Composer.json – Holds the projects assets.

Php artisan serve – Starts the projects on a server and give you a link to access the same on a browser.

Resources/views – This is where files that have to do with HTML are stored.

**Vscode Shorcuts**

If you place your mouse pointer on a named entity, say a class or method etc, then hold “Ctrl” and click it, VSCode will take you to the file and position where it is declared.

**Vscode Extensions**

Laravel blade snippets

Php namespace resolver

Php intelliphense

**Browser Extensions**

Clockwork – To install it into your project, run “composer require isgoingd/clockwork”

**Process To Install A Laravel Project**

Install Composer (globally).

Install Node.js which comes with npm.

Open the terminal.

Navigate to where you want to create the project.

Install the laravel installer using the command:

*composer global require laravel/installer*

Create the laravel project within the directory using:

*laravel new project name*

Navigate into the newly created project directory.

Open it in a code editor, say vscode using:  
 *code .*

Open a server and display the website using:

*php artisan serve*

**Routes**

To specify what happens when a url on the browser is entered, open the *“web.php”* file in the *“Routes”* folder within the main directory.

For debugging purposes, you can use:

*“dd(some variable/string etc)”* - This echoes out the content passed to it, and stops all other execution.

*“ddd(some variable/string etc)”* – This echoes out more information

It is possible to pass data as well through routes which can then be accessed from the views. For example:

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

    return view('listings', [

        'heading' => 'Latest Listings',

        'listings' => [

            [

                'id'=>'1',

                'title' => 'Listing One',

                'description' => 'Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry\'s standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged.'

            ],

            [

                'id'=>'2',

                'title' => 'Listing Two',

                'description' => 'Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry\'s standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged.'

            ]

        ]

    ]);

});

To access the same, from the “listings”(.php) view:

<h1>{{$heading}}</h1>

@unless(count($listings) == 0)

    @foreach($listings as $listing)

        <h2>{{$listing['title']}}</h2>

        <p>{{$listing['description']}}</p>

    @endforeach

    @else

        <p>No Listings</p>

@endunless

**APIs**

Return Json files.

Created by create Routes (as above) in the api file within the resources directory which return items wrapped in a “response” method. E.g.

Route::get(‘/posts’, function() {

return response()->json([‘posts’ => [

[‘title’ => ‘Post One’]

]]);

});

To access the same navigate to the routes/api/… on the urls. E.g. <http://127.0.0.1:8000/api/posts>.

**Blade Directives**

These are used within views.

@php

@endphp

@foreach…:

@endforeach

@if

@else

@endif

@unless

@else

@endunless

{{variable}}

@yield(‘content’)

@extends(‘layout’)

@section(‘content’)

@endsection

**Models**

Located within the “app/Models” directory.

You can create a new model file in two ways:

* You can use the *“php artisan make:model Abcd”* command
* You can a create file manually and name it as “Abcd.php” - A one word convention.

**Database**

To configure the database to use, navigate to the “config/database.php” file.

The default database software is MySQL.

To change this say to “sqlite”, in the “config/database.php” file, change

'default' => env('DB\_CONNECTION', 'mysql'),

to

'default' => env('DB\_CONNECTION', sqlite),

For sqlite, you will also need to:

* create a file called “database.sqlite” within the “database” directory.
* Within the “.env” file, change:

DB\_CONNECTION=mysql

DB\_HOST=127.0.0.1

DB\_PORT=3306

DB\_DATABASE= database.sqlite

# DB\_USERNAME=root

# DB\_PASSWORD=

To work with the database, you can use:

* MySQL workbench – create a new connection, use the appropriate username and password (entered in the “store in vault” section when first creating the connection).
* phpMyAdmin – Ensure you have your XAMPP running, and Apache as well as MySQL running.
* The Terminal use the command: *mysql -u daniel –p*, for example, to log into your user and start working with the database.

**Migrations**

Located within the “database/migrations” folder.

They contain instructions that involve creating and dropping database tables but without using sql.

Each migration files contains a class that has two public methods:

* one that creates/alters a table *up()*
* another that deletes/drops the table - *down()*

Custom migration files can be created using the command:

*php artisan make:migration create\_listings\_table* // An example

To commit all the migrations to a database, use the command:

*php artisan migrate*

To refresh a database, that has data populated into it, to what it was before the population (preserving the schema), use the command:

*php artisan migrate:refresh*

To reset a database (as above) and then repopulate it with seeded data, use the command:

*php artisan migrate:refresh --seed*

**Seeders**

It is possible to create dummy data and populate it into the database.

For example to populate the users table (default), a seeder file “database/seeders/DatabaseSeeder.php” will be used. This will also invoke a UserFactory file that contains the logic for creating the dummy data. This file is the “database/factories/UserFactory” file.

To do the actual population use the command:

*php artisan db:seed*

It is also possible to populate data manually (without using the default faker library). Here you create your own method with the same “database/seeders/DatabaseSeeder.php” file such as within the same run() method as the User seeder:

\App\Models\Listing::create([

            'title' => 'Laravel Senior Developer',

            'tags' => 'laravel, javascript',

            'company' => 'Acme Corp',

            'location' => 'Boston, MA',

            'email' => 'email1@email.com',

            'website' => 'https://www.acme.com',

            'description' => 'Lorem ipsum dolor sit amet consectetur adipisicing elit. Ipsam minima et illo reprehenderit quas possimus voluptas repudiandae cum expedita, eveniet aliquid, quam illum quaerat consequatur! Expedita ab consectetur tenetur delensiti?'

        ]);

        Listing::create([

            'title' => 'Full-Stack Engineer',

            'tags' => 'laravel, backend ,api',

            'company' => 'Stark Industries',

            'location' => 'New York, NY',

            'email' => 'email2@email.com',

            'website' => 'https://www.starkindustries.com',

            'description' => 'Lorem ipsum dolor sit amet consectetur adipisicing elit. Ipsam minima et illo reprehenderit quas possimus voluptas repudiandae cum expedita, eveniet aliquid, quam illum quaerat consequatur! Expedita ab consectetur tenetur delensiti?'

          ]);

**Factories**

To create your own factory file for a custom table in the database use the command:

*php artisan make:factory ListingFactory // For example, creates a file to populate a database table called Listing*

**Components**

When you have html that is being reused a lot, it wise to make it a component that can then be called every time it is need using a single line of code, rather than repeating it over and over.

To do this:

* Take that repeating section of code and move it to another file say, *“resources/views/components/listing-card.blade.php”*
* If it requires any resources to be passed to it, append a *“@props”* directive the top of the created file.
* Wrap the code block with an “x-” tag say *“x-card class="p-6">…</x-card>”*
* In the calling file, where you want the component to be rendered, append a line of code that references the component file, for example:

<x-listing-card :listing="$listing" />

Here, the component file is called “*listing-card.blade.php*” and the same passed with an argument called “listing” which will be captured by the “@props” directive within the same component file.

An example of the above is:

The component:

{{--To get the listing from the listings.blade.php file--}}

@props(['listing'])

<x-card class="p-6">

    <div class="flex">

        <img class="hidden w-48 mr-6 md:block" src="{{asset('images/no-image.png')}}"  {{-- Recommended for public assets --}} alt=""/>

.

.

.

        </div>

    </div>

</x-card>

The calling file:

@unless(count($listings) == 0)

    @foreach($listings as $listing)

        {{--The colon is to bind the listing to the prop accessed from listing-card.blade.php--}}

<x-listing-card :listing="$listing" />

@endforeach

    @else

        <p>No Listings</p>

@endunless

Sometimes the component is a larger block of code (like a template) to which some other code is passed to it. In this case, use the “{{$show}}” to specify where the other code with be inserted.

The file with the code being passed then has to be wrapped in “x-” tags. For example:

<main>

      {{--View Content Here--}}

      {{$slot}}

<main>

The component file called “layout.blade.php”

<x-layout>

<div

.

.

.

</div>

</x-layout>

The calling file

This is similar to using “@yield(‘content’)”, “@extends(‘layout’)”, “@section(‘content’)” and “@endsection” but cleaner/ less verbose.

**Controllers**

Usually we have controllers for all of the resources.

To create a controller use the command:

*php artisan make:controller ListingController*

This will create a file *“app/Http/Controllers/ListingController.php”*

**Scope Filters**

This are used to filter the results displayed on a web page based on some parameters passed, for example, to filter the job listings returned that are tagged as “vue” listings from all the listings.

Procedure:

* Create a public function called “scopeFilter” in the model file that takes as parameters the query and the filter(as an array)

class Listing extends Model

{

    use HasFactory;

    public function scopeFilter($query, array $filters) {

        dd($filters['tag']);

    }

}

* Within the controller, in the appropriate function (index, show etc) append a “filter” function to the return statement that takes the request and the property (wrapped in [ ]) as a parameter. E.g.:

class ListingContoller extends Controller

{

    // To Show All The Listings

    public function index() {

        return view('listings.index', [

            // 'listings' => Listing::latest()->get()

            // 'listings' => Listing::all()

            'listings' => Listing::latest()->filter(request(['tag']))->get()

        ]);

    }

}

* Within the model, inside the “scopeFilter” function:
  + Check if there is a tag. If There is none, do nothing. If the is,
  + Write a query, an SQL “like” query abstraction as:

public function scopeFilter($query, array $filters) {

        if($filters['tag'] ?? null) {

            $query->where('tags', 'like', "%" . request('tag') . "%");

        }

    }

For multiple filters, modify the “filter(request())” in the controller, and add another “if” function that checks the parameter and to it “where” to the query (literally an “orWhere”). For example:

'listings' => Listing::latest()->filter(request(['tag', 'search']))->get()

On the controller.

if($filters['search'] ?? null) {

            $query->where('tags', 'like', "%" . request('search') . "%")

                  ->orWhere('title', 'like', "%" . request('search') . "%")

                  ->orWhere('description', 'like', "%" . request('search') . "%");

        }

On the Model

**Form Validation And Storing Data Into A Database**

Forms need to submit their data to the “store” route.

Their methods need to be “POST”.

Also, you need to have a directive after the from declaration that is *“@csrf”* that prevents *cross-site scripting.*

In the Routes file:

* Create a new route with a “post” method.
* Pass as its parameters:
  + a “base entity” url, say “/listings”. The same should be what is passed to the “action” attribute in the form declaration.
  + The Controller and Class of “store”

// Post Listing

Route::post('/listings', [ListingContoller::class, 'store']);

In the Controller

* Create the “store” method.
* Pass In the dependency injection of “Request $request”.
* You can display the request using *“dd($request->all())”*
* Create a variable and pass to it the “$request” and the “validate()” method that takes an array of ‘field’ => ‘rule’ pairs as constraints to the form fields you want to validate.
* Return a redirect.
* If any of the validation rules fail, it will send an error to the view.

// Store Listing Data

    public function store(Request $request) {

        // You Will have to import the “Rule” class

$formfields = $request->validate([

            'title' => 'required',

            'company' => ['required', Rule::unique('listings','company')],

            'location' => 'required',

            'email' => ['required', 'email'],

            'website' => 'required',

            'tags' => 'required',

            'description' => 'required',

        ]);

        return redirect('/');

    }

* To display these errors, go to the view and:
  + under each field add an “@error(‘field name’)……@enderror” directive,
  + Within this directive render the “{{$message}}” inside a regular html tag

<div class="mb-6">

                <label

                    for="description"

                    class="inline-block text-lg mb-2"

                >

                    Job Description

                </label>

                <textarea

                    class="border border-gray-200 rounded p-2 w-full"

                    name="description"

                    rows="10"

                    placeholder="Include tasks, requirements, salary, etc"

                ></textarea>

                @error('description')

                    <p class="text-red-500 text-xs mt-1">{{$message}}</p>

                @enderror

            </div>

* To save the data into a database, invoke the model and the create method.

Listing::create($formfields);

return redirect('/');

* Before that however, note that by default, database are protected in that before data can be written onto the same, the filed have to be assigned a property known as “fillable”.
* To do this, there are two things you can do:
  + Go to the model and add:

protected $fillable = ['title', 'company', 'location', 'website', 'email', 'description', 'tags'];

* + Open the “app/Providers/AppServiceProvider.php” file, and inside the “boot()” method, add: [Advanced]

// You will have to import the "Model" class

public function boot(): void

{

    Model::unguard();

}

**Flash Messages**

This are pop-up messages that are triggered, e.g. when something is submitted successfully, and this are held for a one-page load.

Procedure:

* In the controller, after the successful operation, you can:
* Create a Session class and invoke the flash message as: (you will have to import the “Session” class)

Session::flash('message', 'Listing Created Successfully');

* Append a “with()” method to the redirect as:

return redirect('/')->with('message', 'listing created successfully');

N/B: The “message” is arbitrary. You can call it “success”, “error”, etc. as you wish.

The above just fires off the message. To see it, you need to echo it in the form of a view. So:

* Create a Component file, say: *“flash-message.blade.php”.*
* Within it, check for the message.
* If set, create a div and format it as you like, say, make it fixed to the top of the page.
* Within it, create a paragraph and within this, pass in the session message using the blade syntax, e.g as: “{{session(‘message’)}}”.

@if(session()->has('message'))

    <div class="fixed top-0 left-1/2 transform -translate-x-1/2 bg-laravel text-white px-48 py-3">

        <p>{{session('message')}}</p>

    </div>

@endif

* Call the component in the view you would like to appear.

</footer>

     <x-flash-message />

</body>

* To hide the Flash Message using Alpine.Js (A lightweight Js Framework)
  + Import the alpine script into your HTML layout

<script src="//unpkg.com/alpinejs" defer></script>

* + Append the Alpine states to the necessary HTML elements, say the created flash message component.

@if(session()->has('message'))

    <div x-data="{show:true}" x-init="setTimeout(()=>show = false, 3000)" x-show="show" class="fixed top-0 left-1/2 transform -translate-x-1/2 bg-laravel text-white px-48 py-3">

        <p>{{session('message')}}</p>

    </div>

@endif

**Retaining the old values in form submission in case of error**

Append the following blade directive to the html:

<input

     type="text"

     class="border border-gray-200 rounded p-2 w-full"

     name="tags"

     placeholder="Example: Laravel, Backend, Postgres, etc"

     value="{{old('tags')}}"

/>

<textarea

     class="border border-gray-200 rounded p-2 w-full"

     name="description"

     rows="10"

     placeholder="Include tasks, requirements, salary, etc"

>{{old('description')}}</textarea>

**Paginating the Results on a page**

Instead of using get() use paginate() or simplePageinate() as:

'listings' => Listing::latest()->filter(request(['tag', 'search']))->paginate(5)

To display the links to the next pages, create a div in the view and within it add:

<div class="mt-6 p-4">

    {{$listings->links()}}

</div>

To style the links, you will have to publish the “pagination” package using the command:

*php artisan vendor:publish*

A list of packages will appear and you will choose the appropriate one.

**File (Image) Uploads**

To upload images and render them later as part of a webpage:

* Create an html input field within a form and add a form attribute of *“enctype=”multipart/form-data””*
* Set up where the file will be stored*.* By default, this file is stored under “storage/app”.
* To change this, open the “config/filesystems.php”.
* In the return statement, change the default disk from “local” to “public”. This is so that a link to the file can be created from within the “public” directory.
* Set up the database to hold a field that stores the link of the file uploaded.
  + Go to the tables migration file say under “database/migrations/2023\_08\_14\_124422\_create\_listings\_table.php”
  + Add a new field.
  + Run the migrations using: *“php artisan migrate:refresh –seed”*
* In the controller, under the “store()” method:
  + To check that the file is passed successfully, you can *“dd($request->file(‘field name’));”*
  + Check if a file is uploaded using the “hasFile()” method on the “$request”.
  + If it is add it to the array containing the other form data and store it at the same time

if($request->hasFile('logo')) {

            $formfields['logo'] = $request->file('logo')->store('logos','public');

        }

* To make the files publicly accessible, run the command: *“php artisan storage:link”*
* To display the file in the webpage(s), say in an image tag, change the src attribute to where the file is stored, e.g:

src="{{$listing->logo ? asset('storage/' . $listing->logo) : asset('images/no-image.png')}}"

**Creating an Edit Feature for data already uploaded**

Create a route to “show edit”. Say:

// Show Edit Listing

Route::get('/listings/{listing}/edit', [ListingContoller::class, 'edit']);

Add a link to the page where the updates will be made on the page showing the individual data item, say listing.

<x-card class="mt-4 p-2 flex space-x-6">

        <a href="/listings/{{$listing->id}}/edit">

            {{--Font Awesome icon--}}

            <i class="fa-solid fa-pencil"></i>Edit

        </a>

    </x-card>

Create a controller for the route

// Show Edit Form For One Listing

    public function edit(Listing $listing) {

        return view('listings.edit', ['Listing' => $listing]);

    }

Create a view, called “edit.blade.php” to hold the edit form. Within it create a html form (copy it from the “create” form and modify it a little. For example:).

* Add to the action attribute of the form

<form method="POST" action="/listings/{{$listing->id}}" enctype="multipart/form-data">

* Specify a “PUT” method using the blade directive

@method('PUT')

* Replace the values of the input field with the listing attributes from the database, except the file input field. Instead append an image after the field to show what is there already.

<input

    type="text"

    class="border border-gray-200 rounded p-2 w-full"

    name="company"

    value="{{$listing->company}}"

/>

<input type="file" class="border border-gray-200 rounded p-2 w-full" name="logo"

accept="image/\*"/>

<img

class="w-48 mr-6 mb-6"

    src="{{$listing->logo ? asset('storage/' . $listing->logo) : asset('images/no-image.png')}}"

    alt=""

/>

**To update the details of the data items:**

Create a “put” route in “web.php”

// Update Listing

Route::put('/listings/{listing}', [ListingContoller::class, 'update']);

Create this “update” method in the controller

// Update Listing Data

    // We need to pass the listing being updated hence the "Listing $listing"

    public function update(Request $request, Listing $listing) {

// Make Sure That the logged in user is the owner of the listing

        if($listing->id != auth()->id) {

            abort(403, 'Unauthorised Action');

        }

// dd($request->file('logo'));

        $formfields = $request->validate([

            'title' => 'required',

            'company' => 'required',

            'location' => 'required',

            'email' => ['required', 'email'],

            'website' => 'required',

            'tags' => 'required',

            'description' => 'required',

        ]);

        if($request->hasFile('logo')) {

            // To store the new photo and get its path

            $newLogoPath = $request->file('logo')->store('logos', 'public');

            // Delete the old logo file if it exists

            if ($listing->logo) {

                // Get the path of the original image and delete it

                Storage::disk('public')->delete($listing->logo);

            }

            // Update the logo field in the form data

            $formfields['logo'] = $newLogoPath;

        }

        // To update the listing

        $listing->update($formfields);

        // To redirect back to the form but with a flash message.

        return back()->with('message', 'listing updated successfully!');

    }

**Deleting a data item from the system**

* Create a post form with a button, at a view. Ensure it has a “@method(‘DELETE’)” directive as well as the “@csrf” directive. Also ensure that the action of the form is to the actual entity say *“/listings/{{$listing->id}}}”*

<form action="/listings/{{$listing->id}}" method="POST">

            @csrf

            @method('DELETE')

            <button class="text-red-500">

                <i class="fa-solid fa-trash"></i>Delete

            </button>

        </form>

* Create a route for the same functionality. Use a “delete” method on the “Route” class and the “destroy” method to the controller.

// Delete Listing

Route::delete('/listings/{listing}', [ListingContoller::class, 'destroy']);

* Create in the controller a destroy method, and pass to it the model, say listing, as a parameter.
* Delete the rental using *“delete()”* method the redirect.

public function destroy(Listing $listing) {

        $listing->delete();

        if ($listing->logo) {

            // Get the path of the original image and delete it

            Storage::disk('public')->delete($listing->logo);

        }

        return redirect('/')->with('message', 'Listing deleted successfully');

    }

**User Creation and Logging In**

Routes:

// Show Register Form

Route::get('/register', [UserController::class, 'create']);

// Register user

Route::post('/users', [UserController::class, 'store']);

Controller:

// Show Register Form

    public function create() {

        return view('users.register');

    }

    // Create a user

    public function store(Request $request) {

        $formfields = $request->validate([

            'name' => ['required', 'min:3'],

            'email' => ['required', 'email', Rule::unique('users', 'email')],

            'password' => 'required|confirmed|min:6'

        ]);

        // Hash password

        $formfields['password'] = bcrypt($formfields['password']);

        // Create user in the database.

        $user = User::create($formfields);

        // Login

        auth()->login($user);

        return redirect('/')->with('message', 'User created and logged in');

    }

}

View

// To display when the user is logged in.

@auth

     <li>

        <span class="font-bold uppercase">Welcome {{auth()->user()->name}}</span>

     </li>

     <li>

        <a href="/listings/manage" class="hover:text-laravel">

<i class="fa-solid fa-arrow-right-to-bracket fa-solid fa-gear"></i>

             Manage Listings

</a>

     </li>

<li>

        <form action="/logout" method="POST" class="inline">

            <button type="submit">

                 <i class="fa-solid fa-door-closed"></i>Logout

           </button>

        </form>

     </li>

@else // To display in guest mode (when no one is logged in)

    <li>

      <a href="/register" class="hover:text-laravel">

<i class="fa-solid fa-user-plus"></i> Register

</a>

    </li>

     <li>

        <a href="/login" class="hover:text-laravel">

<i class="fa-solid fa-arrow-right-to-bracket"></i>

             Login

</a>

    </li>

@endauth

**To log an already registered User In**

View:

<form action="/users/authenticate" method="POST">

            @csrf

.

.

.

Route:

// Show login form

Route::get('/login', [UserController::class, 'login']);

// authenticate user and login

Route::post('/users/authenticate', [UserController::class, 'authenticate']);

Controller:

// Show login form

Public function login() {

  return view('users.login');

}

// authenticate user and login

public function authenticate(Request $request) {

        $formfields = $request->validate([

            'email' => ['required', 'email'],

            'password' => 'required'

        ]);

        if(auth()->attempt($formfields)) {

            $request->session()->regenerate();

            return redirect('/')->with('message', 'You are now logged in');

        }

        return back()->withErrors(['email' => 'Invalid Credentials'])->onlyInput('email');

    }

**To log User out**

// Log User Out

    public function logout(Request $request) {

        // Remove authentication information from the users session

        auth()->logout();

        // To invalidate the user's session

        $request->session()->invalidate();

        // To regenerate the csrf token

        $request->session()->regenerateToken();

        return redirect('/')->with('message', 'You have been logged out!');

    }

**To Limit Guest and Authenticated Users’ Functionality**

This is done on the routes, using middleware.

Procedure:

To the routes allowed only for authenticated users, append *“->middleware(‘auth’)”* to the end.

// Show "Create Post"

Route::get('/listings/create', [ListingContoller::class, 'create'])->middleware('auth');

This triggers a redirect to another named route, usually, “login”. The logic to this redirect is in a file *“app/Http/Middleware/Authenticate.php”.*

If you have no such named route, append *“->name(‘login’)”* to the route that has the same functionality (showing the login form).

// Show login form

Route::get('/login', [UserController::class, 'login'])->name('login');

To the routes allowed only for the guests (unauthenticated users), append *“->middleware(‘guest’)”* to the end. This, by default, triggers a redirect to “/home”. To change this go to the file *“app/Providers/RouteServiceProvider.php”* and change the const variable “HOME” from ‘/home’ to whatever your home page is.

// Register user

Route::post('/users', [UserController::class, 'store'])->middleware('guest');

**To Create Relationships Between Users and Entities**

Done at three levels:

In the database:

* In the entity’s (say, listings) database migrations file, add a foreign key field to hold the user’s id.

$table->foreignId('user\_id')->constrained()->onDelete('cascade');

* In the database seeder, create a user and have that one user have multiples listings.

// Create a specific user

$user = User::factory()->create([

  'name' => 'Ian Dancan',

  'email' => 'ian@gmail.com'

]);

// Create 6 listings linked to the 'Ian' user by the user's id

Listing::factory(6)->create([

  'user\_id' => $user->id

]);

In the eloquent models:

* Go to the entity’s model file, say, “app/Models/Listing.php”.
* Create a new function called “user()”.
* Within this, specify the “belongsTo()” method.

// Relationship to User

public function user() {

  return $this->belongsTo(User::class, 'user\_id');

}

* Go to the user’s model file, say, “app/Models/User.php”.
* Create a function called “listings()”.
* Within this, specify the “hasMany()” method.

// Relationship with listings

    public function listings() {

        return $this->hasMany(Listing::class, 'user\_id');

    }

In the entity’s controller file:

* To the array of details being submitted, add a “user\_id” field and get this id from the auth() helper:

$formfields['user\_id'] = auth()->id();

**Tinker**

A tool that can be used in the command line to query the database, etc.

To open it, use:

*php artisan tinker*

For example,

To see some specific user in the database you can use the command

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

To see the listings associated with the user, you can:

$user->listings

**To view all the entities associated with a logged in user**

Route:

// Show Manage Listings

Route::get('/listings/manage', [ListingContoller::class, 'manage'])->middleware('auth');

Controller:

public function manage() {

        return view('listings.manage', ['listings' => auth()->user()->listings()->get()]);

}

View:

**Process To Create A New Feature In A Laravel Project**

* Create A Route.
* Create A Controller
* Create A View