Lab Exercise 9

Dr. Sarvar Abdullaev
s.abdullaev@inha.uz

April 1, 2020

The purpose of this lab is to learn using Eloquent models, migrations to database, query builder and relationships between models in Laravel.

1 Clone Project, Install Dependencies and Configure Database

Follow below steps in order to ensure that your project is set up correctly:

1. Clone newly created repository from accepted assignment to your local labs folder.
2. Open terminal inside that folder and run following command `composer install` to install all PHP dependencies of cloned project
3. Rename `.env.example` file to `.env` file. In command line run `mv .env.example .env` (Linux or MacOS) or `ren .env.example .env` (Windows)
4. Run following command afterwards: `php artisan key:generate`
5. Once all dependencies are installed, run following command `php artisan serve`. This will start a Laravel's own development web server at `http://localhost:8000`. Open it in your browser. You should be able to see Figure 1 web page:
6. Go to `https://remotemysql.com/signup.html` and provide some email address.It will create a free database account. Save details of your newly created database account into somewhere safe. You can login to your database using these credentials in this `https://remotemysql.com/phpmyadmin/`
7. In your Laravel project folder, open `.env` file and copy your remote database credentials to corresponding environment variables inside `.env` file, and save it.
8. In your laravel project folder, open `config\database.php` file and ensure that your `mysql` configuration is set as shown below:

```
'mysql' => [
            'driver' => 'mysql',
            'host' => env('DB_HOST', 'localhost'),
            'port' => env('DB_PORT', '3306'),
            'database' => env('DB_DATABASE', 'forge'),
            'username' => env('DB_USERNAME', 'forge'),
            'password' => env('DB_PASSWORD', ''),
            'charset' => 'utf8',
            'collation' => 'utf8_unicode_ci',
            'prefix' => '',
            'strict' => true,
            'engine' => null,
            'modes'  => [
                'ONLY_FULL_GROUP_BY',
                'STRICT_TRANS_TABLES','NO_ZERO_IN_DATE', 'NO_ZERO_DATE',
                'ERROR_FOR_DIVISION_BY_ZERO',
                'NO_ENGINE_SUBSTITUTION',
                ],
        ],
```

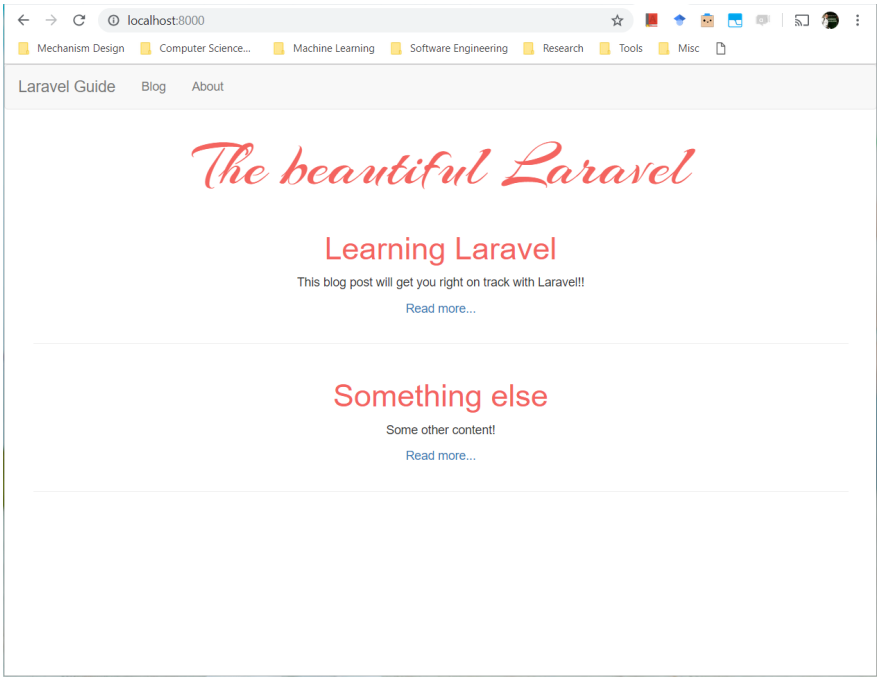

Figure 1: First View

Ensure that `modes` key is set as shown above. It is important because `https://remotemysql.com` does not grant full permission on your database, and your database connection driver should use specified modes.

9. Now your Laravel project is fully configured for using database.

2 Models and Migrations

Complete steps below to create necessary models and migrations for your project:

1. Run `php artisan make:model Post -m`. This will create a *Post* model class located in `app\Post.php` file and a corresponding `create_posts_table.php` migration file located in `database\migrations\` folder.

2. Open `database\migrations\create_posts_table.php` file and write following fields inside it:

```
Schema::create('posts', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
            $table->string('title');
            $table->text('content');
        });
```

3. Open `app\Post.php` and set fillable fields to `['title', 'content']`. This allows setting these fields while instantiating `Post` class.

4. Run `php artisan make:model Tag -m`. This will create a *Post* model class located in `app\Tag.php` file and a corresponding `create_tags_table.php` migration file located in `database\migrations\` folder.

5. Open `database\migrations\create_tags_table.php` file and write following fields inside it:

```
Schema::create('tags', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
```

```
            $table->string('name');
        });
```

6. Open `app\Tag.php` and set fillable fields to `['name']`. This allows setting these fields while instantiating `Tag` class.

7. Run `php artisan make:model Like -m`. This will create a *Post* model class located in `app\Like.php` file and a corresponding `create_likes_table.php` migration file located in `database\migrations\` folder.

8. Run `php artisan migrate` to create all tables in database. Ensure that all of them are created in your database via phpMyAdmin.

3 Database Queries and Seeding Data

In this section, we will modify our views to use model objects instead of associative arrays, build queries to the database and populate database with initial data.

3.1 Adapting Views to Models

Our views were designed to display parameters passed as an associative array (i.e. dictionary). However now we will switch to objects, so we will modify our views to display object properties instead of dictionary values.

1. Open `app\Post.php` and remove all unnecessary methods, so it looks as shown below:

```
class Post extends Model
{
    protected $fillable = ['title', 'content'];
}
```

2. Open `views\blog\index.blade.php` and change `$post['title']` and `$post['body']` to `$post->title` and `$post->body` respectively.

3. Open `views\blog\post.blade.php` and change `$post['title']` and `$post['body']` to `$post->title` and `$post->body` respectively.

4. Open `views\admin\edit.blade.php` and change `$post['title']` and `$post['body']` to `$post->title` and `$post->body` respectively.

5. Open `views\admin\index.blade.php` and replace `['id' => array_search($post, $posts)]` with `['id' => $post->id]` inside *Edit* link. Also change `$post['title']` to `$post->title`.

6. In `views\admin\index.blade.php` and next to *Edit* link add new *Delete* link as shown in below code:

```
<a href="{{ route('admin.delete', ['id' => $post->id]) }}">Delete</a>
```

   This link refers to a new route `admin.delete` which has not been created yet.

7. Open `routes\web.php` and add new route inside `admin` group as shown in below code:

```
Route::get('delete/{id}', [
      'uses' => 'PostController@getAdminDelete',
      'as' => 'admin.delete'
 ]);
```

3.2 Modifying Controller to use Database

Our `PostController` methods were programmed to use session variable to store user blog posts. However we will program them to use database queries instead of session variables.

8. Open `app\Http\Controllers\PostController.php` and make following changes in `getIndex()` function

```
public function getIndex()
    {
        $posts = Post::orderBy('created_at', 'desc')->get();
        return view('blog.index', ['posts' => $posts]);
    }
```

This pulls all posts ordered by their created date in a descending order.

9. In `PostController.php`, make following changes inside `getAdminIndex()` function

```
public function getAdminIndex()
    {
        $posts = Post::orderBy('title', 'asc')->get();
        return view('admin.index', ['posts' => $posts]);
    }
```

This pulls all posts ordered by their title.

10. In `PostController.php`, make following changes inside `getPost($id)` function

```
public function getPost($id)
    {
        $post = Post::where('id', $id)->first();
        return view('blog.post', ['post' => $post]);
    }
```

This finds a post by its ID using where statement.

11. In `PostController.php`, make following changes inside `getAdminEdit($id)` function

```
public function getAdminEdit($id)
    {
        $post = Post::find($id);
        return view('admin.edit', ['post' => $post, 'postId' => $id]);
    }
```

This finds a post by its ID.

12. In `PostController.php`, make following changes inside `postAdminCreate(Request $request)` function

```
public function postAdminCreate(Request $request)
    {
        $this->validate($request, [
            'title' => 'required|min:5',
            'content' => 'required|min:10'
        ]);
        $post = new Post([
            'title' => $request->input('title'),
            'content' => $request->input('content')
        ]);
        $post->save();

        return redirect()->route('admin.index')
                ->with('info', 'Post created, Title is: ' . $request->input('title'));
    }
```

This creates a new post object and saves it in database.

13. In `PostController.php`, make following changes inside `postAdminUpdate(Request $request)` function

```
public function postAdminUpdate(Request $request)
    {
        $this->validate($request, [
            'title' => 'required|min:5',
            'content' => 'required|min:10'
        ]);
        $post = Post::find($request->input('id'));
        $post->title = $request->input('title');
        $post->content = $request->input('content');
        $post->save();
        return redirect()->route('admin.index')
                ->with('info', 'Post edited, new Title is: ' . $request->input('title'));
    }
```

    This finds a post by its ID, sets its fields to given values and submits changes to database.

14. In `PostController.php`, add following function which handles post deletes.

```
public function getAdminDelete($id)
    {
        $post = Post::find($id);
        $post->delete();
        return redirect()->route('admin.index')->with('info', 'Post deleted!');
    }
```

15. Now you can test your blog app by adding, removing or editing posts from `http://localhost:8000/admin` section.

3.3 Seeding Database with Initial data

In this section, we will populate our database with initial data using Laravel's seeding tool.

1. Run `php artisan make:seeder PostTableSeeder`. This will create `PostTableSeeder.php` inside `database\seeds` folder.

2. Create couple of sample records for `posts` table as shown below:

```
public function run()
    {
        $post = new \App\Post([
            'title' => 'Learning Laravel',
            'content' => 'This blog post will get you right on track with Laravel!'
        ]);
        $post->save();

        $post = new \App\Post([
            'title' => 'Something else',
            'content' => 'Some other content'
        ]);
        $post->save();
    }
```

3. Run `php artisan make:seeder TagTableSeeder`. This will create `TagTableSeeder.php` inside `database\seeds` folder.

4. Create couple of sample records for `tags` table as shown below:

```
public function run()
    {
        $tag = new \App\Tag();
        $tag->name = 'Tutorial';
        $tag->save();
```

```
        $tag = new \App\Tag();
        $tag->name = 'Industry News';
        $tag->save();
    }
```

5. Open `DatabaseSeeder.php` in the same folder, and make following modifications:

```
public function run()
    {
        $this->call(PostTableSeeder::class);
        $this->call(TagTableSeeder::class);
    }
```

   This will register corresponding seeders into main database seeder.

6. Run `php artisan db:seed` to create sample records in your database. You can verify that by viewing database tables in phpMyAdmin.

4 Working with Model Relationships

In this section, we will establish relationships between different models, and use them to create and retrieve related objects.

4.1 Establishing Relationships between Models

We will first establish relationships between corresponding models. This may require additional changes in database structure, so we should refresh all migration files after making changes to them.

1. Open `database\migrations\create_likes_table` and add additional `$table->integer('post_id')` field into its schema. It is required in order to establish one-to-many relationship between `posts` and `likes` tables on a database level. Note that one post can have many likes, but each like belongs to only one post. Therefore, there is a natural one-to-many relationship between `posts` and `likes` tables where `likes` table takes the primary key of `posts` table as foreign key attribute `post_id`. This naming convention is very important, because Laravel will conclude that `post_id` is a foreign key by just reading the name of the field. Final version of the migration file should be the same as given below:

```
Schema::create('likes', function (Blueprint $table) {
          $table->increments('id');
          $table->timestamps();
          $table->integer('post_id');
      });
```

2. Open `app\Like.php` file and add following function to connect `Like` model to `Post` model:

```
public function post()
    {
        return $this->belongsTo('App\Post', 'post_id');
    }
```

   This allows an instance of `Like` class to obtain an associated `Post` object upon request.

3. Open `app\Post.php` file and add following function to connect `Post` model to `Like` model:

```
public function likes()
   {
       return $this->hasMany('App\Like', 'post_id');
   }
```

   This allows an instance of `Post` class to retrieve all associated `Like` objects upon request.

4. Now we will establish a many-to-many relationship between `posts` and `tags` tables on a database level. Indeed, every post can be tagged with multiple tags, and every tag can be applied to multiple posts, so they are in a many-to-many relationship with each other. In relational database, many-to-many relationships are resolved using special pivot tables which consists of foreign keys from both participating tables. So we should create a new `post_tag` table and include `post_id` and `tag_id` fields inside it. This will tell Eloquent that `posts` and `tags` tables are in many-to-many relationship with each other. Now let's create `post_tag` table by generating corresponding migration file using command below:

```
php artisan make:migration create_post_tag_table
```

5. Open newly created `create_post_tag_table.php` file and add following fields into this table as shown below:

```
Schema::create('post_tag', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
            $table->integer('post_id');
            $table->integer('tag_id');
        });
```

6. Open `app\Post.php` file and add following function to connect `Post` model to `Tag` model:

```
public function tags()
    {
        return $this->belongsToMany('App\Tag', 'post_tag', 'post_id', 'tag_id')
               ->withTimestamps();
    }
```

7. Open `app\Tag.php` file and add following function to connect `Tag` model to `Post` model:

```
public function posts()
   {
       return $this->belongsToMany('App\Post', 'post_tag', 'tag_id', 'post_id')
               ->withTimestamps();
   }
```

8. Run `php artisan migrate:refresh` to drop and create whole database structure again.

9. Run `php artisan db:seed` to populate your database with initial data.

4.2 Adapting Views to Model Relationships

1. Open `views\blog\index.blade.php` and on top of post content, list all tags of the post. It can be done by adding following snippet:

```
<p style="font-weight: bold">
                @foreach($post->tags as $tag)
                    - {{ $tag->name }} -
                @endforeach
</p>
```

   You can access all tags of the post by simply calling tags property.

2. Open `views\blog\post.blade.php` and on top of post content, show total number of likes for this post. Also there should be a link which allows user to like a given post. It can be done by adding following snippet:

```
<div class="row">
        <div class="col-md-12">
            <p>{{ count($post->likes) }} Likes |
                <a href="{{ route('blog.post.like', ['id' => $post->id]) }}">Like</a></p>
        </div>
 </div>
```

You can access all likes of selected post and even apply `count()` function on them.

3. Open `routes\web.php` and create a new route `blog.post.like` as shown here:

```
Route::get('like/{id}', [
    'uses' => 'PostController@getLikePost',
    'as' => 'blog.post.like'
]);
```

This route invokes `getPostLike()` method of `PostController` which is not defined yet.

4. Open `PostController.php` and add new function `getPostLike($id)` as shown below:

```
public function getLikePost($id)
   {
       $post = Post::where('id', $id)->first();
       $like = new Like();
       $post->likes()->save($like);
       return redirect()->back();
   }
```

Above you create new instance of `Like` class and save it directly inside the property of corresponding `$post` object. This will automatically link newly created `$like` object to `$post`.

5. Open `views\admin\create.blade.php` and add list of checkboxes indicating available tags below content input box. This can be achieved as shown below:

```
@foreach($tags as $tag)
    <div class="checkbox">
        <label>
            <input type="checkbox" name="tags[]" value="{{ $tag->id }}"> {{ $tag->name }}
        </label>
    </div>
@endforeach
```

Now our view for creating posts depends on `$tags` variable which should be provided while generating it. Let's fix this in next step.

6. Open `PostController.php` and in function `getAdminCreate()`, pull all tags from the database and pass them to the view as shown here:

```
public function getAdminCreate()
    {
        $tags = Tag::all();
        return view('admin.create', ['tags' => $tags]);
    }
```

7. In `PostController.php`, we also need to assign selected tags to created post. So after creating the post inside `postAdminCreate(Request $request)` function, we should pass the IDs of selected tags as an array as shown below:

```
$post->tags()->attach($request->input('tags') === null ? [] : $request->input('tags'));
```

We first check if `tags` parameters is sent at all, and then pass an array of tag IDs to be assigned to newly created post record.

8. Open `views\admin\edit.blade.php` and add list of checkboxes indicating available tags below content input box. This can be achieved as shown below:

```
@foreach($tags as $tag)
 <div class="checkbox">
     <label>
         <input type="checkbox" name="tags[]" value="{{ $tag->id }}"
         {{ $post->tags->contains($tag->id) ? 'checked' : '' }}> {{ $tag->name }}
     </label>
 </div>
@endforeach
```

This not only displays list of checkboxes that correspond to tags, but also check/uncheck them based on the assigned tags of post that is being updated.

9. Similar to `create.blade.php`, we also need to pass `$tags` variable to `edit.blade.php`. Open `PostController.php` and in function `getAdminEdit($id)`, pull all tags from the database and pass them to the view as shown here:

```
public function getAdminEdit($id)
    {
        $post = Post::find($id);
        $tags = Tag::all();
        return view('admin.edit', ['post' => $post, 'postId' => $id, 'tags' => $tags]);
    }
```

10. In `PostController.php`, we also need to assign/unassign selected/unselected tags to updated post. So after saving the changes of the post inside `postAdminUpdate(Request $request)` function, we should synchronize the IDs of selected tags with IDs of tags assigned to update post:

```
$post->tags()->sync($request->input('tags') === null ? [] : $request->input('tags'));
```

11. In `PostController.php`, we also need to unassign all tags and remove all likes if we are deleting a post. So we should rewrite `getAdminDelete($id)` function as follows:

```
public function getAdminDelete($id)
    {
        $post = Post::find($id);
        $post->likes()->delete();
        $post->tags()->detach();
        $post->delete();
        return redirect()->route('admin.index')->with('info', 'Post deleted!');
    }
```

Now your blog can handle models with multiple relationships and it should be working as expected. You should be able to create, edit and delete posts from Admin section of your blog. Also posts show related tags, and can be liked by users.

5 Final Solution

You can compare your solution with the final solution here: `https://github.com/iuthub/ip2019_lab_9/tree/solution`