

Criterion C - Development

Virtolio is a solution to the demand for an accurate, reliable and intuitive stock market simulator. Virtolio heavily employs the Eloquent ORM or object relational mapper in order to function. An ORM, in this case, allows Virtolio to fetch items such as the user or the user's portfolio as an object, and use that object within the program. This allows database objects to be manipulated with ease in controllers (which are called by routes).

1) Routes and Middleware

The first action done by the program is routing. After Laravel itself is bootstrapped, then it is waiting for calls to different routes.

```
<?php

use Virtolio\Models\Portfolio;
use Illuminate\Http\Request;
use Illuminate\Http\Input;

/**
 * Index pages
 */
Route::get('/', [
    'uses' => '\Virtolio\Http\Controllers\HomeController@index',
    'as' => 'home',
]);

Route::group(['middleware' => ['web']], function () {

    /**
     * Authentication
     */
    Route::get('/signup', [
        'uses' => '\Virtolio\Http\Controllers\AuthController@getSignup',
        'as' => 'auth.signup',
        'middleware' => ['guest'],
    ]);

    Route::post('/signup', [
        'uses' => '\Virtolio\Http\Controllers\AuthController@postSignup',
        'middleware' => ['guest'],
    ]);
});
```

Figure 1-1

When a user is directed or lands on a certain route, a specific controller is called. For example, when the user lands on the home page or root of the website (or '/'), then:

- Route file will call HomeController (specifically index function of controller)

- 'as' parameter allows a route name to be set (links can be generated by referencing name rather than entire URL)

Further down, the Route facade is again called, this time calling the group enclosure.

- Group encloses members of the web middleware (middleware runs before the controller)
- web middleware is defined to include things such as CSRF token, allowing token validation to be used across the group

2) Controllers

The routes file picks up different URLs and directs them to the appropriate controller.

- Each controller can then interact with the database via the setup models

Figure 2-1 illustrates the get method of the Portfolio controller.

```
1  <?php
2
3  namespace Virtolio\Http\Controllers;
4
5  use Illuminate\Support\Facades\Redirect;
6  use Illuminate\Support\Facades\URL;
7
8  use Input;
9  use Auth;
10 use Validator;
11 use Virtolio\Models\User;
12 use Virtolio\Models\Item;
13 use Virtolio\Models\Portfolio;
14 use Illuminate\Http\Request;
15
16 class PortfolioController extends Controller {
17
18     /*
19     * @return portfolios page with the user object
20     */
21     public function getPortfolios() {
22
23         $user = Auth::user();
24
25         return view('portfolios.index')
26             ->with('user', $user);
27     }
28 }
```

Figure 2-1

Nearly all the controller extend the base Laravel controller. Furthermore, they use or import different functionality, such as the Input or Auth classes. In Figure 2-1, the `getPortfolios()` method can be seen.

- Authenticated user is grabbed and stored in a variable
- The function then returns a specific view, in this case the portfolios index view, with the authenticated user.
- Note that there is no need to grab individual portfolios or items, because that can be done by calling `$user->portfolios()`, as the `$user` is already identified

Figure 2-2 is another example of a controller, in this case a post method for the user signup.

```

/*
takes a request object and validates it
upon successfull validation, a new user is created
@param request
@return redirect user to home with an info flash
*/
public function postSignup(Request $request) {
    $this->validate($request, [
        'email' => 'required|unique:users|email|max:255',
        'username' => 'required|unique:users|alpha_dash|max:20',
        'password' => 'required|min:6',
        'password_again' => 'required|same:password',
    ]);

    User::create([
        'email' => $request->input('email'),
        'username' => $request->input('username'),
        'password' => bcrypt($request->input('password')),
    ]);

    return redirect()->route('home')->with('info', 'Your account has been create and you can now sign in.');
```

Figure 2-2

This method belongs the user controller. It takes the request parameter, which is passed by the user through the form on the signup view. It validates the request, and uses the User facade (belonging to `Virtolio\Models\User`) to create a new user within the database.

3) Models

Models are a way of interacting with the database in an intuitive manner.

Figure 3-1 showcases the user model that was built.

```
<?php

namespace Virtolio\Models;

use Virtolio\Models\Status;
use Illuminate\Auth\Authenticatable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Foundation\Auth\Access\Authorizable;
use Illuminate\Contracts\Auth\Authenticatable as AuthenticatableContract;
use Illuminate\Contracts\Auth\Access\Authorizable as AuthorizableContract;

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

    protected $table = 'users';

    protected $fillable = [
        'username',
        'email',
        'password',
        'first_name',
        'last_name',
        'location'
    ];

    protected $hidden = [
        'password',
        'remember_token'
    ];
}
```

Figure 3-1

Models are firstly namespaced under Virtolio\Models, and are called by this naming scheme throughout the application. The protected variable of \$table specifies what is being referenced by this model. Some of the table columns (username, email, password, etc.) are part of the fillable protected array. This allows Eloquent to protect against mass assignment.

Within models, relationships are defined as methods.

Figure 3-2 explores methods within the user model.

```
public function portfolios() {  
    return $this->hasMany('Virtolio\Models\Portfolio', 'user_id');  
}
```

Figure 3-2

The `portfolios()` method returns `$this`, which is the current user object, and the many portfolios of a user. The `hasMany()` function expects the first parameter to be the model of the other table. The second parameter is the foreign key. Essentially, there is column in the portfolio table called `user_id`. Each portfolio has a `user_id`, and thus belongs to that user. However, rather than make several database queries to fetch portfolios, by simply using the above structure, the same results can be achieved by calling `$user->portfolios()`.

```

<?php

namespace Virtolio\Models;

use Auth;
use Illuminate\Database\Eloquent\Model;

class Portfolio extends Model {

    protected $table = 'portfolios';

    protected $fillable = [
        'name',
        'starting_cash',
        'current_cash',
        'commission_type',
        'commission_amount',
        'commission_type_string',
    ];

    public function user() {
        return $this->belongsTo('Virtolio\Models\User', 'user_id');
    }

    public function items() {
        return $this->hasMany('Virtolio\Models\Item', 'portfolio_id');
    }

    public function stocks() {
        return $this->hasMany('Virtolio\Models\Item', 'portfolio_id')->where('type', 1);
    }

}

```

Figure 3-3

Figure 3-3 showcases the portfolio model. It also has several functions. For example, the user method references the user table, with the local key of user_id matching the primary key of the users table. This also allows us to find information about the user by way of their portfolio. For example, if we have a portfolio, we can do \$portfolio->user()->first_name. This would return the name of the user who has the portfolio that we looked up.

MySQL 5.6.28-0ubuntu0.14.04.1 virtolio.dev/virtolio/virtolio/portfolios

Select Database: virtolio

Structure Content Relations Triggers Table Info Query

Search: Filter

id	user_id	name	starting_cash	current_cash	commission_type	commission_type_string	commission_amount	created_at	updated_at
5	1	First Virtolio	30000.00	28813.94	2	percent commission	2.00	2016-02-27 17:45:34	2016-02-27 17:45:54
6	1	Another One	40000.00	26437.20	3	fixed commission	20.00	2016-02-27 17:46:13	2016-02-27 17:47:16

TABLE INFORMATION

- created: 2016-02-17
- engine: InnoDB
- rows: 2
- size: 16.0 KiB
- encoding: utf8
- auto_increment: 7

2 rows in table

Figure 3-4

Figure 3-4 is a screen capture of the portfolios table. Each portfolio entry has an id, but also has a user_id. This corresponds to the id of the respective user.

4) Migrations

Migrations are a way of defining a schema for a database table and provide a means of version control.

Figure 4-1 shows the migration made for the portfolios table.

```
<?php

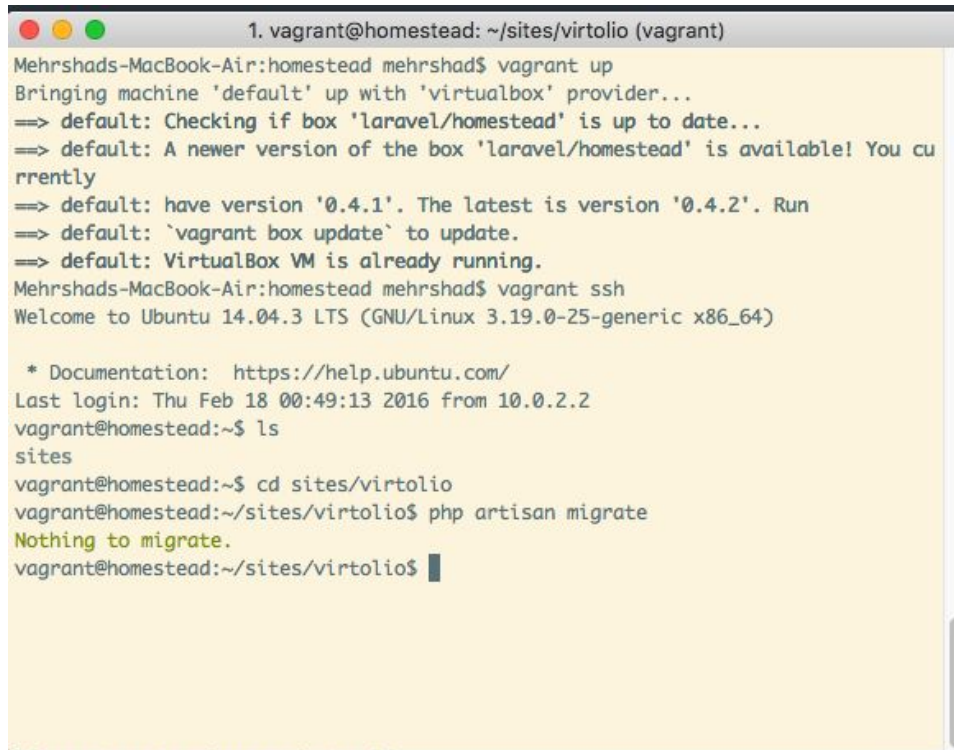
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreatePortfoliosTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('portfolios', function(Blueprint $table) {
            $table->increments('id');
            $table->integer('user_id');
            $table->string('name');
            $table->double('starting_cash', 20, 2);
            $table->double('current_cash', 20, 2)->nullable();
            $table->integer('commission_type');
            $table->string('commission_type_string');
            $table->double('commission_amount', 5, 2)->nullable();
            $table->timestamps();
        });
    }

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

By calling on the up() method, the portfolios table is created with the various columns and with the parameters listed.

Migrations are run through the terminal, using the php artisan command.

A terminal window titled '1. vagrant@homestead: ~/sites/virtolio (vagrant)' showing a series of commands and their outputs. The commands include 'vagrant up', 'vagrant ssh', and 'php artisan migrate'. The output shows the Vagrant process bringing up the machine, checking for updates, and then logging into the Ubuntu VM. Finally, the 'php artisan migrate' command is run, resulting in 'Nothing to migrate.'

```
1. vagrant@homestead: ~/sites/virtolio (vagrant)
Mehrshads-MacBook-Air:homestead mehrshad$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Checking if box 'laravel/homestead' is up to date...
==> default: A newer version of the box 'laravel/homestead' is available! You currently
have version '0.4.1'. The latest is version '0.4.2'. Run
`vagrant box update` to update.
==> default: VirtualBox VM is already running.
Mehrshads-MacBook-Air:homestead mehrshad$ vagrant ssh
Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 3.19.0-25-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
Last login: Thu Feb 18 00:49:13 2016 from 10.0.2.2
vagrant@homestead:~$ ls
sites
vagrant@homestead:~$ cd sites/virtolio
vagrant@homestead:~/sites/virtolio$ php artisan migrate
Nothing to migrate.
vagrant@homestead:~/sites/virtolio$
```

Figure 4-1

New migrations can be made using 'php artisan make:migrate'. This would create a file in the migrations folder under the database folder, and can be used to create a schema to deploy.

5) Views

The components seen thus far have been purely logical, back-end items. The actual interface that the user sees is made up of front-end components such as HTML and CSS3. In order for the views to access the information that the back-end has, the blade templating engine has been used.

```
@foreach ($user->portfolios as $portfolio)

    @if ($portfolio == $user->mostRecentPortfolio())
        <div class="tab-pane active" id="tab_{{ $user->mostRecentPortfolio()->id }}">
    @else
        <div class="tab-pane" id="tab_{{ $portfolio->id }}">
    @endif
    <div class="row">
        <div class="col-md-6">
            <h4>Currently viewing "{{ $portfolio->name }}" portfolio. </h4>
        </div>
        <div class="col-md-6">

            <h6 class="pull-right"> Last modified {{ $portfolio->updated_at }}.
            Starting balance: {{ number_format($portfolio->starting_cash, 2, '.', ',') }}. Commision: {{ $portfolio->commission_type_string }}.
            Amount:
            @if ($portfolio->commission_amount == 0)
                Zero.
            @else
                {{ $portfolio->commission_amount }}.
            @endif
            </h6>

        </div>
    </div>
    <br>
    <div class="row portfolio_row" data-portfolio="{{ $portfolio->id }}">=
    <hr>

    <div class="row">=

    <hr>

    <div class="row">=

</div><!-- /.tab-pane -->
@endforeach
```

Figure 5-1

The screen capture above shows the logic that can be used within a view, using the blade engine.

- The routes file would call on a controller, that would have logic and pull results from a model. Those results were then put in a variable and passed to the view.
- In this case, the portfolio controller for the method of get returns the portfolio view, with the variable of user that holds all the user's information. As this information is passed to the view, it can be called using a foreach loop, and items can be iterated through.

- Relationships for the user have already been defined, so it is possible to call on the \$user->portfolios rather than create a separate database query. The user has many portfolios, so each one is iterated through, and elements such as the id, name, and cash are displayed to the user.

Front-end scripts, such as javascript (and ajax) are also contained within views.

Figure 5-2 showcases the ajax scripting found in the search page.

```
$(document).ready(function() {
    $('#getdata').on('click', function (e) {
        $('#status').html("Loading...");
        e.preventDefault();
        $request = $.ajax({
            type: "POST",
            url: '{ route('test.post') }',
            data: {'ticker': $('#query').val()},
            dataType: "JSON",
            cache: false,
            success: function(data) {
                $('#results-t').html("");
                if(data) {
                    var len = data.length;
                    var txt = "";
                    if(len > 0){
                        for(var i=0;i<len;i++) {
                            var attachment = 'stock/' + data[i].Symbol;

                            var url = "<a href=\" " + attachment + "\" class=\"btn btn-block btn-success btn-sm\">" + data[i].Symbol + " page</a>";

                            txt += "<tr><td>" + data[i].Symbol + "</td><td>" + data[i].Name + "</td><td>" + data[i].Exchange + "</td><td>" + url + "</td></tr>";
                        }
                        if(txt != "") {
                            $('#results').removeClass("hidden");
                            $('#results-t').append(txt);
                        }
                    }
                }
                $('#status').html("");
            },
            error: function(XMLHttpRequest, textStatus, errorThrown) {
                $('#status').html("An error has occurred. Please try again.");
            },
        });
    });
});
```

Figure 5-2

The search page allows a user to search for stocks and select them.

- If a POST request was made for every search and results loaded, it would be a tremendous amount of overhead, and would have too many page refreshes.
- By using AJAX, the user can search and get results populated almost immediately.

- The AJAX calls on a local route, which then forwards the request to an external market data API.
- The results are fetched and returned in the form of JSON, which are then iterated through and displayed as a table below the search box.
- With AJAX, there are still security and usability precautions taken place. For whatever reason, either if the route is not working or the external API is down, the AJAX request can return an error prompting the user to try again.

Word Count: 1060