Dependencies automatically resolved by service container...

## $this->users = $users;

## }

## /\*\*

## \* Bind data to the view.

## \*

## \* @param View $view

## \* @return void

## \*/

## public function compose(View $view)

## {

## $view->with('count', $this->users->count());

## }

## }

## Just before the view is rendered, the composer's compose method is called with the Illuminate\View\View instance. You may use the with method to bind data to the view.

## {tip} All view composers are resolved via the service container, so you may type-hint any dependencies you need within a composer's constructor.

## Attaching A Composer To Multiple Views

## You may attach a view composer to multiple views at once by passing an array of views as the first argument to the composer method:

## View::composer(

## ['profile', 'dashboard'],

## 'App\Http\ViewComposers\MyViewComposer'

## );

## The composer method also accepts the \* character as a wildcard, allowing you to attach a composer to all views:

## View::composer('\*', function ($view) {

## //

## });

## View Creators

## View creators are very similar to view composers; however, they are executed immediately after the view is instantiated instead of waiting until the view is about to render. To register a view creator, use the creator method:

## View::creator('profile', 'App\Http\ViewCreators\ProfileCreator');

## Control Structures

Blade provides convenient syntax for common PHP control structures.Each of the control structures begins with @[structure] and ends with @[endstructure]. Notice that within the tags, we are just typing normal HTML and including variables with the Blade syntax.

## If Statements

## You may construct if statements using the @if, @elseif, @else, and @endif directives. These directives function identically to their PHP counterparts:

## @if (count($records) === 1)

## I have one record!

## @elseif (count($records) > 1)

## I have multiple records!

## @else

## I don't have any records!

## @endif

## For convenience, Blade also provides an @unless directive:

## @unless (Auth::check())

## You are not signed in.

## @endunless

## Loops

## In addition to conditional statements, Blade provides simple directives for working with PHP's loop structures. Again, each of these directives functions identically to their PHP counterparts:

## @for ($i = 0; $i < 10; $i++)

## The current value is {{ $i }}

## @endfor

## @foreach ($users as $user)

## <p>This is user {{ $user->id }}</p>

## @endforeach

## @forelse ($users as $user)

## <li>{{ $user->name }}</li>

## @empty

## <p>No users</p>

## @endforelse

## @while (true)

## <p>I'm looping forever.</p>

## @endwhile

## {tip} When looping, you may use the loop variable to gain valuable information about the loop, such as whether you are in the first or last iteration through the loop.

## When using loops you may also end the loop or skip the current iteration:

## @foreach ($users as $user)

## @if ($user->type == 1)

## @continue

## @endif

## <li>{{ $user->name }}</li>

## @if ($user->number == 5)

## @break

## @endif

## @endforeach

## You may also include the condition with the directive declaration in one line:

## @foreach ($users as $user)

## @continue($user->type == 1)

## <li>{{ $user->name }}</li>

## @break($user->number == 5)

## @endforeach

## The Loop Variable

## When looping, a $loop variable will be available inside of your loop. This variable provides access to some useful bits of information such as the current loop index and whether this is the first or last iteration through the loop:

## @foreach ($users as $user)

## @if ($loop->first)

## This is the first iteration.

## @endif

## @if ($loop->last)

## This is the last iteration.

## @endif

## <p>This is user {{ $user->id }}</p>

## @endforeach

## If you are in a nested loop, you may access the parent loop's $loop variable via the parent property:

## @foreach ($users as $user)

## @foreach ($user->posts as $post)

## @if ($loop->parent->first)

## This is first iteration of the parent loop.

## @endif

## @endforeach

## @endforeach

## The $loop variable also contains a variety of other useful properties:

|  |  |
| --- | --- |
| Property | Description |
| $loop->index | The index of the current loop iteration (starts at 0). |
| $loop->iteration | The current loop iteration (starts at 1). |
| $loop->remaining | The iteration remaining in the loop. |
| $loop->count | The total number of items in the array being iterated. |
| $loop->first | Whether this is the first iteration through the loop. |
| $loop->last | Whether this is the last iteration through the loop. |
| $loop->depth | The nesting level of the current loop. |
| $loop->parent | When in a nested loop, the parent's loop variable. |

## Since Laravel 5.2.22, we can also use the directives @continue and @break

|  |  |
| --- | --- |
| Property | Description |
| @break | Stop the current loop. |
| @continue | Stop the current iteration and start the next one. |

## Example :

## @foreach ($users as $user)

## @continue ($user->id == 2)

## <p>{{ $user->id }} {{ $user->name }}</p>

## @break ($user->id == 4)

## @endforeach

## Comments

## Blade also allows you to define comments in your views. However, unlike HTML comments, Blade comments are not included in the HTML returned by your application:

## {{-- This comment will not be present in the rendered HTML --}}

## PHP

## In some situations, it's useful to embed PHP code into your views. You can use the Blade @php directive to execute a block of plain PHP within your template:

## @php

## //

## @endphp

## {tip} While Blade provides this feature, using it frequently may be a signal that you have too much logic embedded within your template.

## Controllers

## Introduction

## Instead of defining all of your request handling logic as Closures in route files, you may wish to organize this behavior using Controller classes. Controllers can group related request handling logic into a single class. Controllers are stored in the app/Http/Controllers directory.

## Basic Controllers

## Defining Controllers

## Below is an example of a basic controller class. Note that the controller extends the base controller class included with Laravel. The base class provides a few convenience methods such as the middleware method, which may be used to attach middleware to controller actions:

## <?php

## namespace App\Http\Controllers;

## use App\User;

## use App\Http\Controllers\Controller;

## class UserController extends Controller

## {

## /\*\*

## \* Show the profile for the given user.

## \* @param int $id

## \* @return Response

## \*/

## public function show($id)

## {

## return view('user.profile', ['user' => User::findOrFail($id)]);

## }

## }

## You can define a route to this controller action like so:

## Route::get('user/{id}', 'UserController@show');

## Now, when a request matches the specified route URI, the show method on the UserController class will be executed. Of course, the route parameters will also be passed to the method.

## {tip} Controllers are not required to extend a base class. However, you will not have access to convenience features such as the middleware, validate, and dispatch methods.

## Controllers & Namespaces

## It is very important to note that we did not need to specify the full controller namespace when defining the controller route. Since the RouteServiceProvider loads your route files within a route group that contains the namespace, we only specified the portion of the class name that comes after the App\Http\Controllers portion of the namespace.

## If you choose to nest your controllers deeper into the App\Http\Controllers directory, simply use the specific class name relative to the App\Http\Controllers root namespace. So, if your full controller class is App\Http\Controllers\Photos\AdminController, you should register routes to the controller like so:

## Route::get('foo', 'Photos\AdminController@method');

## Single Action Controllers

## If you would like to define a controller that only handles a single action, you may place a single \_\_invoke method on the controller:

## <?php

## namespace App\Http\Controllers;

## use App\User;

## use App\Http\Controllers\Controller;

## class ShowProfile extends Controller

## {

## /\*\*

## \* Show the profile for the given user.

## \*

## \* @param int $id

## \* @return Response

## \*/

## public function \_\_invoke($id)

## {

## return view('user.profile', ['user' => User::findOrFail($id)]);

## }

## }

## When registering routes for single action controllers, you do not need to specify a method:

## Route::get('user/{id}', 'ShowProfile');

## Controller Middleware

## Middleware may be assigned to the controller's routes in your route files:

## Route::get('profile', 'UserController@show')->middleware('auth');

## However, it is more convenient to specify middleware within your controller's constructor. Using the middleware method from your controller's constructor, you may easily assign middleware to the controller's action. You may even restrict the middleware to only certain methods on the controller class:

## class UserController extends Controller

## {

## /\*\*

## \* Instantiate a new controller instance.

## \*

## \* @return void

## \*/

## public function \_\_construct()

## {

## $this->middleware('auth');

## $this->middleware('log')->only('index');

## $this->middleware('subscribed')->except('store');

## }

## }

## Controllers also allow you to register middleware using a Closure. This provides a convenient way to define a middleware for a single controller without defining an entire middleware class:

## $this->middleware(function ($request, $next) {

## // ...

## return $next($request);

## });

## {tip} You may assign middleware to a subset of controller actions; however, it may indicate your controller is growing too large. Instead, consider breaking your controller into multiple, smaller controllers.

## Resource Controllers

## Laravel resource routing assigns the typical "CRUD" routes to a controller with a single line of code. For example, you may wish to create a controller that handles all HTTP requests for "photos" stored by your application. Using the make:controller Artisan command, we can quickly create such a controller:

## php artisan make:controller PhotoController --resource

## This command will generate a controller at app/Http/Controllers/PhotoController.php. The controller will contain a method for each of the available resource operations.

## Next, you may register a resourceful route to the controller:

## Route::resource('photos', 'PhotoController');

## This single route declaration creates multiple routes to handle a variety of actions on the resource. The generated controller will already have methods stubbed for each of these actions, including notes informing you of the HTTP verbs and URIs they handle.

## Actions Handled By Resource Controller

|  |  |  |  |
| --- | --- | --- | --- |
| Verb | URI | Action | Route Name |
| GET | /photos | index | photos.index |
| GET | /photos/create | create | photos.create |
| POST | /photos | store | photos.store |
| GET | /photos/{photo} | show | photos.show |
| GET | /photos/{photo}/edit | edit | photos.edit |
| PUT/PATCH | /photos/{photo} | update | photos.update |
| DELETE | /photos/{photo} | destroy | photos.destroy |

## Spoofing Form Methods

## Since HTML forms can't make PUT, PATCH, or DELETE requests, you will need to add a hidden \_method field to spoof these HTTP verbs. The method\_field helper can create this field for you:

## {{ method\_field('PUT') }}

## Partial Resource Routes

## When declaring a resource route, you may specify a subset of actions the controller should handle instead of the full set of default actions:

## Route::resource('photo', 'PhotoController', ['only' => [

## 'index', 'show'

## ]]);

## Route::resource('photo', 'PhotoController', ['except' => [

## 'create', 'store', 'update', 'destroy'

## ]]);

## Naming Resource Routes

## By default, all resource controller actions have a route name; however, you can override these names by passing a names array with your options:

## Route::resource('photo', 'PhotoController', ['names' => [

## 'create' => 'photo.build'

## ]]);

## Naming Resource Route Parameters

## By default, Route::resource will create the route parameters for your resource routes based on the "singularized" version of the resource name. You can easily override this on a per resource basis by passing parameters in the options array. The parameters array should be an associative array of resource names and parameter names:

## Route::resource('user', 'AdminUserController', ['parameters' => [

## 'user' => 'admin\_user'

## ]]);

## The example above generates the following URIs for the resource's show route:

## /user/{admin\_user}

## Supplementing Resource Controllers

## If you need to add additional routes to a resource controller beyond the default set of resource routes, you should define those routes before your call to Route::resource; otherwise, the routes defined by the resource method may unintentionally take precedence over your supplemental routes:

## Route::get('photos/popular', 'PhotoController@method');

## Route::resource('photos', 'PhotoController');

## {tip} Remember to keep your controllers focused. If you find yourself routinely needing methods outside of the typical set of resource actions, consider splitting your controller into two, smaller controllers.

## Dependency Injection & Controllers

## Constructor Injection

## The Laravel service container is used to resolve all Laravel controllers. As a result, you are able to type-hint any dependencies your controller may need in its constructor. The declared dependencies will automatically be resolved and injected into the controller instance:

## <?php

## namespace App\Http\Controllers;

## use App\Repositories\UserRepository;

## class UserController extends Controller

## {

## /\*\*

## \* The user repository instance.

## \*/

## protected $users;

## /\*\*

## \* Create a new controller instance.

## \*

## \* @param UserRepository $users

## \* @return void

## \*/

## public function \_\_construct(UserRepository $users)

## {

## $this->users = $users;

## }

## }

## Of course, you may also type-hint any Laravel contract. If the container can resolve it, you can type-hint it. Depending on your application, injecting your dependencies into your controller may provide better testability.

## Method Injection

## In addition to constructor injection, you may also type-hint dependencies on your controller's methods. A common use-case for method injection is injecting the Illuminate\Http\Request instance into your controller methods:

## <?php

## namespace App\Http\Controllers;

## use Illuminate\Http\Request;

## class UserController extends Controller

## {

## /\*\*

## \* Store a new user.

## \*

## \* @param Request $request

## \* @return Response

## \*/

## public function store(Request $request)

## {

## $name = $request->name;

## //

## }

## }

## If your controller method is also expecting input from a route parameter, simply list your route arguments after your other dependencies. For example, if your route is defined like so:

## Route::put('user/{id}', 'UserController@update');

## You may still type-hint the Illuminate\Http\Request and access your id parameter by defining your controller method as follows:

## <?php

## namespace App\Http\Controllers;

## use Illuminate\Http\Request;

## class UserController extends Controller

## {

## /\*\*

## \* Update the given user.

## \* @param Request $request

## \* @param string $id

## \* @return Response

## \*/

## public function update(Request $request, $id)

## {

## //

## }

## }

## Route Caching

## {note} Closure based routes cannot be cached. To use route caching, you must convert any Closure routes to controller classes.

## If your application is exclusively using controller based routes, you should take advantage of Laravel's route cache. Using the route cache will drastically decrease the amount of time it takes to register all of your application's routes. In some cases, your route registration may even be up to 100x faster. To generate a route cache, just execute the route:cache Artisan command:

## php artisan route:cache

## After running this command, your cached routes file will be loaded on every request. Remember, if you add any new routes you will need to generate a fresh route cache. Because of this, you should only run the route:cache command during your project's deployment.

## You may use the route:clear command to clear the route cache:

## php artisan route:clear

## Complete database connectivity with DB

## Introduction

## Laravel makes interacting with databases extremely simple across a variety of database backends using either raw SQL, the fluent query builder, and the Eloquent ORM. Currently, Laravel supports four databases:

## - MySQL - Postgres - SQLite - SQL Server

## Configuration

## The database configuration for your application is located at config/database.php. In this file you may define all of your database connections, as well as specify which connection should be used by default. Examples for most of the supported database systems are provided in this file.

## By default, Laravel's sample environment configuration is ready to use with Laravel Homestead, which is a convenient virtual machine for doing Laravel development on your local machine. Of course, you are free to modify this configuration as needed for your local database.

## SQLite Configuration

## After creating a new SQLite database using a command such as touch database/database.sqlite, you can easily configure your environment variables to point to this newly created database by using the database's absolute path:

## DB\_CONNECTION=sqlite

## DB\_DATABASE=/absolute/path/to/database.sqlite

## SQL Server Configuration

## Laravel supports SQL Server out of the box; however, you will need to add the connection configuration for the database to your config/database.php configuration file:

## 'sqlsrv' => [

## 'driver' => 'sqlsrv',

## 'host' => env('DB\_HOST', 'localhost'),

## 'database' => env('DB\_DATABASE', 'forge'),

## 'username' => env('DB\_USERNAME', 'forge'),

## 'password' => env('DB\_PASSWORD', ''),

## 'charset' => 'utf8',

## 'prefix' => '',

## ],

## Read & Write Connections

## Sometimes you may wish to use one database connection for SELECT statements, and another for INSERT, UPDATE, and DELETE statements. Laravel makes this a breeze, and the proper connections will always be used whether you are using raw queries, the query builder, or the Eloquent ORM.

## To see how read / write connections should be configured, let's look at this example:

## 'mysql' => [

## 'read' => [

## 'host' => '192.168.1.1',

## ],

## 'write' => [

## 'host' => '196.168.1.2'

## ],

## 'driver' => 'mysql',

## 'database' => 'database',

## 'username' => 'root',

## 'password' => '',

## 'charset' => 'utf8',

## 'collation' => 'utf8\_unicode\_ci',

## 'prefix' => '',

## ],

## Note that two keys have been added to the configuration array: read and write. Both of these keys have array values containing a single key: host. The rest of the database options for the read and write connections will be merged from the main mysql array.

## You only need to place items in the read and write arrays if you wish to override the values from the main array. So, in this case, 192.168.1.1 will be used as the host for the "read" connection, while 192.168.1.2 will be used for the "write" connection. The database credentials, prefix, character set, and all other options in the main mysql array will be shared across both connections.

## Using Multiple Database Connections

## When using multiple connections, you may access each connection via the connection method on the DB facade. The name passed to the connection method should correspond to one of the connections listed in your config/database.php configuration file:

## $users = DB::connection('foo')->select(...);

## You may also access the raw, underlying PDO instance using the getPdo method on a connection instance:

## $pdo = DB::connection()->getPdo();

## Running Raw SQL Queries

## Once you have configured your database connection, you may run queries using the DB facade. The DB facade provides methods for each type of query: select, update, insert, delete, and statement.

## Running A Select Query

## To run a basic query, you may use the select method on the DB facade:

## <?php

## namespace App\Http\Controllers;

## use Illuminate\Support\Facades\DB;

## use App\Http\Controllers\Controller;

## class UserController extends Controller

## {

## /\*\*

## \* Show a list of all of the application's users.

## \* @return Response

## \*/

## public function index()

## {

## $users = DB::select('select \* from users where active = ?', [1]);

## return view('user.index', ['users' => $users]);

## }

## }

## The first argument passed to the select method is the raw SQL query, while the second argument is any parameter bindings that need to be bound to the query. Typically, these are the values of the where clause constraints. Parameter binding provides protection against SQL injection.

## The select method will always return an array of results. Each result within the array will be a PHP StdClass object, allowing you to access the values of the results:

## foreach ($users as $user) {

## echo $user->name;

## }

## Using Named Bindings

## Instead of using ? to represent your parameter bindings, you may execute a query using named bindings:

## $results = DB::select('select \* from users where id = :id', ['id' => 1]);

## Running An Insert Statement

## To execute an insert statement, you may use the insert method on the DB facade. Like select, this method takes the raw SQL query as its first argument and bindings as its second argument:

## DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);

## Running An Update Statement

## The update method should be used to update existing records in the database. The number of rows affected by the statement will be returned:

## $affected = DB::update('update users set votes = 100 where name = ?', ['John']);

## Running A Delete Statement

## The delete method should be used to delete records from the database. Like update, the number of rows affected will be returned:

## $deleted = DB::delete('delete from users');

## Running A General Statement

## Some database statements do not return any value. For these types of operations, you may use the statement method on the DB facade:

## DB::statement('drop table users');

## Listening For Query Events

## If you would like to receive each SQL query executed by your application, you may use the listen method. This method is useful for logging queries or debugging. You may register your query listener in a service provider:

## <?php

## namespace App\Providers;

## use Illuminate\Support\Facades\DB;

## use Illuminate\Support\ServiceProvider;

## class AppServiceProvider extends ServiceProvider

## {

## /\*\*

## \* Bootstrap any application services.

## \* @return void

## \*/

## public function boot()

## {

## DB::listen(function ($query) {

## // $query->sql

## // $query->bindings

## // $query->time

## });

## }

## /\*\*

## \* Register the service provider.

## \* @return void

## \*/

## public function register()

## {

## //

## }

## }

## Database Transactions

## You may use the transaction method on the DB facade to run a set of operations within a database transaction. If an exception is thrown within the transaction Closure, the transaction will automatically be rolled back. If the Closure executes successfully, the transaction will automatically be committed. You don't need to worry about manually rolling back or committing while using the transaction method:

## DB::transaction(function () {

## DB::table('users')->update(['votes' => 1]);

## DB::table('posts')->delete();

## });

## Handling Deadlocks

## The transaction method accepts an optional second argument which defines the number of times a transaction should be reattempted when a deadlock occurs. Once these attempts have been exhausted, an exception will be thrown:

## DB::transaction(function () {

## DB::table('users')->update(['votes' => 1]);

## DB::table('posts')->delete();

## }, 5);

## Manually Using Transactions

## If you would like to begin a transaction manually and have complete control over rollbacks and commits, you may use the beginTransaction method on the DB facade:

## DB::beginTransaction();

## You can rollback the transaction via the rollBack method:

## DB::rollBack();

## Lastly, you can commit a transaction via the commit method:

## DB::commit();

## {tip} Using the DB facade's transaction methods also controls transactions for the query builder and Eloquent ORM.

## Eloquent: Getting Started

## Introduction

## The Eloquent ORM included with Laravel provides a beautiful, simple ActiveRecord implementation for working with your database. Each database table has a corresponding "Model" which is used to interact with that table. Models allow you to query for data in your tables, as well as insert new records into the table.

## Before getting started, be sure to configure a database connection in config/database.php. For more information on configuring your database, check out the documentation.

## Defining Models

## To get started, let's create an Eloquent model. Models typically live in the app directory, but you are free to place them anywhere that can be auto-loaded according to your composer.json file. All Eloquent models extend Illuminate\Database\Eloquent\Model class.

## The easiest way to create a model instance is using the make:model Artisan command:

## php artisan make:model User

## 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 User --migration

## php artisan make:model User -m

## Eloquent Model Conventions

## Now, let's look at an example Flight model, which we will use to retrieve and store information from our flights database table:

## <?php

## namespace App;

## use Illuminate\Database\Eloquent\Model;

## class Flight extends Model

## {

## //

## }

## Table Names

## Note that we did not tell Eloquent which table to use for 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. You may specify a custom table by defining a table property on your model:

## <?php

## namespace App;

## 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 table has a primary key column named id. You may define a $primaryKey property to override this convention.

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

## Timestamps

## By default, Eloquent expects created\_at and updated\_at columns to exist on your tables. If you do not wish to have these columns automatically managed by Eloquent, set the $timestamps property on your model to false:

## <?php

## namespace App;

## use Illuminate\Database\Eloquent\Model;

## class Flight extends Model

## {

## /\*\*

## \* Indicates if the model should be timestamped.

## \*

## \* @var bool

## \*/

## public $timestamps = false;

## }

## If you need to customize the format of your 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;

## 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 set the CREATED\_AT and UPDATED\_AT constants in your model:

## <?php

## class Flight extends Model

## {

## const CREATED\_AT = 'creation\_date';

## const UPDATED\_AT = 'last\_update';

## }

## Database Connection

## By default, all Eloquent models will use the default database connection configured for your application. If you would like to specify a different connection for the model, use the $connection property:

## <?php

## namespace App;

## use Illuminate\Database\Eloquent\Model;

## class Flight extends Model

## {

## /\*\*

## \* The connection name for the model.

## \*

## \* @var string

## \*/

## protected $connection = 'connection-name';

## }

## Retrieving Models

## Once you have created a model and its associated database table, you are ready to start retrieving data from your database. Think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model. For example:

## <?php

## use App\Flight;

## $flights = App\Flight::all();

## foreach ($flights as $flight) {

## echo $flight->name;

## }

## Adding Additional Constraints

## The Eloquent all method will return all of the results in the model's table. Since each Eloquent model serves as a query builder, you may also add constraints to queries, and then use the get method to retrieve the results:

## $flights = App\Flight::where('active', 1)

## ->orderBy('name', 'desc')

## ->take(10)

## ->get();

## {tip} Since Eloquent models are query builders, you should review all of the methods available on the query builder. You may use any of these methods in your Eloquent queries.

## Collections

## For Eloquent methods like all and get which retrieve multiple results, an instance of Illuminate\Database\Eloquent\Collection will be returned. The Collection class provides a variety of helpful methods for working with your Eloquent results:

## $flights = $flights->reject(function ($flight) {

## return $flight->cancelled;

## });

## Of course, you may also simply loop over the collection like an array:

## foreach ($flights as $flight) {

## echo $flight->name;

## }

## Chunking Results

## If you need to process thousands of Eloquent records, use the chunk command. The chunk method will retrieve a "chunk" of Eloquent models, feeding them to a given Closure for processing. Using the chunk method will conserve memory when working with large result sets:

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

## foreach ($flights as $flight) {

## //

## }

## });

## The first argument passed to the method is the number of records you wish to receive per "chunk". The Closure passed as the second argument will be called 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.

## Using Cursors

## The cursor method allows you to iterate through your database records using a cursor, which will only execute a single query. When processing large amounts of data, the cursor method may be used to greatly reduce your memory usage:

## foreach (Flight::where('foo', 'bar')->cursor() as $flight) {

## //

## }

## Retrieving Single Models / Aggregates

## Of course, in addition to retrieving all of the records for a given table, you may also retrieve single records using find and first. Instead of returning a collection of models, these methods return a single model instance:

## // Retrieve a model by its primary key...

## $flight = App\Flight::find(1);

## // Retrieve the first model matching the query constraints...

## $flight = App\Flight::where('active', 1)->first();

## You may also call the find method with an array of primary keys, which will return a collection of the matching records:

## $flights = App\Flight::find([1, 2, 3]);

## 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, a Illuminate\Database\Eloquent\ModelNotFoundException will be thrown:

## $model = App\Flight::findOrFail(1);

## $model = App\Flight::where('legs', '>', 100)->firstOrFail();

## If the exception is not caught, a 404 HTTP response is automatically sent back to the user. It is not necessary to write explicit checks to return 404 responses when using these methods:

## Route::get('/api/flights/{id}', function ($id) {

## return App\Flight::findOrFail($id);

## });

## Retrieving Aggregates

## You may also use the count, sum, max, and other aggregate methods provided by the query builder. These methods return the appropriate scalar value instead of a full model instance:

## $count = App\Flight::where('active', 1)->count();

## $max = App\Flight::where('active', 1)->max('price');

## Inserting & Updating Models

## Inserts

## To create a new record in the database, simply create a new model instance, set attributes on the model, then call the save method:

## <?php

## namespace App\Http\Controllers;

## use App\Flight;

## use Illuminate\Http\Request;

## use App\Http\Controllers\Controller;

## class FlightController extends Controller

## {

## /\*\*

## \* Create a new flight instance.

## \*

## \* @param Request $request

## \* @return Response

## \*/

## public function store(Request $request)

## {

## // Validate the request...

## $flight = new Flight;

## $flight->name = $request->name;

## $flight->save();

## }

## }

## In this example, we simply assign the name parameter from the incoming HTTP request to the name attribute of the App\Flight model instance. When we call the save method, a record will be inserted into the database. The 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.

## 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, set any attributes you wish to update, and then call the save method. Again, the updated\_at timestamp will automatically be updated, so there is no need to manually set its value:

## $flight = App\Flight::find(1);

## $flight->name = 'New Flight Name';

## $flight->save();

## Mass Updates

## Updates can also be performed against any number of 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:

## App\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.

## {note} When issuing a mass update via Eloquent, the saved 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.

## Mass Assignment

## You may also use the create method to save a new model in a single line. The inserted model instance will be returned to you from the method. However, before doing so, you will need to specify either a fillable or guarded attribute on the model, as all Eloquent models protect against mass-assignment by default.

## A mass-assignment vulnerability occurs when a user passes an unexpected HTTP parameter through a request, and that parameter changes a column in your database you did not expect. For example, a malicious user might send an is\_admin parameter through an HTTP request, which is then passed into 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;

## use Illuminate\Database\Eloquent\Model;

## class Flight extends Model

## {

## /\*\*

## \* The attributes that are mass assignable.

## \*

## \* @var array

## \*/

## protected $fillable = ['name'];

## }

## Once we have made the attributes mass assignable, we can use the create method to insert a new record in the database. The create method returns the saved model instance:

## $flight = App\Flight::create(['name' => 'Flight 10']);

## If you already have a model instance, you may use the fill method to populate it with an array of attributes:

## $flight->fill(['name' => 'Flight 22']);

## Guarding Attributes

## While $fillable serves as a "white list" of attributes that should be mass assignable, you may also choose to use $guarded. The $guarded property should contain an array of attributes that you do not want to be mass assignable. All other attributes not in the array will be mass assignable. So, $guarded functions like a "black list". Of course, you should use either $fillable or $guarded - not both. In the example below, all attributes except for price will be mass assignable:

## <?php

## namespace App;

## use Illuminate\Database\Eloquent\Model;

## class Flight extends Model

## {

## /\*\*

## \* The attributes that aren't mass assignable.

## \*

## \* @var array

## \*/

## protected $guarded = ['price'];

## }

## If you would like to make all attributes mass assignable, you may define the $guarded property as an empty array:

## /\*\*

## \* The attributes that aren't mass assignable.

## \*

## \* @var array

## \*/

## protected $guarded = [];

## Other Creation Methods

## firstOrCreate/ firstOrNew

## There are two other methods you may use to create models by mass assigning attributes: firstOrCreate and firstOrNew. 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 given attributes.

## 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 call save manually to persist it:

## // Retrieve the flight by the attributes, or create it if it doesn't exist...

## $flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);

## // Retrieve the flight by the attributes, or instantiate a new instance...

## $flight = App\Flight::firstOrNew(['name' => 'Flight 10']);

## updateOrCreate

## You may also come across situations where you want to update an existing model or create a new model if none exists. Laravel provides an updateOrCreate method to do this in one step. Like the firstOrCreate method, updateOrCreate persists the model, so there's no need to call save():

## // If there's a flight from Oakland to San Diego, set the price to $99.

## // If no matching model exists, create one.

## $flight = App\Flight::updateOrCreate(

## ['departure' => 'Oakland', 'destination' => 'San Diego'],

## ['price' => 99]

## );

## Deleting Models

## To delete a model, call the delete method on a model instance:

## $flight = App\Flight::find(1);

## $flight->delete();

## Deleting An Existing Model By 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 retrieving it. To do so, call the destroy method:

## App\Flight::destroy(1);

## App\Flight::destroy([1, 2, 3]);

## App\Flight::destroy(1, 2, 3);

## Deleting Models By Query

## Of course, you may also run a delete statement on a set of models. In this example, we will delete all flights that are marked as inactive. Like mass updates, mass deletes will not fire any model events for the models that are deleted:

## $deletedRows = App\Flight::where('active', 0)->delete();

## {note} When executing a mass delete statement via Eloquent, the deleting and deleted model events will not be fired 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 and inserted into the database. If a model has a non-null deleted\_at value, the model has been soft deleted. To enable soft deletes for a model, use the Illuminate\Database\Eloquent\SoftDeletes trait on the model and add the deleted\_at column to your $dates property:

## <?php

## namespace App;

## use Illuminate\Database\Eloquent\Model;

## use Illuminate\Database\Eloquent\SoftDeletes;

## class Flight extends Model

## {

## use SoftDeletes;

## /\*\*

## \* The attributes that should be mutated to dates.

## \*

## \* @var array

## \*/

## protected $dates = ['deleted\_at'];

## }

## Of course, you should add the deleted\_at column to your database table. The Laravel schema builder contains a helper method to create this column:

## Schema::table('flights', function ($table) {

## $table->softDeletes();

## });

## Now, when you call the delete method on the model, the deleted\_at column will be set to the current date and time. And, 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, use the trashed method:

## if ($flight->trashed()) {

## //

## }

## 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 appear in a result set using the withTrashed method on the query:

## $flights = App\Flight::withTrashed()

## ->where('account\_id', 1)

## ->get();

## The withTrashed method may also be used on a relationship query:

## $flight->history()->withTrashed()->get();

## Retrieving Only Soft Deleted Models

## The onlyTrashed method will retrieve only soft deleted models:

## $flights = App\Flight::onlyTrashed()

## ->where('airline\_id', 1)

## ->get();

## Restoring Soft Deleted Models

## Sometimes you may wish to "un-delete" a soft deleted model. To restore a soft deleted model into an active state, use the restore method on a model instance:

## $flight->restore();

## You may also use the restore method in a query to quickly restore multiple models. Again, like other "mass" operations, this will not fire any model events for the models that are restored:

## App\Flight::withTrashed()

## ->where('airline\_id', 1)

## ->restore();

## Like the withTrashed method, the restore method may also be used on relationships:

## $flight->history()->restore();

## Permanently Deleting Models

## Sometimes you may need to truly remove a model from your database. To permanently remove a soft deleted model from the database, use the forceDelete method:

## // Force deleting a single model instance...

## $flight->forceDelete();

## // Force deleting all related models...

## $flight->history()->forceDelete();

## 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 pull "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.

## Writing Global Scopes

## Writing a global scope is simple. Define a class that implements the Illuminate\Database\Eloquent\Scope interface. This interface requires you to implement one method: apply. The apply method may add where constraints to the query as needed:

## <?php

## namespace App\Scopes;

## use Illuminate\Database\Eloquent\Scope;

## use Illuminate\Database\Eloquent\Model;

## use Illuminate\Database\Eloquent\Builder;

## class AgeScope implements Scope

## {

## /\*\*

## \* Apply the scope to a given Eloquent query builder.

## \*

## \* @param \Illuminate\Database\Eloquent\Builder $builder

## \* @param \Illuminate\Database\Eloquent\Model $model

## \* @return void

## \*/

## public function apply(Builder $builder, Model $model)

## {

## $builder->where('age', '>', 200);

## }

## }

## {tip} There is not a predefined folder for scopes in a default Laravel application, so feel free to make your own Scopes folder within your Laravel application's app directory.

## Applying Global Scopes

## To assign a global scope to a model, you should override a given model's boot method and use the addGlobalScope method:

## <?php

## namespace App;

## use App\Scopes\AgeScope;

## use Illuminate\Database\Eloquent\Model;

## class User extends Model

## {

## /\*\*

## \* The "booting" method of the model.

## \*

## \* @return void

## \*/

## protected static function boot()

## {

## parent::boot();

## static::addGlobalScope(new AgeScope);

## }

## }

## After adding the scope, a query to User::all() will produce the following SQL:

## select \* from `users` where `age` > 200

## 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:

## <?php

## namespace App;

## use Illuminate\Database\Eloquent\Model;

## use Illuminate\Database\Eloquent\Builder;

## class User extends Model

## {

## /\*\*

## \* The "booting" method of the model.

## \*

## \* @return void

## \*/

## protected static function boot()

## {

## parent::boot();

## static::addGlobalScope('age', function (Builder $builder) {

## $builder->where('age', '>', 200);

## });

## }

## }

## Removing Global Scopes

## If you would like to remove a global scope for a given query, you may use the withoutGlobalScope method. The method accepts the class name of the global scope as its only argument:

## User::withoutGlobalScope(AgeScope::class)->get();

## If you would like to remove several or even all of the global scopes, you may use the withoutGlobalScopes method:

## // Remove all of the global scopes...

## User::withoutGlobalScopes()->get();

## // Remove some of the global scopes...

## User::withoutGlobalScopes([

## FirstScope::class, SecondScope::class

## ])->get();

## Local Scopes

## Local scopes allow you to define common sets of 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, simply prefix an Eloquent model method with scope.

## Scopes should always return a query builder instance:

## <?php

## namespace App;

## use Illuminate\Database\Eloquent\Model;

## class User extends Model

## {

## /\*\*

## \* Scope a query to only include popular users.

## \*

## \* @param \Illuminate\Database\Eloquent\Builder $query

## \* @return \Illuminate\Database\Eloquent\Builder

## \*/

## public function scopePopular($query)

## {

## return $query->where('votes', '>', 100);

## }

## /\*\*

## \* Scope a query to only include active users.

## \*

## \* @param \Illuminate\Database\Eloquent\Builder $query

## \* @return \Illuminate\Database\Eloquent\Builder

## \*/

## public function scopeActive($query)

## {

## return $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 do not need to include the scope prefix when calling the method. You can even chain calls to various scopes, for example:

## $users = App\User::popular()->active()->orderBy('created\_at')->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. Scope parameters should be defined after the $query parameter:

## <?php

## namespace App;

## use Illuminate\Database\Eloquent\Model;

## class User extends Model

## {

## /\*\*

## \* Scope a query to only include users of a given type.

## \*

## \* @param \Illuminate\Database\Eloquent\Builder $query

## \* @param mixed $type

## \* @return \Illuminate\Database\Eloquent\Builder

## \*/

## public function scopeOfType($query, $type)

## {

## return $query->where('type', $type);

## }

## }

## Now, you may pass the parameters when calling the scope:

## $users = App\User::ofType('admin')->get();

## Events

## Eloquent models fire several events, allowing you to hook into various points in the model's lifecycle using the following methods: creating, created, updating, updated, saving, saved, deleting, deleted, restoring, restored. Events allow you to easily execute code each time a specific model class is saved or updated in the database.

## Whenever a new model is saved for the first time, the creating and created events will fire. If a model already existed in the database and the save method is called, the updating / updated events will fire. However, in both cases, the saving / saved events will fire.

## For example, let's define an Eloquent event listener in a service provider. Within our event listener, we will call the isValid method on the given model, and return false if the model is not valid. Returning false from an Eloquent event listener will cancel the save / update operation:

## <?php

## namespace App\Providers;

## use App\User;

## use Illuminate\Support\ServiceProvider;

## class AppServiceProvider extends ServiceProvider

## {

## /\*\*

## \* Bootstrap any application services.

## \*

## \* @return void

## \*/

## public function boot()

## {

## User::creating(function ($user) {

## return $user->isValid();

## });

## }

## /\*\*

## \* Register the service provider.

## \*

## \* @return void

## \*/

## public function register()

## {

## //

## }

## }

## 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. Observers classes have method names which reflect the Eloquent events you wish to listen for. Each of these methods receives the model as their only argument. Laravel does not include a default directory for observers, so you may create any directory you like to house your observer classes:

## <?php

## namespace App\Observers;

## use App\User;

## class UserObserver

## {

## /\*\*

## \* Listen to the User created event.

## \*

## \* @param User $user

## \* @return void

## \*/

## public function created(User $user)

## {

## //

## }

## /\*\*

## \* Listen to the User deleting event.

## \*

## \* @param User $user

## \* @return void

## \*/

## public function deleting(User $user)

## {

## //

## }

## }

## To register an observer, use the observe method on the model you wish to observe. You may register observers in the boot method of one of your service providers. In this example, we'll register the observer in the AppServiceProvider:

## <?php

## namespace App\Providers;

## use App\User;

## use App\Observers\UserObserver;

## use Illuminate\Support\ServiceProvider;

## class AppServiceProvider extends ServiceProvider

## {

## /\*\*

## \* Bootstrap any application services.

## \*

## \* @return void

## \*/

## public function boot()

## {

## User::observe(UserObserver::class);

## }

## /\*\*

## \* Register the service provider.

## \*

## \* @return void

## \*/

## public function register()

## {

## //

## }

## }

## Laravel Eloquent Database Relationships

## Eloquent Relationships Laravel: Database tables are related to one another. Eloquent manages and work with easy relationships.

## It supports some different types of relationship, which are as follows:

* One to one
* One to many
* Many to many
* Has one through
* Has many through
* One to one (polymorphic)
* One to many (polymorphic)

## Defining Eloquent Relationships

Eloquent relationships are specified as a method on our Eloquent model classes.

Eloquent models relationships also serve as powerful query builders, defining relationships as a method, provides powerful method chaining and query capabilities.

Example:

$user->posts()->where(‘active’, 10)->get();

**One to One**

One to One: A one to one relationships is an essential relation

## A one-to-one relationship is an essential relation. For defining these relationships, we are going to place a phone method on the owner model.

## The phone method will call the hasOne method and return its result:

## hasOne('App\Phone');

## }

## }

## The hasOne method is the name which is passed as the first argument, it is a related model. The relationship is specified; we retrieve the related record by using Eloquent`s dynamic properties.

## Dynamic properties allow us to access relationship methods as if they were properties defined on the model:

## $phone = User::find(1)->phone;

## Eloquent consider as the foreign key of the relationship, which is based on the model name.

## In that case, the Phone model is automatically assumed to have a user\_id foreign key.

## We pass a second argument to the hasOne method.

## return $this->hasOne(‘App\Phone’, ‘foreign\_key’);

## The foreign key should have a value matching the id column of the parent.

## Eloquent will consider the value of the user`s id column in the user\_id column.

## To use the relationship of a value other than id, we pass the third argument to the hasOne method defining our custom key.

## return $this->hasOne(‘App\Phone’, ‘foreign\_key’, ‘local\_key’);

## The Inverse of the Relationship

## We can easily access the Phone model from our Owner.

## To define a relationship on the Phone model that will let us access the Owner.

## We can define the inverse of a hasOne relationship by using the belongsTo method:

## In the above example, Eloquent will try to match the user\_id from the Phone model to an id on the Owner model.

## Eloquent

## consider as the default foreign key name by the name of the relationship method and the method name with \_id.

## The foreign key on the Phone model is not user\_id, we pass a custom key as the second argument to the belongsTo method:

## For Example:

## /\*\*

## \* Get the user that owns the phone.

## \*/

## public function owner()

## {

## return $this->belongsTo('App\User', 'foreign\_key');

## }

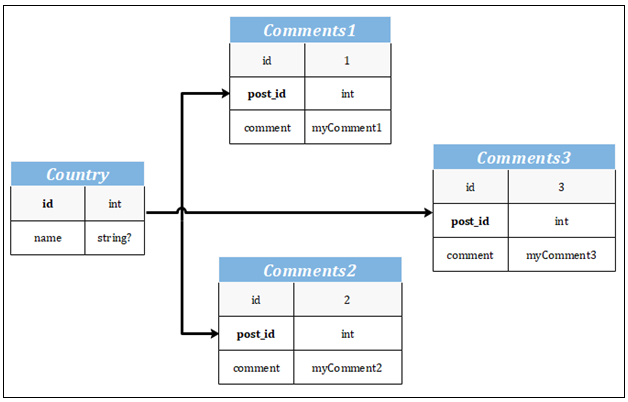
## Our parent model do not use id (as primary key), the third argument to the belongsTo method by defining our parent table as a custom key:

## public function owner()

## {

## return $this->belongsTo('App\User', 'foreign\_key', 'other\_key');

## }



**One To Many**

A one-to-many relationship defines relationships where a single model owns any amount of the other models. One-to-many relationships are defined by placing our Eloquent model

hasMany('App\Comment');

}

}

Eloquent will automatically decide the properforeign key column on the Comment model.

Eloquent will assume the foreign key on the Comment model as post\_id.

The relationship has been specified; we canenter the collection of the comments by entering the comments correctly.

Eloquent provides “dynamic properties”, wecan enter relationship methods as they defined as the properties on the model.

## $comments = App\Post::find(1)->comments;

## foreach ($comments as $comment)

## {

## //statement

## }

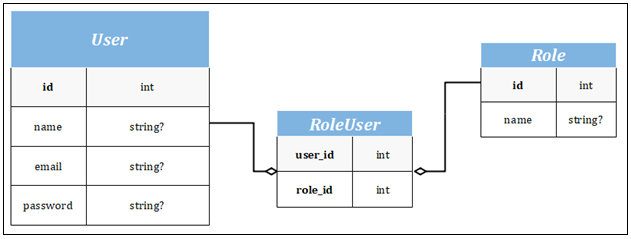
## We can add constraints to the comments by easily call the comments method and continuing the chain conditions onto the query.

## $comment = App\Post::find(1)->comments()->where('title', 'foo')->first();

## The hasOne method, we also override the foreign and local keys by passing an additional arguments to the hasMany method.

## return $this->hasMany('App\Comment', 'foreign\_key');

## return $this->hasMany('App\Comment', 'foreign\_key', 'local\_key');



**Many to Many**

Many-to-many relations are complicated than hasOne and hasMany relationships.

Forexample: - Many users have the role of “Admin”. Tospecify the many-to-many relationship, the three database tables which are asfollows: users,roles,and role\_user.

The role\_user table is derived from the alphabetical order of therelated model names, and it contains the user\_id and role\_id columns.

## Many-to-many relationships are specified by writing a method which will return the result of the belongsToMany method.

## Let`s define the roles method on our User model:

## belongsToMany('App\Role');

## }

## The relationship is specified, we access the user`s roles by using the roles dynamic property:

## $user = App\User::find(1);

## foreach ($user->roles as $role)

## {

## //

## }

## We call the roles method to continue query constraints onto the relationship

## $roles = App\User::find(1)->roles()->orderBy('name')->get();

## The table name of the relationship joins table, and Eloquent will join the two related model names in the alphabetical order.

## We are free to override this convention as a second argument to the belongsToMany method:

## return $this->belongsToMany(‘App\Role’ , ‘role\_user’);

## We also customize the column names of the keys on the table by passing additional arguments to the belongsToMany method.

## The foreign key name of the model is the third argument, which is defining the relationship. The fourth argument joins the foreign key name of the model.

## return $this->belongsToMany('App\Role', 'role\_user', 'user\_id', 'role\_id');

## Defining Custom Intermediate Table Models

## To define a custom model for representing the intermediate table of our relationship, we call the using method when defining the relationship.

## Custom many-to-many pivot model extends the Illuminate\Database\Eloquent\Relations\Pivot class, for custom polymorphic many-to-many pivot models extend the Illuminate\Database\Eloquent\Relations\MorphPivot class.

## Example: - we define a Role that uses a custom RoleUser pivot model:

## belongsToMany('App\User')->using('App\RoleUser');

## }

## }

## To define the RoleUser model, we extend the Pivot class:

## We combine using with Pivot to retrieve columns from the intermediate table.

## For example: -

## We retrieve the created\_by and update\_by columns by the RoleUser pivot table by passing the column names to the with Pivot method:

## belongsToMany('App\User')

## ->using('App\RoleUser')

## ->withPivot([

## 'created\_by',

## 'updated\_by'

## ]);

## }

## }

## Note: Pivot models do not use the SoftDeletes. If we need to delete pivot records consider as

## converting our pivot model to an actual Eloquent model.

## Has One Through

## The “has-one-through” relationship model links across a single intermediate relation.

## For Example: -

## Each supplier has one user, and each user is associated with one user record, then the supplier model accesses the user`s history through the user.

## The database tables necessary, which defines this relationship:

## users

## id - integer

## supplier\_id - integer

## suppliers

## id - integer

## history

## id - integer

## user\_id – integer

## The history table does not contain supplier\_id columns; the hasOneThrough relation provides access to the owner`s history to the supplier model.

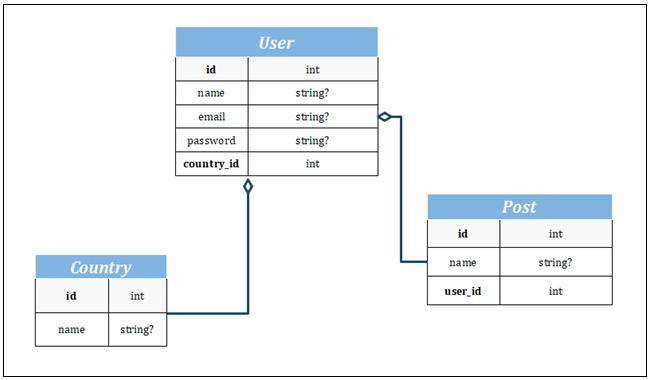
## Let`s define it on the Supplier model:

## hasOneThrough('App\History', 'App\User');

## }

## }

## The first argument that is passed to the hasOneThrough method is the name of the final model that we will access, and the second argument is the name of the intermediate model.



**Has Many Through**

The “has-many-through” provides a simple shortcut for accessing distant relations with an intermediate relation.

Let`s look at the tables requirement fordefining relationship:

Countries

id – integer

name – string

users

id – integer

country\_id – integer

name – string

posts

id – integer

user\_id – integer

title - string

The posts do not contain a country\_id column;the hasManyThrough relationprovides access to a country`s posts with $country->posts.

## Eloquent go through the country\_id to the intermediate of the users table.

## After finding the matching user IDs, it will used to query the posts table.

## Let`s define it on the Country model:

## The first argument which is passed to the hasManyThrough method is the name of the final model for access, while the second argument is the name of the intermediate model.

## Polymorphic Relationships

## A polymorphic relationship allows the target model that belongs to more than one type of model by using a single association.

## One to One (Polymorphic)

## A one-to-one polymorphic relation is similar to a simple one-to-one relation; the target model belongs to more than one type of model on a single association.

## posts

## id – integer

## name - string

## users

## id - integer

## name - string

## images

## id - integer

## url – string

## imageable\_id – integer

## imageable\_type - string

## The imageable\_id column will contain the ID value of the user; the imageable\_type column will contain the class name of the parent model. The imagebale\_type column is used by Eloquent to determine the “type” of parent model for return when accessing the imageable relation.

## Model Structure

## The model definitions needed to build the relationship:

## morphTo();

## }

## }

## class Post extends Model

## {

## public function image()

## {

## return $this->morphOne('App\Image', 'imageable');

## }

## }

## class User extends Model

## {

## public function image()

## {

## return $this->morphOne('App\Image', 'imageable');

## }

## }

## Retrieving the Relationship

## Once the database table and models are defined, we can access the relationships with the models.

## For example: - for retrieve the image for a post, we use the image dynamic property:

## $post = App\Post::find(1);

## $image = $post->image;

## We also retrieve the parent from the polymorphic model by easily accessing the name of the method that performs the call to morphTo.

## The imageable method on the Image model will access that method as a dynamic property:

## $image = App\Image::find(1);

## $imageable = $image->imageable;

## The imageable relation on the Image model will return Post or User instance, it depends on which type of model owns the image.

## One to Many (Polymorphic)

## A one-to-many polymorphic relation is the same as a simple one-to-many relation.

## In this, the target model belongs to more than one type of model on a single association.

## By using polymorphic relationships, we use a single comments table for both.

## Table Structure

## posts

## id – integer

## title – string

## body - text

## videos

## id - integer

## title - string

## url - string

## comments

## id – integer

## body – text

## commentable\_id – integer

## commentable\_type - string

## Model Structure

## morphTo();

## }

## }

## class Post extends Model

## {

## public function comments()

## {

## return $this->morphMany('App\Comment', 'commentable');

## }

## }

## class Video extends Model

## {

## public function comments()

## {

## return $this->morphMany('App\Comment', 'commentable');

## }

## }

## Retrieving the Relationship

## Once the database table and models are defined, we access the relationships with our models.

## $post = App\Post::find(1);

## foreach ($post->comments as $comment) {

## //

## }

## We can also retrieve the owner from the polymorphic model by allowing the name of the method, which performs the call to morphTo.

## The commentable method on the Comment model will access that method as a dynamic property:

## $comment = App\Comment::find(1);

## $commentable = $comment->commentable;

## The comment-able relation of the Comment model returns a Post or Video depends on the type of model that owns the comment.

References

1. <https://laravel-guide.readthedocs.io/en/latest/controllers/>
2. <https://laravel-guide.readthedocs.io/en/latest/database/>
3. <https://laravel-guide.readthedocs.io/en/latest/eloquent/>
4. <https://www.tutorialandexample.com/laravel-eloquent-database-relationships>