Seminar 3 report

Internet Applications, ID1354

Niclas Fölster - nfolster@kth.se

Sun 3 December 2017

1 Introduction

This task involves improving the code-structure for the website and make it more streamlined. There are two mandatory tasks. The first one was a choice between implementing **MVC architecture without a framework**, the second was to **use a PHP framework**, but it must follow the MVC architectural pattern.

2 Literature Study

Since I used Laravel, I followed a guide on it by Jeffrey Way. I also used the Laravel documentation online.

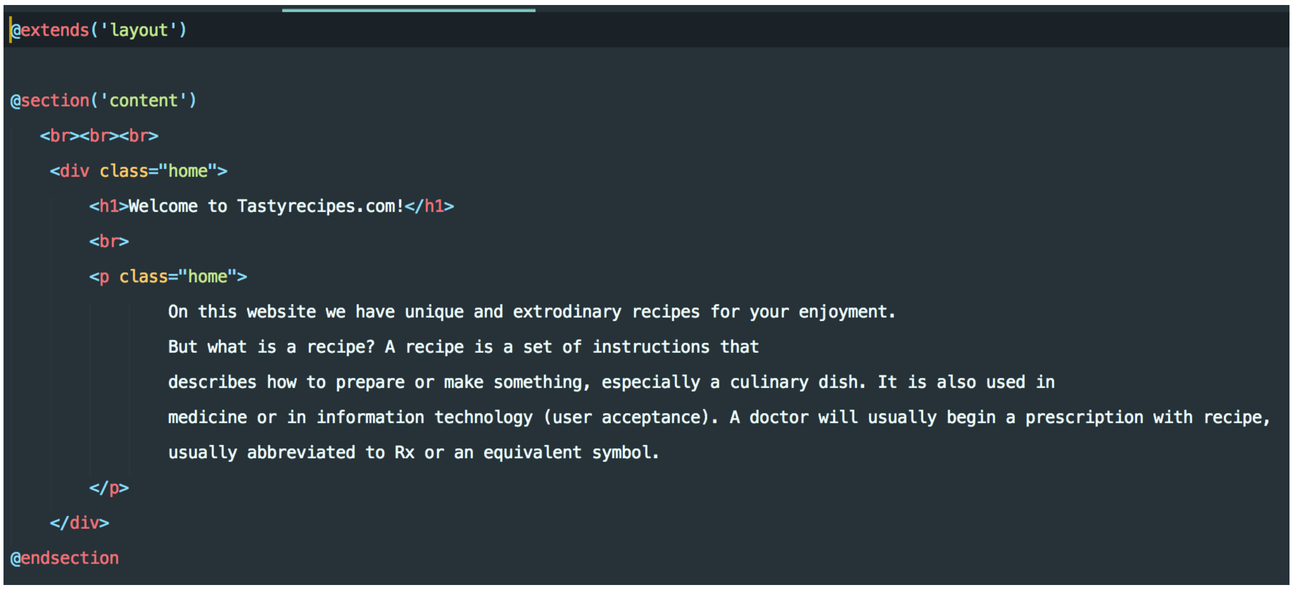
3 Method

I used the Laravel framework. So I followed a guide on how to Install it. I had to install several things through the terminal, including Composer and some other things. I also stopped using Mamp, and used Sequel Pro to manage my Database. I also added some packages to Sublime Text to make it easier to use with Laravel.

4 Result

I used Laravel, which is a very big framework, and tried to use it as good as I could considering the time I had. I do not understand the whole framework. Far from it. But I will do my best to explain what I do know.

Lets start with the views. Laravel uses something called Blade, which makes writing php-code in html documents very easy. For example instead of <?php echo 'hello world' ?> you can just write {{ hello world }}. I also removed duplicated code by using a layout page.



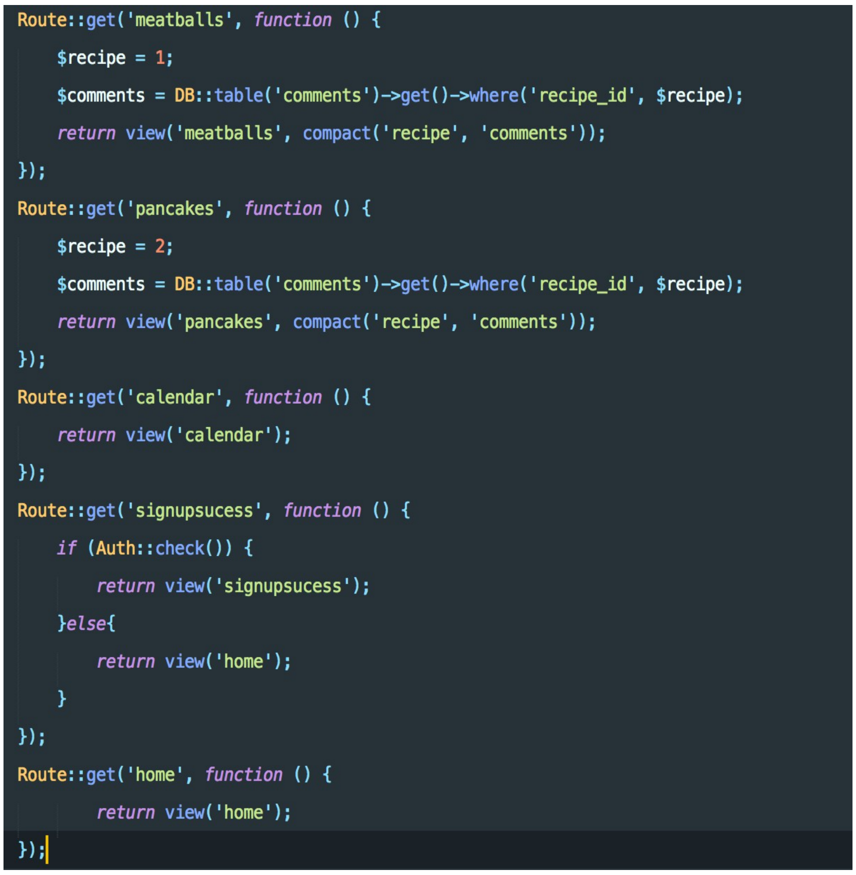
```
@extends('layout')

@section('content')
   <br><br><br>
    <div class="home">
        <h1>Welcome to Tastyrecipes.com!</h1>
        <br>
        <p class="home">
                On this website we have unique and extrodinary recipes for your enjoyment.
                But what is a recipe? A recipe is a set of instructions that
                describes how to prepare or make something, especially a culinary dish. It is also used in
                medicine or in information technology (user acceptance). A doctor will usually begin a prescription with recipe,
                usually abbreviated to Rx or an equivalent symbol.
        </p>
    </div>
@endsection
```

Illustration 1: extending a layout

So in the layout-page, I just specify where the section 'content' shall appear with @yield('content')

With this I do not have to have the html-skeleton and the navbar, and also the side menu on the recipe page more than once, which means no duplicated code! All the views are now in a views-folder.

All my links now go through routes in in my webs.php.

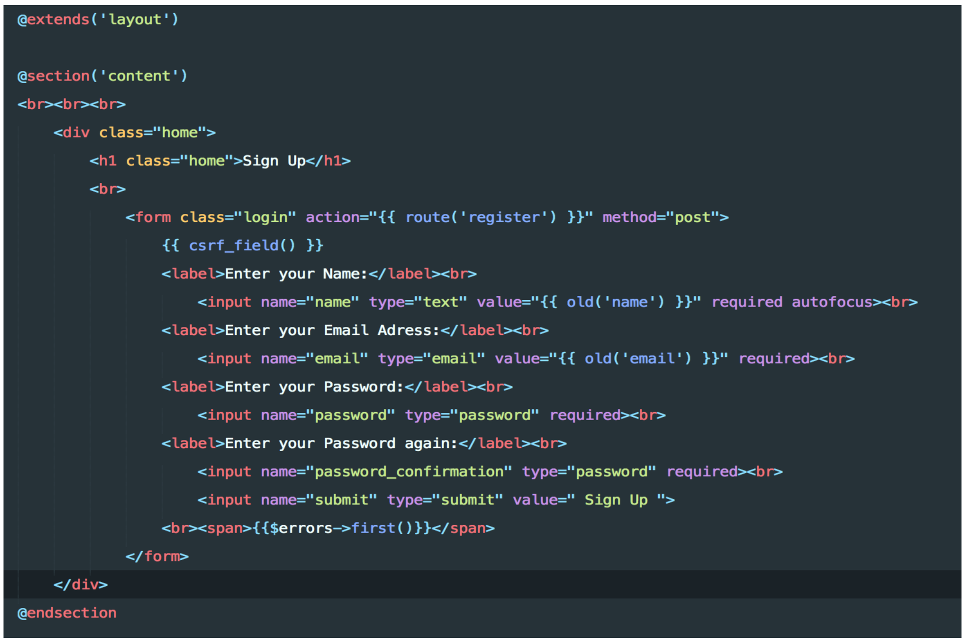


```php
Route::get('meatballs', function () {
    $recipe = 1;
    $comments = DB::table('comments')->get()->where('recipe_id', $recipe);
    return view('meatballs', compact('recipe', 'comments'));
});
Route::get('pancakes', function () {
    $recipe = 2;
    $comments = DB::table('comments')->get()->where('recipe_id', $recipe);
    return view('pancakes', compact('recipe', 'comments'));
});
Route::get('calendar', function () {
    return view('calendar');
});
Route::get('signupsucess', function () {
    if (Auth::check()) {
        return view('signupsucess');
    }else{
        return view('home');
    }
});
Route::get('home', function () {
        return view('home');
});
```

Illustration 2: Part of Webs.php

With Laravel I do not have to specify the whole url, which is very nice. I can also easily send data to the views here.

For the authentication, I used Laravels premade authentication views, controllers and prefabs. This made it very easy and secure to handle log in and error handling. I could use Auth::check() to determine if the current user is authenticated. It also was automatically connected to the users table in the database. So I only did a view with a form in the right folder.

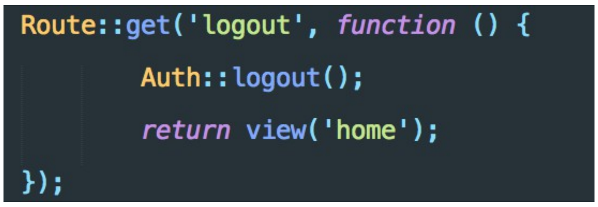


```html
@extends('layout')

@section('content')
<br><br><br>
    <div class="home">
        <h1 class="home">Sign Up</h1>
        <br>
            <form class="login" action="{{ route('register') }}" method="post">
                {{ csrf_field() }}
                <label>Enter your Name:</label><br>
                    <input name="name" type="text" value="{{ old('name') }}" required autofocus><br>
                <label>Enter your Email Adress:</label><br>
                    <input name="email" type="email" value="{{ old('email') }}" required><br>
                <label>Enter your Password:</label><br>
                    <input name="password" type="password" required><br>
                <label>Enter your Password again:</label><br>
                    <input name="password_confirmation" type="password" required><br>
                    <input name="submit" type="submit" value=" Sign Up ">
                <br><span>{{$errors->first()}}</span>
            </form>
    </div>
@endsection
```

Illustration 3: Registration

With this I could also use Auth::user()->name, to get the name of the authenticated user. For this seminar I changed what users had too. Instead of a username, you log in with an email.

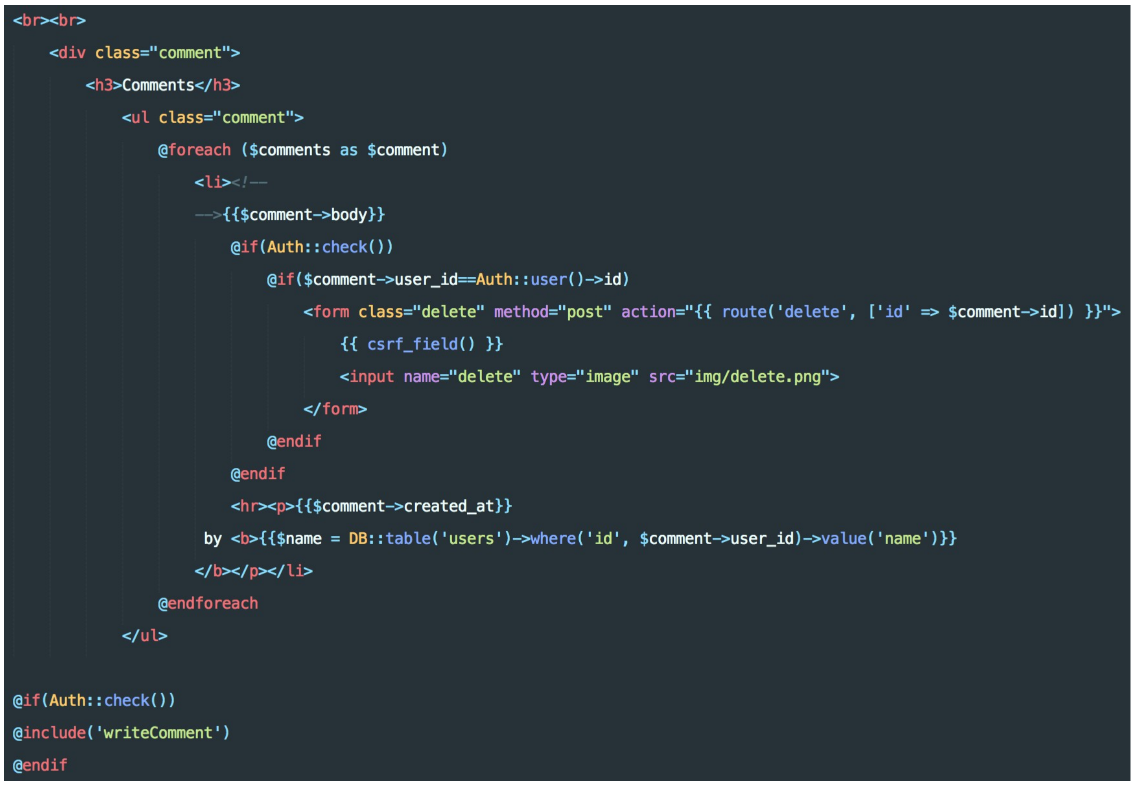
So when you log out you just get routed to log out and then Auth:logout() is called.



```
Route::get('logout', function () {
        Auth::logout();
        return view('home');
});
```

Illustration 4: Logout

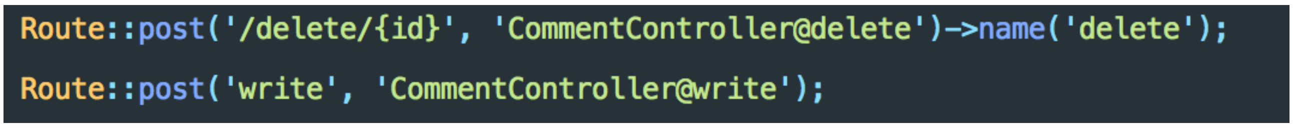
For the Comments they first get database data when routing (see Illustration 1).



```
<br><br>
    <div class="comment">
        <h3>Comments</h3>
            <ul class="comment">
                @foreach ($comments as $comment)
                    <li><!--
                    -->{{$comment->body}}
                        @if(Auth::check())
                            @if($comment->user_id==Auth::user()->id)
                                <form class="delete" method="post" action="{{ route('delete', ['id' => $comment->id]) }}">
                                    {{ csrf_field() }}
                                    <input name="delete" type="image" src="img/delete.png">
                                </form>
                            @endif
                        @endif
                        <hr><p>{{$comment->created_at}}
                     by <b>{{$name = DB::table('users')->where('id', $comment->user_id)->value('name')}}
                    </b></p></li>
                @endforeach
            </ul>

@if(Auth::check())
@include('writeComment')
@endif
```

Illustration 5: Comments.

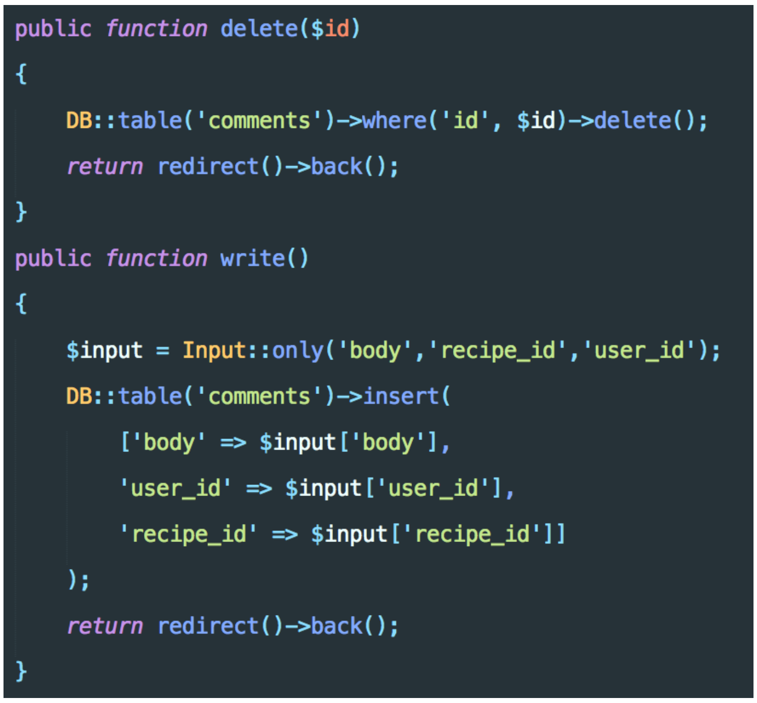
Then as you see above, when you press the delete button, you get routed to delete, which takes us to our routes, and from there to the controller. This time around I also have ids for both users and comments, to make it more secure, as per your recommendation.



```
Route::post('/delete/{id}', 'CommentController@delete')->name('delete');
Route::post('write', 'CommentController@write');
```

Illustration 6: Controller calls for comments

Here in Webs.php the Controller is called.



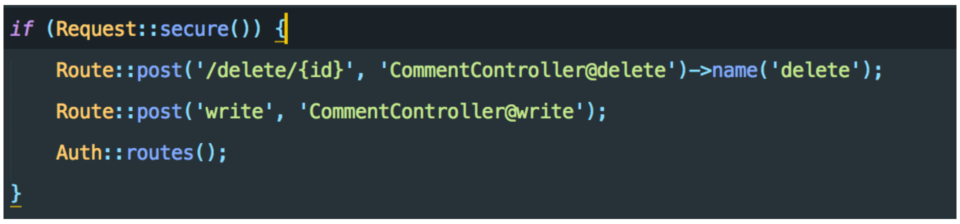
```
public function delete($id)
{
    DB::table('comments')->where('id', $id)->delete();
    return redirect()->back();
}
public function write()
{
    $input = Input::only('body','recipe_id','user_id');
    DB::table('comments')->insert(
        ['body' => $input['body'],
        'user_id' => $input['user_id'],
        'recipe_id' => $input['recipe_id']]
    );
    return redirect()->back();
}
```

Illustration 7: CommentController functions.

Here in the Comment controller, comments are inserted and deleted.

Security

HTTPS is used when sending passwords etc from client to server, otherwise the password is sent as raw text, which is very unsecured. I think I have added HTTPS check by adding the Request::secure(). This checks if HTTPS is used. This way all comments and authentication goes through this check. So if HTTPS is not used, it will not go through.

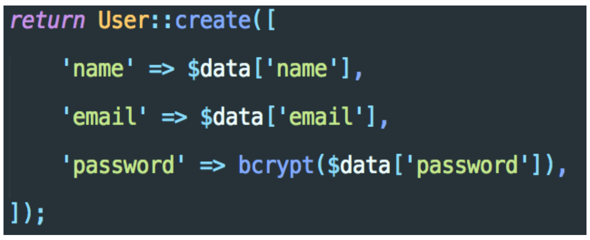


```
if (Request::secure()) {
    Route::post('/delete/{id}', 'CommentController@delete')->name('delete');
    Route::post('write', 'CommentController@write');
    Auth::routes();
}
```

Illustration 8: HTTPS check

1. Password Encryption

I made sure laravels pre-created authentication used password encryption, which they did.

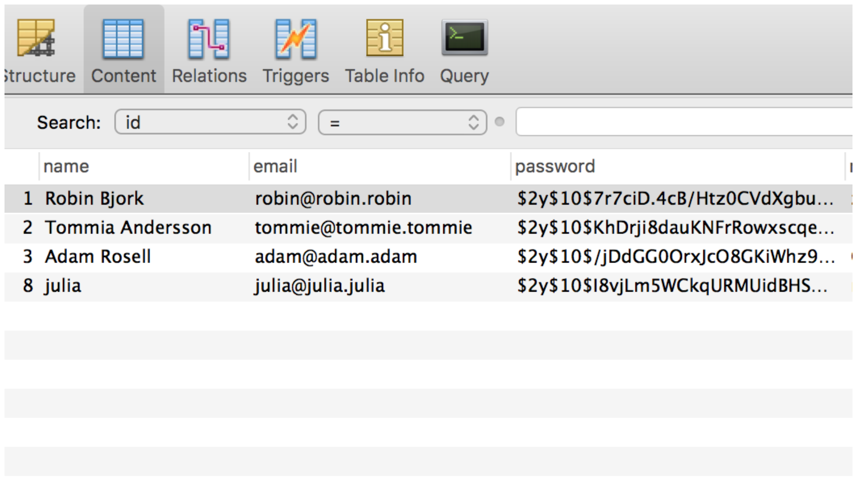


```
return User::create([
    'name' => $data['name'],
    'email' => $data['email'],
    'password' => bcrypt($data['password']),
]);
```

Illustration 9: bcrypt function for password

They use bcrypt, which is a standard and very good password hashing function. For example is it resistant to brute-force search attacks.

So in the database, only the hashkeys are shown.

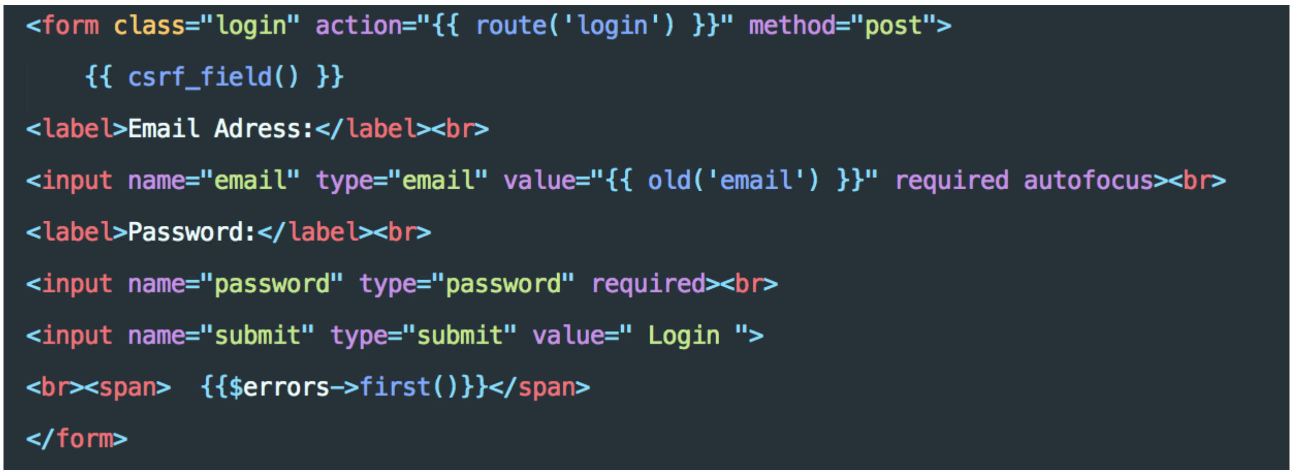


	name	email	password
1	Robin Bjork	robin@robin.robin	$2y$10$7r7ciD.4cB/Htz0CVdXgbu…
2	Tommia Andersson	tommie@tommie.tommie	$2y$10$KhDrji8dauKNFrRowxscqe…
3	Adam Rosell	adam@adam.adam	$2y$10$/jDdGG0OrxJcO8GKiWhz9…
8	julia	julia@julia.julia	$2y$10$I8vjLm5WCkqURMUidBHS…

Illustration 10: database users table.

2. Cross Site Scripting

In Laravel it is required to, whenever you use a Form where visitors can enter data, to use CSRF (cross-site request forgery) tokens.



```
<form class="login" action="{{ route('login') }}" method="post">
    {{ csrf_field() }}
<label>Email Adress:</label><br>
<input name="email" type="email" value="{{ old('email') }}" required autofocus><br>
<label>Password:</label><br>
<input name="password" type="password" required><br>
<input name="submit" type="submit" value=" Login ">
<br><span>  {{$errors->first()}}</span>
</form>
```

Illustration 11: csrf_field

This protects against both Cross-site Request Forgery and Cross-site Scripting. As they explain it:

Cross-site Request Forgery:

"Imagine a situation in which a malicious third-party crafts a special link (or a form masquerading as a link) which when clicked initiates a request to another site where you are registered and happen to be authenticated into (by way of a session cookie). Suppose this link endpoint performed a sensitive task such as updating your profile to include a spam message. Because you are authenticated, the site will presume the request is indeed coming from you, and update the profile accordingly.

CSRF (cross-site request forgery) tokens are used to ensure that third-parties cannot initiate such a request. This is done by generating a token that must be passed along with the form contents. This token will then be compared with a value additionally saved to the user session. If it matches, the request is deemed valid, otherwise it is deemed invalid."

Cross-site Scripting:

“Laravel’s `{{}}` syntax will automatically escape any HTML entities passed along via a view variable. This is a very big deal, considering that a malicious user might pass the following string into a comment or user profile:

```
My list <script>alert("spam spam spam!")</script>
```

If this string were allowed to be saved to the database without filtering, and then subsequently displayed in a web page without escaping, it would in fact display an annoying alert window. This is an example of an attack known as cross-site scripting. In the grand scheme of things this is but a minor annoyance compared to more sophisticated attacks which might prompt the user to supply some sensitive information via a JavaScript modal which was subsequently sent to a third-party website.

Fortunately, when a variable is rendered within the `{{}}` escape tags, Laravel would instead render the string like so, thus preventing the possibility of cross-site scripting:

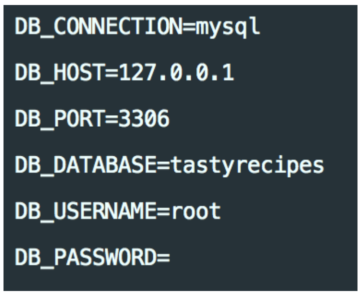
```
My list &lt;script&gt;alert("spam spam spam!")&lt;/script&gt;”
```

In Illustration 5 for example, you can see that all data that users entered are inside these `{{}}` .

3. Database Security

The Database security part consists mainly of two parts. Database Access Control, and preventing SQL-injections.

Database Access Control is predefined in laravel. There is a single secure plain-text file named .env to store all these kinds of passwords etc. This makes Database Access Control very simple with Laravel.



```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=tastyrecipes
DB_USERNAME=root
DB_PASSWORD=
```

.env file

Laravels Eloquent ORM(Object-relational Mapping) uses PDO(PHP Data Objects) parameter binding to avoid SQL injection. Parameter binding ensures that evil users can’t pass in query data which could modify the query’s intent.

Git link: https://github.com/nfolster/ID1354_Projekt

5 Discussion

I have completed both the mandatory tasks, and one of the optional tasks, since I am using a Database. Using and learning Laravel was overwhelming and still is. I can see the cons of using such a large framework. I could not follow the tutorials fully since there was many extra functionalities and things I did not need for this small website, so I had do look up many things individually. It took a long time and much work to learn all the necessary parts to complete this assignment.

Three very positive things with it however was Blade, which made writing php and html code very simple and efficient. Second was that I could create pre made functionality for authentication. I only had to implement my design into it. This saved me a lot of work. The third was the security. Many security measures were automatically implemented, or it existed pre-defined functions for it.

Most of the problems I faced with was with things I did not know or had time to learn about the framework. This resulted in a few not-so-good-looking fixes. Nothing impactful however.

6 Comments About the Course

I will write this in Swedish. Denna del är helt klart den svåraste hittils. Men det var väldigt lärorikt och mycket intressant. Jag la ner ca 30-40h på detta.