Intermediate Task List

#Introduction

This quickstart guide provides an intermediate introduction to the Laravel framework and includes content on database migrations, the Eloquent ORM, routing, authentication, authorization, dependency injection,

validation, views, and Blade templates. This is a great starting point if you are familiar with the basics of the Laravel framework or PHP frameworks in general.

To sample a basic selection of Laravel features, we will build a task list we can use to track all of the tasks we want to accomplish (the typical "to-do list" example). In contrast to the "basic" quickstart, this tutorial will allow users to create accounts and authenticate with the application. The complete, finished source code for this project is available on GitHub.

#Installation

Of course, first you will need a fresh installation of the Laravel framework. You may use the Homestead virtual machine or the local PHP environment of your choice to run the framework. Once your local environment is ready, you may install the Laravel framework using Composer:

```
composer create-project laravel/laravel quickstart --prefer-dist
```

You're free to just read along for the remainder of this quickstart; however, if you would like to download the source code for this quickstart and run it on your local machine, you may clone its Git repository and install its dependencies:

```
git clone https://github.com/laravel/quickstart-intermediate quickstart
cd quickstart

composer install
php artisan migrate
```

For more complete documentation on building a local Laravel development environment, check out the full Homestead and installation documentation.

#Prepping The Database

Database Migrations

First, let's use a migration to define a database table to hold all of our tasks. Laravel's database migrations provide an easy way to define your database table structure and modifications using fluent, expressive PHP

code. Instead of telling your team members to manually add columns to their local copy of the database, your teammates can simply run the migrations you push into source control.

The `users` Table

Since we are going to allow users to create their accounts within the application, we will need a table to store all of our users. Thankfully, Laravel already ships with a migration to create a basic `users` table, so we do not need to manually generate one. The default migration for the `users` table is located in the `database/migrations` directory.

The `tasks` Table

Next, let's build a database table that will hold all of our tasks. The Artisan CLI can be used to generate a variety of classes and will save you a lot of typing as you build your Laravel projects. In this case, let's use the `make:migration` command to generate a new database migration for our `tasks` table:

```
php artisan make:migration create_tasks_table --create=tasks
```

The migration will be placed in the `database/migrations` directory of your project. As you may have noticed, the `make:migration` command already added an auto-incrementing ID and timestamps to the migration file. Let's edit this file and add an additional `string` column for the name of our tasks, as well as a `user_id` column which will link our `tasks` and `users` tables:

```
<?php

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

class CreateTasksTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('tasks', function (Blueprint $table) {
            $table->increments('id');
```

```
            $table->integer('user_id')->index();
            $table->string('name');
            $table->timestamps();
        });
    }

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

To run our migrations, we will use the `migrate` Artisan command. If you are using Homestead, you should run this command from within your virtual machine, since your host machine will not have direct access to the database:

```
php artisan migrate
```

This command will create all of our database tables. If you inspect the database tables using the database client of your choice, you should see new `tasks` and `users` tables which contains the columns defined in our migration. Next, we're ready to define our Eloquent ORM models!

Eloquent Models

Eloquent is Laravel's default ORM (object-relational mapper). Eloquent makes it painless to retrieve and store data in your database using clearly defined "models". Usually, each Eloquent model corresponds directly with a single database table.

The `User` Model

First, we need a model that corresponds to our `users` database table. However, if you look in the `app` directory of your project, you will see that Laravel already ships with a `User` model, so we do not need to generate one manually.

The `Task` Model

So, let's define a `Task` model that corresponds to our `tasks` database table we just created. Again, we can use an Artisan command to generate this model. In this case, we'll use the `make:model` command:

```
php artisan make:model Task
```

The model will be placed in the `app` directory of your application. By default, the model class is empty. We do not have to explicitly tell the Eloquent model which table it corresponds to because it will assume the database table is the plural form of the model name. So, in this case, the `Task` model is assumed to correspond with the `tasks` database table.

Let's add a few things to this model. First, we will state that the `name` attribute on the model should be "mass-assignable":

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Task extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

We'll learn more about how to use Eloquent models as we add routes to our application. Of course, feel free to consult the complete Eloquent documentation for more information.

Eloquent Relationships

Now that our models are defined, we need to link them. For example, our `User` can have many `Task` instances, while a `Task` is assigned to one `User`. Defining a relationship will allow us to fluently walk through our relations like so:

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

foreach ($user->tasks as $task) {
    echo $task->name;
}
```

The `tasks` Relationship

First, let's define the `tasks` relationship on our `User` model. Eloquent relationships are defined as methods on models. Eloquent supports several different types of relationships, so be sure to consult the full Eloquent documentation for more information. In this case, we will define a `tasks` function on the `User` model which calls the `hasMany` method provided by Eloquent:

```
<?php

namespace App;

// Namespace Imports...

class User extends Model implements AuthenticatableContract,
                                    AuthorizableContract,
                                    CanResetPasswordContract
{
    use Authenticatable, Authorizable, CanResetPassword;

    // Other Eloquent Properties...

    /**
     * Get all of the tasks for the user.
     */
    public function tasks()
    {
        return $this->hasMany(Task::class);
    }
}
```

The `user` Relationship

Next, let's define the `user` relationship on the `Task` model. Again, we will define the relationship as a method on the model. In this case, we will use the `belongsTo` method provided by Eloquent to define the relationship:

```
<?php

namespace App;

use App\User;
use Illuminate\Database\Eloquent\Model;

class Task extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];

    /**
     * Get the user that owns the task.
     */
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Wonderful! Now that our relationships are defined, we can start building our controllers!

#Routing

In the basic version of our task list application, we defined all of our logic using Closures within our `routes.php` file. For the majority of this application, we will use controllers to organize our routes. Controllers will allow us to break out HTTP request handling logic across multiple files for better organization.

Displaying A View

We will have a single route that uses a Closure: our `/` route, which will simply be a landing page for application guests. So, let's fill out our `/` route. From this route, we want to render an HTML template that contains the "welcome" page:

In Laravel, all HTML templates are stored in the `resources/views` directory, and we can use the `view` helper to return one of these templates from our route:

```
Route::get('/', function () {
    return view('welcome');
});
```

Of course, we need to actually define this view. We'll do that in a bit!

Authentication

Remember, we also need to let users create accounts and login to our application. Typically, it can be a tedious task to build an entire authentication layer into a web application. However, since it is such a common need, Laravel attempts to make this procedure totally painless.

First, notice that there is already a `app/Http/Controllers/Auth/AuthController` included in your Laravel application. This controller uses a special `AuthenticatesAndRegistersUsers` trait which contains all of the necessary logic to create and authenticate users.

Authentication Routes

So, what's left for us to do? Well, we still need to create the registration and login templates as well as define the routes to point to the authentication controller. First, let's add the routes we need to our `app/Http/routes.php` file:

```
// Authentication Routes...
Route::get('auth/login', 'Auth\AuthController@getLogin');
Route::post('auth/login', 'Auth\AuthController@postLogin');
Route::get('auth/logout', 'Auth\AuthController@getLogout');

// Registration Routes...
Route::get('auth/register', 'Auth\AuthController@getRegister');
Route::post('auth/register', 'Auth\AuthController@postRegister');
```

Authentication Views

Authentication requires us to create `login.blade.php` and `register.blade.php` within the `resources/views/auth` directory. Of course, the design and styling of these views is unimportant; however, they should at least contain some basic fields.

The `register.blade.php` file should contain a form that includes `name`, `email`, `password`, and `password_confirmation` fields and makes a `POST` request to the `/auth/register` route.

The `login.blade.php` file should contain a form that includes `email` and `password` fields and makes a `POST` request to `/auth/login`.

> **Note:** If you would like to view complete examples for these views, remember that the entire application's source code is available on GitHub.

The Task Controller

Since we know we're going to need to retrieve and store tasks, let's create a `TaskController` using the Artisan CLI, which will place the new controller in the `app/Http/Controllers` directory:

```
php artisan make:controller TaskController --plain
```

Now that the controller has been generated, let's go ahead and stub out some routes in our `app/Http/routes.php` file to point to the controller:

```
Route::get('/tasks', 'TaskController@index');
Route::post('/task', 'TaskController@store');
Route::delete('/task/{task}', 'TaskController@destroy');
```

Authenticating All Task Routes

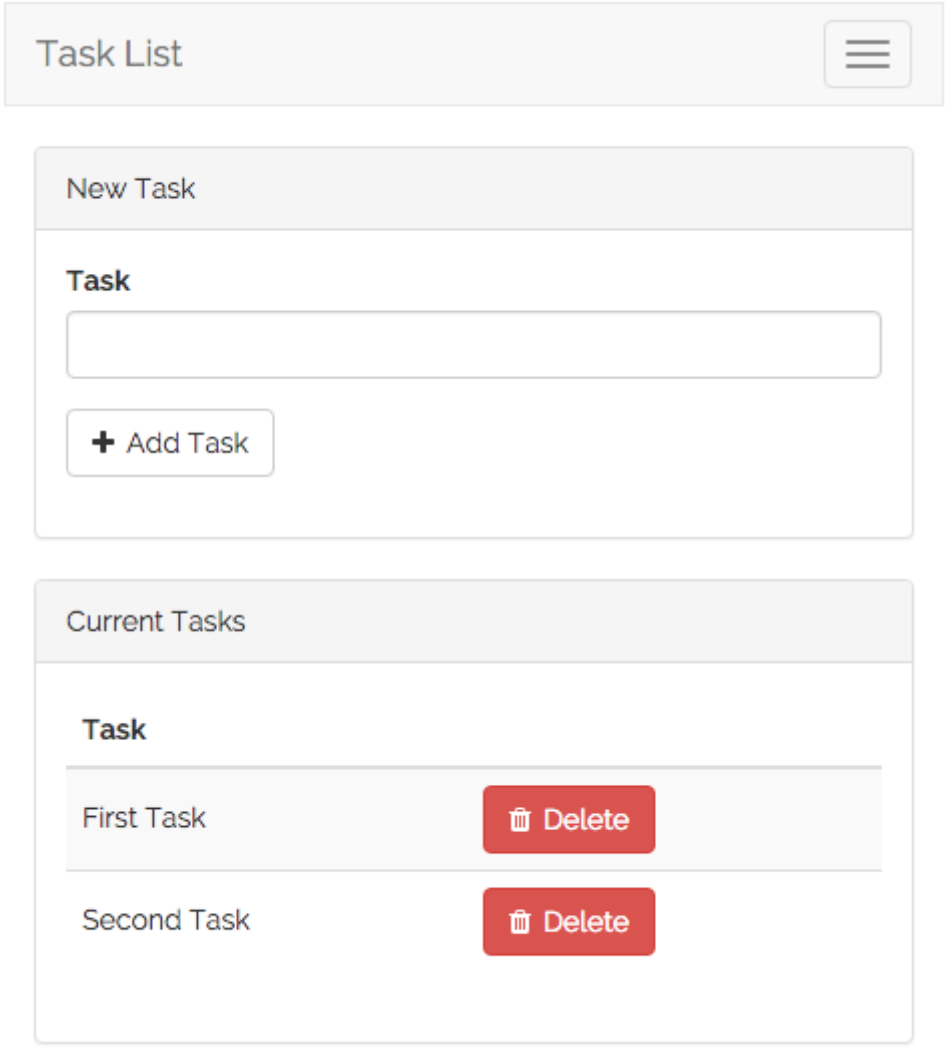
For this application, we want all of our task routes to require an authenticated user. In other words, the user must be "logged into" the application in order to create a task. So, we need to restrict access to our task routes to only authenticated users. Laravel makes this a cinch using middleware.

To require an authenticated users for all actions on the controller, we can add a call to the `middleware` method from the controller's constructor. All available route middleware are defined in the `app/Http/Kernel.php` file. In this case, we want to assign the `auth` middleware to all actions on the controller:

```php
<?php

namespace App\Http\Controllers;



use App\Http\Requests;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class TaskController extends Controller
{
    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
    }
}
```

#Building Layouts & Views

This application only has a single view which contains a form for adding new tasks as well as a listing of all current tasks. To help you visualize the view, here is a screenshot of the finished application with basic Bootstrap CSS styling applied:



Defining The Layout

Almost all web applications share the same layout across pages. For example, this application has a top navigation bar that would be typically present on every page (if we had more than one). Laravel makes it easy to share these common features across every page using Blade **layouts**.

As we discussed earlier, all Laravel views are stored in `resources/views`. So, let's define a new layout view in `resources/views/layouts/app.blade.php`. The `.blade.php` extension instructs the framework to use the Blade templating engine to render the view. Of course, you may use plain PHP templates with Laravel. However, Blade provides convenient short-cuts for writing cleaner, terse templates.

Our `app.blade.php` view should look like the following:

```
// resources/views/layouts/app.blade.php


<!DOCTYPE html>
```

```
<html lang="en">
    <head>
        <title>Laravel Quickstart - Intermediate</title>

        <!-- CSS And JavaScript -->
    </head>

    <body>
        <div class="container">
            <nav class="navbar navbar-default">
                <!-- Navbar Contents -->
            </nav>
        </div>

        @yield('content')
    </body>
</html>
```

Note the `@yield('content')` portion of the layout. This is a special Blade directive that specifies where all child pages that extend the layout can inject their own content. Next, let's define the child view that will use this layout and provide its primary content.

Defining The Child View

Great, our application layout is finished. Next, we need to define a view that contains a form to create a new task as well as a table that lists all existing tasks. Let's define this view in `resources/views/tasks/index.blade.php`, which will correspond to the `index` method in our `TaskController`.

We'll skip over some of the Bootstrap CSS boilerplate and only focus on the things that matter. Remember, you can download the full source for this application on GitHub:

```
// resources/views/tasks/index.blade.php

@extends('layouts.app')

@section('content')
```

```
    <!-- Bootstrap Boilerplate... -->

    <div class="panel-body">
        <!-- Display Validation Errors -->
        @include('common.errors')

        <!-- New Task Form -->
        <form action="/task" method="POST" class="form-horizontal">
            {{ csrf_field() }}

            <!-- Task Name -->
            <div class="form-group">
                <label for="task-name" class="col-sm-3 control-label">Task</label>

                <div class="col-sm-6">
                    <input type="text" name="name" id="task-name" class="form-control">
                </div>
            </div>

            <!-- Add Task Button -->
            <div class="form-group">
                <div class="col-sm-offset-3 col-sm-6">
                    <button type="submit" class="btn btn-default">
                        <i class="fa fa-plus"></i> Add Task
                    </button>
                </div>
            </div>
        </form>
    </div>

    <!-- TODO: Current Tasks -->
@endsection
```

A Few Notes Of Explanation

Before moving on, let's talk about this template a bit. First, the `@extends` directive informs Blade that we are using the layout we defined at `resources/views/layouts/app.blade.php`. All of the content between `@section('content')` and `@endsection` will be injected into the location of the `@yield('content')` directive within the `app.blade.php` layout.

Now we have defined a basic layout and view for our application. Let's go ahead and return this view from the `index` method of our `TaskController`:

```
/**
 * Display a list of all of the user's task.
 *
 * @param  Request  $request
 * @return Response
 */
public function index(Request $request)
{
    return view('tasks.index');
}
```

Next, we're ready to add code to our `POST /task` route's controller method to handle the incoming form input and add a new task to the database.

> **Note:** The `@include('common.errors')` directive will load the template located at `resources/views/common/errors.blade.php`. We haven't defined this template, but we will soon!

Adding Tasks

Validation

Now that we have a form in our view, we need to add code to our `TaskController@store` method to validate the incoming form input and create a new task. First, let's validate the input.

For this form, we will make the `name` field required and state that it must contain less than `255` characters. If the validation fails, we want to redirect the user back to the `/tasks` URL, as well as flash the old input and errors into the session:

```
/**
 * Create a new task.
 *
 * @param  Request  $request
 * @return Response
```

```
     */
    public function store(Request $request)
    {
        $this->validate($request, [
            'name' => 'required|max:255',
        ]);

        // Create The Task...
    }
```

If you followed along with the basic quickstart, you'll notice this validation code looks quite a bit different! Since we are in a controller, we can leverage the convenience of the `ValidatesRequests` trait that is included in the base Laravel controller. This trait exposes a simple `validate` method which accepts a request and an array of validation rules.

We don't even have to manually determine if the validation failed or do manual redirection. If the validation fails for the given rules, the user will automatically be redirected back to where they came from and the errors will automatically be flashed to the session. Nice!

The `$errors` Variable

Remember that we used the `@include('common.errors')` directive within our view to render the form's validation errors. The `common.errors` will allow us to easily show validation errors in the same format across all of our pages. Let's define the contents of this view now:

```
// resources/views/common/errors.blade.php

@if (count($errors) > 0)
    <!-- Form Error List -->
    <div class="alert alert-danger">
        <strong>Whoops! Something went wrong!</strong>

        <br><br>

        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
```

```
@endif
```

Note: The `$errors` variable is available in **every** Laravel view. It will simply be an empty instance of `ViewErrorBag` if no validation errors are present.

Creating The Task

Now that input validation is handled, let's actually create a new task by continuing to fill out our route. Once the new task has been created, we will redirect the user back to the `/tasks` URL. To create the task, we are going to leverage the power of Eloquent's relationships.

Most of Laravel's relationships expose a `create` method, which accepts an array of attributes and will automatically set the foreign key value on the related model before storing it in the database. In this case, the `create` method will automatically set the `user_id` property of the given task to the ID of the currently authenticated user, which we are accessing using `$request->user()`:

```
/**
 * Create a new task.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $this->validate($request, [

        'name' => 'required|max:255',
    ]);

    $request->user()->tasks()->create([
        'name' => $request->name,
    ]);

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

Great! We can now successfully create tasks. Next, let's continue adding to our view by building a list of all existing tasks.

#Displaying Existing Tasks

First, we need to edit our `TaskController@index` method to pass all of the existing tasks to the view. The `view` function accepts a second argument which is an array of data that will be made available to the view, where each key in the array will become a variable within the view. For example, we could do this:

```
/**
 * Display a list of all of the user's task.
 *
 * @param  Request  $request
 * @return Response
 */
public function index(Request $request)
{
    $tasks = Task::where('user_id', $request->user()->id)->get();

    return view('tasks.index', [
        'tasks' => $tasks,
    ]);
}
```

However, let's explore some of the dependency injection capabilities of Laravel to inject a `TaskRepository` into our `TaskController`, which we will use for all of our data access.

Dependency Injection

Laravel's service container is one of the most powerful features of the entire framework. After reading this quickstart, be sure to read over all of the container's documentation.

Creating The Repository

As we mentioned earlier, we want to define a `TaskRepository` that holds all of our data access logic for the `Task` model. This will be especially useful if the application grows and you need to share some Eloquent queries across the application.

So, let's create an `app/Repositories` directory and add a `TaskRepository` class. Remember, all Laravel `app` folders are auto-loaded using the PSR-4 auto-loading standard, so you are free to create as many extra directories as needed:

```
<?php

namespace App\Repositories;

use App\User;
use App\Task;

class TaskRepository
{
    /**
     * Get all of the tasks for a given user.
     *
     * @param  User  $user
     * @return Collection
     */
    public function forUser(User $user)
    {
        return Task::where('user_id', $user->id)
                    ->orderBy('created_at', 'asc')
                    ->get();
    }
}
```

Injecting The Repository

Once our repository is defined, we can simply "type-hint" it in the constructor of our `TaskController` and utilize it within our `index` route. Since Laravel uses the container to resolve all controllers, our dependencies will automatically be injected into the controller instance:

```
<?php

namespace App\Http\Controllers;

use App\Task;
use App\Http\Requests;
```

```php
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
use App\Repositories\TaskRepository;

class TaskController extends Controller
{
    /**
     * The task repository instance.
     *
     * @var TaskRepository
     */
    protected $tasks;

    /**
     * Create a new controller instance.
     *
     * @param  TaskRepository  $tasks
     * @return void
     */
    public function __construct(TaskRepository $tasks)
    {
        $this->middleware('auth');

        $this->tasks = $tasks;
    }

    /**
     * Display a list of all of the user's task.
     *
     * @param  Request  $request
     * @return Response
     */
    public function index(Request $request)
    {
        return view('tasks.index', [
            'tasks' => $this->tasks->forUser($request->user()),
        ]);
    }
}
```

Displaying The Tasks

Once the data is passed, we can spin through the tasks in our `tasks/index.blade.php` view and display them in a table. The `@foreach` Blade construct allows us to write concise loops that compile down into blazing fast plain PHP code:

```
@extends('layouts.app')

@section('content')
    <!-- Create Task Form... -->

    <!-- Current Tasks -->
    @if (count($tasks) > 0)
        <div class="panel panel-default">
            <div class="panel-heading">
                Current Tasks
            </div>

            <div class="panel-body">
                <table class="table table-striped task-table">

                    <!-- Table Headings -->
                    <thead>
                        <th>Task</th>
                        <th> </th>
                    </thead>

                    <!-- Table Body -->
                    <tbody>
                        @foreach ($tasks as $task)
                            <tr>
                                <!-- Task Name -->
                                <td class="table-text">
                                    <div>{{ $task->name }}</div>
                                </td>

                                <td>
```

```
                                <!-- TODO: Delete Button -->
                            </td>
                        </tr>
                    @endforeach
                </tbody>
            </table>
        </div>
    </div>
@endif
@endsection
```

Our task application is almost complete. But, we have no way to delete our existing tasks when they're done. Let's add that next!

#Deleting Tasks

Adding The Delete Button

We left a "TODO" note in our code where our delete button is supposed to be. So, let's add a delete button to each row of our task listing within the `tasks/index.blade.php` view. We'll create a small single-button form for each task in the list. When the button is clicked, a `DELETE /task` request will be sent to the application which will trigger our `TaskController@destroy` method:

```
<tr>
    <!-- Task Name -->
    <td class="table-text">
        <div>{{ $task->name }}</div>
    </td>

    <!-- Delete Button -->
    <td>
        <form action="/task/{{ $task->id }}" method="POST">
            {{ csrf_field() }}
            {{ method_field('DELETE') }}

            <button>Delete Task</button>
        </form>
```

```
        </form>
    </td>
</tr>
```

A Note On Method Spoofing

Note that the delete button's form `method` is listed as `POST`, even though we are responding to the request using a `Route::delete` route. HTML forms only allow the `GET` and `POST` HTTP verbs, so we need a way to spoof a `DELETE` request from the form.

We can spoof a `DELETE` request by outputting the results of the `method_field('DELETE')` function within our form. This function generates a hidden form input that Laravel recognizes and will use to override the actual HTTP request method. The generated field will look like the following:

```
<input type="hidden" name="_method" value="DELETE">
```

Route Model Binding

Now, we're almost ready to define the `destroy` method on our `TaskController`. But, first, let's revisit our route declaration for this route:

```
Route::delete('/task/{task}', 'TaskController@destroy');
```

Without adding any additional code, Laravel would inject the given task ID into the `TaskController@destroy` method, like so:

```
/**
 * Destroy the given task.
 *
 * @param  Request  $request
 * @param  string  $taskId
 * @return Response
 */
public function destroy(Request $request, $taskId)
{
    //
}
```

However, the very first thing we will need to do in this method is retrieve the `Task` instance from the database using the given ID. So, wouldn't it be nice if Laravel could just inject the `Task` instance that matches the ID in the first place? Let's make it happen!

In your `app/Providers/RouteServiceProvider.php` file's `boot` method, let's add the following line of code:

```
$router->model('task', 'App\Task');
```

This small line of code will instruct Laravel to retrieve the `Task` model that corresponds to a given ID whenever it sees `{task}` in a route declaration. Now we can define our destroy method like so:

```
/**
 * Destroy the given task.
 *
 * @param  Request  $request
 * @param  Task  $task
 * @return Response
 */
public function destroy(Request $request, Task $task)
{
    //
}
```

Authorization

Now, we have a `Task` instance injected into our `destroy` method; however, we have no guarantee that the authenticated user actually "owns" the given task. For example, a malicious request could have been concocted in an attempt to delete another user's tasks by passing a random task ID to the `/tasks/{task}` URL. So, we need to use Laravel's authorization capabilities to make sure the authenticated user actually owns the `Task` instance that was injected into the route.

Creating A Policy

Laravel uses "policies" to organize authorization logic into simple, small classes. Typically, each policy corresponds to a model. So, let's create a `TaskPolicy` using the Artisan CLI, which will place the generated file in `app/Policies/TaskPolicy.php`:

```
php artisan make:policy TaskPolicy
```

Next, let's add a `destroy` method to the policy. This method will receive a `User` instance and a `Task` instance. The method should simply check if the user's ID matches the `user_id` on the task. In fact, all policy methods should either return `true` or `false`:

```
<?php

namespace App\Policies;

use App\User;
use App\Task;
use Illuminate\Auth\Access\HandlesAuthorization;

class TaskPolicy
{
    use HandlesAuthorization;

    /**
     * Determine if the given user can delete the given task.
     *
     * @param  User  $user
     * @param  Task  $task
     * @return bool
     */
    public function destroy(User $user, Task $task)
    {
        return $user->id === $task->user_id;
    }
}
```

Finally, we need to associate our `Task` model with our `TaskPolicy`. We can do this by adding a line in the `app/Providers/AuthServiceProvider.php` file's `$policies` property. This will inform Laravel which policy should be used whenever we try to authorize an action on a `Task` instance:

```
/**
 * The policy mappings for the application.
 *
 * @var array
 */
```

```
protected $policies = [
    Task::class => TaskPolicy::class,
];
```

Authorizing The Action

Now that our policy is written, let's use it in our `destroy` method. All Laravel controllers may call an `authorize` method, which is exposed by the `AuthorizesRequest` trait:

```
/**
 * Destroy the given task.
 *
 * @param  Request  $request
 * @param  Task  $task
 * @return Response
 */
public function destroy(Request $request, Task $task)
{
    $this->authorize('destroy', $task);

    // Delete The Task...
}
```

Let's examine this method call for a moment. The first argument passed to the `authorize` method is the name of the policy method we wish to call. The second argument is the model instance that is our current concern. Remember, we recently told Laravel that our `Task` model corresponds to our `TaskPolicy`, so the framework knows on which policy to fire the `destroy` method. The current user will automatically be sent to the policy method, so we do not need to manually pass it here.

If the action is authorized, our code will continue executing normally. However, if the action is not authorized (meaning the policy's `destroy` method returned `false`), a 403 exception will be thrown and an error page will be displayed to the user.

> **Note:** There are several other ways to interact with the authorization services Laravel provides. Be sure to browse the complete authorization documentation.

Deleting The Task

Finally, let's finish adding the logic to our `destroy` method to actually delete the given task. We can use Eloquent's `delete` method to delete the given model instance in the database. Once the record is deleted, we will redirect the user back to the `/tasks` URL:

```
/**
 * Destroy the given task.
 *
 * @param  Request  $request
 * @param  Task  $task
 * @return Response
 */
public function destroy(Request $request, Task $task)
{
    $this->authorize('destroy', $task);

    $task->delete();

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