Chapter 3: Building A Support Ticket System

In this chapter, we will build a support ticket system to learn about Laravel main features, such as Eloquent ORM, Eloquent Relationships, Migrations, Requests, Laravel Collective, sending emails, etc.

While the project design is simple, it provides an excellent way to learn Laravel.

What do we need to get started?

I assume that you have followed the instructions provided in the previous chapter and you've created a basic website. You will need that basic application to start building the support ticket system.

What will we build?

We'll start by laying down the basic principle behind the application's creation, and a summary of how it works.

Our ticket system is simple:

- When users visit the contact page, they will be able to submit a ticket to contact us.
- Once they've created a ticket, the system will send us an email to let us know that there is a new ticket.
- The ticket system automatically generates a unique link to let us access the ticket.
- We can view all the tickets.
- We can be able to reply, edit, change tickets' status or delete them.

Let's start building things!

Laravel Database Configuration

Our application requires a database to work. Laravel supports many database platforms, including:

1. MySQL
2. SQLite
3. PostgreSQL
4. SQL Server
5. MariaDB

> **Note:** A database is a collection of data. We use database to store, manage and update our data easier.

The great thing is, we can choose any of them to develop our applications. In this book, we will use **MySQL**.

You can configure databases using **database.php** file, which is placed in the **config directory**.

config/database.php

```
<?php

return [

    /*
    |--------------------------------------------------------------------------
    | Default Database Connection Name
    |--------------------------------------------------------------------------
    |
    | Here you may specify which of the database connections below you wish
    | to use as your default connection for all database work. Of course
    | you may use many connections at once using the Database library.
    |
    */

    'default' => env('DB_CONNECTION', 'mysql'),

    /*
    |--------------------------------------------------------------------------
    | Database Connections
    |--------------------------------------------------------------------------
    |
    | Here are each of the database connections setup for your application.
```

```php
    | Of course, examples of configuring each database platform that is
    | supported by Laravel is shown below to make development simple.
    |
    |
    | All database work in Laravel is done through the PHP PDO facilities
    | so make sure you have the driver for your particular database of
    | choice installed on your machine before you begin development.
    |
    */

    'connections' => [

        'sqlite' => [
            'driver' => 'sqlite',
            'database' => env('DB_DATABASE', database_path('database.sqlite')),
            'prefix' => '',
        ],

        'mysql' => [
            'driver' => 'mysql',
            'host' => env('DB_HOST', '127.0.0.1'),
            'port' => env('DB_PORT', '3306'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'unix_socket' => env('DB_SOCKET', ''),
            'charset' => 'utf8mb4',
            'collation' => 'utf8mb4_unicode_ci',
            'prefix' => '',
            'strict' => true,
            'engine' => null,
        ],

        'pgsql' => [
            'driver' => 'pgsql',
            'host' => env('DB_HOST', '127.0.0.1'),
            'port' => env('DB_PORT', '5432'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'charset' => 'utf8',
            'prefix' => '',
            'schema' => 'public',
            'sslmode' => 'prefer',
        ],

        'sqlsrv' => [
            'driver' => 'sqlsrv',
            'host' => env('DB_HOST', 'localhost'),
            'port' => env('DB_PORT', '1433'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'charset' => 'utf8',
            'prefix' => '',
        ],

    ],

    /*
    |--------------------------------------------------------------------------
    | Migration Repository Table
    |--------------------------------------------------------------------------
    |
    | This table keeps track of all the migrations that have already run for
    | your application. Using this information, we can determine which of
    | the migrations on disk haven't actually been run in the database.
    |
    */

    'migrations' => 'migrations',

    /*
    |--------------------------------------------------------------------------
    | Redis Databases
    |--------------------------------------------------------------------------
    |
    | Redis is an open source, fast, and advanced key-value store that also
    | provides a richer set of commands than a typical key-value systems
    | such as APC or Memcached. Laravel makes it easy to dig right in.
    |
    */

    'redis' => [

        'client' => 'predis',

        'default' => [
            'host' => env('REDIS_HOST', '127.0.0.1'),
            'password' => env('REDIS_PASSWORD', null),
            'port' => env('REDIS_PORT', 6379),
            'database' => 0,
        ],

    ],

];
```

Try to read the comments to understand how to use this file. The most two important settings are:

1. **default**: You can set the type of database you would like to use here. By default, it is mysql. If you want to use a different database, you can set it to: **pgsql**, **sqlite**.
2. **connections**: Fill your database authentication credentials here. The **env() function** is used to retrieve configuration variables **(DB_HOST, DB_DATABASE, DB_USERNAME, DB_PASSWORD)** from **.env file**. If it can't find any variables, it will use the value of the function's second parameter **(localhost, forge)**.

If you use Homestead, Laravel has created a **homestead database** for you already. If you don't use Homestead, you will need to create a database manually.

Laravel uses PHP Data Objects (PDO). When we execute a Laravel SQL query, rows are returned in a form of a stdClass object. For instance, we may access our data using:

```
$user->name
```

MySQL Strict Mode

In new versions of Laravel, **MySQL Strict Mode** is `enabled` by default.

> **Information**: If you want to know what **Strict Mode** is, check out this article: Upgrading to MySQL 5.7? Beware of the new STRICT mode

In some cases, you might want to turn the **Strict Mode** `off`. Here's how we can do it:

Find:

```
        'mysql' => [
            'driver' => 'mysql',
            'host' => env('DB_HOST', '127.0.0.1'),
            'port' => env('DB_PORT', '3306'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'unix_socket' => env('DB_SOCKET', ''),
            'charset' => 'utf8mb4',
            'collation' => 'utf8mb4_unicode_ci',
            'prefix' => '',
            'strict' => true,
            'engine' => null,
        ],
```

Change **strict** from `true` to `false`:

```
'strict' => false,
```

> **Note:** If you encounter some database errors, be sure to try to turn the **Strict Mode** off first. It's still safe to turn the **Strict Mode** off.

Create a database

> **Note:** You need a basic understanding of SQL to develop Laravel applications. At least, you should know how to read, update, modify and delete a database and its tables. If you don't know anything about database, a good place to learn is: http://www.w3schools.com/sql

To develop multiple applications, we will need multiple databases. In this section, we will learn how to create a database.

Default database information

Laravel has created a **homestead** database for us. To connect to **MySQL** or **Postgres** database, you can use these settings:

host: 127.0.0.1

database: homestead

username: homestead

password: secret

port: 33060 (MySQL) or 54320 (Postgres)

Create a database using the CLI

You can easily create a new database via the command line. First, **vagrant ssh** to your Homestead. Use this command to connect to **MySQL**:

```
    mysql -u homestead -p
```

When it asks for the password, use **secret**.

To create a new database, use this command:

```
CREATE DATABASE your_database_name;
```

Feel free to change **your_database_name** to your liking.

To see all databases, run this command:

```
show databases;
```

Finally, you may leave MySQL using this command:

```
exit
```

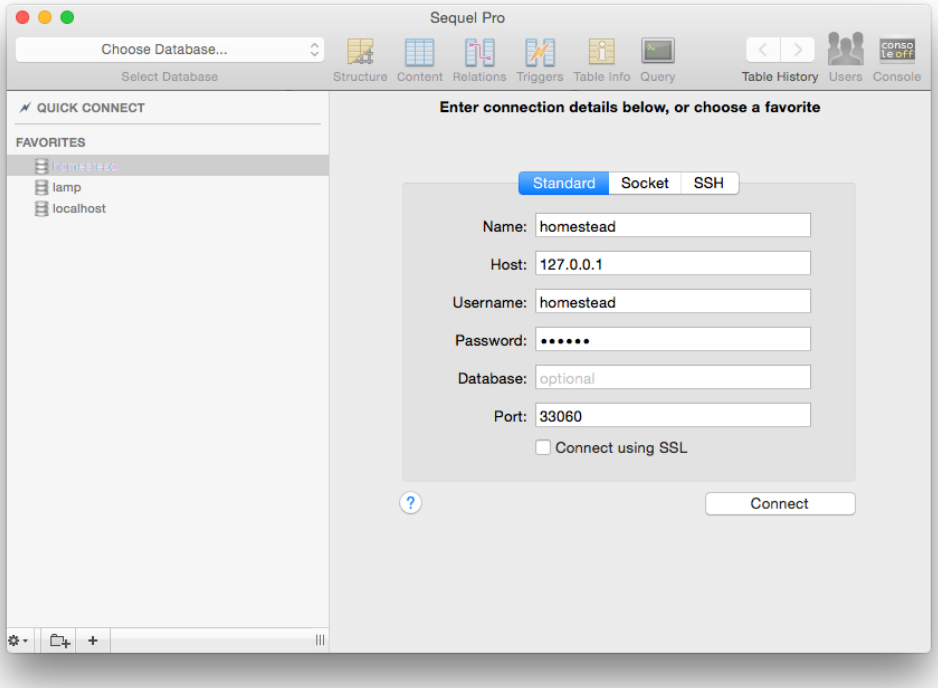
Even though we can easily create a new database via the **CLI**, we should use a **Graphical User Interface (GUI)** to manage databases easily.

Create a database on Mac

On Mac, the most popular GUI to manage databases is **Sequel Pro**. It's free, fast and very easy to use. You can download it here:

www.sequelpro.com

After that, you can connect to MySQL or Postgres database using database credentials in the **Default database information** section.



Once connected, you can easily create a new database by clicking **Choose Database...** and then **Add Database**.

Alternatively, you may use Navicat.

Create a database on Windows

On Windows, three popular GUIs for managing databases are:

SQLYog (Free)

https://www.webyog.com/product/sqlyog

SQLYog has a free open-source version. You can download it here:

https://github.com/webyog/sqlyog-community

Click the **Download SQLyog Community Version** to download.

HeidiSQL (Free)

http://www.heidisql.com

Navicat

http://www.navicat.com

Feel free to choose to use any GUI that you like. After that, you can connect to MySQL or Postgres database using database credentials in the **Default database information** section.

Using Migrations

One of the best features of Laravel is **Migrations**.

Whether you're working with a team or alone, you may need to find a way to keep track your database schema. **Laravel Migrations** is the right way to go.

Laravel uses migration files to know what we change in our database. The great thing is, you can easily revert or apply changes to your applications by just running a command. For example, we can use this command to reset the database:

```
php artisan migrate:reset
```

It's easy, right?

This feature is very useful. You should always use Migrations to build your application's database schema.

You can find Migrations documentation at:

http://laravel.com/docs/master/migrations

Meet Laravel Artisan

Artisan is **Laravel's CLI** (Command Line Interface). We often use **Artisan commands** to develop our Laravel applications. For instance, we use Artisan commands to generate migration files, seed our database, see the application namespace, etc.

You've used **Artisan** before! In the last chapter, we've used **Artisan** to generate controllers:

```
php artisan make:controller PagesController
```

Artisan official docs:

http://laravel.com/docs/master/artisan

To see a list of available Artisan commands, go to your application's root, run:

```
php artisan list
```

You should see all commands:

```

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
      --ansi            Force ANSI output
      --no-ansi         Disable ANSI output
  -n, --no-interaction  Do not ask any interactive question
      --env[=ENV]       The environment the command should run under
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug

Available commands:
  clear-compiled       Remove the compiled class file
  down                 Put the application into maintenance mode
  env                  Display the current framework environment
  help                 Displays help for a command
  inspire              Display an inspiring quote
  list                 Lists commands
  migrate              Run the database migrations
  optimize             Optimize the framework for better performance
  serve                Serve the application on the PHP development server
  tinker               Interact with your application
  up                   Bring the application out of maintenance mode
 app
  app:name             Set the application namespace
 auth
  auth:clear-resets    Flush expired password reset tokens
 cache
  cache:clear          Flush the application cache
  cache:forget         Remove an item from the cache
  cache:table          Create a migration for the cache database table
 config
  config:cache         Create a cache file for faster configuration loading
  config:clear         Remove the configuration cache file
 db
  db:seed              Seed the database with records
 event
  event:generate       Generate the missing events and listeners based on registration
```

```
 key
  key:generate         Set the application key
 make
  make:auth            Scaffold basic login and registration views and routes
  make:command         Create a new Artisan command
  make:controller      Create a new controller class
  make:event           Create a new event class
  make:job             Create a new job class
  make:listener        Create a new event listener class
  make:mail            Create a new email class
  make:middleware      Create a new middleware class
  make:migration       Create a new migration file
  make:model           Create a new Eloquent model class
  make:notification    Create a new notification class
  make:policy          Create a new policy class
  make:provider        Create a new service provider class
  make:request         Create a new form request class
  make:seeder          Create a new seeder class
  make:test            Create a new test class
 migrate
  migrate:install      Create the migration repository
  migrate:refresh      Reset and re-run all migrations
  migrate:reset        Rollback all database migrations
  migrate:rollback     Rollback the last database migration
  migrate:status       Show the status of each migration
 notifications
  notifications:table  Create a migration for the notifications table
 queue
  queue:failed         List all of the failed queue jobs
  queue:failed-table   Create a migration for the failed queue jobs database table
  queue:flush          Flush all of the failed queue jobs
  queue:forget         Delete a failed queue job
  queue:listen         Listen to a given queue
  queue:restart        Restart queue worker daemons after their current job
  queue:retry          Retry a failed queue job
  queue:table          Create a migration for the queue jobs database table
  queue:work           Start processing jobs on the queue as a daemon
 route
  route:cache          Create a route cache file for faster route registration
  route:clear          Remove the route cache file
  route:list           List all registered routes
 schedule
  schedule:run         Run the scheduled commands
 session
  session:table        Create a migration for the session database table
 storage
  storage:link         Create a symbolic link from "public/storage" to "storage/app/public"
 vendor
  vendor:publish       Publish any publishable assets from vendor packages
 view
  view:clear           Clear all compiled view files
```

Create a new migration file

Now, let's try to generate a new migration file!

We're going to create a **tickets table**. Go to your **application root** (~/Code/Laravel), execute this command:

```
php artisan make:migration create_tickets_table
```

You'll see:

```
vagrant@homestead:~/Code/Laravel$ php artisan make:migration create_tickets_table
Created Migration: 2017_09_09_095351_create_tickets_table
```

Laravel will create a new migration template and place it in your **database/migrations** directory.

The name of the new migration template is: **create_tickets_table**. You can name it whatever you want. Check the migrations directory, you'll find a file look like this:

2018_10_01_180019_create_tickets_table

You may notice that Laravel put a **timestamp** before the name of the file, it helps to determine the order of the migrations.

Open the file:

```
<?php

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

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

```
        Schema::create('tickets', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
        });
    }

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

Basically, a migration is just a standard class. There are two important methods:

1. **up** method: you use this method to add new tables, column to the database.
2. **down** method: well, you might have guessed already, this method is used to reverse what you've created.

Errors when creating or deleting a migration

Before reading the next section, let's take a look at some common errors when creating or deleting a migration.

If you see this error:

```
    [ErrorException]
    include(/home/vagrant/Code/Laravel/database/migrations/2016_09_09_193947_create_tickets_table.php):
    failed to open stream: No such file or directory
```

You will need to run this command to regenerate the list of all classes that need to be included in your app:

```
composer dump-autoload -o
```

> **Tip:** In the future, if you see this error again, you can run the command above to fix it.

If you see this error:

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

You may be running a version of MySQL older than the **5.7.7** release. To fix the bug, you need to edit the **AppServiceProvider.php** file, and set a default string length inside the `boot` method:

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

public function boot()
{
    Schema::defaultStringLength(191);
}
```

> **Note:** You can read more about this issue at https://laravel.com/docs/master/migrations#creating-indexes

Understand Schema to write migrations

It's time to create our tickets table by filling the up method!

Schema is a class that we can use to define and manipulate tables. We use **Schema::create** method to create the **tickets table**:

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

Schema::create method has two parameters. The first one is the **name of the table**.

The second one is a **Closure**. The Closure has one parameter: **$table**. You can name the parameter whatever you like.

We use the **$table** parameter to create **database columns**, such as id, name, date, etc.

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

increments('id') command is used to create **id column** and defines it to be an auto-increment primary key field in the **tickets** table.

timestamps is a special method of Laravel. It creates **updated_at** and **created_at** column. Laravel uses these columns to know when a row is **created** or **changed**.

To see a full list of Schema methods and column type, check out the **official docs**:

https://laravel.com/docs/master/migrations

You don't need to remember them all. We will learn some of them by creating some migrations in this book.

Our tickets will have these columns:

- id
- title
- content
- slug: URL friendly version of the ticket title
- status: current status of the ticket (answered or pending)
- user_id: who created the ticket

Here's how we can write our first migrations:

create_tickets_table.php

```php
<?php

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

class CreateTicketsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('tickets', function (Blueprint $table) {
            $table->increments('id');
            $table->string('title', 255);
            $table->text('content');
            $table->string('slug')->nullable();
            $table->tinyInteger('status')->default(1);
            $table->integer('user_id')->nullable();
            $table->timestamps();
        });
    }

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

Finally, run this command to create the tickets table and its columns:

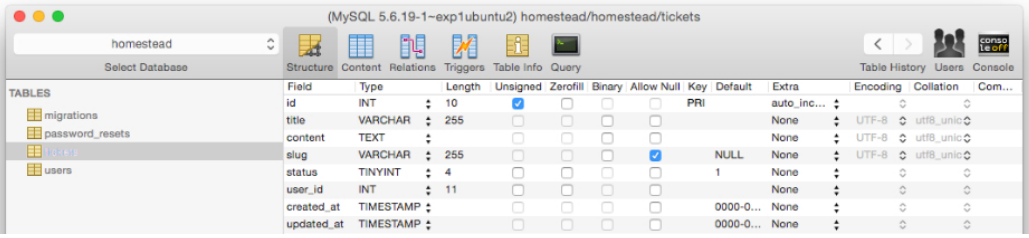
```
php artisan migrate
```

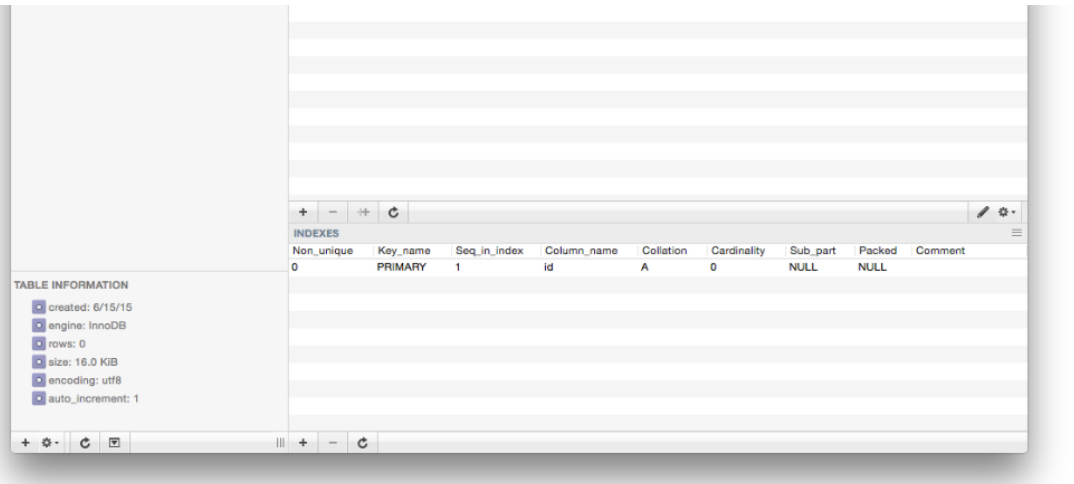
```
vagrant@homestead:~/Code/Laravel$ php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table
Migrating: 2018_10_01_180148_create_tickets_table
Migrated:  2018_10_01_180148_create_tickets_table
```

The first time you run this command, Laravel will create a **migration table** to keep track of what migrations you've created.

By default, Laravel also creates **create_users_table** migration and **create_password_resets_table** migration for us. These migrations will create **users** and **password_resets** tables. If you want to implement the default authentication, leave the files there. Otherwise, you can just delete them, or run **php artisan fresh** command to completely remove the default authentication feature.

Well, I guess it worked! It looks like the tables have been created. Let's check the **homestead** database:





> **Note:** I use the **homestead** database. If you like to use another one, feel free to change it using the **.env** file.

Well done! You've just created a new **tickets table** to store our data.

Create a new Eloquent model

Laravel has a very nice feature: **Eloquent ORM**.

Eloquent provides a simple way to use ActiveRecord pattern when working with databases. By using this technique, we can wrap our database into objects.

What does it mean?

In **Object Oriented Programming (OOP)**, we usually create multiple objects. Objects can be anything that has properties and actions. For example, a mobile phone can be an object. Each model of a phone has its own blueprint. You can buy a new case for your phone, or you can change its home screen. But no matter you customize it, it's still based on the blueprint that was created by the manufacturer.

We call that blueprint: **model**. Basically, model's just a class. Each model has its own variables (features of each mobile phone) and methods (actions that you take to customize the phone).

Model is known as the **M** in the **MVC** system (Model-View-Controller).

Now let's get back to our **tickets table**. If we can turn the **tickets** table to be a model, we can then easily access and manage it. Eloquent helps us to do the magic.

We may use Eloquent ORM to create, edit, manipulate, deletes our tickets without writing a single line of SQL!

To get started, let's create our first **Ticket** model by running this Artisan command:

```
php artisan make:model Ticket
```

> **Note:** a model name should be singular, and a table name should be plural.

Yes! It's that simple! You've created a model!

You can find the **Ticket model (Ticket.php)** in the **app** directory.

Here is a new tip. You can also generate the tickets migration at the same time by adding the **-m** option:

```
php artisan make:model Ticket -m
```

Cool?

Ok, let's open our new Ticket model:

app/Ticket.php

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Ticket extends Model
{
    //
}
```

As you may notice, the **Ticket** model is just a **PHP** class that extends the **Model** class. Now we can use this file to tell Laravel about its **relationships**. For instance, each ticket is created by a user, we can tell tickets belongs to users by writing like this:

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

We also can be able to use this model to access any tickets' data, such as: title, content, etc.

```
public function getTitle()
{
    return $this->title
}
```

Eloquent is clever. It automatically finds and connects our models with our database tables if you name them correctly (Ticket model and tickets table, in this case).

For some reasons, if you want to use a different name, you can let Eloquent know that by defining a table property like this:

```
protected $table = 'yourCustomTableName';
```

Read more about Eloquent here:

http://laravel.com/docs/master/eloquent

Once we have the Ticket model, we can build a form to let the users create a new ticket!

Create a page to submit tickets

Now as we have the Ticket model, let's write the code for creating new tickets.

To create a new ticket, we will need to use **Controller action** (also known as Controller method) and **view** to display the new ticket form.

Create a view to display the submit ticket form

Go to our **views** directory, create a new directory called **tickets**.

Because we will have many ticket views (such as: create ticket view, edit ticket view, delete ticket view, etc.), we should store all the views in the **tickets** directory.

Next, create a new Blade template called **create.blade.php**

views/tickets/create.blade.php

```
@extends('master')
@section('title', 'Create a ticket')

@section('content')
    <div class="container col-md-8 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header ">
                <h5 class="float-left">Create a ticket</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body mt-2">
                <form>
                    <fieldset>
                        <div class="form-group">
                            <label for="title" class="col-lg-12 control-label">Title</label>
                            <div class="col-lg-12">
                                <input type="text" class="form-control" id="title" placeholder="Title">
                            </div>
                        </div>
                        <div class="form-group">
                            <label for="content" class="col-lg-12 control-label">Content</label>
                            <div class="col-lg-12">
                                <textarea class="form-control" rows="3" id="content"></textarea>
                                <span class="help-block">Feel free to ask us any question.</span>
                            </div>
                        </div>

                        <div class="form-group">
                            <div class="col-lg-10 col-lg-offset-2">
                                <button class="btn btn-default">Cancel</button>
                                <button type="submit" class="btn btn-primary">Submit</button>
                            </div>
                        </div>
                    </fieldset>
                </form>
            </div>
        </div>
    </div>
@endsection
```

Unfortunately, we can't see this view yet, we have to use a Controller action to display it. Open **PagesController.php**, edit:

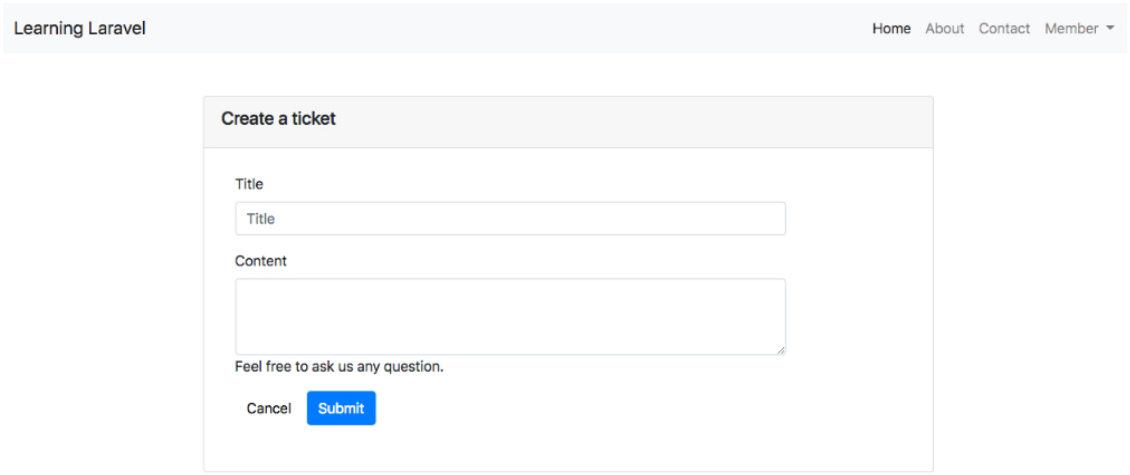
```
public function contact()
{
    return view('contact');
}
```

to:

```
public function contact()
{
    return view('tickets.create');
}
```

Instead of displaying the contact view, we redirect users to **tickets.create** view.

After saving these changes, we should see a new responsive **submit ticket** form when visiting the **contact** page:



Create a new controller for the tickets

Even though we can use **PagesController** to manage all the pages, we should create **TicketsController** to manage our tickets.

TicketsController will be responsible for creating, editing and deleting tickets. It will help to organize and maintain our application much easier.

You have known how to create a controller, let's create the **TicketsController** by running this command:

```
php artisan make:controller TicketsController --resource
```

As mentioned before, by using the **--resource** flag, Laravel creates some **RESTful actions** (create, edit, update, etc.) for us.

Update the **create** action as follows:

```
public function create()
{
    return view('tickets.create');
}
```

And don't forget to update the **web.php** file:

```
Route::get('/contact', 'TicketsController@create');
```

Great! You now have the **TicketsController**. Pretty simple, right?

Introducing HTTP Requests

In previous versions of Laravel, developers usually place validation anywhere they want. That is not a good practice.

Luckily, Laravel 5 has a new feature called **Requests** (aka HTTP Requests or Form Requests). When users send a request (submit a ticket, for example), we can use the new **Request** class to define some rules and validate the request. If the validator passes, then everything will be executed as normal. Otherwise, the user will be automatically redirected back to where they are.

As you can see, it's really convenient for us to validate our application's forms.

We will use **Request** to validate the **create ticket** form.

To create a new **Request**, simply run this Artisan command:

```
php artisan make:request TicketFormRequest
```

```
vagrant@homestead:~/Code/Laravel$ php artisan make:request TicketFormRequest
Request created successfully.
```

A new **TicketFormRequest** will be generated! You can find it in the **app/Http/Requests** directory.

Open the file, you can see that there are two methods: **authorize** and **rules**.

authorize() method

```
public function authorize()
{
    return false;
}
```

By default, it returns **false**. That means no one can be able to perform the request. To be able to submit the tickets, we have to turn it to **true**

```
public function authorize()
{
    return true;
}
```

rules() method

```
public function rules()
{
    return [
        //
    ];
}
```

We use this method to define our validation rules.

Currently, our **create ticket** form has two fields: title and content, we can set the following rules:

```
public function rules()
{
    return [
        'title' => 'required|min:3',
        'content'=> 'required|min:10',
    ];
}
```

required|min:3 validation rule means that the users must fill the title field, and the title should have a minimum three character length.

There are many validation rules, you can see a list of available rules at:

http://laravel.com/docs/master/validation#available-validation-rules

Install Laravel Collective packages

> **Note:** This is an optional section, you may **SKIP THIS SECTION**. The package might not support the latest version of Laravel yet.

Since Laravel 5, some Laravel components have been removed from the core framework. If you've used older versions of Laravel before, you may love these features:

- **HTML:** HTML helpers for creating common HTML and form elements
- **Annotations:** route and events annotations.
- **Remote:** a simple way to SSH into remote servers and run commands.

Fortunately, bringing all these features back is very easy. You just need to install LaravelCollective package!

https://laravelcollective.com

Don't know how to install the package? Let me show you.

Install a package using Composer

First, you need to open your **composer.json** file, which is placed in your application root.

composer.json

```
{
    "name": "laravel/laravel",
    "description": "The Laravel Framework.",
    "keywords": ["framework", "laravel"],
    "license": "MIT",
    "type": "project",
    "require": {
        "php": ">=7.0.0",
        "fideloper/proxy": "~3.3",
        "laravel/framework": "5.5.*",
        "laravel/tinker": "~1.0"
    },
    "require-dev": {
        "filp/whoops": "~2.0",
        "fzaninotto/faker": "~1.4",
        "mockery/mockery": "0.9.*",
        "phpunit/phpunit": "~6.0"
    },
    "autoload": {
        "classmap": [
            "database/seeds",
            "database/factories"
        ],
        "psr-4": {
            "App\\": "app/"
        }
    },
    "autoload-dev": {
        "psr-4": {
            "Tests\\": "tests/"
        }
    },
    "extra": {
        "laravel": {
            "dont-discover": [
            ]
        }
    },
    "scripts": {
        "post-root-package-install": [
            "@php -r \"file_exists('.env') || copy('.env.example', '.env');\""
        ],
        "post-create-project-cmd": [
            "@php artisan key:generate"
        ],
        "post-autoload-dump": [
            "Illuminate\\Foundation\\ComposerScripts::postAutoloadDump",
            "@php artisan package:discover"
        ]
    },
    "config": {
        "preferred-install": "dist",
        "sort-packages": true,
        "optimize-autoloader": true
    }
}
```

This is a **JSON (Javascript Object Notation)** file. We use **JSON** to store and exchange data. JSON is very easy to read. If you can read HTML or XML, I'm sure that you can read JSON.

If you can't read it, learn more about JSON here:

https://w3schools.com/json

In this section, we will add the **HTML** package to our app. The instructions can be found here:

https://laravelcollective.com/docs/master/html

To install a Laravel package using Composer, you just need to add the following code:

find:

```
"require": {
    "php": ">=5.6.4",
    "laravel/framework": "5.5.*",
    "laravel/tinker": "~1.0"
},
```

add:

```
"require": {
    "php": ">=5.6.4",
    "laravel/framework": "5.5.*",
    "laravel/tinker": "~1.0",
    "laravelcollective/html": "5.5.*"
},
```

> **Note:** Your version could be different. For example, if you're using **Laravel 5.6**, the code should be "laravelcollective/html": "5.6.*". Check for the latest version here.

Save the file and run this command at your application root:

```
composer update
```

Done! You've just installed **LaravelCollective/HTML** package!

Create a service provider and aliases

After installing the HTML package via Composer. You need to follow some extra steps to let Laravel know where to find the package and use it.

To use the package, you have to add a **service provider** to the **providers** array of the **config/app.php**.

Find:

```
'providers' => [

    /*
     * Laravel Framework Service Providers...
     */
    Illuminate\Auth\AuthServiceProvider::class,
    Illuminate\Broadcasting\BroadcastServiceProvider::class,
    Illuminate\Bus\BusServiceProvider::class,
    Illuminate\Cache\CacheServiceProvider::class,
    Illuminate\Foundation\Providers\ConsoleSupportServiceProvider::class,
    Illuminate\Cookie\CookieServiceProvider::class,
    Illuminate\Database\DatabaseServiceProvider::class,
    Illuminate\Encryption\EncryptionServiceProvider::class,
    Illuminate\Filesystem\FilesystemServiceProvider::class,
    Illuminate\Foundation\Providers\FoundationServiceProvider::class,
    Illuminate\Hashing\HashServiceProvider::class,
    Illuminate\Mail\MailServiceProvider::class,
    Illuminate\Notifications\NotificationServiceProvider::class,
    Illuminate\Pagination\PaginationServiceProvider::class,
    Illuminate\Pipeline\PipelineServiceProvider::class,
    Illuminate\Queue\QueueServiceProvider::class,
    Illuminate\Redis\RedisServiceProvider::class,
    Illuminate\Auth\Passwords\PasswordResetServiceProvider::class,
    Illuminate\Session\SessionServiceProvider::class,
    Illuminate\Translation\TranslationServiceProvider::class,
    Illuminate\Validation\ValidationServiceProvider::class,
    Illuminate\View\ViewServiceProvider::class,

    /*
     * Package Service Providers...
     */
    Laravel\Tinker\TinkerServiceProvider::class,

    /*
     * Application Service Providers...
     */
    App\Providers\AppServiceProvider::class,
    App\Providers\AuthServiceProvider::class,
    // App\Providers\BroadcastServiceProvider::class,
    App\Providers\EventServiceProvider::class,
    App\Providers\RouteServiceProvider::class,

],
```

Add the following line to the **$provider** array:

```
Collective\Html\HtmlServiceProvider::class,
```

Your **$provider** array should look like this:

```
'providers' => [

    /*
     * Laravel Framework Service Providers...
     */
    Illuminate\Auth\AuthServiceProvider::class,
    Illuminate\Broadcasting\BroadcastServiceProvider::class,
    Illuminate\Bus\BusServiceProvider::class,
    Illuminate\Cache\CacheServiceProvider::class,
    Illuminate\Foundation\Providers\ConsoleSupportServiceProvider::class,
    Illuminate\Cookie\CookieServiceProvider::class,
    Illuminate\Database\DatabaseServiceProvider::class,
    Illuminate\Encryption\EncryptionServiceProvider::class,
    Illuminate\Filesystem\FilesystemServiceProvider::class,
    Illuminate\Foundation\Providers\FoundationServiceProvider::class,
    Illuminate\Hashing\HashServiceProvider::class,
    Illuminate\Mail\MailServiceProvider::class,
    Illuminate\Notifications\NotificationServiceProvider::class,
    Illuminate\Pagination\PaginationServiceProvider::class,
    Illuminate\Pipeline\PipelineServiceProvider::class,
    Illuminate\Queue\QueueServiceProvider::class,
    Illuminate\Redis\RedisServiceProvider::class,
    Illuminate\Auth\Passwords\PasswordResetServiceProvider::class,
    Illuminate\Session\SessionServiceProvider::class,
```

```
    Illuminate\Translation\TranslationServiceProvider::class,
    Illuminate\Validation\ValidationServiceProvider::class,
    Illuminate\View\ViewServiceProvider::class,

    /*
     * Package Service Providers...
     */
    Laravel\Tinker\TinkerServiceProvider::class,
    Collective\Html\HtmlServiceProvider::class,

    /*
     * Application Service Providers...
     */
    App\Providers\AppServiceProvider::class,
    App\Providers\AuthServiceProvider::class,
    // App\Providers\BroadcastServiceProvider::class,
    App\Providers\EventServiceProvider::class,
    App\Providers\RouteServiceProvider::class,

],
```

Then find the aliases array:

```
'aliases' => [

    'App' => Illuminate\Support\Facades\App::class,
    'Artisan' => Illuminate\Support\Facades\Artisan::class,
    'Auth' => Illuminate\Support\Facades\Auth::class,
    'Blade' => Illuminate\Support\Facades\Blade::class,
    'Broadcast' => Illuminate\Support\Facades\Broadcast::class,
    'Bus' => Illuminate\Support\Facades\Bus::class,
    'Cache' => Illuminate\Support\Facades\Cache::class,
    'Config' => Illuminate\Support\Facades\Config::class,
    'Cookie' => Illuminate\Support\Facades\Cookie::class,
    'Crypt' => Illuminate\Support\Facades\Crypt::class,
    'DB' => Illuminate\Support\Facades\DB::class,
    'Eloquent' => Illuminate\Database\Eloquent\Model::class,
    'Event' => Illuminate\Support\Facades\Event::class,
    'File' => Illuminate\Support\Facades\File::class,
    'Gate' => Illuminate\Support\Facades\Gate::class,
    'Hash' => Illuminate\Support\Facades\Hash::class,
    'Lang' => Illuminate\Support\Facades\Lang::class,
    'Log' => Illuminate\Support\Facades\Log::class,
    'Mail' => Illuminate\Support\Facades\Mail::class,
    'Notification' => Illuminate\Support\Facades\Notification::class,
    'Password' => Illuminate\Support\Facades\Password::class,
    'Queue' => Illuminate\Support\Facades\Queue::class,
    'Redirect' => Illuminate\Support\Facades\Redirect::class,
    'Redis' => Illuminate\Support\Facades\Redis::class,
    'Request' => Illuminate\Support\Facades\Request::class,
    'Response' => Illuminate\Support\Facades\Response::class,
    'Route' => Illuminate\Support\Facades\Route::class,
    'Schema' => Illuminate\Support\Facades\Schema::class,
    'Session' => Illuminate\Support\Facades\Session::class,
    'Storage' => Illuminate\Support\Facades\Storage::class,
    'URL' => Illuminate\Support\Facades\URL::class,
    'Validator' => Illuminate\Support\Facades\Validator::class,
    'View' => Illuminate\Support\Facades\View::class,

],
```

add these two aliases to the aliases array:

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

You should have:

```
'aliases' => [

    'App' => Illuminate\Support\Facades\App::class,
    'Artisan' => Illuminate\Support\Facades\Artisan::class,
    'Auth' => Illuminate\Support\Facades\Auth::class,
    'Blade' => Illuminate\Support\Facades\Blade::class,
    'Broadcast' => Illuminate\Support\Facades\Broadcast::class,
    'Bus' => Illuminate\Support\Facades\Bus::class,
    'Cache' => Illuminate\Support\Facades\Cache::class,
    'Config' => Illuminate\Support\Facades\Config::class,
    'Cookie' => Illuminate\Support\Facades\Cookie::class,
    'Crypt' => Illuminate\Support\Facades\Crypt::class,
    'DB' => Illuminate\Support\Facades\DB::class,
    'Eloquent' => Illuminate\Database\Eloquent\Model::class,
    'Event' => Illuminate\Support\Facades\Event::class,
    'File' => Illuminate\Support\Facades\File::class,
    'Gate' => Illuminate\Support\Facades\Gate::class,
    'Hash' => Illuminate\Support\Facades\Hash::class,
    'Lang' => Illuminate\Support\Facades\Lang::class,
    'Log' => Illuminate\Support\Facades\Log::class,
    'Mail' => Illuminate\Support\Facades\Mail::class,
    'Notification' => Illuminate\Support\Facades\Notification::class,
    'Password' => Illuminate\Support\Facades\Password::class,
    'Queue' => Illuminate\Support\Facades\Queue::class,
```

```
    'Redirect' => Illuminate\Support\Facades\Redirect::class,
    'Redis' => Illuminate\Support\Facades\Redis::class,
    'Request' => Illuminate\Support\Facades\Request::class,
    'Response' => Illuminate\Support\Facades\Response::class,
    'Route' => Illuminate\Support\Facades\Route::class,
    'Schema' => Illuminate\Support\Facades\Schema::class,
    'Session' => Illuminate\Support\Facades\Session::class,
    'Storage' => Illuminate\Support\Facades\Storage::class,
    'URL' => Illuminate\Support\Facades\URL::class,
    'Validator' => Illuminate\Support\Facades\Validator::class,
    'View' => Illuminate\Support\Facades\View::class,
    'Form' => Collective\Html\FormFacade::class,
    'Html' => Collective\Html\HtmlFacade::class,
],
```

Now you can use the **LaravelCollective/HTML** package!

How to use HTML package

The **HTML** package helps us to build forms easier and faster. Let's see an example:

Normal HTML code:

```
<form action="contact">
    <label>First name:</label>
    <input type="text" name="firstname" value="Enter your first name">
    <br />
    <label>Last name:</label>
    <input type="text" name="lastname" value="Enter your last name">
    <br />
    <input type="submit" value="Submit">
</form>
```

Using HTML package:

```
{!! Form::open(array ('url' => 'contact')) !!}
{!! Form::label('First name') !!}
{!! Form::text('firstname', 'Enter your first name') !!}
<br />
{!! Form::label('Last name') !!}
{!! Form::text('lastname', 'Enter your last name') !!}
<br />
{!! Form::submit() !!}
{!! Form::close() !!}
```

As you see, we use **Form::open()** to create our opening form tag and **Form::close()** to close the form.

Text fields and labels can be generated using **Form::text** and **Form::label** method.

You can learn more about how to use HTML package by reading Laravel 4 official docs:

http://laravel.com/docs/4.2/html

If you don't like to take advantage of the HTML package to build your forms, you don't have to use it. I just want you to understand its syntax because some Laravel developers are still using it these days. It would be better if you know both methods.

Submit the form data

Having learned about **Requests**, working with **Controllers** and **View** to build the **create ticket** form, now you're ready to process the submitted data.

if you click the submit button now, nothing happens. We need to use other **HTTP method** to submit the form.

Two commonly used HTTP methods for a client to communicate with a server are: **GET** and **POST.**

We've used **GET** to display the form:

```
Route::get('/contact', 'TicketsController@create');
```

But we won't use **GET** to submit data. **GET** requests should only be used to retrieve data.

We always use **POST** method to handle the form submissions endpoints. When we use **POST**, requests are never cached, parameters are not saved in users' browser history. Therefore, **POST** is safer than **GET**.

Let's open the **web.php** file, add:

```
Route::post('/contact', 'TicketsController@store');
```

Good! Now when users make a **POST** request to the contact page, this route tells Laravel to execute the TicketsController's **store** action.

The store action is still empty. You can update it to display the form data:

TicketsController.php

```
public function store(TicketFormRequest $request)
{
    return $request->all();
}
```

We use the **TicketFormRequest** as **a parameter** of the **store action** here to tell Laravel that we want to apply validation to the store action.

Laravel requires us to **type-hint** the **Illuminate\Http\Request** class on our controller constructor to obtain an instance of the current **HTTP request**. Simply put, we need to add this line at the top of the **TicketsController.php** file:

```
use App\Http\Requests\TicketFormRequest;
```

find:

```
class TicketsController extends Controller
{
```

add above:

```
use App\Http\Requests\TicketFormRequest;
class TicketsController extends Controller
{
```

One more step, you need to update the ticket form to send POST requests. Open **tickets/create.blade.php**.

Find:

```
<form>
```

Update to:

```
<form method="post">
```

You also need to tell Laravel the name of the fields:

Find:

```
<input type="text" class="form-control" id="title" placeholder="Title">
```

Update to:

```
<input type="text" class="form-control" id="title" placeholder="Title" name="title">
```

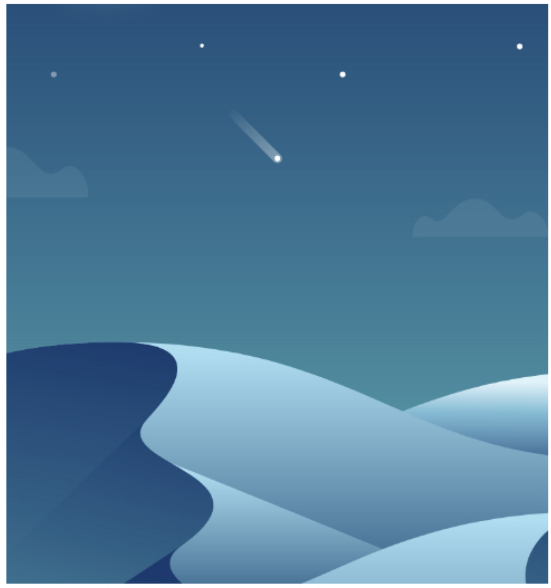
Find:

```
<textarea class="form-control" rows="3" id="content"></textarea>
```

Update to:

```
<textarea class="form-control" rows="3" id="content" name="content"></textarea>
```

Now, try to click the submit button!



Oops! There is an error - known as **TokenMismatchException**.

What is it?

For security purposes, Laravel requires a token to be sent when using the **POST** method. If you don't send any token, it will throw an error.

To fix this, you need to add a **hidden token field** below your form opening tag:

```
<form method="post">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
```

Good! Refresh your browser, fill the form and hit submit again, you'll see:


```
{
    "_token": "I5p1O4YxBShS3UBl46AgWtrDZUPxlDimtTrBF8L5",
    "title": "My first ticket",
    "content": "This is a test"
}
```


Yayyy! We can see the ticket data! Everything is working!

> **Note:** If the page is refreshed and you don't see the ticket data, that means the validation rules are working. You have to follow the rules and fill the title and the content correctly.

One last step is to display the errors when the users don't fill the form or the form is not valid.

Find:

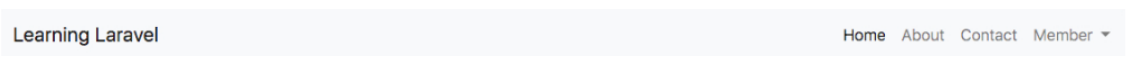
```
<form method="post">
```

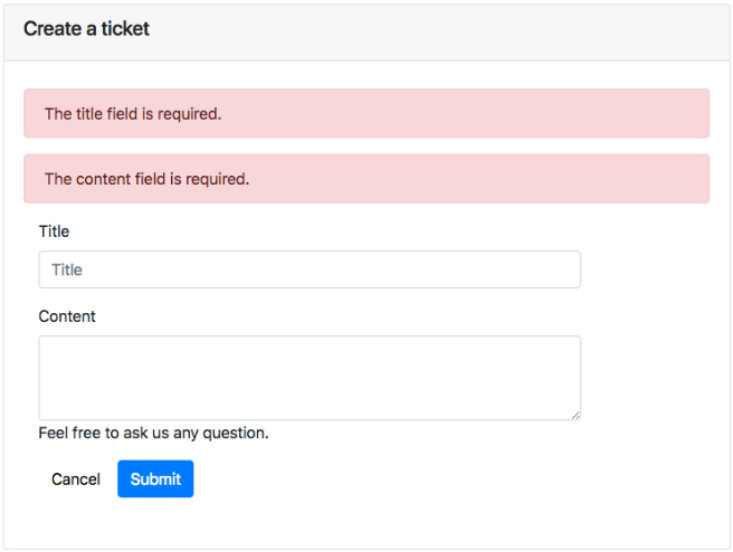
add below:

```
@foreach ($errors->all() as $error)
    <p class="alert alert-danger">{{ $error }}</p>
@endforeach
```

Basically, if the validator fails, Laravel will store all errors in the session. We can easily access the errors via **$errors** object.

Now, let's go back to the form, don't fill anything and hit the submit button:





Here is the new **tickets/create.blade.php** file:

```
@extends('master')
@section('title', 'Create a ticket')

@section('content')
    <div class="container col-md-8 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header ">
                <h5 class="float-left">Create a ticket</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body mt-2">
                <form method="post">
                    @foreach ($errors->all() as $error)
                        <p class="alert alert-danger">{{ $error }}</p>
                    @endforeach
                    @if (session('status'))
                        <div class="alert alert-success">
                            {{ session('status') }}
                        </div>
                    @endif
                    <input type="hidden" name="_token" value="{{ csrf_token() }}">
                    <fieldset>
                        <div class="form-group">
                            <label for="title" class="col-lg-12 control-label">Title</label>
                            <div class="col-lg-12">
                                <input type="text" class="form-control" id="title" placeholder="Title" name="title">
                            </div>
                        </div>
                        <div class="form-group">
                            <label for="content" class="col-lg-12 control-label">Content</label>
                            <div class="col-lg-12">
                                <textarea class="form-control" rows="3" id="content" name="content"></textarea>
                                <span class="help-block">Feel free to ask us any question.</span>
                            </div>
                        </div>

                        <div class="form-group">
                            <div class="col-lg-10 col-lg-offset-2">
                                <button class="btn btn-default">Cancel</button>
                                <button type="submit" class="btn btn-primary">Submit</button>
                            </div>
                        </div>
                    </fieldset>
                </form>
            </div>
        </div>
    </div>
@endsection
```

Using .env file

In the next section, we will learn how to insert data into the database. Before working with databases, you should understand **.env** file first.

What is the .env file?

Our applications usually run in different environments. For example, we develop our apps on a local server, and deploy it on a production server. The database settings and server credentials of each environment might be different. Laravel 5 provides us an easy way to handle different configuration settings by simply editing the **.env** file.

The .env file helps us to load custom configurations variables without editing .htaccess files or virtual hosts, and keep our sensitive credentials more secure.

Learn more about .env file here:

https://github.com/vlucas/phpdotenv

How to edit it?

The .env file is very easy to configure. Let's open it:

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:7mE0BuvaO41HhPKztH+ZGhXval8SMbABs7wEwI7/w10=
APP_DEBUG=true
APP_URL=http://localhost

LOG_CHANNEL=stack

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

BROADCAST_DRIVER=log
CACHE_DRIVER=file
QUEUE_CONNECTION=sync
SESSION_DRIVER=file
SESSION_LIFETIME=120

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null

PUSHER_APP_ID=
PUSHER_APP_KEY=
PUSHER_APP_SECRET=
PUSHER_APP_CLUSTER=mt1

MIX_PUSHER_APP_KEY="${PUSHER_APP_KEY}"
MIX_PUSHER_APP_CLUSTER="${PUSHER_APP_CLUSTER}"
```

As you see, the file is very clear. Let's try to edit a few settings.

Currently, you're using the default **homestead** database. If you've created a different database and you want to use it instead, edit this line:

```
DB_DATABASE=homestead
```

to

```
DB_DATABASE=yourCustomDatabaseName
```

If you don't want to display full error messages, turn the **APP_DEBUG** to false.

If you're using Sendinblue to send emails, replace these lines with your **Sendinblue credentials**:

```
MAIL_DRIVER=smtp
MAIL_HOST=mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

For example:

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp-relay.sendinblue.com
MAIL_PORT=587
MAIL_USERNAME=learninglaravel
MAIL_PASSWORD=secret
```

Now, let's move to the fun part!

Insert data into the database

Insert data into the database

In the previous section, you've learned how to receive and validate the users' requests.

Once you have the submitted form data, inserting the data into the database is pretty easy.

First, let's begin by putting this line at the top of your **TicketsController** file:

```
use App\Ticket;
```

Be sure to put it above the class name:

```
use App\Ticket;
class TicketsController extends Controller
{
```

This tells Laravel that you want to use your **Ticket model** in this class.

Now you can use Ticket model to store the form data. Update the **store** action as follows:

```
public function store(TicketFormRequest $request)
{
    $slug = uniqid();
    $ticket = new Ticket(array(
        'title' => $request->get('title'),
        'content' => $request->get('content'),
        'slug' => $slug
    ));

    $ticket->save();

    return redirect('/contact')->with('status', 'Your ticket has been created! Its unique id is: '.$slug);
}
```

Let's see the code line by line:

```
$slug = uniqid();
```

We use the uniqid() function to generate a unique ID based on the microtime. You may use **md5()** function to generate the slugs or create your custom slugs.

This is the ticket's unique ID.

```
$ticket = new Ticket(array(
    'title' => $request->get('title'),
    'content' => $request->get('content'),
    'slug' => $slug
));
```

Next, we create a new **Ticket model** instance, set attributes on the model.

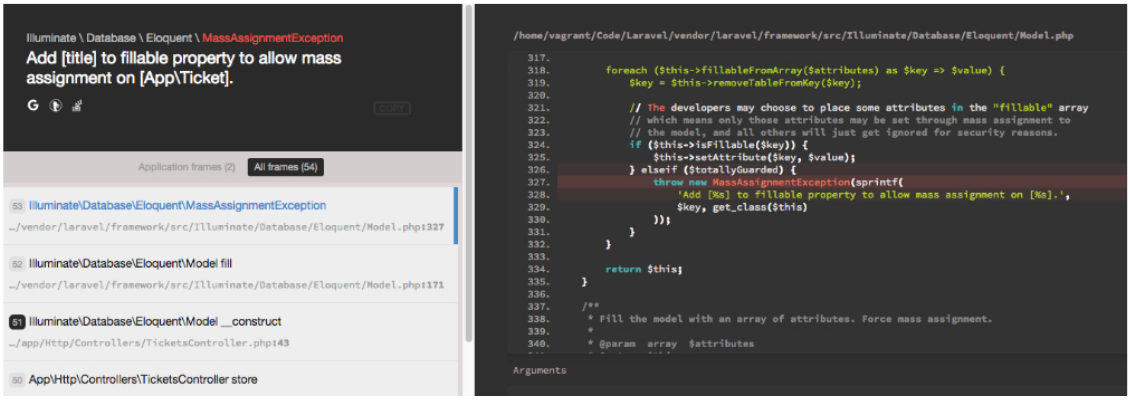
```
$ticket->save();
```

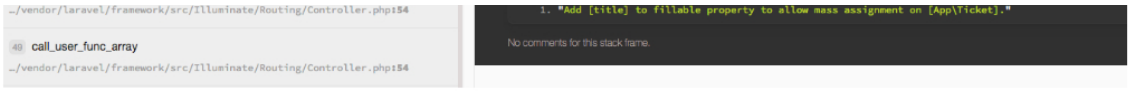
Then we call the **save** method to save the data to our database.

```
return redirect('/contact')->with('status', 'Your ticket has been created! Its unique id is: '.$slug);
```

Once the ticket has been saved, we redirect users to the contact page with a **message**.

Finally, try to create a new ticket and submit it.





Oh no!

There is an error: "**MassAssignmentException**"

Don't worry, it's a Laravel feature that protect against **mass-assignment**.

What is mass-assignment?

According to the Laravel official docs:

> *"mass-assignment vulnerability occurs when user's pass unexpected HTTP parameters through a request, and then 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 mapped onto your model's create method, allowing the user to escalate themselves to an administrator"*

Read more about it here:

https://laravel.com/docs/master/eloquent#mass-assignment

To save the ticket, open the **Ticket model**. (Ticket.php file)

Then place the following contents into the Ticket Model:

```
class Ticket extends Model
{
    protected $fillable = ['title', 'content', 'slug', 'status', 'user_id'];
}
```

The **$fillable** property make the columns **mass assignable**.

Alternatively, you may use the **$guarded** property to make all attributes **mass assignable** except for your chosen attributes. For example, I use the **id** column here:

```
protected $guarded = ['id'];
```

> **Note:** You must use either **$fillable** or **$guarded**.

One more thing to do, we need to update the **tickets/create.blade.php** view to display the status message:

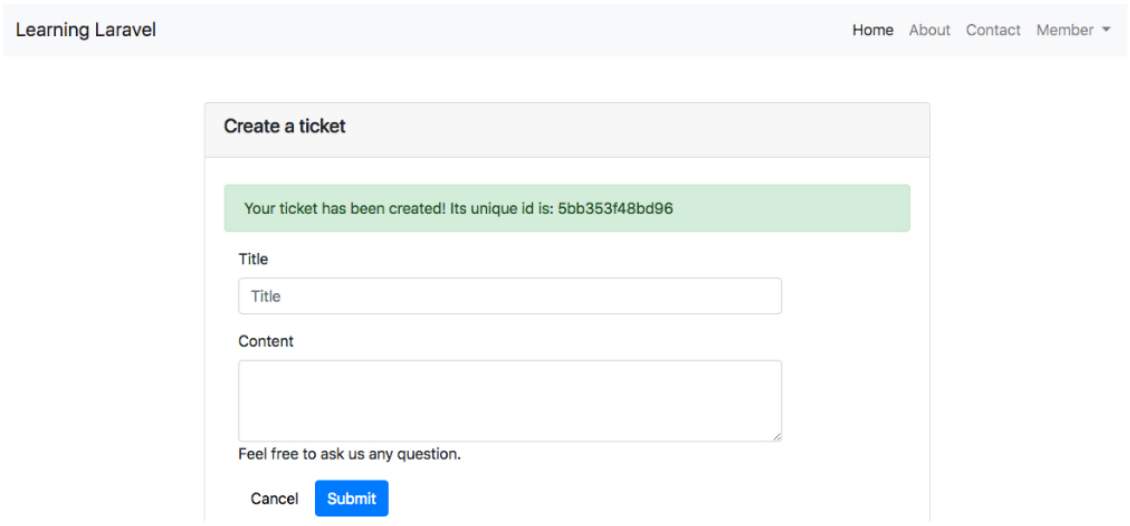
Find:

```
<form method="post">

    @foreach ($errors->all() as $error)
        <p class="alert alert-danger">{{ $error }}</p>
    @endforeach
```

Add Below:

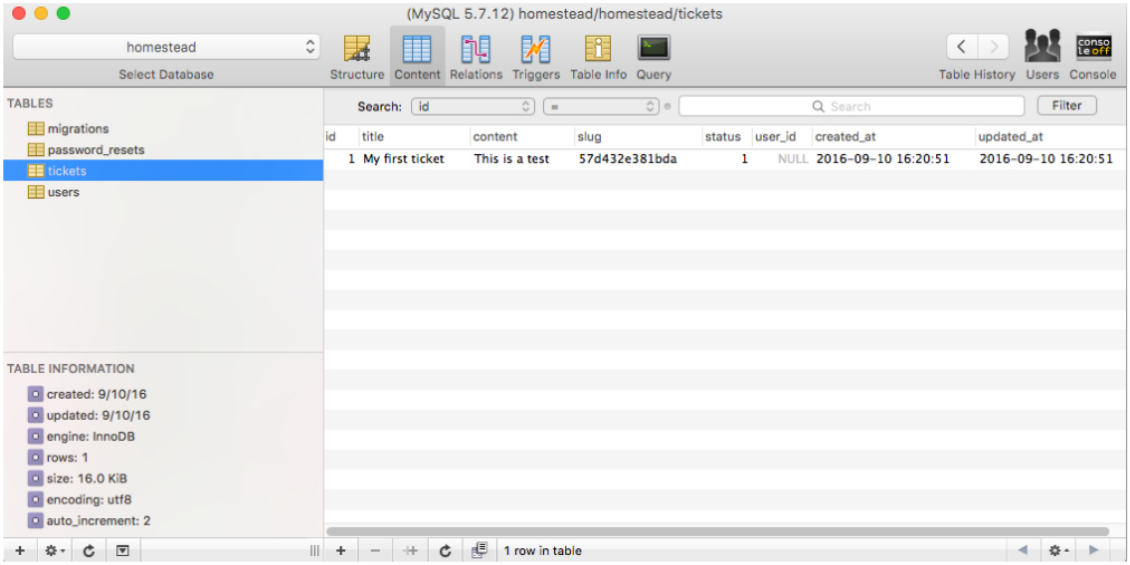
```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

Good! Try to create a new ticket again.



Well done! You've just saved a new ticket to the database!

Be sure to check your application database, you should see some new records in the tickets table:



View all tickets

As you continue developing the app, you'll find that you need a way to display all tickets, so you can be able to view, modify or delete them easily.

The following are the steps to list all tickets:

First, we'll modify the **web.php** file:

```
Route::get('/tickets', 'TicketsController@index');
```

When users access **homestead.test/tickets**, we use **TicketsController** to execute the index action. Feel free to change the link path or the action's name to whatever you like.

Then, open the **TicketsController** file, and update the **index** action:

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

We use **Ticket::all()** to get all tickets in our database and store them in the **$tickets** variable.

Before we return the **tickets.index** view, we use the **compact()** method to convert the result to an array, and pass it to the view.

Alternatively, you can use:

```
return view('tickets.index')->with('tickets', $tickets);
```

or

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

Those three methods are the same.

Finally, create a new view called **index.blade.php** and place it in the **tickets** directory:

views/tickets/index.blade.php

```
@extends('master')
@section('title', 'View all tickets')
@section('content')

    <div class="container col-md-8 col-md-offset-2 mt-5">
        <div class="card">
            <div class="card-header ">
                <h5 class="float-left">Tickets</h5>
```

```
                <div class="clearfix"></div>
            </div>
            <div class="card-body mt-2">
                @if ($tickets->isEmpty())
                    <p> There is no ticket.</p>
                @else
                    <table class="table">
                        <thead>
                        <tr>
                            <th>ID</th>
                            <th>Title</th>
                            <th>Status</th>
                        </tr>
                        </thead>
                        <tbody>
                        @foreach($tickets as $ticket)
                            <tr>
                                <td>{{ $ticket->id }} </td>
                                <td>{{ $ticket->title }}</td>
                                <td>{{ $ticket->status ? 'Pending' : 'Answered' }}</td>
                            </tr>
                        @endforeach
                        </tbody>
                    </table>
                @endif
            </div>
        </div>
    </div>

@endsection
```

We perform the following steps to load the tickets:

```
@if ($tickets->isEmpty())
    <p> There is no ticket.</p>
@else
```

First, we check if the **$tickets** variable is empty or not. If it's empty, then we display a message to our users.

```
@else
    <table class="table">
        <thead>
            <tr>
                <th>ID</th>
                <th>Title</th>
                <th>Status</th>
            </tr>
        </thead>
        <tbody>
                @foreach($tickets as $ticket)
                    <tr>
                        <td>{{ $ticket->id }} </td>
                        <td>{{ $ticket->title }}</td>
                        <td>{{ $ticket->status ? 'Pending' : 'Answered' }}</td>
                    </tr>
                @endforeach
        </tbody>
    </table>
@endif
```

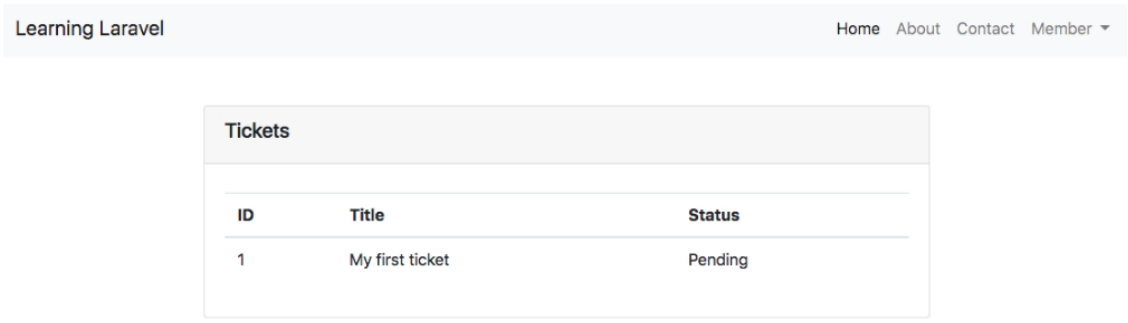
If the **$tickets** is not empty, we use **foreach() loop** to display all tickets.

```
<td>{{ $ticket->status ? 'Pending' : 'Answered' }}</td>
```

Here is how we can write the **if else** statement in a short way. If the ticket's status is **1**, we say that it's **pending**. If the ticket's status is **0**, we say that it's **answered**.

Feel free to change the name of the status to your liking.

Go to **homestead.test/tickets**, you should be able to view all the tickets that you've created!



View a single ticket

At this point, viewing a ticket is much easier. When we click on the title of the ticket, we want to display its content and status.

As usual, open **web.php** file, and add:

```
Route::get('/ticket/{slug?}', 'TicketsController@show');
```

You should notice that we use a special route (**/ticket/{slug?}**) here. By doing this, we tell Laravel that any **route parameter** named **slug** will be bound to the **show** action of our **TicketsController**. Simply put, when we visit **ticket/558467e731bb8**, Laravel automatically detects the **slug** (which is **558467e731bb8**) and pass it to the action.

> **Note:** you can change **{slug?}** to **{slug}** or whatever you like. Be sure to put your custom name in the **{ }** brackets.

Next, open **TicketsController**, update the **show action** as follows:

```
public function show($slug)
{
    $ticket = Ticket::whereSlug($slug)->firstOrFail();
    return view('tickets.show', compact('ticket'));
}
```

We pass the **slug** of the ticket we want to display in the **show** action. Then we can use this slug to find the correct ticket via our **Ticket** model's **firstOrFail** method.

The **firstOrFail** method will retrieve the first result of the query. If there is no result, it will throw a **ModelNotFoundException**.

If you don't want to throw an exception, you can use the **first()** method.

```
$ticket = Ticket::whereSlug($slug)->first();
```

Finally, we return the **tickets.show** view with the ticket.

We don't have the **show** view yet. Let's create it:

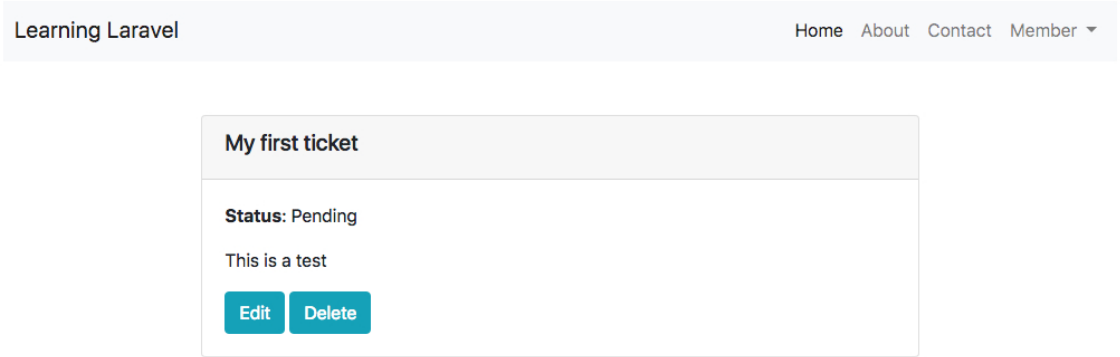
views/tickets/show.blade.php

```
@extends('master')
@section('title', 'View a ticket')
@section('content')

    <div class="container col-md-8 col-md-offset-2 mt-5">
        <div class="card">
            <div class="card-header ">
                <h5 class="float-left">{{ $ticket->title }}</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body">
                <p> <strong>Status</strong>: {{ $ticket->status ? 'Pending' : 'Answered' }}</p>
                <p> {{ $ticket->content }} </p>
                <a href="#" class="btn btn-info">Edit</a>
                <a href="#" class="btn btn-info">Delete</a>
            </div>
        </div>
    </div>

@endsection
```

Pretty simple, right?

We just display the ticket's title, status and content. We also add the **edit** and **delete** button here to easily edit and remove the ticket.

Now if you access http://homestead.test/ticket/yourSlug, you'll see:



> **Note:** your **ticket's slug** may be different. Be sure to use a **correct slug** to view the ticket.

Using a helper function

Laravel has many **helper** functions. These PHP functions are really useful. We can use helper functions to manage paths, modify strings, configure our application, etc.

You can find them here:

https://laravel.com/docs/master/helpers

Now, as we have a view to display all the tickets, let's explore how we can link the **title of the ticket** to the **TicketController's show** action.

Open **tickets/index.blade.php**.

Find:

```
@foreach($tickets as $ticket)
    <tr>
        <td>{{ $ticket->id }} </td>
        <td>{{ $ticket->title }}</td>
        <td>{{ $ticket->status ? 'Pending' : 'Answered' }}</td>
    </tr>
@endforeach
```

Update to:

```
@foreach($tickets as $ticket)
    <tr>
        <td>{{ $ticket->id }}</td>
        <td>
            <a href="{{ action('TicketsController@show', $ticket->slug) }}">{{ $ticket->title }} </a>
        </td>
        <td>{{ $ticket->status ? 'Pending' : 'Answered' }}</td>
    </tr>
@endforeach
```

Here, we use **action** function to generate a URL for the **TicketsController's show** action:

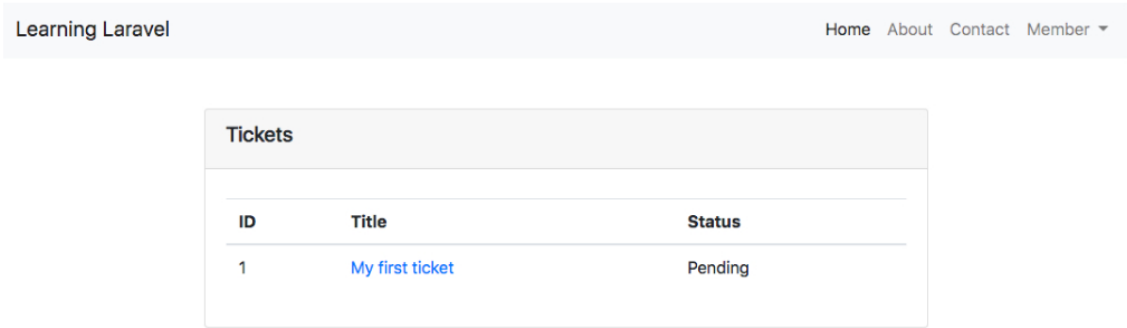
```
action('TicketsController@show', $ticket->slug)
```

The second argument is a **route parameter**. We use **slug** to find the ticket, so we put the ticket's slug here.

Alternatively, you can write the code like this:

```
action('TicketsController@show', ['slug' => $ticket->slug])
```

Now, when you access **homestead.test/tickets**, you can click on the title to view the ticket.



Edit a ticket

It's time to move on to create our ticket edit form.

Open **web.php**, add this route:

```
Route::get('/ticket/{slug?}/edit','TicketsController@edit');
```

When users go to **/ticket/{slug?}/edit**, we redirect them to the **TicketsController's edit action**.

Let's modify the **edit** action:

```
public function edit($slug)
{
    $ticket = Ticket::whereSlug($slug)->firstOrFail();
    return view('tickets.edit', compact('ticket'));
}
```

We find the ticket using its slug, then we use the **tickets.edit** view to display the edit form.

Let's create our **edit view** at **resouces/views/tickets/edit.blade.php**:

```
@extends('master')
@section('title', 'Edit a ticket')

@section('content')
    <div class="container col-md-8 col-md-offset-2">
        <div class="card mt-5">
            <div class="card-header ">
                <h5 class="float-left">Edit ticket</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body mt-2">
                <form method="post">
                    @foreach ($errors->all() as $error)
                        <p class="alert alert-danger">{{ $error }}</p>
                    @endforeach
                    @if (session('status'))
                        <div class="alert alert-success">
                            {{ session('status') }}
                        </div>
                    @endif
                    <input type="hidden" name="_token" value="{{ csrf_token() }}">
                    <fieldset>
                        <div class="form-group">
                            <label for="title" class="col-lg-12 control-label">Title</label>
                            <div class="col-lg-12">
                                <input type="text" class="form-control" id="title" placeholder="Title" name="title" value="{{ $ticke
                            </div>
                        </div>
                        <div class="form-group">
                            <label for="content" class="col-lg-12 control-label">Content</label>
                            <div class="col-lg-12">
                                <textarea class="form-control" rows="3" id="content" name="content">{{ $ticket->content }}</textarea
                                <span class="help-block">Feel free to ask us any question.</span>
                            </div>
                        </div>
                        <div class="form-group">
                            <label>
                                <input type="checkbox" name="status" {{ $ticket->status?"":"checked"}} > Close this ticket?
                            </label>
                        </div>
                        <div class="form-group">
                            <div class="col-lg-10 col-lg-offset-2">
                                <button class="btn btn-default">Cancel</button>
                                <button type="submit" class="btn btn-primary">Update</button>
                            </div>
                        </div>
                    </fieldset>
                </form>
            </div>
        </div>
    </div>
@endsection
```

This view is very similar to the **create** view, but we add a new checkbox to modify **ticket's status**.

```
<div class="form-group">
    <label>
        <input type="checkbox" name="status" {{ $ticket->status?"":"checked"}} > Close this ticket?
    </label>
</div>
```

Try to understand this line:

```
{{ $ticket->status?"":"checked"}}
```

If the status is **1 (pending)**, we display **nothing**, the checkbox is **not checked**. If the status is **0 (answered)** , we display a **checked attribute**, the checkbox is **checked**.

Good! Now, let's open the **show** view, update the **edit button** as follows:

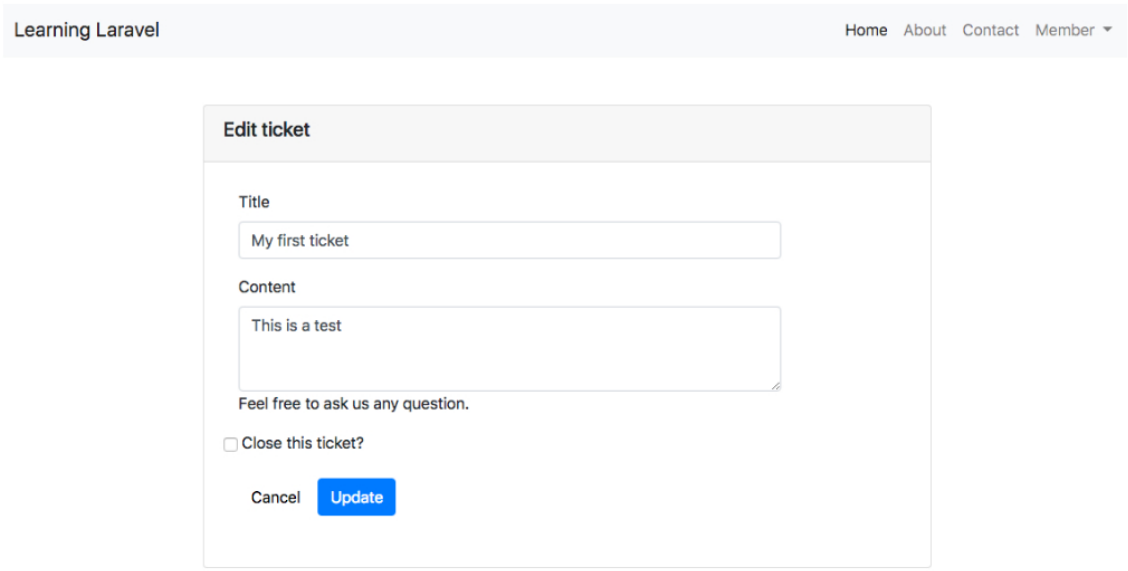
Find:

```
<a href="#" class="btn btn-info">Edit</a>
```

Update to:

```
<a href="{{ action('TicketsController@edit', $ticket->slug) }}" class="btn btn-info">Edit</a>
```

We use the **action helper** again! When you click on the edit button, you should be able to access the edit form:



Unfortunately, we can't update the ticket yet. Remember what you've done to create a new ticket?

We need to use **POST method** to submit the form.

Open **web.php**, add:

```
Route::post('/ticket/{slug?}/edit','TicketsController@update');
```

Then use **update action** to handle the submission and store the changes.

```
public function update($slug, TicketFormRequest $request)
{
    $ticket = Ticket::whereSlug($slug)->firstOrFail();
    $ticket->title = $request->get('title');
    $ticket->content = $request->get('content');
    if($request->get('status') != null) {
        $ticket->status = 0;
    } else {
        $ticket->status = 1;
    }
    $ticket->save();
    return redirect(action('TicketsController@edit', $ticket->slug))->with('status', 'The ticket '.$slug.' has been updated!');

}
```

As you notice, you can save the ticket by using the following code:

```
$ticket = Ticket::whereSlug($slug)->firstOrFail();
$ticket->title = $request->get('title');
$ticket->content = $request->get('content');
if($request->get('status') != null) {
    $ticket->status = 0;
} else {
    $ticket->status = 1;
}
$ticket->save();
```

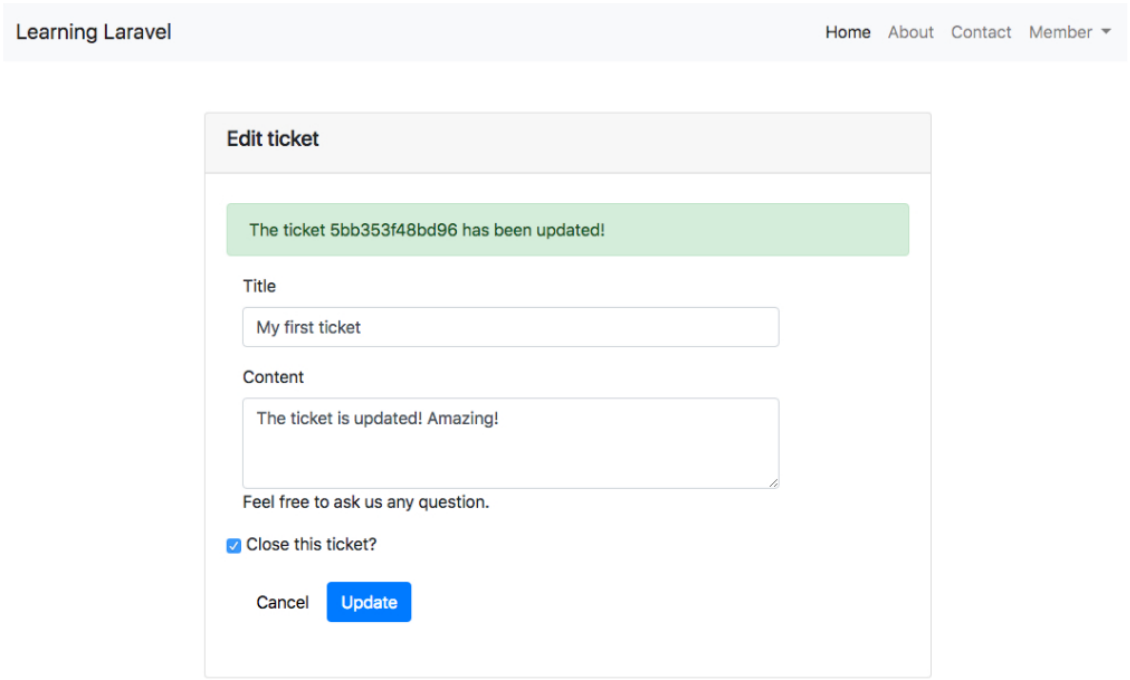
This is how we can check if the users select the status checkbox or not:

```
if($request->get('status') != null) {
    $ticket->status = 0;
} else {
    $ticket->status = 1;
}
```

Finally, we redirect users to the ticket page with a status message:

```
return redirect(action('TicketsController@edit', $ticket->slug))->with('status', 'The ticket '.$slug.' has been updated!');
```

Try to edit the ticket now and hit the **update button**!



Amazing! The ticket has been updated!

Delete a ticket

You've learned how to create, read and update a ticket. Next, you will learn how to delete it. By the end of this section, you'll have a nice **CRUD** application!

First step, let's open the **web.php** file and add a new route:

```
Route::post('/ticket/{slug?}/delete','TicketsController@destroy');
```

When we send a POST request to this route, Laravel will take the slug and execute the **TicketsController's destroy** action.

> **Note:** You may use the **GET** method here.

Open **TicketsController** and update the destroy action:

```
public function destroy($slug)
{
    $ticket = Ticket::whereSlug($slug)->firstOrFail();
    $ticket->delete();
    return redirect('/tickets')->with('status', 'The ticket '.$slug.' has been deleted!');
}
```

We find the ticket using the provided slug. After that, we use **$ticket->delete()** method to delete the ticket.

As always, we then redirect users to the **all tickets** page. Let's update the **index.blade.php** to display the status message:

Find:

```
<div class="card-body mt-2">
```

Add below:

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

Finally, in order to remove the ticket, all we need to do is create a form to submit a delete request.

Open **show.blade.php** and find:

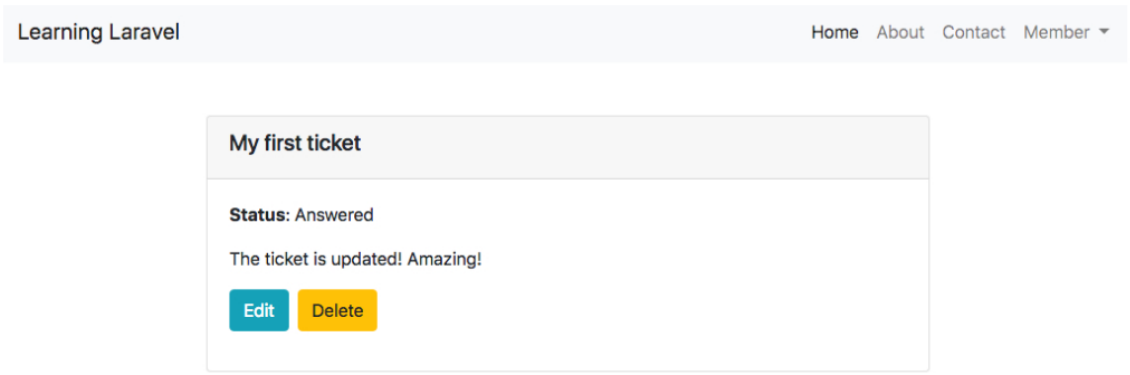
```
<a href="{{ action('TicketsController@edit', $ticket->slug) }}" class="btn btn-info">Edit</a>
```

```
<a href="#" class="btn btn-info">Delete</a>
```

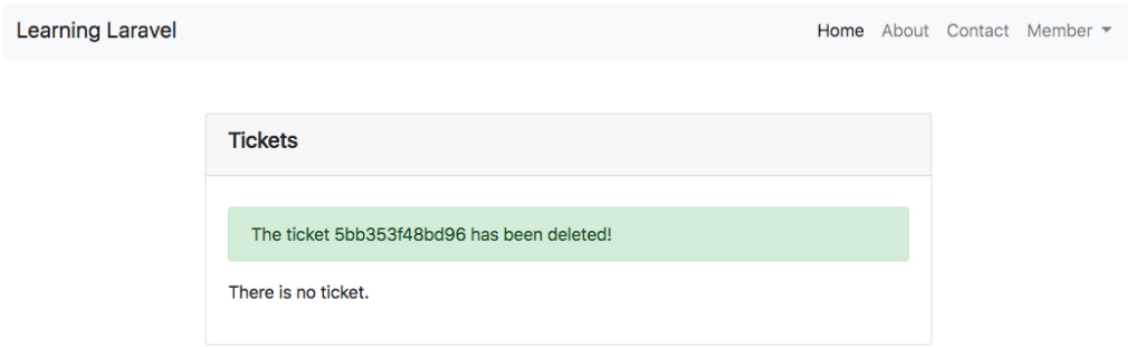
Update to:

```
<a href="{{ action('TicketsController@edit', $ticket->slug) }}" class="btn btn-info float-left mr-2">Edit</a>
<form method="post" action="{{ action('TicketsController@destroy', $ticket->slug) }}" class="float-left">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
    <div>
        <button type="submit" class="btn btn-warning">Delete</button>
    </div>
</form>
<div class="clearfix"></div>
```

The code above creates a nice delete form for you. When you view a ticket, you should see a different delete button:



Now, click the **delete button**, you should be able to remove the ticket!



You've just deleted a ticket!

Congratulations! You now have a **CRUD** (Create, Read, Update, Delete) application!

Sending an email

When users submit a ticket, we may want to receive an email to get notified. In this section, you will learn how to send emails using Laravel.

Laravel provides many methods to send emails. You may use a plain PHP method to send emails, or you may use some email service providers such as Mailgun, Sendinblue, Sendgrid, Mandrill, Amazon SES, etc.

To send emails on a production server, simply edit the **mail.php** configuration file, which is placed in the **config** directory.

Here is the file without comments:

```
return [

    'driver' => env('MAIL_DRIVER', 'smtp'),

    'host' => env('MAIL_HOST', 'smtp.mailgun.org'),

    'port' => env('MAIL_PORT', 587),

    'from' => [
    'address' => env('MAIL_FROM_ADDRESS', 'hello@example.com'),
    'name' => env('MAIL_FROM_NAME', 'Example'),
     ],

    'encryption' => env('MAIL_ENCRYPTION', 'tls'),

    'username' => env('MAIL_USERNAME'),

    'password' => env('MAIL_PASSWORD'),

    'sendmail' => '/usr/sbin/sendmail -bs',
```

```
    'markdown' => [
    'theme' => 'default',

    'paths' => [
        resource_path('views/vendor/mail'),
    ],
],
];
```

To send emails on a local development server (Homestead), simply edit the **.env** file.

```
MAIL_DRIVER=mail
MAIL_HOST=mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

As usual, you may learn how to use Mailgun, Mandrill and SES drivers at the official docs:

https://laravel.com/docs/master/mail

Because working on Homestead, we will learn how to send emails on Homestead using **Gmail** and Sendinblue for FREE!

Sending emails using Gmail

> **Note:** Even though we can use Gmail, it's recommended to use a transactional email service (Sendinblue, Mandrill, etc.) to send emails. You may **SKIP THIS SECTION**.

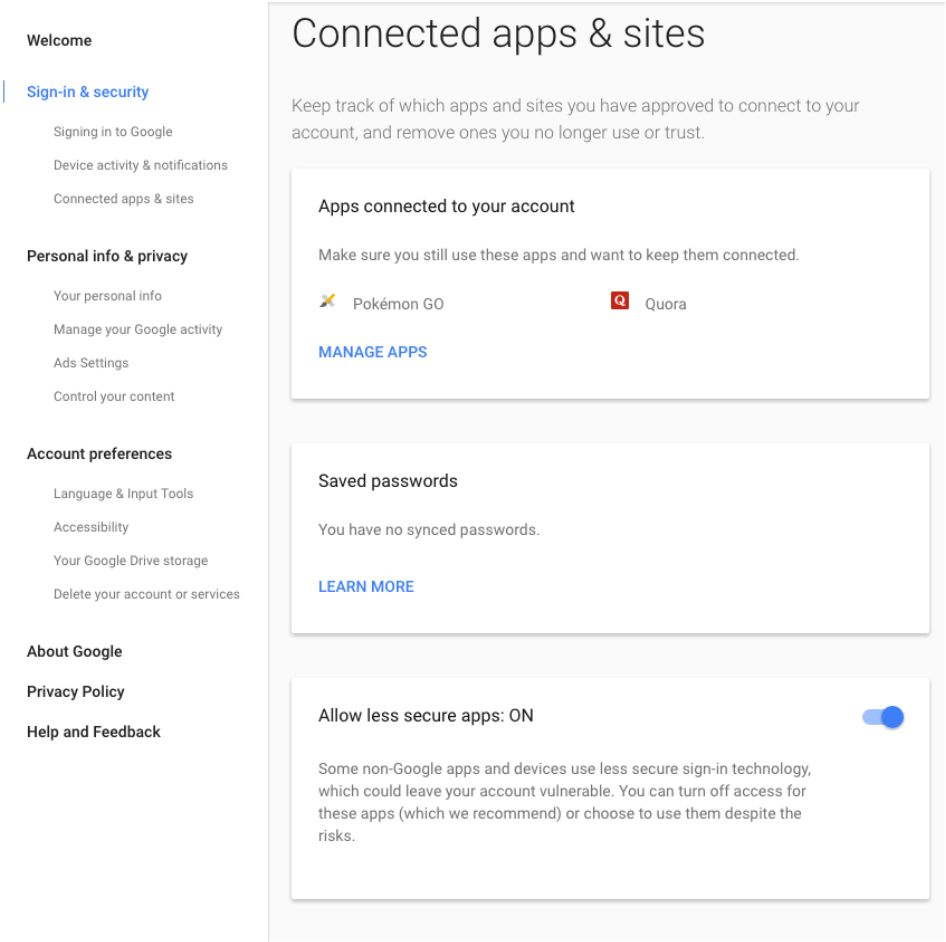
If you have a Gmail account, it's very easy to send emails using Laravel 5!

First, go to:

https://myaccount.google.com/security#connectedapps

Take a look at the **Sign-in & security -> Connected apps & sites -> Allow less secure apps** settings.

You must turn the option "Allow less secure apps" **ON**.



Once complete, edit the **.env** file:

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=587
```

```
MAIL_USERNAME=yourEmail
MAIL_PASSWORD=yourPassword
MAIL_ENCRYPTION=tls
```

Well done! You're now ready to send emails using Gmail!

If you get this error when sending email: **"Failed to authenticate on SMTP server with username "youremail@gmail.com" using 3 possible authenticators"**

You may try one of these methods:

- Go to https://accounts.google.com/UnlockCaptcha, click **continue** and unlock your account for access through other media/sites.
- Using a double quote password: **"your password"**
- Try to use only your Gmail username: yourGmailUsername

Sending emails using Sendinblue or other mail service providers

Go to **Sendinblue**, register a new account:

https://www.sendinblue.com/?ae=484

> **Note:** If you don't want to use Sendinblue, you can try other email service providers, such as Sendgrid.

When your account is activated, edit the **.env** file:

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp-relay.sendinblue.com
MAIL_PORT=587
MAIL_USERNAME=yourSendinblueUsername
MAIL_PASSWORD=yourPassword
```

Good job! You're now ready to send emails using Sendinblue!

Sending a test email

To send a test email, open **web.php** file and add this route:

```
Route::get('sendemail', function () {

    $data = array(
        'name' => "Learning Laravel",
    );

    Mail::send('emails.welcome', $data, function ($message) {

        $message->from('yourEmail@domain.com', 'Learning Laravel');

        $message->to('yourEmail@domain.com')->subject('Learning Laravel test email');

    });

    return "Your email has been sent successfully";

});
```

As you see, we use the **send** method on the **Mail** facade. There are three arguments:

1. The name of the view that we use to send emails.
2. An array of data that we want to pass to the email.
3. A closure that we can use to customize our email subjects, sender, recipients, etc.

When you visit http://homestead.test/sendemail, Laravel will try to send an email. If the email is sent successfully, Laravel will display a message.

> **Note:** Be sure to replace **yourEmail@domain.com** with your **real email address**.

Finally, we don't have the **welcome.blade.php** view yet, let's create it and put it in the **emails** directory.

views/emails/welcome.blade.php

```
<!DOCTYPE html>
<html lang="en-US">
<head>
    <meta charset="utf-8">
</head>
<body>
<h2>Learning Laravel!</h2>

<div>
    Welcome to {{ $name }} website!
</div>

```

```
</body>
</html>
```

Because we've passed an array containing the **$name** key in the above route, we could display the **name** within this **welcome view** using:

```
{{ $name }}
```

or

```
<?php echo $name ?>
```

Done! Now go to http://homestead.test/sendemail, you should see:

```
Your email has been sent successfully
```

Check your inbox, you should receive a new email!

Feel free to customize your email address, recipients, subjects, etc.

Sending an email when there is a new ticket

Now that we have set up everything, let's send an email when users create a new ticket!

Open **TicketsController.php** and update the **store** action.

Find:

```
return redirect('/contact')->with('status', 'Your ticket has been created! Its unique id is: '.$slug);
```

Add above:

```
$data = array(
    'ticket' => $slug,
);

Mail::send('emails.ticket', $data, function ($message) {
    $message->from('yourEmail@domain.com', 'Learning Laravel');

    $message->to('yourEmail@domain.com')->subject('There is a new ticket!');
});
```

> **Note:** Be sure to replace **yourEmail@domain.com** with your **real email address**. Don't add the code above if you don't want to send an email.

And don't forget to tell Laravel that you want to use the **Mail** facade here:

Find:

```
class TicketsController extends Controller
```

Add above:

```
use Illuminate\Support\Facades\Mail;
```

As you may notice, we don't have the **emails/ticket.blade.php** view yet. Let's create it!

```
<!DOCTYPE html>
<html lang="en-US">
<head>
    <meta charset="utf-8">
</head>
<body>
<h2>Learning Laravel!</h2>

<div>
    You have a new ticket. The ticket id is {{ $ticket }}!
</div>

</body>
</html>
```

It's time to **create a new ticket**!

If everything works out well, you should see a new email in your inbox! Here is the email's content:

Learning Laravel!

You have a new ticket. The ticket id is 558c1f6ead809!

Laravel 5.6+ allows us to send emails using Markdown. If you love Markdown, you may read the documentation to learn how to use this feature:

https://laravel.com/docs/master/mail#markdown-mailables

Reply to a ticket

Welcome to the last section!

In this section, we will learn how to create a form for users to post a reply.

Create a new comments table

First, we need a table to store all the ticket responses. I name this table **comments**.

Let's run this migration command:

```
php artisan make:migration create_comments_table
```

Then, open **yourTimestamps_create_comments_table.php**, use **Schema** to create some columns:

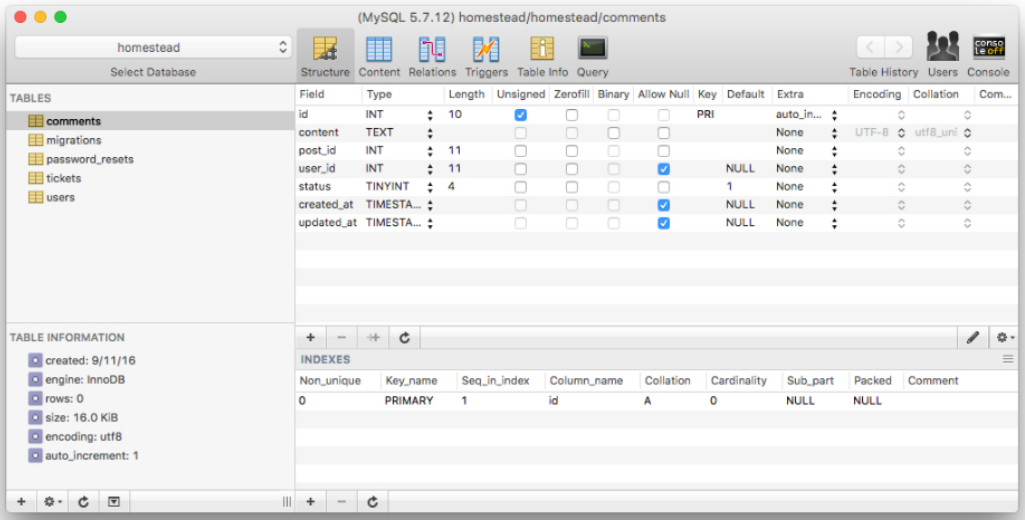
```
<?php

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

class CreateCommentsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('comments', function (Blueprint $table) {
            $table->increments('id');
            $table->text('content');
            $table->integer('post_id');
            $table->integer('user_id')->nullable();
            $table->tinyInteger('status')->default(1);
            $table->timestamps();
        });
    }

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

```
}
```

You should know how to read this file by now. Let's run the **migrate** command to create the comments table and its columns:

```
php artisan migrate
```

Great! Check your database now to make sure that you have created the **comments** table.



Introducing Eloquent: Relationships

In Laravel, you can maintain a relationship between tables easily using Eloquent. Here are the relationships that Eloquent supports:

- One to One
- One to Many
- Many to Many
- Has Many Through
- Polymorphic Relations
- Many To Many Polymorphic Relations

What is a relationship?

Usually, tables are related to each other. For instance, our tickets may have many comments (ticket responses). That is **One to Many** relationship.

Once we've defined a **One to Many** relationship between tickets and comments table, we can easily access and list all comments or any related records.

Learn more about Eloquent relationships here:

https://laravel.com/docs/master/eloquent-relationships

Create a new Comment model

As you know, we need a Comment model! Run this command to create it:

```
php artisan make:model Comment
```

Once completed, open it and add this relationship:

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

In addition, we may want to make all columns **mass assignable** except the id column:

```
protected $guarded = ['id'];
```

Instead of using the **$fillable** property, we use the **$guarded** property here.

You now have:

app/Comment.php

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    protected $guarded = ['id'];

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

By doing this, we let Eloquent know that this **Comment model** belongs to the **Ticket model**.

Next, open the **Ticket model** and add:

```php
public function comments()
{
    return $this->hasMany('App\Comment', 'post_id');
}
```

You now have:

app/Ticket.php

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Ticket extends Model
{
    protected $fillable = ['title', 'content', 'slug', 'status', 'user_id'];

    public function comments()
    {
        return $this->hasMany('App\Comment', 'post_id');
    }
}
```

As you may have guessed, we tell that the Ticket model has many comments and Eloquent can use the **post_id** (ticket id) to find all related comments.

That's it! You've defined a **One to Many** relationship between two tables.

Create a new comments controller

With the relation defined, we will create a new **CommentsController** to handle form submissions and save comments to our database.

Before doing that, let's modify our **web.php** first:

```php
Route::post('/comment', 'CommentsController@newComment');
```

When we send a **POST** request to this route, Laravel will execute the **CommentsController's newComment** action.

It's time to run this command to generate our controller:

```
php artisan make:controller CommentsController
```

Open the new **CommentsController** and add a new action:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests\CommentFormRequest;
use App\Comment;

class CommentsController extends Controller
{
    public function newComment(CommentFormRequest $request)
    {
```

```
            $comment = new Comment(array(
                'post_id' => $request->get('post_id'),
                'content' => $request->get('content')
            ));

            $comment->save();

            return redirect()->back()->with('status', 'Your comment has been created!');
        }
    }
```

Don't forget to add:

```
use App\Http\Requests\CommentFormRequest;
use App\Comment;
```

Here is a little tip, you can use **redirect()->back()** to redirect users back to the previous page!

As you see, we still use **Request** here for the validation.

Create a new CommentFormRequest

We don't have the **CommentFormRequest** yet, so let's create it as well:

```
php artisan make:request CommentFormRequest
```

```
vagrant@homestead:~/Code/Laravel$ php artisan make:migration create_comments_table --create=comments
Created Migration: 2015_06_26_145222_create_comments_table
vagrant@homestead:~/Code/Laravel$ php artisan migrate
Migrated: 2015_06_26_145222_create_comments_table
vagrant@homestead:~/Code/Laravel$ php artisan make:model Comment
Model created successfully.
vagrant@homestead:~/Code/Laravel$ php artisan make:controller CommentsController
Controller created successfully.
vagrant@homestead:~/Code/Laravel$ php artisan make:request CommentFormRequest
Request created successfully.
vagrant@homestead:~/Code/Laravel$
```

Now, define our rules:

app/Requests/CommentFormRequest.php

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class CommentFormRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'content'=> 'required|min:3',
        ];
    }
}
```

> **Note:** Check the **authorize()** function, don't forget to update it to **return true**

Create a new reply form

Good job! Now open the **tickets.show** view and find:

```
        </form>
        <div class="clearfix"></div>
    </div>
</div>
```

Add this form below:

```
<div class="card mt-3">
    <form method="post" action="/comment">

        @foreach($errors->all() as $error)
            <p class="alert alert-danger">{{ $error }}</p>
        @endforeach

        @if(session('status'))
            <div class="alert alert-success">
                {{ session('status') }}
            </div>
        @endif

        <input type="hidden" name="_token" value="{{ csrf_token() }}">
        <input type="hidden" name="post_id" value="{{ $ticket->id }}">

        <fieldset>
            <legend class="ml-3">Reply</legend>
            <div class="form-group">
                <div class="col-lg-12">
                    <textarea class="form-control" rows="3" id="content" name="content"></textarea>
                </div>
            </div>

            <div class="form-group">
                <div class="col-lg-10 col-lg-offset-2">
                    <button type="reset" class="btn btn-default">Cancel</button>
                    <button type="submit" class="btn btn-primary">Post</button>
                </div>
            </div>
        </fieldset>
    </form>
</div>
```

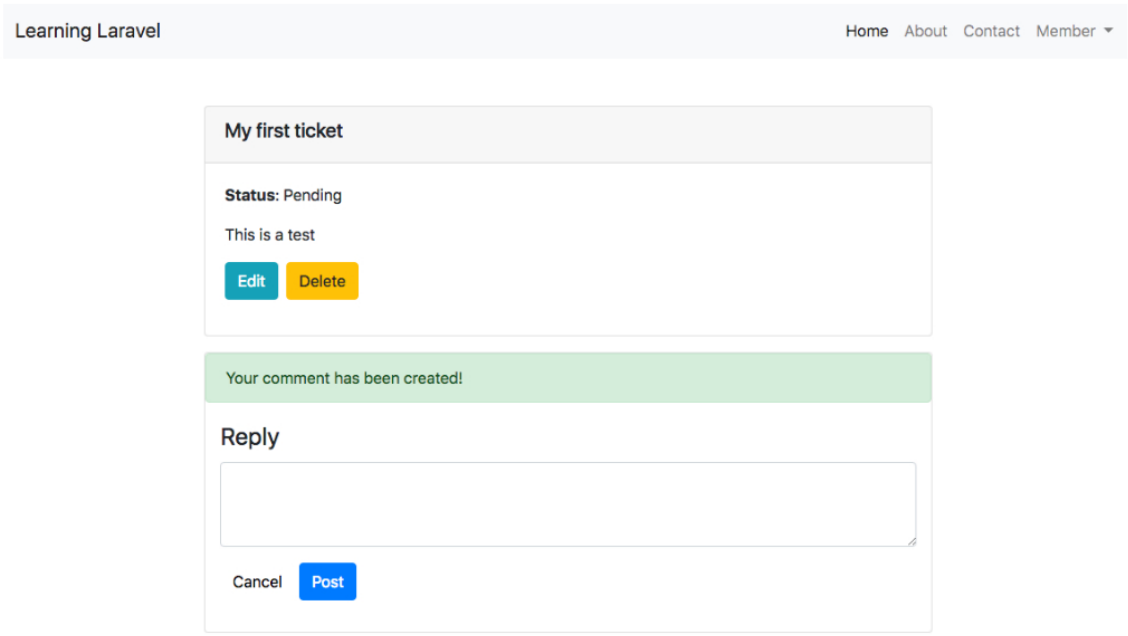
Here is the new **tickets.show** view:

```
@extends('master')
@section('title', 'View a ticket')
@section('content')

    <div class="container col-md-8 col-md-offset-2 mt-5">
        <div class="card">
            <div class="card-header ">
                <h5 class="float-left">{{ $ticket->title }}</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body">
                <p> <strong>Status</strong>: {{ $ticket->status ? 'Pending' : 'Answered' }}</p>
                <p> {{ $ticket->content }} </p>
                <a href="{{ action('TicketsController@edit', $ticket->slug) }}" class="btn btn-info float-left mr-2">Edit</a>
                <form method="post" action="{{ action('TicketsController@destroy', $ticket->slug) }}" class="float-left">
                    <input type="hidden" name="_token" value="{{ csrf_token() }}">
                    <div>
                        <button type="submit" class="btn btn-warning">Delete</button>
                    </div>
                </form>
                <div class="clearfix"></div>
            </div>
        </div>
        <div class="card mt-3">
            <form method="post" action="/comment">

                @foreach($errors->all() as $error)
                    <p class="alert alert-danger">{{ $error }}</p>
                @endforeach

                @if(session('status'))
                    <div class="alert alert-success">
                        {{ session('status') }}
                    </div>
                @endif

                <input type="hidden" name="_token" value="{{ csrf_token() }}">
                <input type="hidden" name="post_id" value="{{ $ticket->id }}">

                <fieldset>
                    <legend class="ml-3">Reply</legend>
                    <div class="form-group">
                        <div class="col-lg-12">
                            <textarea class="form-control" rows="3" id="content" name="content"></textarea>
                        </div>
                    </div>

                    <div class="form-group">
                        <div class="col-lg-10 col-lg-offset-2">
                            <button type="reset" class="btn btn-default">Cancel</button>
                            <button type="submit" class="btn btn-primary">Post</button>
                        </div>
                    </div>
                </fieldset>
            </form>
        </div>

    </div>
```

```
@endsection
```

This form is very similar to the **create ticket form**, we just need to add a new **hidden input** to submit the **ticket id** (post_id) as well.

When you have the form, let's try to reply to a ticket.



Yes! You've created a new response!

Display the comments

One last step, we're going to modify the **show action** of our **TicketsController** to list all ticket's comments and pass them to the view.

Open **TicketsController**. The changes in the **show action** are listed as follows:

```
public function show($slug)
{
    $ticket = Ticket::whereSlug($slug)->firstOrFail();
    $comments = $ticket->comments()->get();
    return view('tickets.show', compact('ticket', 'comments'));
}
```

As you may see in the code above, we just need to use this line to list all comments:

```
$comments = $ticket->comments()->get();
```

Amazing, right? You don't even use a single SQL code!

Now, open the **show.blade.php** view, and add this code above the reply form:

```
@foreach($comments as $comment)
    <div class="card mt-3">
        <div class="card-body">
            {{ $comment->content }}
        </div>
    </div>
@endforeach
```

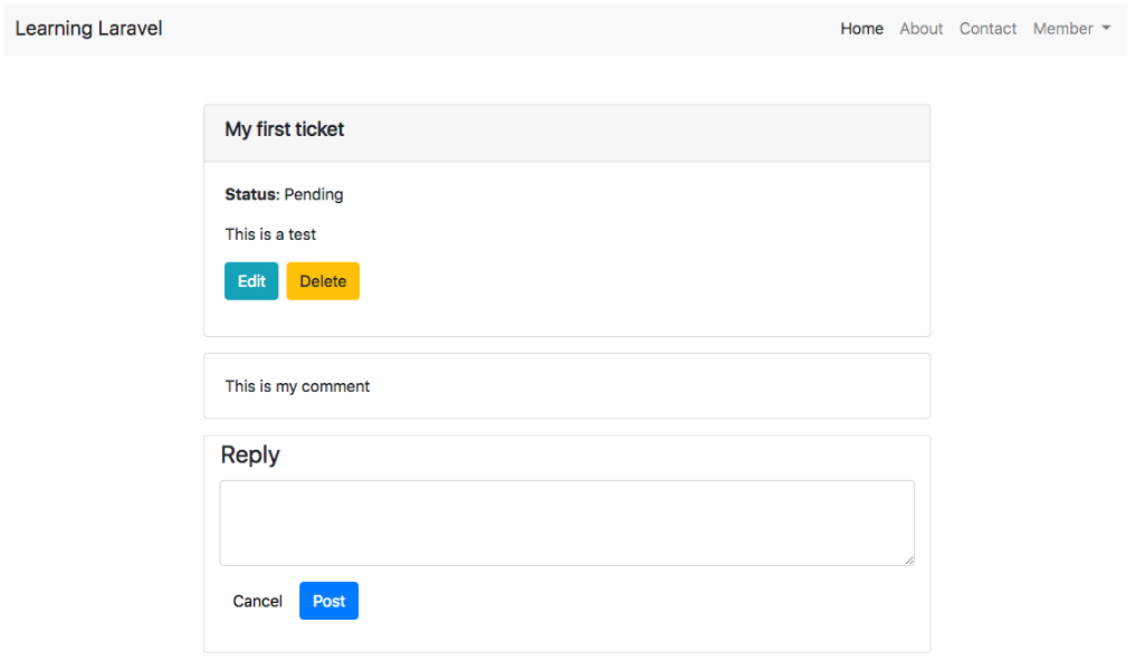
Here is the new **show.blade.php**:

views/tickets/show.blade.php

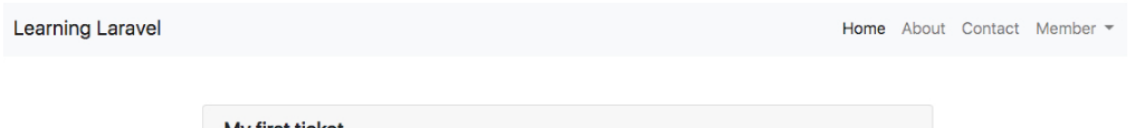
```
@extends('master')
@section('title', 'View a ticket')
@section('content')

    <div class="container col-md-8 col-md-offset-2 mt-5">
        <div class="card">
            <div class="card-header ">
                <h5 class="float-left">{{ $ticket->title }}</h5>
                <div class="clearfix"></div>
            </div>
            <div class="card-body">
                <p> <strong>Status</strong>: {{ $ticket->status ? 'Pending' : 'Answered' }}</p>
                <p> {{ $ticket->content }} </p>
                <a href="{{ action('TicketsController@edit', $ticket->slug) }}" class="btn btn-info float-left mr-2">Edit</a>
```

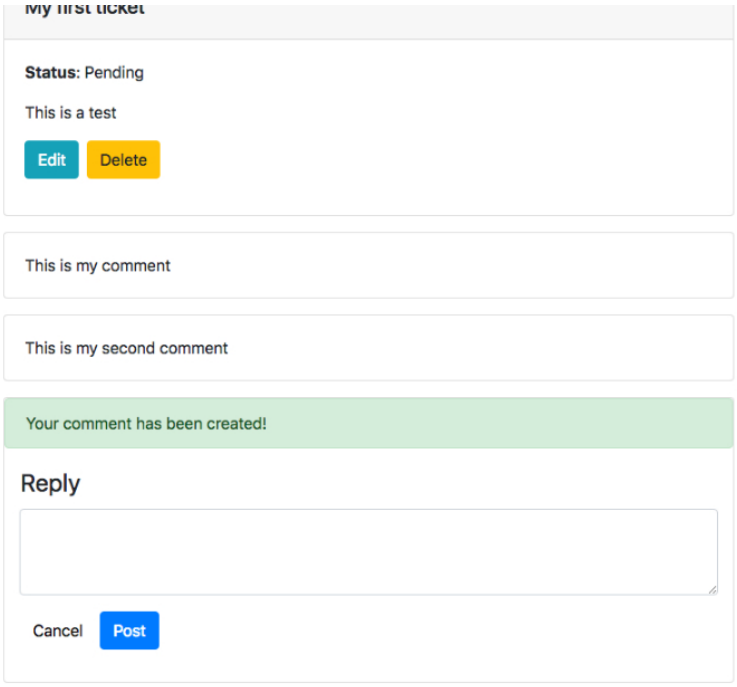
```
                    <form method="post" action="{{ action('TicketsController@destroy', $ticket->slug) }}" class="float-left">
                        <input type="hidden" name="_token" value="{{ csrf_token() }}">
                        <div>
                            <button type="submit" class="btn btn-warning">Delete</button>
                        </div>
                    </form>
                    <div class="clearfix"></div>
                </div>
            </div>
            @foreach($comments as $comment)
                <div class="card mt-3">
                    <div class="card-body">
                        {{ $comment->content }}
                    </div>
                </div>
            @endforeach

            <div class="card mt-3">
                <form method="post" action="/comment">

                    @foreach($errors->all() as $error)
                        <p class="alert alert-danger">{{ $error }}</p>
                    @endforeach

                    @if(session('status'))
                        <div class="alert alert-success">
                            {{ session('status') }}
                        </div>
                    @endif

                    <input type="hidden" name="_token" value="{{ csrf_token() }}">
                    <input type="hidden" name="post_id" value="{{ $ticket->id }}">

                    <fieldset>
                        <legend class="ml-3">Reply</legend>
                        <div class="form-group">
                            <div class="col-lg-12">
                                <textarea class="form-control" rows="3" id="content" name="content"></textarea>
                            </div>
                        </div>

                        <div class="form-group">
                            <div class="col-lg-10 col-lg-offset-2">
                                <button type="reset" class="btn btn-default">Cancel</button>
                                <button type="submit" class="btn btn-primary">Post</button>
                            </div>
                        </div>
                    </fieldset>
                </form>
            </div>

        </div>

    @endsection
```

Refresh your browser now!



To make sure that everything is working, reply again!





Congratulations! You now have a fully working support ticket system!

Chapter 3 Summary

In this chapter, you've gone through the different steps involved in creating a ticket support system. Even though the app is simple, it provides us many things to learn:

- You've known how to create databases.
- You've learned one of the most important Laravel features: migrations. Now you can create any database structures that you want.
- You've understood how to use Request to validate forms.
- If you want to use different packages, you've known how to install them.
- You've learned about Laravel's helper functions.
- Sending emails using Gmail and Sendinblue is easy, right?
- You've known how to define Eloquent Relationships and work with those relationships easily.

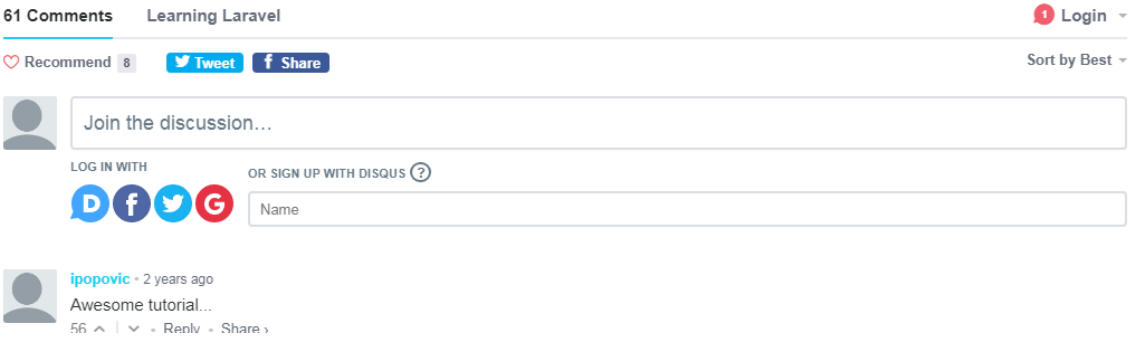
Basically, you may now be able to create a simple blog system. Feel free to build a different application or customize this application to meet your needs.

The next chapter is where all the fun begin! You will learn to create a complete blog application that has an admin control panel. You may use this application to write your blog posts or you may use it as a starter template for all your amazing applications.

Chapter 3 Source Code

You can view and download the source code at:

> Learning Laravel 5 Book: Chapter 3 Source Code



learninglaravel Mod ipopovic • 2 years ago
Good to hear that. Thank you :)
Reply • Share ›

Sabrina Markon • 8 months ago
This tutorial was brilliant and packs a lot of Laravel concepts into one useful application. Thank you so much!
29 Reply • Share ›

Chaturvedi Ravi Arvind • 7 months ago
all things are perfect.. but why I am not able to redirect to any other page using navbar from homestead.show/($slug?)
2 Reply • Share ›

Zoran Koprek Chaturvedi Ravi Arvind • 6 months ago
Hi, i used:
Example
{{ url('example') }}

in navigation to get correct url. I am building other application so the same problem came for me. Hope this is correct way.
1 Reply • Share ›

Chaturvedi Ravi Arvind • 7 months ago
awsome tutorial
1 Reply • Share ›

Anand ND • 2 years ago
wonderful tutorial , fantastic
1 Reply • Share ›

learninglaravel Mod Anand ND • 2 years ago
Thank you. Have fun!
Reply • Share ›

Carlos Frostte • 2 years ago
If error 1071 appears when migrating: https://laravel.com/docs/5.... (Index Lengths & MySQL / MariaDB)
1 Reply • Share ›

learninglaravel Mod Carlos Frostte • 2 years ago
A nice tip! Thank you.
Reply • Share ›

Mohammed Almosawi • 2 years ago
This is an awesome tutorial. I've learned a lot about Laravel.
Thank you very much.
1 Reply • Share ›

learninglaravel Mod Mohammed Almosawi • 2 years ago
You're welcome! Happy coding!
Reply • Share ›

Timur TimOne Timerkhaov • 24 days ago
Great job thanks
Reply • Share ›

Arnel Forbes • 4 months ago
see more
Reply • Share ›

learninglaravel Mod Arnel Forbes • 4 months ago
You forgot to put the CSRF token field there. :)
Reply • Share ›

Arnel Forbes learninglaravel • 4 months ago
@csrf

i do have this, but its still work, i just dont know what ive done wrong

Reply • Share ›

learninglaravel Mod ➚ Arnel Forbes • 4 months ago

I think it's related to CSRF. You can turn it off first by following these tips:

https://stackoverflow.com/q...

If it's working this time then you have to check your .env file. Also, make sure that you're using Homestead.

You can enable the CSRF feature again later.

Lastly, make sure to clear your browser cache.

- Ryan

Reply • Share ›

Arnel Forbes ➚ learninglaravel • 3 months ago

i am using windows and apache

Reply • Share ›

Hary Dhimas • 8 months ago

can u tell me why when i add this

use App\Http\Requests\TicketFormRequest
to my ticketcontroller and save it , sublime automatically delete it?

Reply • Share ›

Hary Dhimas ➚ Hary Dhimas • 8 months ago

solved
it seems that my sublime is smart enough, i use Request on the typehint method param, not TicketFormRequest

Reply • Share ›

Alexey Verhola • 8 months ago

Thanks for awesome chapter!!

Reply • Share ›

Justin O'Reilly • a year ago

Sweet!

Reply • Share ›

kifftfer • a year ago

What a great tutorial ... I got given a bunch of non-working, half-missing Laravel 4 code. Never having seen php before this guide has really helped me get into Laravel and php. I also want to thank you for the fast and helpful responses! I have bought the books and now look forward to trying to make sense of the disaster that I was handed over :-|.

It's easy to see why this is such a popular framework!

Also, I suppose this isn't etiquette but if anyone wants to potentially earn a few bucks and help me with my project, implementing a children's cancer database and front-end. Then please IM.

Reply • Share ›

Dan Niel • a year ago

Hi,... can you u teach me how to get user_id and store it intickets table?

Reply • Share ›

learninglaravel Mod ➚ Dan Niel • a year ago

Hi, you can use \Auth::user()->id to get the user id. For example:

$id = \Auth::user()->id;

Remember to import Auth.

After that, you can store it anywhere. (like saving the title)

Reply • Share ›

mohit99 • a year ago

This book is the only reliable up to date 100% accurate Laravel 5.5 learning resource available on Internet - great work

Reply • Share ›

learninglaravel Mod ➚ mohit99 • a year ago

Thank you :)

1 Reply • Share ›

Prakan Puvibunsuk • a year ago

In the last coding of show.blade.php line 33
<form class="form-horizontal" method="post" action="/comment">

The above coding did not work on my system (lubuntu 16.04) therefore I had to change code to the following
<form class="form-horizontal" method="post" action="{!! action('CommentsController@newComment') !!}">

Another minor problem is ticket and comment are not relate in database level
therefore comments data still exist even the ticket is deleted.

I'd like to know if Eloquent ORM support cascade delete, if so how to code.

Anyway this is one of the best tutorial.

Thanks,

Reply • Share ›

GMFreelancer • 2 years ago

Is there any package to make comments easier?

Reply • Share ›

gbenga wale • 2 years ago

I get this error message while trying to send email (Class 'Illuminate\Support\Facade\Mail' not found).
in the controller, I have included 'use Illuminate\Support\Facade\Mail;' in the begining of the ticket controller and in the store function, I have this

'Mail::send('welcome_email', $data, function ($message) {
$message->from('walegbenga807@gmail.com', 'Coa Blog');

$message->to('nigeriawonderboy@gmail.com')->subject('There is a new post!');
});
return redirect('/')->with('status', 'ticket created');'

Reply • Share ›

learninglaravel Mod ➚ gbenga wale • 2 years ago

Try to use:
use Mail;

Reply • Share ›

gbenga wale ➚ gbenga wale • 2 years ago

now working, I added s to the facade like 'use Illuminate\Support\Facades\Mail;' but it says

'FatalErrorException in HandleExceptions.php line 59:
Maximum execution time of 30 seconds exceeded'

Reply • Share ›

learninglaravel Mod ➚ gbenga wale • 2 years ago

The Maximum execution time of 30 seconds exceeded error is not related to Laravel but rather your PHP configuration. Are you using Homestead or WAMP/MAMP?
You have to edit the php.ini, and increase the max_execution_time.

For example:

max_execution_time = 120;

Reply • Share ›

gbenga wale ➚ learninglaravel • 2 years ago

ok. I will try that, Am using XAMPP

Reply • Share ›

Kevin Jones • 2 years ago

Again, very nice tutorial. I still plan to purchase the ebook soon. Here's a question about architecture. How can I add jquery (or basic javascript) to be applied to just a particular blade? I added a "delete" confirmation (modal and javascript), but had to include the javascript in the master.blade.php (below the jquery library). Putting the javascript (which includes some jquery) in the show.blade.php doesn't work, because the jquery library is loaded AFTER (per the master.blade.php) the show.blade.php's contents are loaded in the DOM. Thoughts? Ideas?

Reply • Share ›

learninglaravel Mod ➚ Kevin Jones • 2 years ago

You may create another Blade file called scripts, and then include it where you want. Or you may use Stacks to inject the script. Stacks is a new feature of Blade (only available in new versions of Laravel).

You may read about Stacks here:

https://laravel.com/docs/5....

Reply • Share ›

Kevin Jones • 2 years ago

In the create.blade.php file in view/tickets, the "Cancel" button does not appear to have any functionality attached. I'd like the "Cancel" button to return the user "back" one level (e.g. redirect()->back()), but the "Request" logic preempts getting to the controller code unless I randomly populate the form first. What is the best way to implement the "Cancel" as a true "go back button" when also employing the Requests feature?

Thank you in advance. Nice tutorial by the way. I want to work through the some of the procedure to this example (and probably the blog as well), then I am certainly going to purchase this ebook!

Reply • Share ›

learninglaravel Mod ➚ Kevin Jones • 2 years ago

Laravel has a helper called url, which you can use to make the Cancel button work.

{{ url()->previous() }}

Reply • Share ›

gbenga wale • 2 years ago

migrations table created and users table but tickets, comments and other
tables are not created. This is the error it gave and this is what I
ran with composer (php artisan make:migration create_tickets_table --create=tickets). And this is the error

$ php artisan migrate
Migration table created successfully.

[Illuminate\Database\QueryException]
SQLSTATE[42000]: Syntax error or access violation: 1071 Specified key was t
oo long; max key length is 767 bytes (SQL: alter table `users` add unique `
users_email_unique`(`email`))

[PDOException]
SQLSTATE[42000]: Syntax error or acces

How can I go about it

Reply • Share ›

learninglaravel Mod ➔ gbenga wale • 2 years ago

In new Laravel versions, the database has been changed. Laravel uses the utf8mb4 character set by default.
If you're using an old MySQL version, you might encounter that error.

To fix that, you need to edit the AppServiceProvider.php file, and set a default string length inside the boot method :

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

public function boot()
{
    Schema::defaultStringLength(191);
}
```

The book will be updated soon. Thank you.

Reply • Share ›

Andrew • 2 years ago

{!! trans('main.subtitle') !!} this is on the home page, where is it sourced from

Reply • Share ›

learninglaravel Mod ➔ Andrew • 2 years ago

It's from the main.php file. You'll learn about it in the next chapter.

Reply • Share ›

Andrew • 2 years ago

if I wanted to assign a role when new person registers, how would I go about this

Reply • Share ›

learninglaravel Mod ➔ Andrew • 2 years ago

You can do it in the Auth\RegisterController (the one that you create in the next chapter)

Reply • Share ›

Realtebo ✓verified • 2 years ago

What's the differences from {{ }} and {!!! !!!}?

Reply • Share ›

learninglaravel Mod ➔ Realtebo ✓verified • 2 years ago

If you type {{ <tagname>test</tagname> }}, the output'll be:

```
<tagname>test</tagname>
```

If you type {!! <tagname>test</tagname> !!}, the output'll be:

test

Reply • Share ›

Chris Eccles • 2 years ago

Shouldn't the route for posting comments be "Route::post('/ticket/comment', 'CommentsController@newComment');"

Great tutorial, easy to follow and very informative.

Reply • Share ›

learninglaravel Mod ➔ Chris Eccles • 2 years ago

Hi. You can choose any route that you like.

Thank you :)

Reply • Share ›

Abdulaziz Alzaabi • 2 years ago

Thanks for this great tutorial.. is there a way to download the project.. or at least blade file such as master

Reply • Share ›

learninglaravel Mod ➔ Abdulaziz Alzaabi • 2 years ago

You can download the source code here:

https://github.com/Learning...

Reply • Share ›

Load more comments

Subscribe Add Disqus to your site Disqus' Privacy Policy DISQUS

Adobe Creative Cloud for Teams starting at $29.99 per month.ads via Carbon