BSc Applied Computing

Server-side coding with PHP and Laravel

Introduction to Laravel

In this session…

- Recap OOP PHP
  - Last exercise progress
  - OOP fundamentals (class, namespace and use statements)
  - Frameworks are OOP
  - Frameworks are the easiest way of implementing an API
- Introduction to Laravel
  - Server-side OOP PHP framework for both web applications and APIs
  - Object-relational mapping (ORM)
  - Composer and package managers
  - Tour of a Laravel project
- Laravel task list project
  - Project setup
  - Project development workshop

Quick recap

- OOP fundamentals (class, namespace and use statements)
- PHP is not OOP by default – it has to be used in that way
- OOP splits up the application into manageable functional chunks
- Helps make larger projects more understandable
- Allows a development team work on different aspects of the same application simultaneously without interfering with each other
- Most server-side frameworks use the OOP paradigm (Laravel, CakePHP, etc)
- Creating an API using a framework is much easier if you understand OOP

BSc Applied Computing

Laravel, a PHP server-side framework

Laravel

- **“The PHP Framework for Web Artisans**
  Laravel is a web application framework with expressive, elegant syntax.“
  - Laravel.com

- A Rapid Application Development (RAD) framework
- Simplifies the development of complex web application projects
- Laravel is an object-oriented framework for server-side web application projects and works equally well as a server-side API
- It is fairly easy to learn when compared with its main rivals
- But, as with everything, there is still a learning curve!

Object-relational mapping (ORM)

- Frameworks are often used as they provide many of the essential aspects of complex web projects without the need for the developer to code them or port them from other projects
- One important aspect that frameworks often provide is object-relational mapping (ORM)
- An ORM handles connections and all interactions with the database for you
- ORM interact with a database and create a virtual database of data records represented as lists of objects which are then presented to the framework
- Laravel's ORM implementation, **Eloquent**, completely removes the need for any direct database interaction to be coded by the developer once a data structure has been defined
- This means no monitoring, opening or closing connections, no SQL statements, no dealing with different types of response from the database, no mapping result objects to PHP structures so the data becomes available – all of this is done for you!
- Learning and using an ORM is a significant time saving for larger web application projects

ORM example

- Rather than the lengthily process needed to save a new record in the last project that we built:

```
// all ok - make db connection
$this->m_dbConnector->Open();

// build SQL
$sql = "INSERT into users (email,password) VALUES ('" . $_POST['email'] . "','" . $_POST['password'] . "')";

// run query, check for errors
if ($this->m_dbConnector->Insert($sql)) {
```

- In Eloquent:

```
User::create(['email' => $data['email'],'password' => Hash::make($data['password'])]);
```

- **Note:** this example also demonstrates another function that is provided for us by Laravel – password encryption

Composer, a PHP dependency manager

- Current web development practice involves multiple packages
- Packages are built on other packages
  - Lightbox.js requires jQuery for example
- Fluid sites (or under development) require updated packages
- Easy to break dependencies by updating a single package if that then requires updates to its dependency packages
- Dependancy managers resolve this
- They keep track of the packages installed and oversee updates making sure that any new requirements of an updated package are met
- Different languages have different managers… including:
  - Maven for Java
  - NuGet for C# / Visual Studio
  - Composer for PHP
- You were asked to make sure that Composer was installed for this session
- If it is still not installed, please install it now

Tour of a Laravel project

- A fresh **Laravel** install will include a complex folder structure
- You do not need to be familiar with all of these, but there are some areas which you are likely to want to use at some point:
  - App/HTTP/Controllers – application logic
  - App/HTTP/Middleware – authentication logic
  - App/Models – data object classes default folder (can be changed if preferred)
  - Routes – all URL request are routed to controllers and methods (web.php and api.php)
  - Public – images, CSS and JavaScript resources
  - Resources/Views – HTML templates known as blades which combine to form views
  - Database/Migrations – Data structure information used to configure the database tables

BSc Applied Computing

Laravel task list project

Project setup

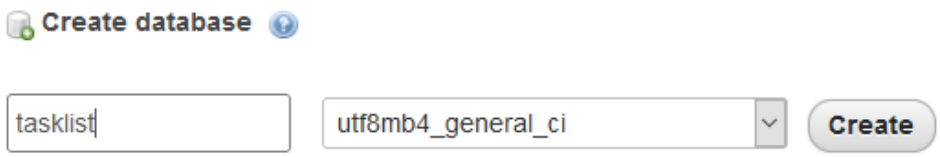
- You should create a new virtual host for this project (use the instructions from week 1 if needed)
- **Project URL:** laravel.tasklist
- **Project folder:** LaravelTaskList
  (do not create the folder yet, just add to **httpd-vhosts.conf** including a reference to a subfolder that Laravel will create for us, **public**)

```
<VirtualHost *:80>
    DocumentRoot C:\xampp\htdocs\LaravelTaskList\public
    ServerName laravel.tasklist

    <Directory "C:\xampp\htdocs\LaravelTaskList\public">
        Options Indexes FollowSymLinks Includes ExecCGI
        AllowOverride All
        Require all granted
    </Directory>
</VirtualHost>
```

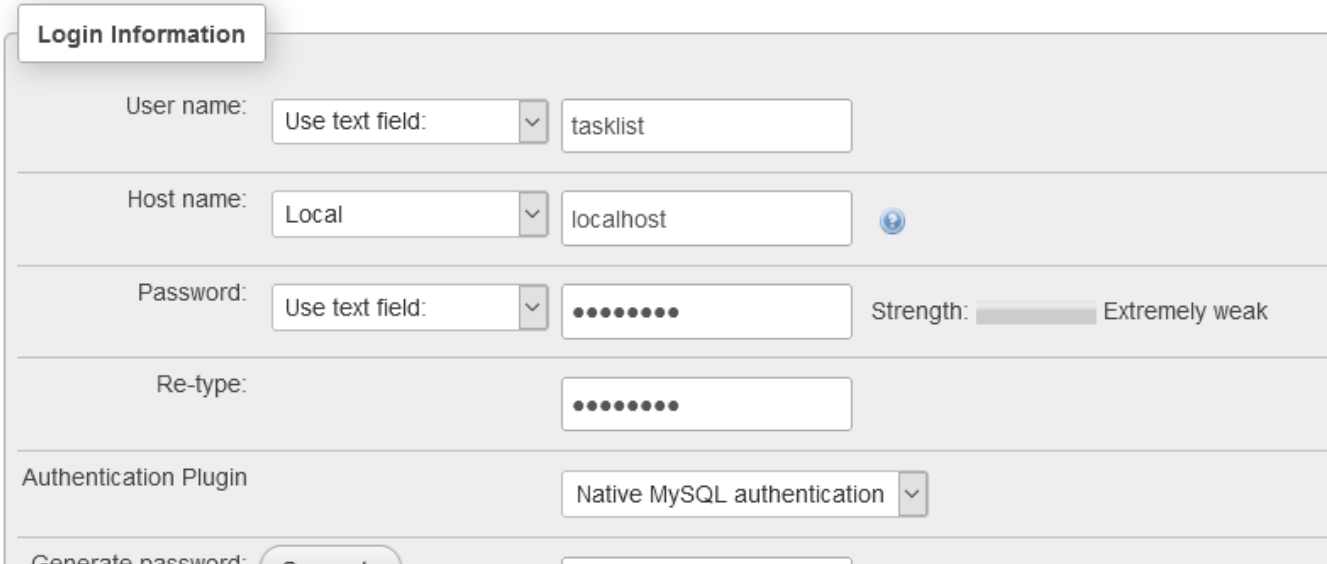
- This tutorial is largely the same as the Laravel Quickstart tutorial but updated for Laravel 7 to take advantage of simpler authentication etc.
- This project will be developed further in the next session so it is imperative that you complete this stage
- This tutorial will use the **Visual Code** editor
- **All new code will be highlighted**, other code will be included to indicate placement of new code only and should not be entered
- Further explanation of code is included in the notes pane of specific slides

Database setup

- Even using Laravel, the database itself and the user account for the webserver still need to be set up using PHPMyAdmin*

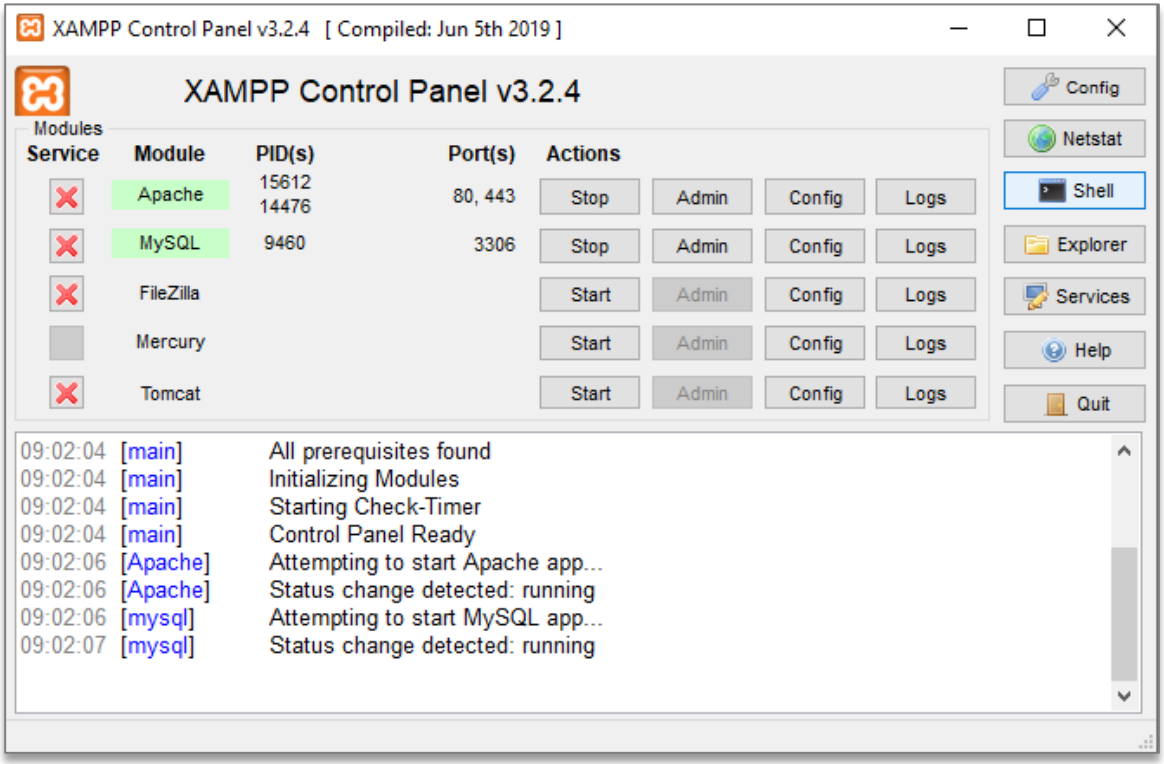


- Set up a new database with the following details
- **Database:** tasklist
- **Username:** tasklist
- **Password:** tasklist
- **Hostname:** localhost



Creating a new Laravel project

- Open the XAMPP control panel and find the Shell button
- This opens a command line window which allows us to use PHP and composer to create a new project



Creating a new Laravel project

- The shell opens in the **c:\xampp** folder
- Navigate to the htdocs folder with the command **cd htdocs**
- Create a new Laravel project with **composer** using the command
  - **composer create-project laravel/laravel LaravelTaskList 8.6.12 –prefer-dist**

```
C:\htdocs>cd c:\xampp\htdocs

c:\xampp\htdocs>composer create-project laravel/laravel LaravelTaskList 8.6.12 --prefer-dist
```

- This will create a new project folder called **LaravelTaskList** and download a fresh Laravel install to it

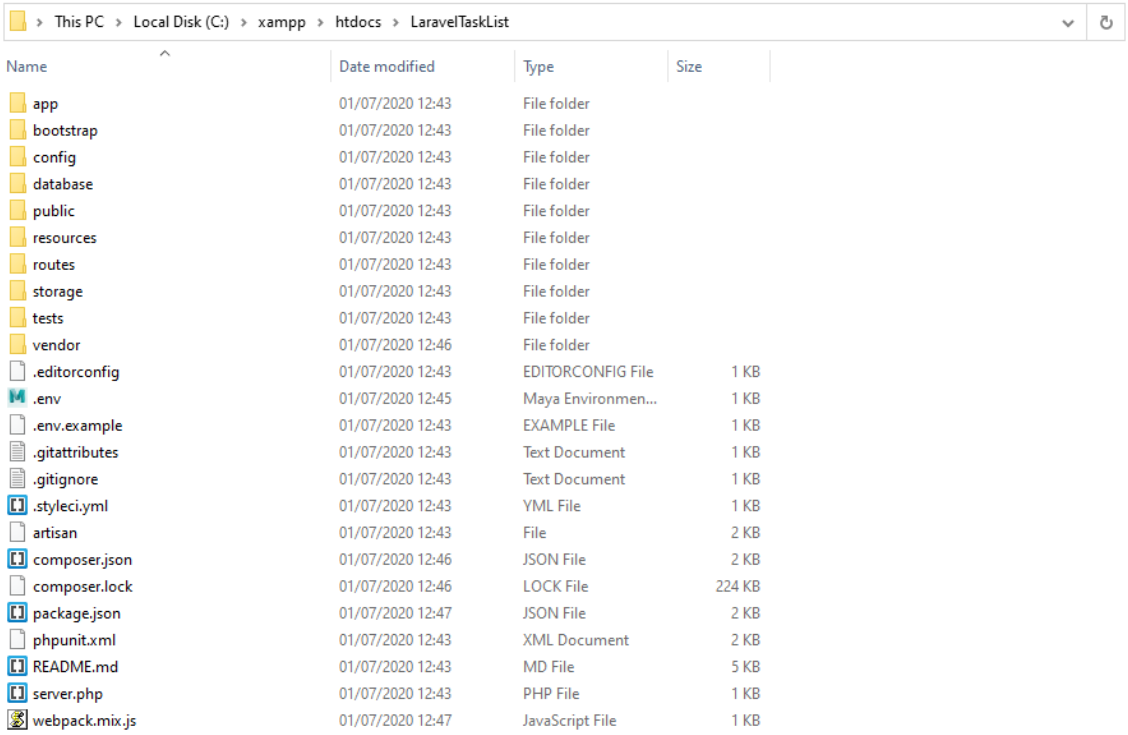
Adding authentication

- Laravel has a bespoke user authentication module*
- Move into the project folder with the command
  - **cd LaravelTaskList**
- Tell composer that we want to add authentication with the command
  - **composer require laravel/ui**
- Install the authentication scaffolding with the command
  - **php artisan ui vue --auth**

```
User@DESKTOP-3LR8GQL c:\xampp\htdocs
# cd LaravelTaskList

User@DESKTOP-3LR8GQL c:\xampp\htdocs\LaravelTaskList
# composer require laravel/ui
Using version ^2.1 for laravel/ui
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing laravel/ui (v2.1.0): Downloading (100%)
Writing lock file
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi
Discovered Package: facade/ignition
Discovered Package: fideloper/proxy
Discovered Package: fruitcake/laravel-cors
Discovered Package: laravel/tinker
Discovered Package: laravel/ui
Discovered Package: nesbot/carbon
Discovered Package: nunomaduro/collision
Package manifest generated successfully.
44 packages you are using are looking for funding.
Use the `composer fund` command to find out more!

User@DESKTOP-3LR8GQL c:\xampp\htdocs\LaravelTaskList
# php artisan ui vue --auth
```

Project folder

- Installing Laravel may take a moment
- When it is complete, take a look in the project folder, it should look something like this



Database configuration

- Laravel needs access to the database that we have set up for this project, this is contained in the **.env** file
- Navigate to the project folder in Windows Explorer or similar, right click and select **Open as a Visual Code Project**
- Locate and open the **.env** file in the project root folder – make the highlighted changes below then save this file

```
1  APP_NAME=LaravelTaskList
2  APP_ENV=local
3  APP_KEY=base64:vLQG5DIVL9xeASb+ZVRD9cCbfIva6iT5nHZStNKD0wA=
4  APP_DEBUG=true
5  APP_URL=http://laravel.tasklist
6
7  LOG_CHANNEL=stack
8
9  DB_CONNECTION=mysql
10 DB_HOST=localhost
11 DB_PORT=3306
12 DB_DATABASE=tasklist
13 DB_USERNAME=tasklist
14 DB_PASSWORD=tasklist
```

Database Migrations

- “Laravel's database migrations provide an easy way to define your database table structure and modifications” – Laravel Quickstart
- Useful for team projects as these files can be used to setup all databases identically – similar in concept to the **dbsetup.php** script used last week
- This project requires 2 tables, **users** and **tasks**
- As we have included the **Auth** module, we now have a default **users** table migration file already set up
- Migration files contain two important methods, up for data to be added to the database and down for how this can be removed from the database
- The default users migration from the new project is on the next slide
- Migrations can add whole tables or adjust the structure of existing tables

Users migration

- The **up** function contains the table structure to be added
- The **down** function contains information about how this can be removed should this be required later

```php
<?php

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

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

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

Creating a Tasks migration

- Laravel includes a command line interface known as **Artisan**
- **Artisan** is very useful for creating new Laravel files as it will setup the object structure, includes and place the new file in the appropriate folder
- This cuts down on an LOT of potential errors
- As Artisan is a command line interface, open the **XAMPP** control panel and use the **Shell** button
- Make sure that the shell is navigated to the project folder (the path will be shown and should end in **LaravelTaskList**)
- Type the following command to create the tasks table migration

```
User@DESKTOP-3LR8GQL c:\xampp\htdocs\LaravelTaskList
# php artisan make:migration create_tasks_table --create=tasks
Created Migration: 2020_07_02_085127_create_tasks_table
```

Creating a Tasks migration

- A new file has been created in the **database/migrations** folder
- It will have a filename of the current date/time and will include **create_tasks_table**
- Open this file in Visual Code

```php
<?php

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

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

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

Configuring the Tasks migration

- The tasks table requires a **string** to hold the task **name** and a **user id** field to associate the tasks with the user that owns it
- Add two lines to the **up** method which specify these columns – add the highlighted code below and save the migration file

```
 9      /**
10       * Run the migrations.
11       *
12       * @return void
13       */
14      public function up()
15      {
16          Schema::create('tasks', function (Blueprint $table) {
17              $table->id();
18              $table->integer('user_id')->index();
19              $table->string('name');
20              $table->timestamps();
21          });
22      }
```

Running the migrations

- Return to the XAMPP Shell, make sure that the shell is located inside the current project folder, then run the following Artisan command
  - **php artisan migrate**
- This will configure the database specified in **.env** with all of the configurations in the migrate folder

```
Setting environment for using XAMPP for Windows.
User@DESKTOP-3LR8GQL c:\xampp
# cd htdocs\LaravelTaskList

User@DESKTOP-3LR8GQL c:\xampp\htdocs\LaravelTaskList
# php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table (0.02 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table (0.02 seconds)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated:  2019_08_19_000000_create_failed_jobs_table (0.01 seconds)
Migrating: 2020_07_02_085127_create_tasks_table
Migrated:  2020_07_02_085127_create_tasks_table (0.02 seconds)

User@DESKTOP-3LR8GQL c:\xampp\htdocs\LaravelTaskList
#
```

Testing migrations

- Test that the migrations have been successful by checking the strucuture of the tasks table
- The table should have been added to the database so check the tasks table in PHPMyAdmin

	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
☐	1	**id**	bigint(20)		UNSIGNED	No	*None*		AUTO_INCREMENT	Change Drop More
☐	2	**user_id**	int(11)			No	*None*			Change Drop More
☐	3	**name**	varchar(255)	utf8mb4_unicode_ci		No	*None*			Change Drop More
☐	4	**created_at**	timestamp			Yes	*NULL*			Change Drop More
☐	5	**updated_at**	timestamp			Yes	*NULL*			Change Drop More

- If your tasks table looks like the one above, the migration was successful
- Migrations are a very powerful, portable and easy syntax for database management tasks
- They are worth exploring further > migrations documentation

Eloquent Models

- "Eloquent is Laravel's default ORM (object-relational mapper). Eloquent makes it painless to retrieve and store data in your database using clearly defined "models". Usually, each Eloquent model corresponds directly with a single database table." – Laravel Quickstart
- Models by default live in the **app** folder – as we have included the **Auth** module, the **Users** model has already been created and placed here for us
- In order to create the Tasks model, return to the XAMPP Shell and use the following command
  - **php artisan make:model Task**

```
User@DESKTOP-3LR8GQL c:\xampp\htdocs\LaravelTaskList
# php artisan make:model Task
Model created successfully.
```

Configuring the Task model

- Artisan has made a new **Tasks** object in the **app/Models** folder
- Open this in **Visual Code**, it should look like the file below, fairly sparse

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Task extends Model
{
    use HasFactory;
}

```

Configuring the Task model

- The **name** field should be filled with data that the user can add, all other columns in the **Task** model are auto-numbered (**id**), default value (**created_at** and **updated_at**) or assigned through a relationship (**user_id**)
- Fields which contain user data are known as **mass assignable**
- Add the highlighted code to the **Task** model to set the **name** field as mass assignable, then save the file

```
 8  class Task extends Model {
 9
10      use HasFactory;
11
12      /**
13       * The attributes that are mass-assignable.
14       *
15       * @var array
16       */
17      protected $fillable = ['name'];
18  }
```

Entity relationships

- Defining a relationship between two tables requires editing of both model files, in this case one **user** can have many **tasks**
- First, define the relationship in the **User** model… **User hasMany Task**
- Open **app/Models/User.php** in Visual Code and add the highlighted method and comment to the bottom of this class

```
31      /**
32       * The attributes that should be cast to native types.
33       *
34       * @var array
35       */
36      protected $casts = [
37          'email_verified_at' => 'datetime',
38      ];
39
40      /**
41       * Get all of the tasks for the user.
42       */
43      public function tasks() {
44          return $this->hasMany(Task::class);
45      }
46  }
```

Entity relationships

- Next, establish the relationship for the Task model
- Each task has only one user so… **Task belongsTo User**
- Open **app/Model/Task.php** in Visual Code and add the highlighted method to the end of the class

```
 8 ▼ class Task extends Model {
 9
10       use HasFactory;
11
12 ▼     /**
13        * The attributes that are mass-assignable.
14        *
15        * @var array
16        */
17       protected $fillable = ['name'];
18
19 ▼     /**
20        * Get the user who owns this task
21        */
22 ▼     public function user() {
23           return $this->belongsTo(User::class);
24       }
25   }
```

Entity relationships, example use

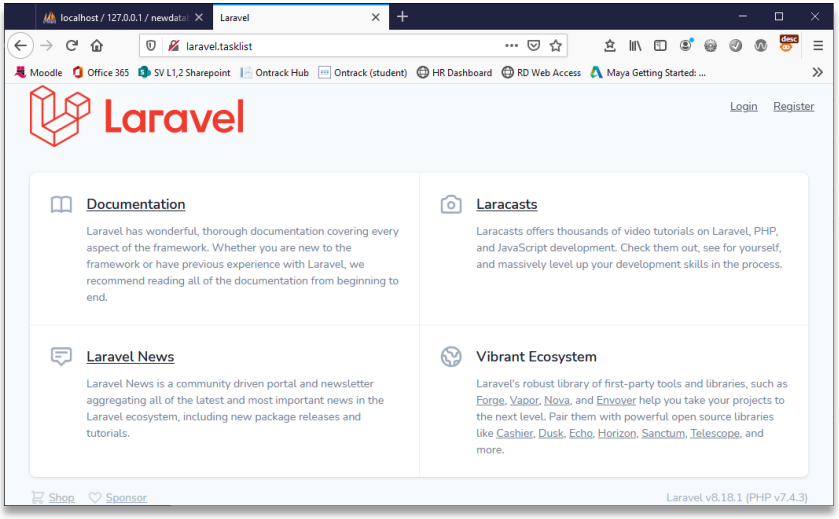
- Now that the relationship between these two models is established, retrieving all of the tasks for any user is a simple as this

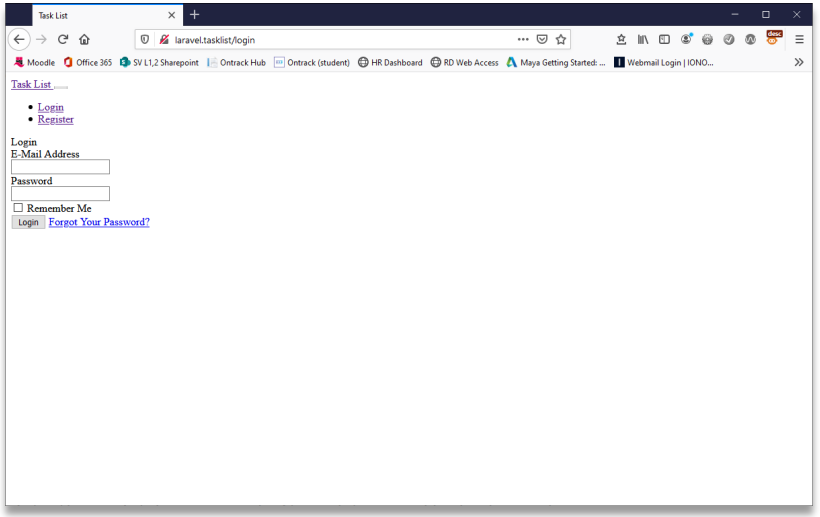
```
$user = App\Models\User::find(1);

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

- **Do not include this code**, it is here as an example of how powerful model relationships are and is not intended for inclusion in the project… not yet!
- This concludes the building of our models and relationships for this project, ready for use in the Controllers later on

Quick test

- Open a browser and navigate to **http://laravel.tasklist** you should see the default Laravel boilerplate page
- Now try **http://laravel.tasklist/login**, you should see a plain login form





Routing

- If you completed last week's project, you would have had a sneaky preview of Laravel Routing
- Laravel project commonly employ close relationships between HTTP routes and controller methods
- Open **routes/web.php** in Visual Code – this is where all of the website HTTP requests are contained*

```
16 ▼ Route::get('/', function () {
17       return view('welcome');
18   });
19
20   Auth::routes();
21
22   Route::get('/home', [App\Http\Controllers\HomeController::class, 'index'])->name('home');
```

Removing the Closure route

- We have no further need of the Laravel boilerplate page
- As we have a /home route defined we will use that as the default page view from here
- In Visual Code, delete the highlighted code from the /routes/web.php file

```
16  Route::get('/', function () {
17      return view('welcome');
18  });
19
20  Auth::routes();
```

```
16  Auth::routes();
17
18  Route::get('/', [App\Http\Controllers\HomeController::class, 'index'])->name('home');
19  Route::get('/home', [App\Http\Controllers\HomeController::class, 'index'])->name('home');
```

Adding Routes to TaskController

- The tasks model will require some application logic to manage the task list data, this will be added to a **TaskController** class
- Before creating the **TaskController**, add the routes to it
- Add the highlighted lines to the bottom of **web.php** to implement routes to the **TaskController** functions then save the file

```
16  Auth::routes();
17
18  Route::get('/', [App\Http\Controllers\HomeController::class, 'index'])->name('home');
19  Route::get('/home', [App\Http\Controllers\HomeController::class, 'index'])->name('home');
20
21  Route::get('/tasks', [App\Http\Controllers\TaskController::class, 'index']);
22  Route::post('/task', [App\Http\Controllers\TaskController::class, 'store']);
23  Route::delete('/task/{task}', [App\Http\Controllers\TaskController::class, 'destroy']);
```

Creating TaskController

- We know that we need a Controller with the name **TaskController** and with methods named **index**, **store** and **destroy**
- Add this controller using the **XAMPP Shell** and the following command
  - **php artisan make:controller TaskController**

```
User@DESKTOP-3LR8GQL c:\xampp\htdocs\LaravelTaskList
# php artisan make:controller TaskController
Controller created successfully.
```

- The new **TaskController** can be found in **app/HTTP/Controllers**, open it in Visual Code and have a look at the default file that has been created

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class TaskController extends Controller {
    //
}
```

Requiring authentication

- To use the functionality provided by **TaskController**, the user would need to be logged in, we need to restrict access to **TaskController** to only those that have logged in
- Laravel, as many other server-side frameworks, make this kind of requirement relatively easy through a layer known as **Middleware**.
- As we have included the Auth module, the Auth Middleware has been pre-built for us and we only need to specify that we want it used in the **TaskController**
- In Laravel, required Middleware is specified for each class in its constructor function
- If you would like to look over the authentication Middleware, it can be found in **app/HTTP/Middleware/authentication.php** and the routing to this Middleware can be found in **app/HTTP/Controllers/Kernel.php** (do not make any changes to these files)

Adding the model reference

- The **TaskController** will need to reference the **Task** class
- To add a reference to **Task** to the top of **TaskController** add the following highlighted code*

```
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request;
6  use App\Models\Task;
7
8  class TaskController extends Controller {
9      //
10 }
```

Requiring authentication

- Add the highlighted code and comments to **TaskController** to implement a constructor function and include the authentication Middleware

```
7  class TaskController extends Controller {
8
9      /**
10      * Create a new controller instance.
11      *
12      * @return void
13      */
14     public function __construct() {
15         $this->middleware('auth');
16     }
```

Routes and views (blades)

- Each route links a URL to a controller function as previously mentioned
- The controller functions perform application logic and return a view for the browser to display
- In Laravel views are split into a number of template files known as **Blades**
- All blades have a filename which includes **<name>.blade.php** – this instructs Laravel to use the blade templating engine when rendering the view which allows access to a simple and elegant syntax for defining these views
- Blades are to be found in the **resources/views** folder and many will already exist here in folders for the authentication views such as **login** or **register** and layouts which contain the overall page structure into which specific views are added (much like the brief HTML framework in the **index.php** file in the previous project)
- As we add functionality to our controller, we will also add blades into the resources folder to display the results
- This project will include one view with a form to add new tasks and a table to display stored tasks (much as the previous tasks list project had)

Application layout blade

- The application will use **resources/views/layout/app.blade.php** to include structural HTML, include CSS and JS references and so on
- The file that has been created for us contains references to a number of Bootstrap files which come packaged but not compiled or installed
- Compiling this requires **NPM** a **Node.js** package manager*
- If you want to compile the bootstrap files you will need to install **Node.js** before the next step (if you do not want to do this, the rest of the project will work fine, it just won't look very nice)
- With **Node** installed and using the **XAMPP Shell**, enter the commands
  - **npm install**
  - **npm run dev**

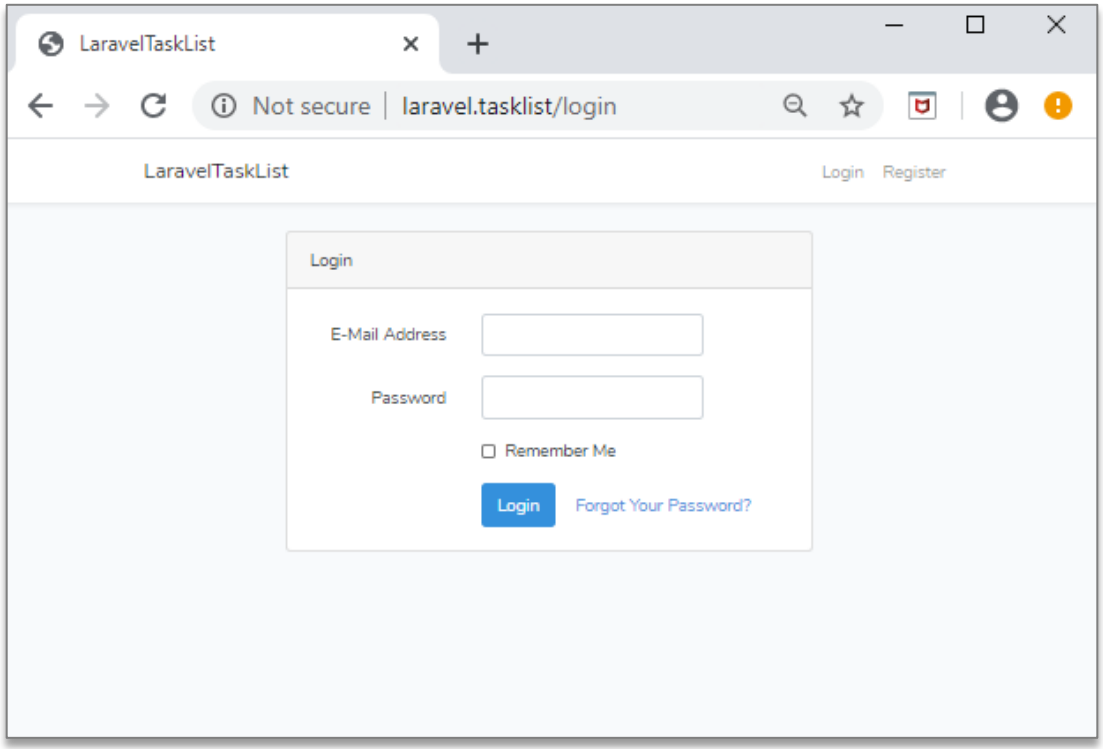
```
User@DESKTOP-3LR8GQL c:\xampp\htdocs\LaravelTaskList
# npm install
```

```
User@DESKTOP-3LR8GQL c:\xampp\htdocs\LaravelTaskList
# npm run dev
```

- This should build and deploy the JS and CSS that the bootstrap template is looking for

Testing the bootstrap template

- Simply reload your browser, the login form should now look a little bit more finished

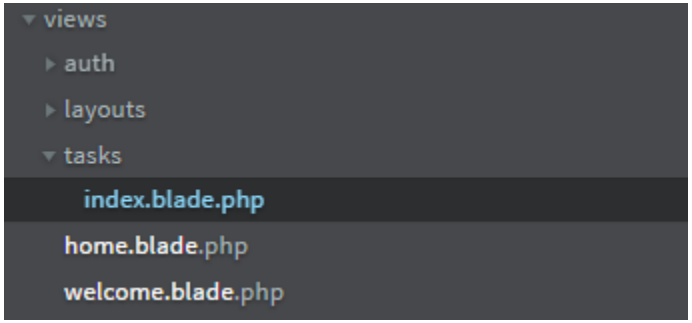


App.blade.php

- The application will use **resources/views/layout/app.blade.php** to include structural HTML, include CSS and JS references and so on
- **Line 79** contains the statement **@yield('content')**, this is a special Blade directive that specifies where all child pages that extend the layout can inject their own content.
- **Line 42** contains **@guest** and **line 72** contains **@endguest** – all of the code between these statements will display only if the user is NOT logged in – this is an example of how code blocks are defined in the blade syntax
- CSS is included on **Line 20** – the **{{ assets('css/app.css') }}** statement returns a URL to a file named **app.css** which is contained in the **public/css** folder – you can hard-code these if preferred but any changes to the project environment would then also have to be mirrored in the blades
- Often, properties are passed to the view by the controller and are displayed in page with the double brace indicators **{{ --something-- }}** – for example, **line 57** includes **{{ Auth::user()->name }}** which prints the name attribute of the currently authenticated user object into the page (effectively displaying the name of the person logged in)

Adding a tasks view

- We are ready to add our tasks view blade
- Add a new **folder** inside the **resources/views** folder named **tasks**
- Add a new **file** inside the **resources/views/tasks** named **index.blade.php**



Adding a tasks view

- Open **index.blade.php** in Visual Code and add the following code

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

@section('content')

<div class="row justify-content-center">
    <div class="col-md-8 message">

        <div class="card">
            <div class="card-header">New Task</div>

            <div class="card-body">
                <!-- display validation errors -->
                @include('common.errors')

                <!-- TODO: new task -->
            </div>

            <!-- TODO: current tasks -->
        </div>
    </div>
</div>
@endsection
```

Adding the new task form

- Replace the TODO: new task form comment with the following highlighted code to add a HTML form to **index.blade.php**

```
11            <div class="card-body">
12                <!-- display validation errors -->
13                @include('common.errors')
14
15                <!-- new task -->
16                <form action="/task" method="post" class="form-horizontal">
17                    {{ csrf_field() }}
18                    <!-- Task name -->
19                    <div class="form-group">
20                        <label for="task-name" class="col-sm-3 control-label">Task</label>
21                        <div class="col-sm-6">
22                            <input type="text" name="name" id="task-name" class="form-control" />
23                        </div>
24                    </div>
25                    <!-- add button -->
26                    <div class="form-group">
27                        <div class="col-sm-offset-3 col-sm-6">
28                            <button type="submit" class="btn btn-primary">Add Task</button>
29                        </div>
30                    </div>
31                </form>
32
33            </div>
```

Returning the index view

- As we now have a view to return we can instruct the **TaskController** to do this by creating the **index** method specified earlier in **routes/web.php**
- Open **TaskController** in Visual Code and add the highlighted code

```
7  class TaskController extends Controller {
8
9      /**
10      * Create a new controller instance.
11      *
12      * @return void
13      */
14     public function __construct() {
15         $this->middleware('auth');
16     }
17
18     /**
19      * Display a list of all of the user's task.
20      *
21      * @param  Request  $request
22      * @return Response
23      */
24     public function index(Request $request) {
25         return view('tasks.index');
26     }
27 }
```

Validating a request

- We have set up a route for **post** requests sent to **/form** and have built a form that can send these, now we handle these requests
- Initially, we will create the store method specified in **routes/web.php** and make sure that we validate the incoming data
- Eloquent deals with SQL injections for us and as we have included the CSRF field we are also protected from that kind of attack
- We should still validate the incoming data to make sure that we have been sent a suitable task name
- Open **TaskController** in Visual Code and add the highlighted code on the next slide to implement the **store** method and add the validate clause to check that **name** has content (required) and is no longer than 255 characters

Validating a request

```
18      /**
19       * Display a list of all of the user's task.
20       *
21       * @param  Request  $request
22       * @return Response
23       */
24      public function index(Request $request) {
25          return view('tasks.index');
26      }
27
28      /**
29       * Create a new task.
30       *
31       * @param  Request  $request
32       * @return Response
33       */
34      public function store(Request $request)
35      {
36          $this->validate($request, [
37              'name' => 'required|max:255',
38          ]);
39
40          // Create The Task...
41      }
42  }
```

The $errors variable

- Remember that we used the **@include('common.errors')** directive in **index.blade.php** to render form validation errors
- The **common.errors.blade.php** file will allow us to easily show validation errors in the same format across all of our pages
- We have yet created this file so let’s do so now
- Make a new folder named **common** in **resources/views**
- Add a new file, **errors.blade.php** in the **resources/views/common** folder
- The code for this file is on the following slide

The $errors variable

- Open the **errors.blade.php** file in Visual Code and add the following code saving when complete

```
1  @if (count($errors) > 0)
2      <!-- form error list -->
3      <div class="alert alert-danger">
4          <strong>Whoops! Something went wrong!</strong>
5
6          <ul>
7              @foreach ($errors->all() as $error)
8                  <li>{{ $error }}</li>
9              @endforeach
10         </ul>
11     </div>
12 @endif
```

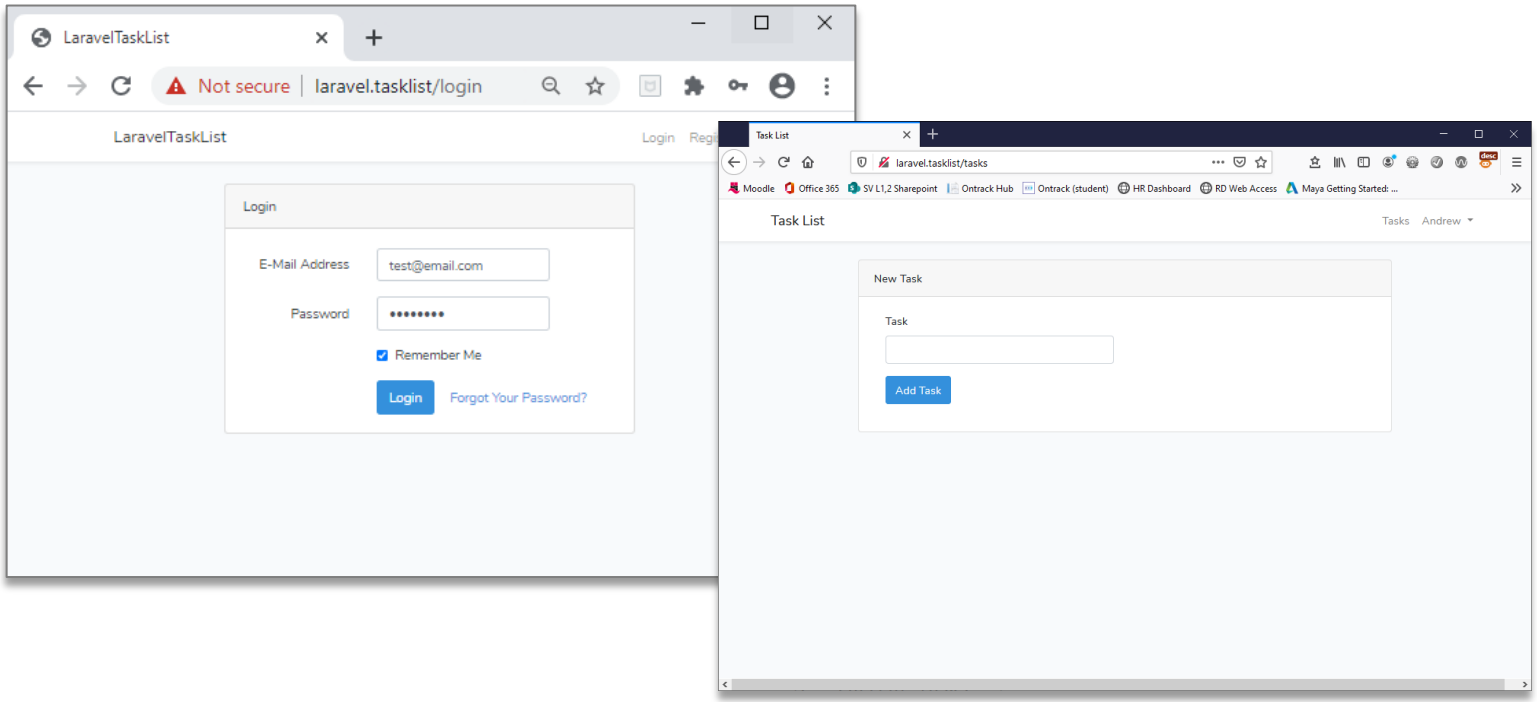
Completing the store method

- Now that any errors are trapped and handled, we can complete the **store** method in the **TaskController**
- Open **TaskController** in Visual Code and add the highlighted code

```
28      /**
29       * Create a new task.
30       *
31       * @param  Request  $request
32       * @return Response
33       */
34      public function store(Request $request) {
35          $this->validate($request, [
36              'name' => 'required|max:255',
37          ]);
38
39          $request->user()->tasks()->create([
40              'name' => $request->name,
41          ]);
42
43          return redirect('/tasks');
44      }
45  }
```

Testing the current functionality

- Return to and refresh your browser, you should now be able to register a user, log in and create a new task
- Check that the data is being added correctly in **PHPMyAdmin**



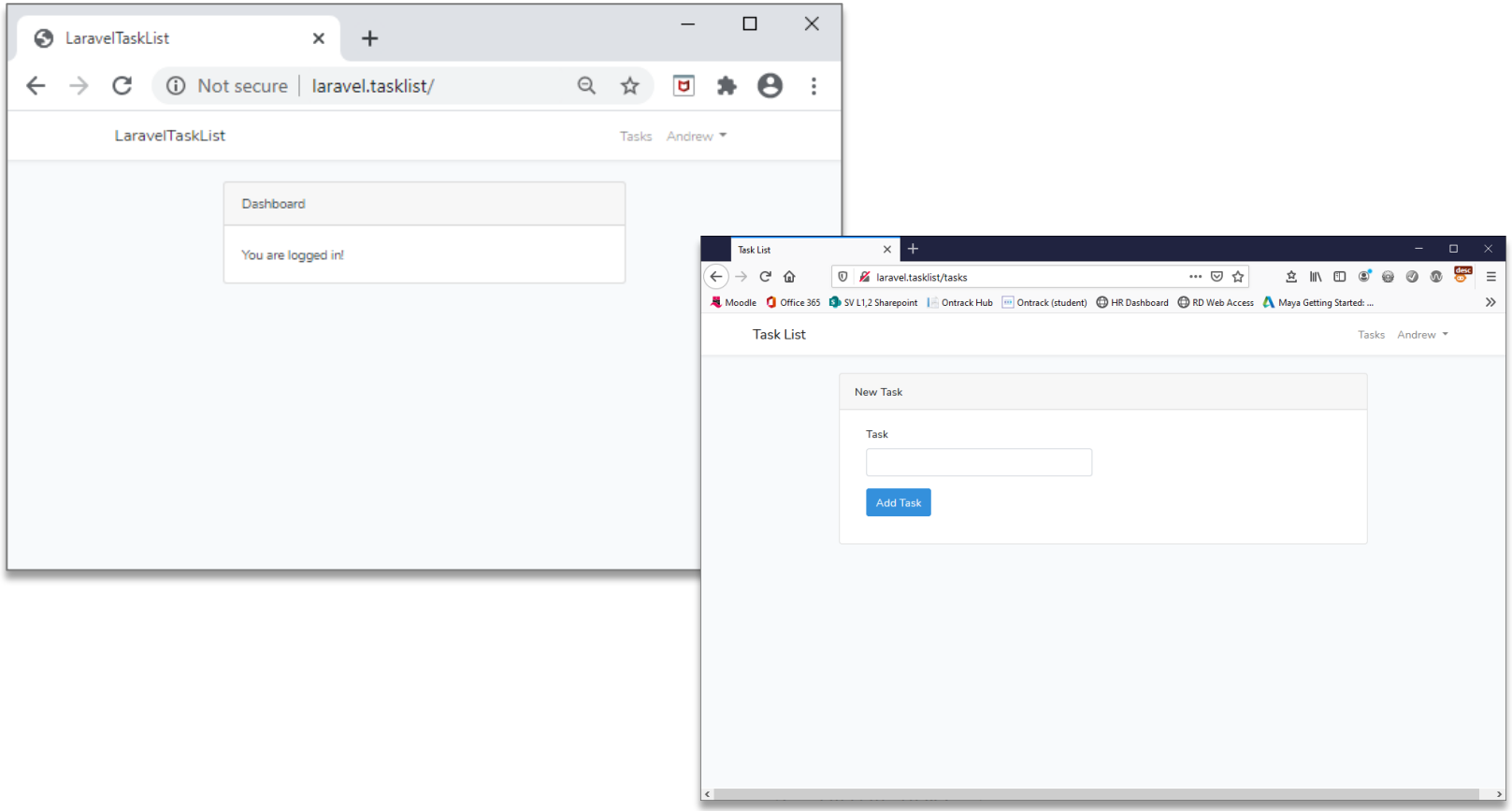
Testing shows up a few issues

- There is no control to get from **home** to **tasks**
  - This can also be fixed with a simple addition to **resources/views/layout/app.blade.php**
  - Open this blade, add the highlighted code below and save

```
46                          @if (Route::has('register'))
47                              <li class="nav-item">
48                                  <a class="nav-link" href="{{
                                    route('register') }}">{{ __('Register')
                                    }}</a>
49                              </li>
50                          @endif
51                      @else
52                          <li class="nav-item">
53                              <a class="nav-link" href="/tasks">Tasks</a>
54                          </li>
55                          <li class="nav-item dropdown">
56                              <a id="navbarDropdown" class="nav-link
```

Retest

- Both of these issues should now be resolved



Named routes

- The link that was just added for the tasks page was hard-coded
- This works fine as it is but becomes an issue if the URL has to change
- Laravel has a system in place to deal with this kind of maintenance known as named Routes
- Each route in **routes/web.php** can be extended with a name and that name used in blades, if the route changes, so does the hyperlink
- Open **routes/web.php** in Visual Code and add the highlighted code

```
16  Auth::routes();
17
18  Route::get('/home', 'HomeController@index')->name('home');
19  Route::get('/', 'HomeController@index')->name('home');
20
21  Route::get('/tasks', 'TaskController@index')->name('tasks');
22  Route::post('/task', 'TaskController@store');
23  Route::delete('/task/{task}', 'TaskController@destroy');
```

Named routes

- Return to **app.blade.php** and replace the **/tasks** location in the hyperlink that we just added with the code highlighted below

```
@else
    <li class="nav-item">
        <a class="nav-link" href="{{ route('tasks') }}">Tasks</a>
    </li>
    <li class="nav-item dropdown">
```

- Save all files and retest the site, you will see no difference in the presented code but now if we change the route for **/tasks** we will not need to pick through blades which might implement this URL and change those as well

Building a list of existing tasks

- Now we can add tasks, we need a way to display all of the existing tasks which have been added to the database
- In order to do this **TaskController** needs to collect all existing tasks and package them before passing them to the view
- Open **TaskController** in Visual Code and alter the contents of the index method to the code shown below

```
19      /**
20       * Display a list of all of the user's task.
21       *
22       * @param  Request  $request
23       * @return Response
24       */
25      public function index(Request $request) {
26          $tasks = Task::where('user_id', auth()->user()->id)->get();
27
28          return view('tasks.index', [
29              'tasks' => $tasks,
30          ]);
31      }
```

Displaying the tasks

- Now that a collection of tasks for this user is passed to the view, we need only add display logic for this to the view
- Open **index.blade.php** in Visual Code and add the highlighted code replacing the TODO comment

```
                    <div class="form-group">
                        <div class="col-sm-offset-3 col-sm-6">
                            <button type="submit" class="btn btn-primary">Add Task</button>
                        </div>
                    </div>
                </form>

            </div>

            <!-- current tasks -->
            @if (count($tasks) > 0)
                <div class="card-header">Current Tasks</div>
                <div class="panel-body">


                </div>
            @endif
        </div>
    </div>
</div>
@endsection
```

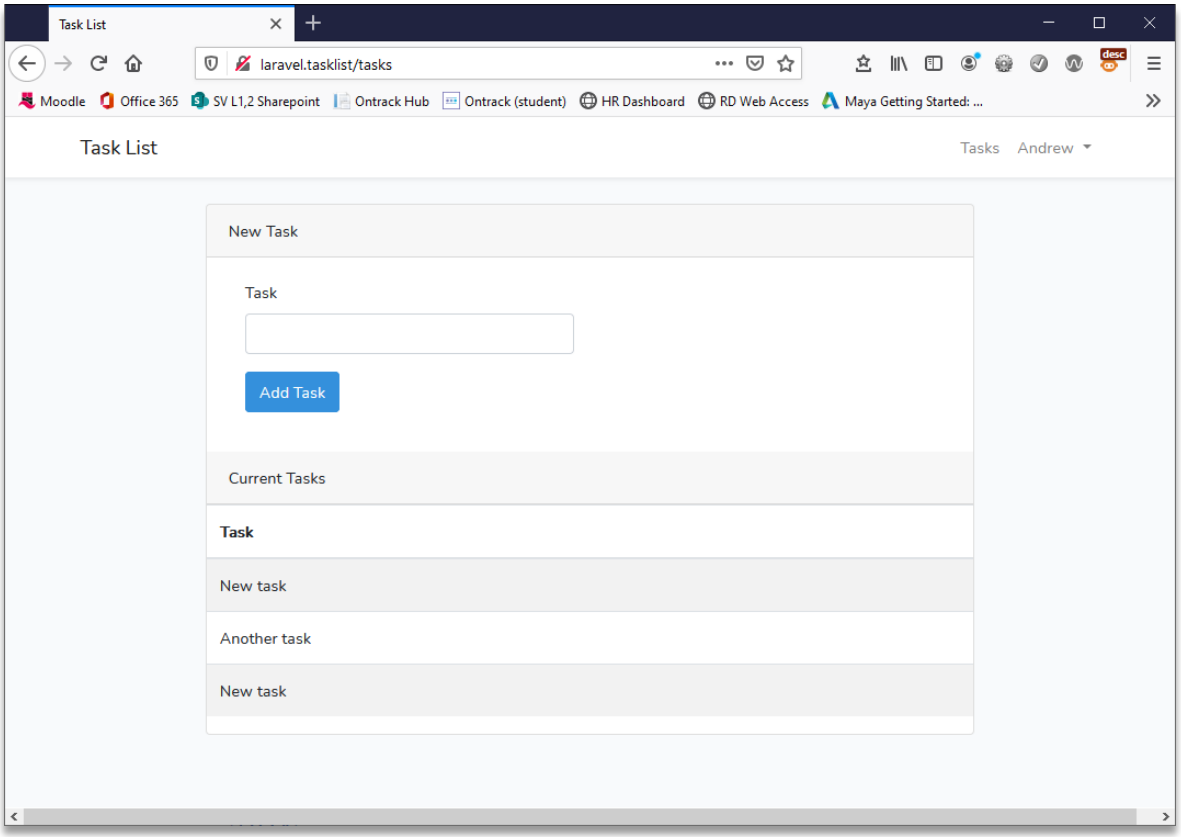
Displaying the tasks

- Next, add the highlighted code to implement the table itself

```
35              <!-- current tasks -->
36              @if (count($tasks) > 0)
37                  <div class="card-header">Current Tasks</div>
38                  <div class="panel-body">
39                      <table class="table table-striped task-table">
40                          <thead>
41                              <th colspan="2">Task</th>
42                          </thead>
43                          <tbody>
44                              @foreach ($tasks as $task)
45                                  <tr>
46                                      <td class="table-text"><div>{{ $task->name }}</div></td>
47                                      <td>
48                                          <!-- TODO: delete button -->
49                                      </td>
50                                  </tr>
51                              @endforeach
52                          </tbody>
53                      </table>
54                  </div>
55              @endif
56          </div>
57      </div>
58  </div>
59  @endsection
```

Testing the table

- Open your browser and refresh the site, you should now be able to enter and see all tasks for the currently logged in user



Adding a delete button

- This is not as simple as it might at first appear
- What is relatively simple is adding the HTML to **index.blade.php**
- Replace the TODO comment with the highlighted code
- Explanations are the notes pane

```
44                          @foreach ($tasks as $task)
45                              <tr>
46                                  <td class="table-text"><div>{{ $task->name }}</div></td>
47                                  <td>
48                                      <!-- delete button -->
49                                      <form action="/task/{{ $task->id }}" method="post">
50                                          {{ csrf_field() }}
51                                          {{ method_field('DELETE') }}
52                                          <button class="btn btn-danger">Delete Task</button>
53                                      </form>
54                                  </td>
55                              </tr>
56                          @endforeach
```

Adding the destroy method

- Now we have a delete button and a route which points this to the **destroy** method of the **TaskController** which we should now add
- Without adding any additional code, Laravel would inject the given task ID into the **TaskController@destroy** method
- Add the following highlighted method definition to **TaskController**

```
48          return redirect('/tasks');
49      }
50      /**
51       * Destroy the given task
52       *
53       * @param Request $request
54       * @param Task $task
55       * @return Response
56       */
57      public function destroy(Request $request, Task $task) {
58
59          //
60      }
```

Authorisation

- Now, we have a Task instance injected into our destroy method, however, we have no guarantee that the authenticated user actually "owns" the given task*
- We will use Laravel's authorization capabilities to make sure the authenticated user actually owns the Task to be deleted
- Laravel uses "policies" to organize authorization logic into simple, small classes
- Typically, each policy corresponds to a model.
- Create a **TaskPolicy** using the **XAMPP Shell** and entering the following **Artisan** command
  - **php artisan make:policy TaskPolicy**

```
User@DESKTOP-3LR8GQL c:\xampp\htdocs\LaravelTaskList
# php artisan make:policy TaskPolicy
Policy created successfully.
```

Coding the TaskPolicy

- A **TaskPolicy** file has been created for us in **app/Policies**
- Presently this only contains a constructor method which we will not need
- Replace the constructor with the code highlighted below, do not forget to add the us[illegible]ve the file

```php
<?php

namespace App\Policies;

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

class TaskPolicy {
    use HandlesAuthorization;

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

Associating Task and TaskPolicy

- At present, these two classes are unaware of each other but we can fix this in **app/Providers/AuthServiceProvider** by adding a line to the **$policies** list
- Open the above file in Visual Code, locate the $policies list and add the highlighted line, save when finished

```
 8  class AuthServiceProvider extends ServiceProvider
 9  {
10      /**
11       * The policy mappings for the application.
12       *
13       * @var array
14       */
15      protected $policies = [
16          // 'App\Model' => 'App\Policies\ModelPolicy',
17          'App\Task' => 'App\Policies\TaskPolicy',
18      ];
```

Authorising the destroy action

- Now that we have our Policy in place let's use it
- Laravel Controllers utilise an **AuthorizeRequests** trait which allows them to take advantage of an **authorize** method
- Open the **TaskController** in Visual Code and add the highlighted code to the destroy method

```
51      /**
52       * Destroy the given task.
53       *
54       * @param  Request  $request
55       * @param  string  $taskId
56       * @return Response
57       */
58      public function destroy(Request $request, Task $task) {
59          // check the the user owns the task
60          $this->authorize('destroy', $task);
61
62      }
```

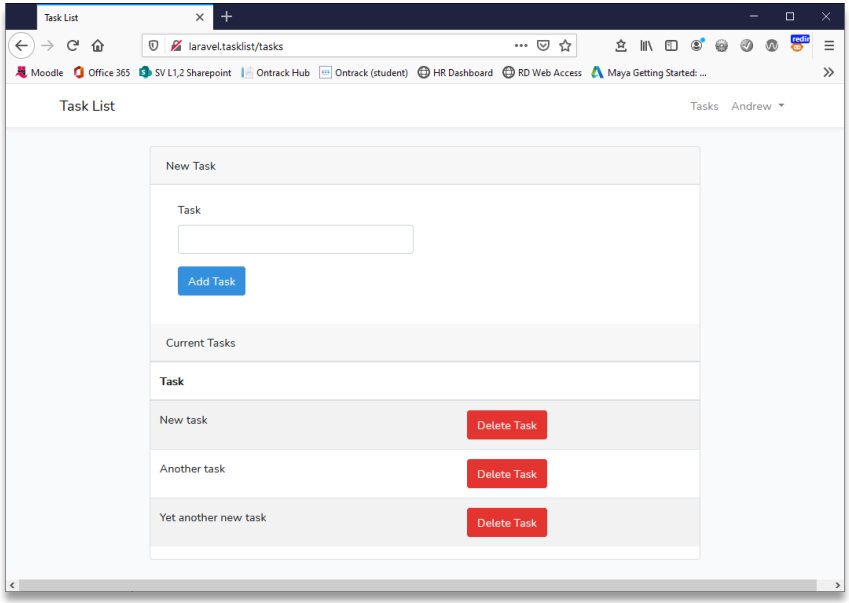
Deleting the Task

- We are now happy that the task belongs to this user
- We use Eloquent's **delete** method to remove the task
- Finally, we re-direct back to the **/tasks** view
- Add the code highlighted below

```
51      /**
52       * Destroy the given task.
53       *
54       * @param  Request  $request
55       * @param  string  $taskId
56       * @return Response
57       */
58      public function destroy(Request $request, Task $task) {
59          // check the the user owns the task
60          $this->authorize('destroy', $task);
61
62          $task->delete();
63
64          return redirect('/tasks');
65      }
66  }
```

Final testing

- Make sure that all files are saved
- Refresh your web browser and test the application
- You should be able to add users and for each user you should be able to see the tasks that only they have entered
- Each user should be able to delete their tasks



Project review

- This has been a weighty introduction to Laravel which has covered a lot of core functionality
- MVC architectural aspects
- Routing
- Blades
- Eloquent models and implicit model bindings
- Authorisation, Validation and Policies
- Method spoofing*
- All of this is useful for building robust server-side web applications
- A good deal of it is also useful when using Laravel to drive an API (Eloquent, Models, Routing, Controllers, Policies etc)
- Please make sure that you complete this project as you will need it next week when we turn this application into a Task API

BSc Applied Computing

Session review

Recap

- Recap OOP PHP
  - Last exercise progress
  - OOP fundamentals (class, namespace and use statements)
  - Frameworks are OOP
  - Frameworks are the easiest way of implementing an API
- Introduction to Laravel
  - Server-side OOP PHP framework for both web applications and APIs
  - Object-relational mapping (ORM)
  - Composer and package managers
  - Tour of a Laravel project
- Laravel task list project
  - Project setup
  - Project development workshop

Any questions?

- Next…
- Modifying the task list application to a web service API