Nature and Travel application

Purpose of the application

The application was designed for nature and travel enthusiasts. The user can explore different places around the world interact with them, comment, and see other people's thought about their recent travels. The application also, provides helpful news about nature, science and health.

Requirements

Below it is a table with detailed information about requirements using the MOSCOW method.

Requirements	MOSCOW
Sign Up	Must have
Log in/ Log out	Must have
Post your experiences about recent travels	Must have
Show countries information	Must have
Show places to visit	Must have
Like a specific country	Must have
Undo the like for a specific country	Must have
Recommend a place to others	Could have
Comment about a specific country	Must have
Display all comments in chronological order	Must have
Sort countries by continents	Must have
Sort countries by popularity	Must have
Dynamically update the popularity according to user's like	Must have
Show a heavily personalized dashboard	Must have
Display all user's comment in dashboard	Must have
Display all post made by users latest first	Must have
Display all liked countries according to user in dashboard	Must have
Display a map for a specific country	Must have
Send feedbacks and suggestion of how to improve the app	Must have
Rate the app	Must have
Show all feedbacks when a user logs in with admin privileges	Must have
Implementation of the admin panel	Must have
Show the average rating in admin panel	Must have
Create a country in admin panel	Must have
Create a place to visit in admin panel	Must have
Display all users in admin panel	Must have
Ability to delete a user via the admin panel	Must have
Ability to promote a user to admin	Must have
Ability to demote a user from admin to a regular one	Must have

Implementation of the news room tab	Should have
Show the top trending news regarding nature	Should have
Show the top trending news regarding science-health	Should have
Ability to save an article to read later	Should have
Show all saved articles per user in dashboard	Should have
Implementation of access control and security	Must have
Filtering HTTP requests using middleware	Must have
Chat rooms per country	Won't have (this time)
Display top hotels to stay	Won't have (this time)
Display the weather	Won't have (this time)

Installation

There are two possible ways to install the application to test the results. Deploying the website to a live server with a domain name is beyond the scope of the assignment.

First way

This is the safest off all and I totally recommend this. Because I built the app using Laravel framework it is required to install the PHP dependency manager "Composer". You can download and install via https://getcomposer.org/ Once done, import the .sql file to the local MySQL server and run it. Note: You must create a database "travelapp" and then import the sql dump file to create the tables. For the database connection you need to provide the credentials in the .env file. I had my password as 123456 but change to match yours. After that, open the terminal (if Windows open the cmd) and navigate inside the root directory. Then run the command "php artisan serve" and hit enter. The local development server will start and it can be accessed in http://127.0.0.1:8000

```
C:\Users\elkap\Desktop\travelapp>php artisan serve
```

Second way

This is the unsafe way but I optimized the app to work for faster testing. It needs Apache server and Maria DB. It can be both downloaded using XAMPP. Note that it requires the latest release of XAMPP that supports PHP > 7.2. In general the application works if the PHP server meets the following requirements.

- PHP >= 7.1.3
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension
- XML PHP Extension

- Ctype PHP Extension
- JSON PHP Extension

After that, simple deploy the folder in xampp htdocs and run http://localhost/travelapp/public/

This has a huge security issue because if you go back one step i.e. http://localhost/travelapp/ all the directory is visible, but is a simple way to test without composer.

Technologies

Below is a table with all the technologies I used during the implementation process. After that a brief explanation why I chose this technology over others.

FROND END
HTML 5
Materialize CSS
JavaScript
BACK END
PHP
Laravel Framework
MySQL
Visual Studio Code
Git/GitHub
XAMPP

HTML 5

The standard web mark-up language. Because the application was built to run in browsers it was a must choice.

Materialize CSS

At first, I considered use Bootstrap as the main CSS framework for the application, but I chose to use a less popular one to experiment a lot. Bootstrap came with Laravel by default, but I override it immediately. Materialize use familiar Google UI practices called Material Design. It has a simple style looking yet beautiful and colorful.

JavaScript

At some point I wrote my own JavaScript code but mainly I used the Materialize JS to initialize components such as parallax and dropdown menus.

PHP/Laravel framework

It was a difficult decision to chose PHP over Python at first. Python code is simple looking yet powerful when security becomes an issue. It is more modern than PHP with great frameworks to work such as Django and Flask. But I think Laravel gives developer more freedom to work and it can be more customized than Python's frameworks. Plus, Laravel handles database migrations extremely well and supports many SQL distributions such as MySQL, PostgreSQL, SQLite and more.

MySQL

MySQL is the most popular and powerful database system in the world. I was familiar using MySQL with previous assignments, so I chose this rather than a new one. Great alternative was PostgreSQL.

Visual Studio Code

My main text editor by Microsoft. It is open source and provides great tools for faster coding. The marketplace also, has a great PHP debugger. Other alternatives, Atom and Sublime Text.

Git/GitHub

Every piece of code I wrote was uploaded at https://github.com/ekapsimalis/travelapp It is a source control manager and helped me see the changes occurred during the coding process. I could go back if something didn't work well when implementing new things. Also, it is a great platform to check how I built the project step by step.

XAMPP

The apache server I used to check if the project run smoothly. While all the work was tested by Laravel development server environment as described in the installation first way, I tried to make it work in apache too for faster results and testing.

Work flow

First week

In the first week I designed the basic look of the website. I deleted all pre-defined assets came with the Laravel installation such as Bootstrap CSS and JavaScript. Then I created the first model of the application Users. I created a table to represent the model in the database. At first I wanted only the basic fields to work for testing purpose. So, the primary key was an auto-increment id, then an email field and a password. After that I used the Laravel's authentication system to store the user's data into the session. Every time a user logs in or signs up the data will transfer to all pages inside the application until the user decides to log out. The sign up logic implemented as described below step by step

```
public function postSignUp(Request $request){

    $this->validate($request, [
        'username' => 'required|max:191',
        'email' => 'required|max:191|unique:users',
        'password' =>'required|max:191|'
    ]);
```

When the user submits a form to the sign up page the first step is to verify if the data passed is correct. The username must be filled in with max chars of 191 same as password. The email on

the other hand must be unique in the entire users table to prevent the same person have multiple accounts. When the validation returns TRUE then the data is stored in the database and logs in the user immediately

```
$email = $request['email'];
$password = bcrypt($request['password']);
$username = $request['username'];

$user = new User();
$user->email = $email;
$user->password = $password;
$user->username = $username;
$user->type = 'default';
$user->save();

Auth::login($user);

return redirect()->route('dashboard');
```

In the above code an instance of the class User was created with the data passed by the form as attributes. Then the user is saved in the database with the method `save()`. This method is a pre-defined method in Laravel and it is equivalent to SQL INSERT INTO statement. The type field will later determine if the user has administrative privileges. The `login()` method is similar to session start in plain PHP and finally when all succeeds the user is redirected to the dashboard view.

When a user has already an account he is prompted to provide the credentials for checking. The logic here is to search every row in the database to match the given data. If succeeds again the user is prompt to the dashboard.

```
if (Auth::attempt(['email' => $request['email'], 'password' => $request['password']])){
    return redirect()->route('dashboard');
}
```

Finally, the logout system works the similar way as the login method in Laravel. The only difference is that instead using session start it uses session destroy method in plain PHP.

```
public function getLogOut(){
    Auth::logout();
    return redirect()->route('home');
}
```

When the authentication system finally worked as expected, I proceed to make some dummy html pages for later holding all the data from the database. It is a common practice when building applications with user interaction, to first create the look of the page and then insert the data. The basic rule here is to create a master layout and then all the other pages will inherit the layout with the additional content. Therefore, I created a folder called layouts inside the resources. The basic structure looked like this.

```
<body id="background">
    <header>
        @include('inc._navbar')
    </header>
    <main>
        <div class="container">
            @include('inc._messages')
            @yield('content')
        </div>
    </main>
    <footer class="page-footer brown darken-3">
        @include('inc._footer')
    </footer>
    <script type="text/javascript" src="{{asset('js/materialize.min.js')}}"></script>
    <script type="text/javascript" src="{{asset('js/myscript.js')}}"></script>
/body>
```

Notice the two functions `include()` and `yield()`. The first one separates some basic content such as navbar or footer and import them in the main view. This is a good practice because if something needs change in i.e. navbar it doesn't need to change in all pages, but just in the _navbar partial view. All application's partials views are stored in the inc folder. The second one is the one that separate each view. When a new view created, the only things that needs change is the title and content. Therefore, I created a box of undefined content for later implementation. To access these boxes simple type `@section('name of content')` in every additional view. An example is shown below

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

@section('title')
    Contact us
@endsection
```

The view models can also accept basic logic such as if statements and for loops. This particular function helps to separate views if a user is authenticated or not. A good example is the navbar as show below

```
@if (Auth::guest())
  <li><a href="{{route('login')}}">LogIn</a></li>
  <li><a href="{{route('signup')}}">SignUp</a></li>
@endif
@if (!Auth::guest())
  <li><a href="{{route('news.index')}}" class="blue accent-3">News Room</a></li>
  <li><a href="{{route('dashboard')}}">Hello {{ Auth::user()->username }}</a></li>
  <li><a href="{{route('logout')}}">Log Out</a></li>
  @if (auth()->user()->isAdmin())
    <li><a href="{{route('admin')}}" class="grey darken-1">Admin Panel</a></li>
  @endif
```

The above code can be translated as if the user is guest then show him only the Log in and Sign up routes. Alternatively, the navbar displays the user name, the log out and all the other links that can be accessed only by authenticated users. But what happens if a guest user types manually the URL to access the forbidden content? Then we need access control in all routes of the application. This process took me more than one week, but I will describe it here as it is a core security feature that can't be ignored

Access Control and URL filtering

There are many ways to implement access control in the application. The most common is by creating a middleware that filters a specific group of http requests. I.e. if the user is authenticated but does not have admin privileges, he can't access the admin panel. In this case if a user types manually *localhost/admin* without filtering this GET request, he can access the panel and use all the features. To solve the problem, I manually created a middleware.

```
public function handle($request, Closure $next)
{
    if (auth()->user()->isAdmin()){
        return $next($request);
    }

    return redirect()->route('home');
}
```

This statement checks the request and if the condition results TRUE then procced, otherwise return to the home screen. The method `isAdmin()` was implemented in the user model

```
public function isAdmin(){
    return $this->type === self::ADMIN_TYPE;
}
```

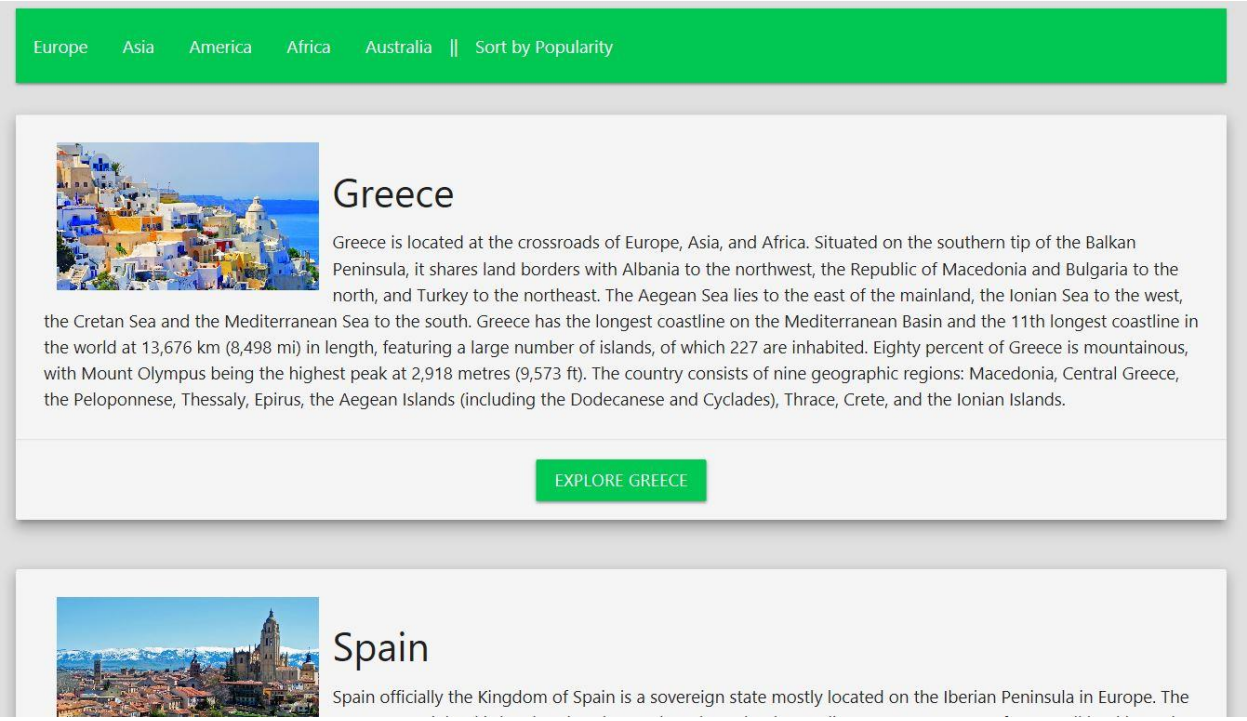
The above function is Boolean type that checks the database users table the ‘type’ column if the variable is default or admin. To apply this middleware, I need to specify in what routes could be applied. Example code is provided below.

```
Route::get('/admin', 'AdminController@admin')->middleware('is_admin')->name('admin');
```

Of course, there are also other ways to apply access control. Most common is via a method when an event occur. Most of the times before the actual event happens an if statement in the controller method prevents unauthorized users to gain control.

Second week

This was the week I built the core functionality of the application. At first, I created the Country model and a database representation. In the starting page I wanted to show all the available countries with the option to sort it by continents. This functionality helps the user to easily navigate through countries when the entries became bigger and bigger. I added a continent field in the database to easily make the queries I needed to sort the countries. This is the general view of the countries index page.



In the controller method `index()` I took all the available countries in the database and displayed it 5 per page. This is called pagination, that creates links to navigate between pages. The other feature is the ability to display all countries from a specific continent. In plain PHP I would have run a query to display all countries where continent is Africa for example. The SQL statement would looked like SELECT * FROM countries WHERE ‘continent’ Africa. In Laravel there is a simple way to run this query.

```php
public function byContinents($continent){
    $countries = DB::table('countries')->where('continent', $continent)->get();
    return view('country.byContinent')->with('countries', $countries);
}
```

The above function takes a continent parameter and search the table countries by the given string. Then returns the result and passed it into the specific view. The view had all the information needed to display the image the name and the description of each country. In order to access all the countries one by one into the html view I used the foreach statement.

```php
@foreach ($countries as $country)
<div class="row">
    <div class="col s12">
        <div class="card z-depth-3 grey lighten-4">
            <div class="card-content">
                <img src="{{ $country->image }}" alt="" class="col s3">
                <h4>{{ $country->name }}</h4>
                <p>{{ $country->description }}</p>
            </div>
            <div class="card-action center-align">
                <a class="waves-effect waves-light btn green accent-4" href="{!! route('show.country', $country
            </div>
        </div>
    </div>
</div>
```

The next step was little different than the previous one. I wanted to have a single page of every country in the database. The URL would be something like */countries/id.* The problem was that the id had to passed in dynamically. In the database table countries, the primary key is and auto increment unsinged integer. So the id had to match the primary key of each country. Because of this condition the route had to know that the second argument is a dynamic link. Laravel definition is made by single curly braces.

```php
Route::get('/countries/{id}', 'CountriesController@show')->name('show.country');
```

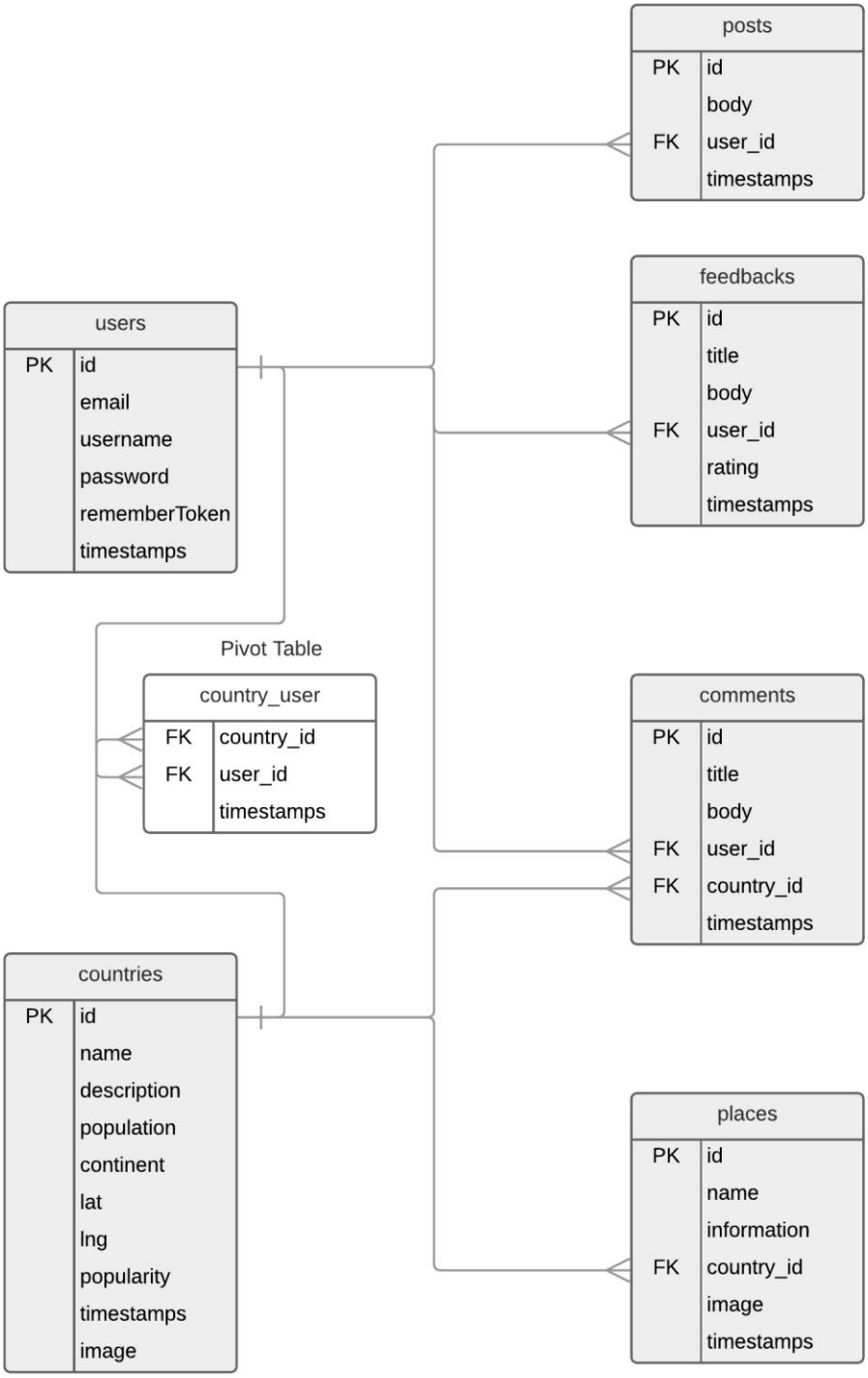
Now the function `show($id)` in Countries controller waits an argument. Back to the method the id is the key that separates one country to another. The logic behind this function is “return the country that have `$id==$country->id`. Laravel again provides a shortcut to this query by using the method `find($id).`

```php
public function show($id){

    $country = Country::find($id);
```

With all the information available, it was simple to create the page of the single country. In addition to just display a specific country I wanted the user to be informed about interesting places to visit if he happened to travel in the country. I again created a table places to store the

information, but I had to make a relationship between the tables. In simple words a country can have many places, but a place must belong to a single country. This is the ‘one to many’ relationship in SQL. I used many of these relationships throughout the building process. So, I made an ER Diagram to explain all the connections in the application.



To implement the relation inside the PHP code I had to define two function in both sides of the models. In this case I show below how I worked, but the same process was done with all the other relations in the database.

```php
public function places(){
    return $this->hasMany('Nature\Place');
}
```

```php
public function country(){
    return $this->belongsTo('Nature\Country');
}
```

More information can be found in https://laravel.com/docs/5.6/eloquent-relationships

One extra thing I wanted to add is a Google maps API to display a country map to the user when visiting a country. I thought it would be a cool feature for the user to navigate the map inside the webpage rather than open a separate window. I wrote some JS code to work but google maps API is a paid service. So I decided to add it but when the map is loaded I get a waning message that the map is not fully functional unless I pay for the service. In the GitHub repo is an open issue to warn that the map has not fully implemented. The code is displayed below

```
@section('head')
    <script src="https://maps.googleapis.com/maps/api/js?key=AIzaSyBHMKDfy6PFdKpQLQv2209hfCD3w0jYvJM"></script>
    {{-- <script src = "https://maps.googleapis.com/maps/api/js?AIzaSyBHMKDfy6PFdKpQLQv2209hfCD3w0jYvJM"></script>
    <script>
        function loadmap(){
            var mapOptions = {
                center:new google.maps.LatLng({{ $country->lat }}, {{ $country->lng }} ),
                zoom:6,
                mapTypeId:google.maps.MapTypeId.ROADMAP
            };

            var map = new google.maps.Map(document.getElementById("sample"),mapOptions);
        }

        google.maps.event.addDomListener(window, 'load', loadMap);
    </script>
@endsection
```

Later I applied a fully working external API to fulfill the assignment's requirement.

At that time, I worked with other sections of the application, fixing bugs that occur or adding more functionality like more validation or run more specific queries. It is very hard to provide all the information in this documentation **BUT** my whole work **STEP** by **STEP** is available for everyone to see and evaluate in https://github.com/ekapsimalis/travelapp There are 29 commits with explanatory comments showing the date and the extra code I added to make this project work as I expected. All the commits made by 25 of July to 20 of August 2018 are my proof of work. I will continue to explain more core aspects of the app but there are so many

details that is impossible to describe in this paper like session flash messages, UI changes, paginations, 404 and 401 pages and many more.

Third week

By the start of the third week, I started to implement the admin panel. The whole idea was the admin to be able to see all the user's feedbacks and rates with the ability to create a country and place. Also, the admin could be able to see all the users that created an account in the app, delete a specific one or promote others to gain admin privileges. In general, the application was built with three layers of access control. Below is a table that describes each layer's privileges

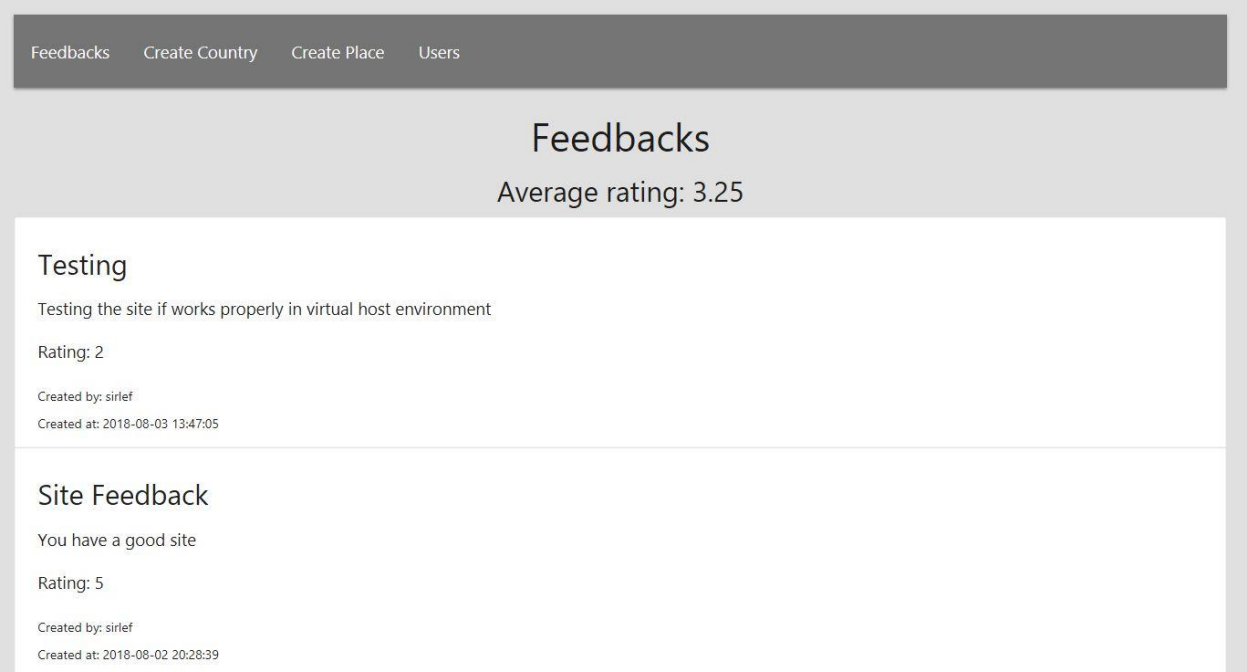
Level	Privileges
Guest	The guest can be able to see the welcome page with top news and the latest entries in the database plus to explore all the counties but limited to only the basic information.
Authenticated user	All the above, plus post a comment, see all places for the specific country, like a country, gain access to the personalized dashboard, reed all the news in the news room tab, save articles for later reading and give feedbacks and ratings
Admin	All the above, plus gain access to the admin panel

At first, when the admin logs in, he can be able to read all the feedbacks and see the average rating providing within the feedback form. The algorithm to display the average rating is shown below

```
$_count = count($allfeedbacks);
$sum = 0;
foreach ($allfeedbacks as $feedback){
    $sum+= $feedback->rating;
}
$score = $sum/$_count;

return view('admin')->with('feedbacks', $feedbacks)->with('score', $score);
```

This method allows to dynamically calculate the average when more and more feedbacks are coming by the users. The variable is stored in the view add accessed directly in the main admin page. The page looks like this.



The admin can also create a country and a place to insert into the database. Using this panel is more convenient than inserting directly in the database, because the validation is more applicable. Again, the admin is prompted to provide all the information needed for a successful country object and if something goes wrong an appropriate error message appears. The logic is similar to the sign-up user described above. Here is the coding implementation

```
$country = new Country();

$country->name = $request['name'];
$country->image = $request['image'];
$country->continent = $request['continent'];
$country->population = $request['population'];
$country->description = $request['description'];
$country->lat = $request['lat'];
$country->lng = $request['lng'];

$country->save();
Session::flash('countrycreated', 'You have successfully created a country');
return redirect()->back();
```

The flash message simple informs the admin that the country had been successfully added. The message lives for one request. If another request made the flash message disappears. All these messages appear at the top of the page below the navigation bar. There are included in the master layout for every page. Again, the detailed code can be found in https://github.com/ekapsimalis/travelapp/commit/3ac3df298a035fcc4be6288a71a37071947bbb623 GitHub Repository.

Another functionality admin can perform, is managing users. The admin can be able to delete a specific user from the database. To apply this, in plain PHP, I had to run the DELETE SQL statement. Again, Laravel provides a simple way for doing this. Below is the actual code.

```
public function deleteUser($id){

    $user = User::find($id);
    $user->delete();

    //Delete the related entries
    $posts = Post::where('user_id', $id);
    $posts->delete();
    $comments = Comment::where('user_id', $id);
    $comments->delete();
    $feedbacks = Feedback::where('user_id', $id);
    $feedbacks->delete();

    Session::flash('deleteUser', 'User has been deleted');
    return redirect()->back();

}
```

Let's examine the code step by step. The first thing to notice is that the function takes one argument and it is the primary key of the user I wanted to delete. This is a GET request meaning that everyone can access this route via URL. To prevent this, I applied the `isAdmin()` middleware described in previous chapter. Then I find the user with the specific id and before delete the user, I delete all the related to him data. After that, I actually delete the user and provide a simple flash message saying all is done correctly. If I hadn't deleted all the data first, I would have faced some null reference exceptions.

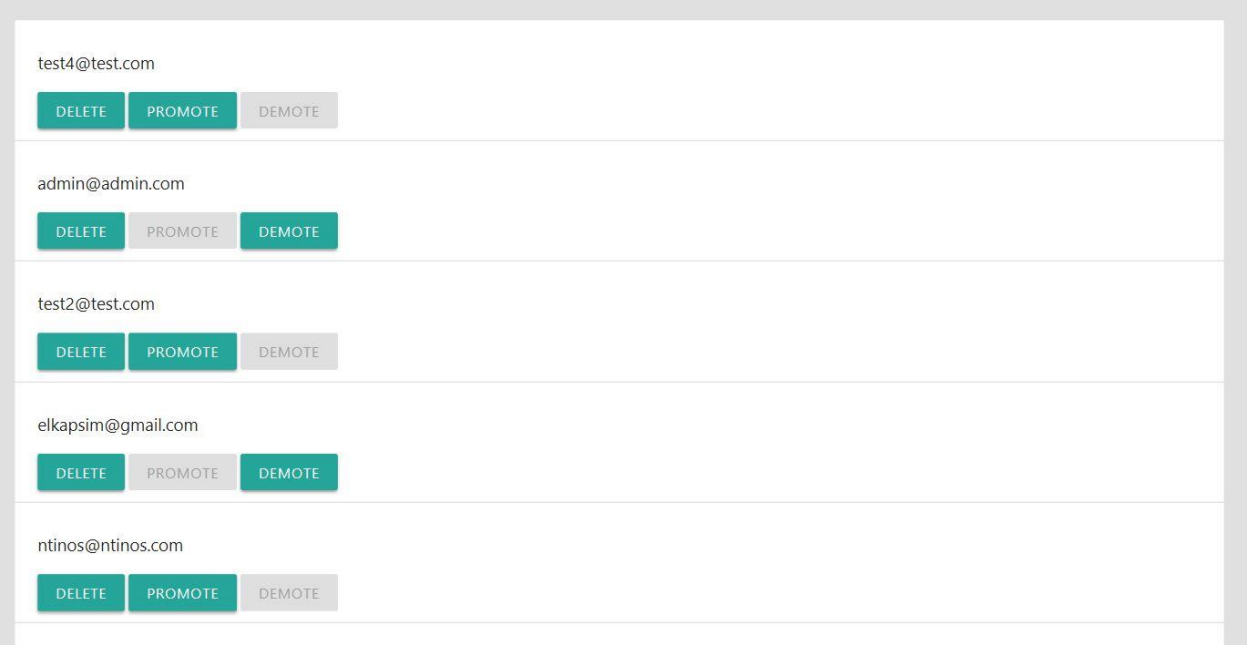
The other feature was the ability to promote a user to admin. Instead of doing this hardcoded inside the database I provided a build-in functionality. The only thing I wanted to change was the type column of the users table. By default all created users has the 'default' value I needed to run the UPDATE SQL statement and then save the user with the new data.

```
public function promoteUser($id){

    $user = User::find($id);
    $user->type = 'admin';
    $user->save();

    return redirect()->back();
}
```

The screen looks like this



The problem here was the ability to disable the button if the user was already an admin. I had to search into the database to check the user's type value and then disable the button. The same problem occurred during the implementation of the news tab using external API.

News API functionality

The final major functionality was the showing of top trending news about nature, science and health. I searched the web to find API services and I found the https://newsapi.org/ The free version provides 1,000 requests per day using a collection of top news around the world. When I got the API key I was able to start using the requests to retrieve a JSON format with the data. The JSON looked like this

```
{
    status: "ok",
    totalResults: 20,
  - articles: [
      - {
          - source: {
                id: null,
                name: "Streetinsider.com"
            },
            author: null,
            title: "PepsiCo (PEP) Agrees to Acquire SodaStream(SODA) for $144/Sh in Deal
            Valued at $3.2B",
            description: null,
            url:
            https://www.streetinsider.com/Corporate+News/PepsiCo+%28PEP%29+Agrees+to+Acquire+S
            ,
            urlToImage: null,
            publishedAt: "2018-08-20T09:48:17Z"
        },
```


To get the results and convert the JSON to PHP object, I used the following code

```
$apiresponse = file_get_contents('https://newsapi.org/v2/top-headlines?sources=national-geographic&apiKey
$ngnews = json_decode($apiresponse);
```

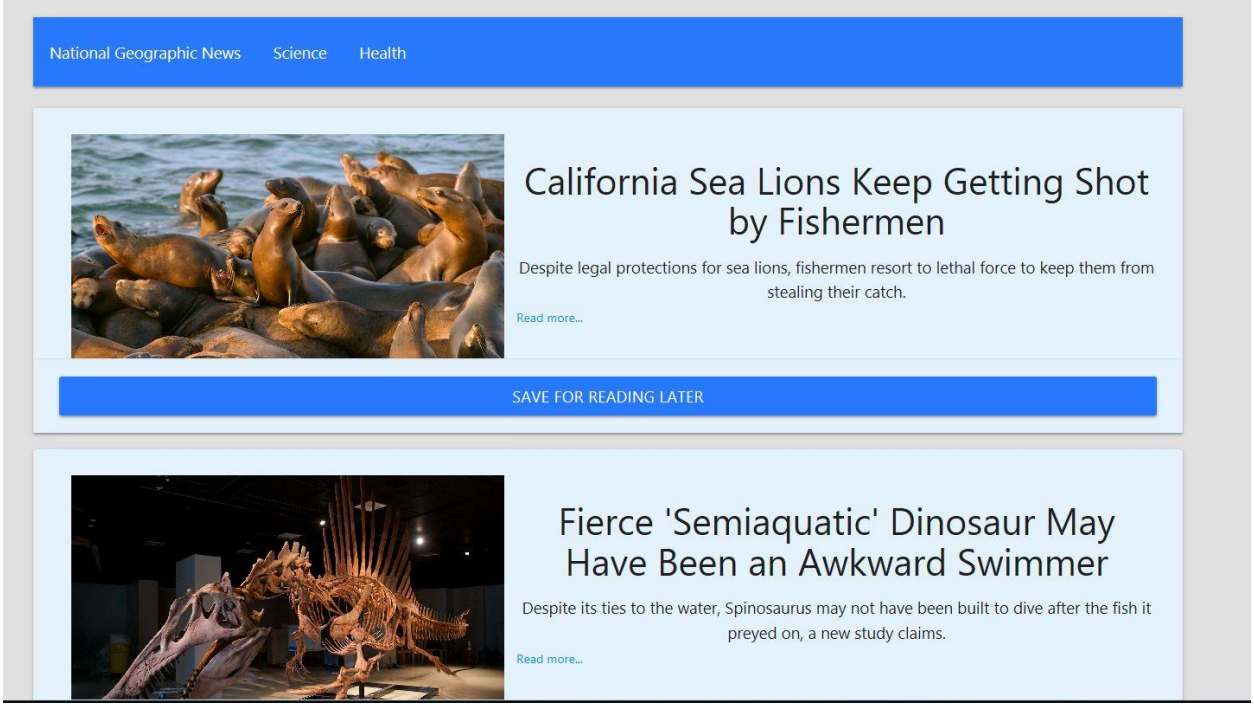
Every time a user clicks the news room tab sends a request to the link and retrieve the data. This is then passed dynamically to the appropriate view. The news is refreshing every hour, so its more likely the news will change after one request. It is impossible to save all the data into the database, so a specific article is lost after a short period. To solve this issue, I created a table in the database for articles that a user wants to save for later reading. This action saves the data and can be accessed even if the article disappears. All the saved articles are shown in the user's dashboard. The table news has all the fields required to display the article plus a column to specify who was the user who pressed save. By doing this I had access to each user's articles.

```
public function save(Request $request){

    $user = Auth::user();

    DB::table('news')->insert(
        ['title' => $request['title'], 'description' => $request['description'], 'url' => $request['url'],
        'urlToImage' => $request['urlToImage'], 'user_id' => $user->id]
    );
    return redirect()->back();
}
```

All the articles are looped through using a foreach statement and displayed using the materialize CSS card component. The result looks like this.



Finally, I had a problem with the save article button. Because all these articles aren't saved in the database I couldn't be able to check if one exists to stop users spam the button. Then I found the following solution. I assumed that the title of each article is unique so every time an article is shown, I had to check it if already exists in the database. To check an unindexed column in the database requires more resources that an indexed one like id. But I didn't find another faster solution, so I proceeded as show bellow

```
<div class="card-action">
    @if(DB::table('news')->where([
        ['title', '=', $article->title],
        ['user_id', '=', Auth::user()->id]
    ])->exists())
        <a href="{{route('news.save')}}" target="_blank" class="waves-effect waves-light
    @else
        <form action="{{route('news.save')}}" method="post">
```

The best way is to send AJAX requests to server if the user clicks the save button instead of reloading the whole page, but I hadn't much time to learn these kind of requests.

Like button Many to Many relationship

Lastly, I wanted to implement a like button in each country for the user. This relationship between the country and the user is more complicated than the previous ones. A user can like many countries and a country is liked by many users. This is the many to many relationship and in order to work I needed a pivot table. In Laravel the many to many relationships can be defined in the appropriate model as follow

```php
public function users(){
    return $this->belongsToMany('Nature\User');
}
```

The pivot table stores the user id and the country id to create the binding. When the user hits like, a new entry is created.

```php
public function like($id){

    // Check if the user is Authenticated
    // Else we may encounter strange errors

    if (!Auth::guest()){

        $country = Country::find($id);
        $user = Auth::user();

        // Check if the record already exist in the database
        if (DB::table('country_user')->where([
            ['country_id', '=', $country->id],
            ['user_id', '=', $user->id]
        ])->exists()){
            Session::flash('like_exist', 'You have already liked the country');
            return redirect()->route('show.country', $country->id);
        }
        else{
            $user->countries()->attach($country);
```

The attach method in Laravel acts like the INSERT INTO the pivot table. By convention Laravel search for a pivot table named country_user and adds all the required data to make the binding possible. In addition to that, there is a counter in the method that auto updates the popularity column in order to be able to sort by popularity in the countries index page

```php
    // Update the popularity column
    $country->popularity = $country->popularity + 1;
    $country->save();
    return redirect()->back();
```

Japan

LIKE

Japan is a sovereign island country in East Asia. Located in the Pacific Ocean, it lies off the eastern coast of the Asian mainland and stretches from the Sea of Okhotsk in the north to the East China Sea and China in the southwest. Coordinates: 35°N 136°E The kanji that make up Japan's name mean "sun origin", and it is often called the "Land of the Rising Sun". Japan is a stratovolcanic archipelago consisting of about 6,852 islands. The four largest are Honshu, Hokkaido, Kyushu, and Shikoku, which make up about ninety-seven percent of Japan's land area and often are referred to as home islands. The country is divided into 47 prefectures in eight regions, with Hokkaido being the northernmost prefecture and Okinawa being the southernmost one. The population of 127 million is the world's tenth largest. Japanese people make up 98.5% of Japan's total population. About 9.1 million people live in Tokyo, the capital of Japan.

Japan

ALREADY LIKED

UNDO

Japan is a sovereign island country in East Asia. Located in the Pacific Ocean, it lies off the eastern coast of the Asian mainland and stretches from the Sea of Okhotsk in the north to the East China Sea and China in the southwest. Coordinates: 35°N 136°E The kanji that make up Japan's name mean "sun origin", and it is often called the "Land of the Rising Sun". Japan is a stratovolcanic archipelago consisting of about 6,852 islands. The four largest are Honshu, Hokkaido, Kyushu, and Shikoku, which make up about ninety-seven percent of Japan's land area and often are referred to as home islands. The country is divided into 47 prefectures in eight regions, with Hokkaido being the northernmost prefecture and Okinawa being the southernmost one. The population of 127 million is the world's tenth largest. Japanese people make up 98.5% of Japan's total population. About 9.1 million people live in Tokyo, the capital of Japan.