Basic ToDo Application in Laravel 5

Boban Joksimoski

December 16, 2018

Contents

1 Introduction

With the release of Laravel 5, there have been a bunch of backwards incompatible changes, new features added as well as the usual influx of new users, a beginner example can be shown using a basic to-do application. The app covers a wide range of concepts, links to relevant learning material where possible and should make for a great introduction to the framework.

2 The example and its aim

Ouphp artisaeon make:controller PhotoController –resourcer `TO-DO` application will consist of one or more projects, each with its own list of tasks. You will be able to create, list, modify and delete both tasks and projects.

2.1 Beginner Steps:

1. Installing and setting up Laravel
2. Installing extra packages that will make development easier
3. Using migrations and seeds
4. Learning how to use resourceful controllers
5. Learning how to use views (including the blade templating language and content layouts)
6. Handling model relations

3 Installation

Installation of Laravel differs based on the OS and the method you wish to use. See the Laravel Installation part in the official Laravel 5 documentation. Our preffered way is by using Composer. Laragon is another application that gives easy setup of Laravel

3.1 Composer

Installing Laravel is extremely quick and painless thanks to Composer. First you'll need to install Composer if you haven't already (you'll only need to do this once):

```
curl -sS https://getcomposer.org/installer | php
mv composer.phar /usr/local/bin/composer
```

3.2 Laravel

Laravel can be installed in lots of different ways. You can install it globally, meaning that you can use it from one location.

```
composer global require "laravel/installer=~1.1"
```

This creates the laravel setup file in the `.composer/vendor/bin/` directory. You can add the laravel command to your path and thus create a new project:

```
laravel new l5todo
```

The other way is by using Composer to directly download and setup composer in the current directory. For exampe, if you are using Wamp, Xampp or similar tool, you can set up laravel in the `htdocs` directory.

```
cd [path to htdocs]
composer create-project laravel/laravel l5todo
```

The last method is used in our example

4 Configuration

Laravel 5 uses a package called DotEnv that stores sensitive information in **.env** files, which are loaded as PHP environment variables at runtime. Sounds complicated but it just means your sensitive credentials go into these files while the rest of your config remains in the standard config files.

5 Setting up the Database

We need a database. Set one up for yourself in a DB of your choice then copy .env.example to .env and update accordingly:

```
DB_HOST=localhost
DB_DATABASE=[the name of the database]
DB_USERNAME=[the user that will access the database]
DB_PASSWORD=[the password for the db user]
```

Finally if you're not using MySQL open /config/database.php and change the default line:

```
'default' => 'mysql',
```

By default `.env` files are ignored by the git version control system, via the default `.gitignore` that is given with Laravel. If not, remember to add your environment files to your .gitignore by adding a .env line!

6 First Steps

As mentioned above, the To-Do application will comprise of one or more projects each with their own task list. We're going to need *Project* and *Task* models, controllers, views, migrations, seeds (optional but useful) and routes. You probably already understand what most/all of these are, however if you don't you should check out the video M-V-Huh? and Basic Model/Controller/View Workflow.

Let's make our way through them one at a time. We want to get our table schema set up in the database. It will look like the following:

Projects

Field	Type	Null	Key
`id`	int(10) unsigned	NO	PRI
`name`	varchar(255)	NO	
`slug`	varchar(255)	NO	
`created_at`	timestamp	NO	
`updated_at`	timestamp	NO	

Tasks

Field	Type	Null	Key
`id`	int(10) unsigned	NO	PRI
`project_id`	int(10) unsigned	NO	MUL
`name`	varchar(255)	NO	
`slug`	varchar(255)	NO	
`completed`	tinyint(1)	NO	
`description`	text	NO	
`created_at`	timestamp	NO	
`updated_at`	timestamp	NO	

First a migration must be set up:

```
php artisan make:migration create_projects_and_tasks_tables --create="projects"
```

We're creating both tables in the one migration so that they can be removed in reverse order to avoid an integrity constraint violation. Open `/database/migrations/<date>_create_projects_and_tasks_tables.php` and set it up as follows:

```
<?php

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

```
use Illuminate\Database\Migrations\Migration;

class CreateProjectsAndTasksTables extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
Schema::create('projects', function (Blueprint $table) {
    $table->increments('id');
    $table->string('name')->default('');
    $table->string('slug')->default('');
    $table->timestamps();
});

Schema::create('tasks', function (Blueprint $table){
    $table->increments('id');
    $table->integer('project_id')->unsigned()->default(0);
    $table->foreign('project_id')->references('id')->on('projects')->onDelete('cascade');
    $table->string('name')->default('');
    $table->string('slug')->default('');
    $table->boolean('completed')->default(false);
    $table->text('description');
    $table->timestamps();
});
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
Schema::dropIfExists('tasks'); // Replace with Schema::drop('tasks') if not working properly
Schema::dropIfExists('projects'); // Replace with Schema::drop('projects') if not working properly
    }
}
```

Perform the migration using the command `php artisan migrate`. If you check the database, your tables should now be all set up.

7 Seeds

We'll seed some projects/tasks to have something to work with when we finally get to the browser. Create `/database/seeds/ProjectsTableSeeder.php` and `TasksTableSeeder.php` using artisan:

```
php artisan make:seeder ProjectsTableSeeder
php artisan make:seeder TasksTableSeeder
```

and populate the files like so:

```
// File: /database/seeds/ProjectsTableSeeder.php

<?php

use Illuminate\Database\Seeder;
```

```
class ProjectsTableSeeder extends Seeder {

    public function run()
    {
// Uncomment the below to wipe the table clean before populating
DB::table('projects')->delete();

$projects = array(
    ['id' => 1, 'name' => 'Project 1', 'slug' => 'project-1', 'created_at' => new DateTime, 'updated_at
    ['id' => 2, 'name' => 'Project 2', 'slug' => 'project-2', 'created_at' => new DateTime, 'updated_at
    ['id' => 3, 'name' => 'Project 3', 'slug' => 'project-3', 'created_at' => new DateTime, 'updated_at
);

// Uncomment the below to run the seeder
DB::table('projects')->insert($projects);
    }

}


// File: /database/seeds/TasksTableSeeder.php

<?php

use Illuminate\Database\Seeder;

class TasksTableSeeder extends Seeder {

    public function run()
    {
// Uncomment the below to wipe the table clean before populating
DB::table('tasks')->delete();

$tasks = array(
    ['id' => 1, 'name' => 'Task 1', 'slug' => 'task-1', 'project_id' => 1, 'completed' => false, 'descr
    ['id' => 2, 'name' => 'Task 2', 'slug' => 'task-2', 'project_id' => 1, 'completed' => false, 'descr
    ['id' => 3, 'name' => 'Task 3', 'slug' => 'task-3', 'project_id' => 1, 'completed' => false, 'descr
    ['id' => 4, 'name' => 'Task 4', 'slug' => 'task-4', 'project_id' => 1, 'completed' => true, 'descri
    ['id' => 5, 'name' => 'Task 5', 'slug' => 'task-5', 'project_id' => 1, 'completed' => true, 'descri
    ['id' => 6, 'name' => 'Task 6', 'slug' => 'task-6', 'project_id' => 2, 'completed' => true, 'descri
    ['id' => 7, 'name' => 'Task 7', 'slug' => 'task-7', 'project_id' => 2, 'completed' => false, 'descr
);

//// Uncomment the below to run the seeder
DB::table('tasks')->insert($tasks);
    }

}
```

The seed classes should be added to `/database/seeds/DatabaseSeeder.php`.

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
```

```
     */
    public function run()
    {
//// Not needed since Laravel 5.2.
// Model::unguard();

// $this->call(UserTableSeeder::class);

$this->call('ProjectsTableSeeder');
$this->call('TasksTableSeeder');

////
// Model::reguard(); // Reguard the models
    }
}
```

We should regenerate the list of all classes using the command `composer dump-autoload`. Please refer to the composer documentation for more detailed explanation.

The data can be seeded using the command:

```
php artisan db:seed
# or
php artisan migrate:refresh --seed
```

The database should be populated now with the data.

NOTE: If you get an error like:

```
[Illuminate\Database\QueryException]‘
SQLSTATE[42000]: Syntax error or access violation: 1071 Specified key was too long; max key length is 7

[PDOException]
SQLSTATE[42000]: Syntax error or access violation: 1071 Specified key was too long; max key length is 7
```

then you must specify the default string length for each column (see laravel 5.4 release notes) in the `boot` method in `AppServiceProvider.php`, located in the `app/Providers` directory:

```
// Include the Schema facade
use Illuminate\Support\Facades\Schema;


public function boot()
{
    // Set default string length
    Schema::defaultStringLength(191);
}
```

8 Models

To work with our projects and tasks tables we need equivalent Models so create them now:

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

This command creates the models for the Project and Task tables in `/app` directory.

9 Artisan – Tinker

Now that we have information in the database it would be a great time to learn about one of artisan's handy features – **tinker**. As explained in the informative article Tinkering with Tinker Like an Artisan, tinker provides a command line for interacting with your Laravel installation. As an example, let's use it to retrieve the number of projects currently in the database:

```
$ php artisan tinker
>App\Project::count();
3
>App\Task::count();
7
```

As you can see tinker has the potential to be quite useful. For example, you can use:

```
$ php artisan tinker
>App\Project::get();
```

10 Controllers

We've gotten to the point now where we can start hitting the browser. To do that we need to set up some Controllers and Routes to point to them. First up the controllers:

```
php artisan make:controller ProjectsController --resource
php artisan make:controller TasksController --resource
```

10.1 Nested Resources

Begin by adding the Project and Task resources to `routes/web.php`:

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

Route::resource('projects', 'ProjectsController');
Route::resource('tasks', 'TasksController');
```

Let's now look at a neat little artisan feature – route:list. In your command line enter the following:
`php artisan route:list`
You'll notice that both projects and tasks are top level urls. In our to-do app, tasks belong to projects though, so it makes sense for URLs to be nested more like `/projects/1/tasks/3` instead of just `/tasks/3`. This can be accomplished using something called nested resources. As with most things in Laravel, the modification required is quick and simple. Open `/app/Http/routes.php` and make the following change:

```
// Route::resource('tasks', 'TasksController');
Route::resource('projects.tasks', 'TasksController');
```

That's it. Do another `php artisan route:list` and see what you have now:

11 Setting Slug-based URLs

We've almost got our routes perfect however in their current state we'll have URLs like `/projects/1/tasks/2`. It would be much better for our visitors if the model IDs were replaced with their respective slug fields instead. So we'd get for example `/projects/my-first-project/tasks/buy-milk`.

Open `routes/web.php` and drop the following in:

```
Route::bind('tasks', function($value, $route){
    return App\Task::whereSlug($value)->first();
});

Route::bind('projects', function($value, $route){
   return App\Project::whereSlug($value)->first();
});
```

The above will override the default behavior for the tasks and projects wildcards in php artisan routes.

So far we have a working database complete with seed data and a bunch of routes for displaying, editing and deleting our projects and tasks. Now we will cover controllers, models (with relationships), views (including the blade templating language and layouts) and route model binding.

12 Laravel Form Helpers

Laravel 4 had a HTML package but it was removed for 5 to cut down on cruft. Later it has been deprecated for the newer `laravelcollective/html` (docs, packagist).

Require the package using Composer:

```
composer require "laravelcollective/html":"^5.5"
```

This changes the `composer.json` file to include the dependency in the `require` section:

```
{
    //... Other content

    "require": {
// ...
"laravelcollective/html": "^5.5"
    },
    //... Other content
}
```

Next up add the service provider and aliases. Open `/config/app.php` and update as follows:

```
'providers' => [
  ...

  Collective\Html\HtmlServiceProvider::class,
],

'aliases' => [

  ...

  'Form'=> Collective\Html\FormFacade::class,
  'HTML'=> Collective\Html\HtmlFacade::class,
],
```

To confirm it's working use the following:

```
php artisan tinker
> echo Form::text('foo');
"<input name=\"foo\" type=\"text\">"
```

In addition to HTML and Form facades, this package provides some handy helper functions such as `link_to_route()` which we'll be using later.

13 Controllers and Blade View Layouts

If you browse to /projects you'll get an empty page. Why is that? Run `php artisan route:list` one more time and look at this line:

```
| Domain | Method   | URI      | Name           | Action                                          | Middl
|--------+----------+----------+----------------+-------------------------------------------------+------
|        | GET/HEAD | projects | projects.index | App\Http\Controllers\ProjectsController@index |
|--------+----------+----------+----------------+-------------------------------------------------+------
```

Looks like the `/projects` URL is loading ProjectsController's index method. So open up `/app/Http/controllers/Projec` and update the method to point to a view we'll create:

```
public function index()
{
  return view('projects.index');
}
```

We're using Blade Templates, so create a `/resources/views/projects/index.blade.php` file and enter some text in there. Hit `/projects` in your browser again. If everything is working correctly you should see the text you entered above. Do the same for the create controller method.

Showing the contents of a view is great, but if we have more than one page on our site we'll want a consistent template across all pages. In other words, we need the view's contents to sit inside a basic HTML template. This is done with controller layouts.

There are a few steps to implementing controller layouts. First we should create a layout view. Laravel used to ship with a pretty decent one called `app.blade.php`. Now it is a separate part that can be included using composer. The procedure is following:

1. Add the Scafold package to Laravel. This can be done in 2 ways:
   (a) Modify the composer.json file in the `require:` section:

   ```
   require : {
       "php": ">=5.6.4",
       "laravel/framework": "5.3.*",
       "laravelcollective/html": "^5.3",
       "marcuscampos/scafold": "^1.0"

   }
   ```

   (b) or type from terminal:

   ```
   composer require marcuscampos/scafold
   ```

2. Update composer:

   ```
   composer update
   ```

3. Add the service provider to your `config/app.php`:

   ```
   MarcusCampos\Scafold\ScafoldServiceProvider::class,
   ```

4. Publish the views and assets:

   ```
   php artisan vendor:publish
   ```

   and (if asked) choose to install all providers and tags.

In the `app.blade.php` file, notice near the bottom the layout contains a `@yield('content')` line. That's the function that will load our actual content. Here we will reference the layout in the view using `@extends('app')` and wrap in a `@section('content')` block like so:

```
@extends('app')

@section('content')
    This is my /resources/views/projects/index.blade.php file!
@endsection
```

In `/resources/views/projects` folder create a `show.blade.php`, `index.blade.php` and `create.blade.php` view with the above markup replacing the filename as necessary. With these in place, refresh `/projects` in your browser. You should now see the `app.blade.php` skeleton around your view content

14 Route Model Binding

By default Laravel will provide an ID value to various resourceful controller methods such as show(), edit(), update() and destroy(). This is fine but it adds alot of extra boilerplate we need to write – grabbing the model instance, checking if it exists etc. Thankfully Laravel provides something called route model binding that helps with this issue. Instead of providing an `$id` variable, the method will be given the `$project` or `$task` object instance instead.

There are couple of ways of providing Route Model Binding. See `https://laravel.com/docs/5.3/routing#route-model-binding` for more information.

We will use the model to provide the route model bindings. Add the following methods to the defined models (`App\Task` and `App\Project`)

```
// File: App\Task.php

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Task extends Model
{
    public function getRouteKeyName()
    {
return 'slug';
    }
}


// File: App\Project.php

<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Project extends Model
{
    public function getRouteKeyName()
    {
return 'slug';
    }
}
```

and in your `TasksController` and `ProjectsController` replace every method definition's `$id` reference with `Task $task` and `Project $project` like so:

```
// public function edit($id)
public function edit(Project $project)
```

Don't forget to add `use App\Task` and `App\Project` at the top of your respective controllers now that we're referencing those models!

At this point you can also pass the object to its respective view in each controllers show, edit and update methods like so as we'll be using them later:

```
public function edit(Project $project)
{
    return view('projects.show', compact('project'));}
}
```

The TasksController will also need some minor modifications. Because we're using nested resources, `php artisan route:list` will tell us that task routes all include a `{projects}` mask in addition to the `{tasks}` mask that some of them receive. As a result the controller methods will be passed a `Project` instance as their first argument. So update them accordingly remembering to update method docs and pass the new `$project` variable.

By this point your controllers should look like so:

```
<?php

// /app/Http/Controllers/ProjectsController.php

namespace App\Http\Controllers;
```

```
use App\Project;
use Illuminate\Http\Request;

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

class ProjectsController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
$projects = Project::all();
return view('projects.index', compact('projects'));
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
return view('projects.create');
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
//
    }

    /**
     * Display the specified resource.
     *
     * @param  \App\Project $project
     * @return \Illuminate\Http\Response
     */
    public function show(Project $project)
    {
return view('projects.show', compact('project'));
    }

    /**
     * Show the form for editing the specified resource.
     *
     * @param  \App\Project $project
     * @return \Illuminate\Http\Response
     */
    public function edit(Project $project)
    {
return view('projects.edit', compact('project'));
    }
```

```
    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \App\Project $project
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, Project $project)
    {
//
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function destroy(Project $project)
    {
//
    }
}
```

```
<?php

// /app/Http/Controllers/TasksController.php

namespace App\Http\Controllers;

use App\Project;
use App\Task;
use Illuminate\Http\Request;

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

class TasksController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @param  \App\Project $project
     * @return \Illuminate\Http\Response
     */
    public function index(Project $project)
    {
return view('tasks.index', compact('project'));
    }

    /**
     * Show the form for creating a new resource.
     *
     * @param  \App\Project $project
     * @return \Illuminate\Http\Response
     */
    public function create(Project $project)
    {
```

```
return view('tasks.create', compact('project'));
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \App\Project $project
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request, Project $project)
    {
//
    }

    /**
     * Display the specified resource.
     *
     * @param  \App\Project $project
     * @param  \App\Task $task
     * @return \Illuminate\Http\Response
     */
    public function show(Project $project, Task $task)
    {
return view('tasks.show', compact('project', 'task'));
    }

    /**
     * Show the form for editing the specified resource.
     * @param  \App\Project $project
     * @param  \App\Task $task
     * @return \Illuminate\Http\Response
     */
    public function edit(Project $project, Task $task)
    {
return view('tasks.edit', compact('project', 'task'));
    }

    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \App\Project $project
     * @param  \App\Task $task
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, Project $project, Task $task)
    {
//
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  \App\Project $project
     * @param  \App\Task $task
     * @return \Illuminate\Http\Response
     */
    public function destroy(Project $project, Task $task)
    {
//
```

```
    }
}
```

If you refresh the url `/projects/project-1` everything should still be working.

15 Displaying Our Models

15.1 Project listing page

It's time to start listing our projects and tasks. Open `/projects` in your browser. Based on php artisan route:list this is our project listing page. Open `/resources/views/projects/index.blade.php` and set it to the following:

```
@extends('app')

@section('content')
    <h2>Projects</h2>

    @if ( !$projects->count() )
You have no projects
    @else
<ul>
    @foreach( $projects as $project )
<li><a href="{{ route('projects.show', $project->slug) }}">{{ $project->name }}</a></li>
    @endforeach
</ul>
    @endif

@endsection
```

There are a few things going on above:

- We are using the blade templating language's if and foreach control-flow functions as well as its print function (the double curly braces).
- We are checking if there are any projects to show. If not, display a message saying so. If there are, list them all
- We are calling the route() helper with a named route (You can see a list of your named routes with php artisan route:list) to link to each projects details page

You'll also need to pass the $projects variable to this view or you'll get an undefined variable error. Open `/app/Http/controllers/ProjectsController.php` and update the `index()` method to:

```
public function index()
{
  $projects = Project::all();
  return view('projects.index', compact('projects'));
}
```

Refresh and you'll now see a listing of your projects.

15.2 Model Relations – The Project Details page

On the project details page we need to display a list of the given projects tasks. To do that we need to define a one-to-many relationship in our Project model allowing it to grab its tasks.

Open `/app/Project.php` and add a `tasks()` method like so:

```
public function tasks()
{
    return $this->hasMany('App\Task');
}
```

Inversely we can also add a many-to-one relationship to our Task model:

```
public function project()
{
   return $this->belongsTo('App\Project');
}
```

To make sure it worked:

```
$ php artisan tinker
>App\Project::whereSlug('project-1')->first()->tasks->count();
5
>App\Project::whereSlug('project-2')->first()->tasks->count();
2
>App\Task::first()->project->name;
Project 1
```

Perfect! The view can now be updated (`/resources/views/projects/show.blade.php`):

```
@extends('app')

@section('content')

    <h2>{{ $project->name }}</h2>
    @if ( !$project->tasks->count() )
        Your project has no tasks.
    @else
        <ul>
            @foreach( $project->tasks as $task )
                <li><a href="{{ route('projects.tasks.show', [$project->slug, $task->slug]) }}">{{ $tas
            @endforeach
        </ul>
    @endif

@endsection
```

Click a project on the project listing page in your browser and your project will now display complete with its task listing.

Finally we have the task show page (`/resources/views/tasks/show.blade.php`). This one is very straightforward:

```
@extends('app')

@section('content')
    <h2>
        {!! link_to_route('projects.show', $project->name, [$project->slug]) !!} -
        {{ $task->name }}
    </h2>

    {{ $task->description }}
@endsection
```

16 Building the Forms

16.1 Adding Navigation Links

Before we do anything it would make life a little easier to add create/edit/delete/back links to our projects and tasks pages.

```
<!-- /resources/views/projects/index.blade.php -->
@extends('app')

@section('content')
```

```
    <h2>Projects</h2>

    @if ( !$projects->count() )
        You have no projects
    @else
        <ul>
            @foreach( $projects as $project )
                <li>
                    {!! Form::open(array('class' => 'form-inline', 'method' => 'DELETE', 'route' => arr
                        <a href="{{ route('projects.show', $project->slug) }}">{{ $project->name }}</a>
                        (
                            {!! link_to_route('projects.edit', 'Edit', array($project->slug), array('cl
                            {!! Form::submit('Delete', array('class' => 'btn btn-danger')) !!}
                        )
                    {!! Form::close() !!}
                </li>
            @endforeach
        </ul>
    @endif

    <p>
        {!! link_to_route('projects.create', 'Create Project') !!}
    </p>
@endsection




<!-- /resources/views/projects/show.blade.php -->
@extends('app')

@section('content')
    <h2>{{ $project->name }}</h2>

    @if ( !$project->tasks->count() )
        Your project has no tasks.
    @else
        <ul>
            @foreach( $project->tasks as $task )
                <li>
                    {!! Form::open(array('class' => 'form-inline', 'method' => 'DELETE', 'route' => arr
                        <a href="{{ route('projects.tasks.show', [$project->slug, $task->slug]) }}">{{
                        (
                            {!! link_to_route('projects.tasks.edit', 'Edit', array($project->slug, $tas

                            {!! Form::submit('Delete', array('class' => 'btn btn-danger')) !!}
                        )
                    {!! Form::close() !!}
                </li>
            @endforeach
        </ul>
    @endif

    <p>
        {!! link_to_route('projects.index', 'Back to Projects') !!} |
        {!! link_to_route('projects.tasks.create', 'Create Task', $project->slug) !!}
    </p>
@endsection
```

For the most part this should be pretty self explanatory. The only tricky concept is the delete link. Resource controllers require a HTTP DELETE method to be sent. This can't be done with a standard link so a form

submit to the given route is required. See Actions Handled by Resource Controller in the documentation for more information.

16.2 Creating the Add and Edit pages

With the listing pages all set up, we need to be able to add and edit projects and tasks. The create and edit forms will be pretty much identical so instead of duplicating them, we will inherit from a single form partial for each model. I'll begin with the project create/edit views:

```
<!-- /resources/views/projects/create.blade.php -->
@extends('app')

@section('content')
  <h2>Create Project</h2>

  {!! Form::model(new App\Project, ['route' => ['projects.store']]) !!}
    @include('projects/partials/_form', ['submit_text' => 'Create Project'])
  {!! Form::close() !!}
@endsection

<!-- /resources/views/projects/edit.blade.php -->
@extends('app')

@section('content')
  <h2>Edit Project</h2>

  {!! Form::model($project, ['method' => 'PATCH', 'route' => ['projects.update', $project->slug]]) !!}
    @include('projects/partials/_form', ['submit_text' => 'Edit Project'])
  {!! Form::close() !!}
@endsection
```

Do the same for tasks but with updated routes:

```
<!-- /resources/views/tasks/create.blade.php -->
@extends('app')

@section('content')
  <h2>Create Task for Project "{{ $project->name }}"</h2>

  {!! Form::model(new App\Task, ['route' => ['projects.tasks.store', $project->slug], 'class'=>'']) !!}
      @include('tasks/partials/_form', ['submit_text' => 'Create Task'])
  {!! Form::close() !!}
@endsection

<!-- /resources/views/tasks/edit.blade.php -->
@extends('app')

@section('content')
  <h2>Edit Task "{{ $task->name }}"</h2>

  {!! Form::model($task, ['method' => 'PATCH', 'route' => ['projects.tasks.update', $project->slug, $ta
      @include('tasks/partials/_form', ['submit_text' => 'Edit Task'])
  {!! Form::close() !!}
@endsection
```

Now the are a few new concepts here:

16.3 Including a Partial

Firstly you'll notice the use of blades `@include` method. This includes the form view that we will define later (for now, just create the files `/resources/views/projects/partials/_form.blade.php` and `/resources/views/tasks/partial`

Because the same form will be used on both create and edit pages we need its submit button to have a 'Create Form' and 'Edit Form' message appropriately so a `submit_text` variable is passed to the view.

16.4 Form Model Binding

The forms require different HTML `<form>` tags so rather those were split out from the _form partial and placed directly in the views. The Add form is a simple POST request to the projects.store named route and the Edit form is a `PATCH` to projects.update. This may seem confusing but it's just the way **RESTful** controllers work.

Also notice the use of `Form::model()`. This is called called form model binding and though this doesn't do much now, it will be used to automatically populate the edit form later when we add the fields with using `Form::input()`.

16.5 CSRF Protection

Form helpers provide alot of functionality for free. If you go to /projects/create and view page source you'll see something like the following:

```
<form method="POST" action="http://l4todo.localhost.com/projects" accept-charset="UTF-8">
  <input name="_token" type="hidden" value="Y8uOo7SeD5tQZExezDf5a7UwiYR4P6qIHEUKJNxI">
</form>
```

See the `_token` field? This is a CSRF token automatically generated by the `{{ Form::model() }}` call which prevents cross-site request forgery. Suffice to say it's a good thing and we didn't even have to do anything special to get it!

16.6 Create the Edit Forms

We need form markup for our projects and tasks. Thanks to form model binding, we can just use Laravel's Form helpers to output all the fields we need.

```
<!-- /resources/views/projects/partials/_form.blade.php -->
<div class="form-group">
  {!! Form::label('name', 'Name:') !!}
  {!! Form::text('name') !!}
</div>
<div class="form-group">
  {!! Form::label('slug', 'Slug:') !!}
  {!! Form::text('slug') !!}
</div>
<div class="form-group">
  {!! Form::submit($submit_text, ['class'=>'btn primary']) !!}
</div>


<!-- /resources/views/tasks/partials/_form.blade.php -->
<div class="form-group">
  {!! Form::label('name', 'Name:') !!}
  {!! Form::text('name') !!}
</div>

<div class="form-group">
  {!! Form::label('slug', 'Slug:') !!}
  {!! Form::text('slug') !!}
</div>

<div class="form-group">
  {!! Form::label('completed', 'Completed:') !!}
  {!! Form::checkbox('completed') !!}
</div>

<div class="form-group">
  {!! Form::label('description', 'Description:') !!}
  {!! Form::textarea('description') !!}
</div>
```

```
<div class="form-group">
  {!! Form::submit($submit_text) !!}
</div>
```

That's about it. In your browser you should now be able to browse to your add and edit pages.

16.7 Making the Forms Work

We have project and task add and edit forms displaying and a pseudo-form for deleting them. Now to make everything work as advertised.

Firstly add to the top of your controllers:

```
use Illuminate\Support\Facades\Input;
use Illuminate\Support\Facades\Redirect;
```

Now for the store, update and destroy methods.

```
// ProjectsController
public function store()
{
  $input = Input::all();
  Project::create( $input );

  return Redirect::route('projects.index')->with('message', 'Project created');
}

public function update(Project $project)
{
  $input = array_except(Input::all(), '_method');
  $project->update($input);

  return Redirect::route('projects.show', $project->slug)->with('message', 'Project updated.');
}

public function destroy(Project $project)
{
  $project->delete();

  return Redirect::route('projects.index')->with('message', 'Project deleted.');
}

// TasksController
public function store(Project $project)
{
  $input = Input::all();
  $input['project_id'] = $project->id;
  Task::create( $input );

  return Redirect::route('projects.show', $project->slug)->with('message', 'Task created.');
}

public function update(Project $project, Task $task)
{
  $input = array_except(Input::all(), '_method');
  $task->update($input);

  return Redirect::route('projects.tasks.show', [$project->slug, $task->slug])->with('message', 'Task u
}

public function destroy(Project $project, Task $task)
{
  $task->delete();
```

```
  return Redirect::route('projects.show', $project->slug)->with('message', 'Task deleted.');
}
```

Again the above is pretty much self explanatory and boilerplate-free thanks to route model binding and Laravel's beautiful expressive syntax.

If you submitted one of the forms now, you'd likely see an error `MassAssignmentException: _token`. A mass assignment is where you pass an array of data to `Model::create()` or `Model::update()` the way we're doing in our controllers, instead of setting one field at a time. To fix this simply add an empty guarded property to each model.

```
class Project extends Model {

protected $guarded = [];



class Task extends Model {

protected $guarded = [];
```

16.8 Flash Messages

One thing to note from the above is my use of the `with()` function. `with()` passes a flash (one time use) variable to the session which can then be read on the next page load. We now need to check for that message and display it to the user. Open `/resources/views/app.blade.php` and add the following:

```
<div class="content">
  @if (Session::has('message'))
    <div class="flash alert-info">
      <p>{{ Session::get('message') }}</p>
    </div>
  @endif
  @yield('content')
</div>
```

Try creating a project. The message will display as it should however refresh the page and it will be gone.

17 Validation

17.1 Server Side Form Validation

As it stands our create and edit forms work but they're not validated.

There are multiple ways of handling form validation – some objectively better than others, however for such a small project as this I like to use the controllers validate() method with a `Illuminate\Http\Request` object like so:

```
// /app/Http/Controllers/ProjectsController.php

use Illuminate\Http\Request;

class ProjectsController extends Controller {

  protected $rules = [
    'name' => ['required', 'min:3'],
    'slug' => ['required'],
  ];

  /**
   * Store a newly created resource in storage.
   *
 * @param \Illuminate\Http\Request $request
```

```
 * @return Response
 */
public function store(Request $request)
{
$this->validate($request, $this->rules);

$input = Input::all();
Project::create( $input );

return Redirect::route('projects.index')->with('message', 'Project created');
}

/**
 * Update the specified resource in storage.
 *
 * @param  \App\Project $project
 * @param \Illuminate\Http\Request $request
 * @return Response
 */
public function update(Project $project, Request $request)
{
$this->validate($request, $this->rules);

$input = array_except(Input::all(), '_method');
$project->update($input);

return Redirect::route('projects.show', $project->slug)->with('message', 'Project updated.');
}


// /app/Http/Controllers/TasksController.php
use Illuminate\Http\Request;

class TasksController extends Controller {

protected $rules = [
'name' => ['required', 'min:3'],
'slug' => ['required'],
'description' => ['required'],
];

/**
 * Store a newly created resource in storage.
 *
 * @param  \App\Project $project
 * @param \Illuminate\Http\Request $request
 * @return Response
 */
public function store(Project $project, Request $request)
{
$this->validate($request, $this->rules);

$input = Input::all();
$input['project_id'] = $project->id;
Task::create( $input );

return Redirect::route('projects.show', $project->slug)->with('Task created.');
}

/**
 * Update the specified resource in storage.
```

```
 *
 * @param  \App\Project $project
 * @param  \App\Task    $task
 * @param \Illuminate\Http\Request $request
 * @return Response
 */
public function update(Project $project, Task $task, Request $request)
{
$this->validate($request, $this->rules);

$input = array_except(Input::all(), '_method');
$task->update($input);

return Redirect::route('projects.tasks.show', [$project->slug, $task->slug])->with('message', 'Task upd
}
```

We need a place to display any generated errors. Open /resources/views/app.blade.php and drop the following above `@yield('content')`:

```
<div class="content">
@if (Session::has('message'))
<div class="flash alert-info">
<p>{{ Session::get('message') }}</p>
</div>
@endif
@if ($errors->any())
<div class='flash alert-danger'>
@foreach ( $errors->all() as $error )
<p>{{ $error }}</p>
@endforeach
</div>
@endif

@yield('content')
</div>
```

See Available Validation Rules for a complete list of rules available. Validation should now be working. If validation on a form fails, `$this->validate()` will redirect back to the current page along with an `ErrorBag` of errors which will then be displayed on the page.