2.1 - Adding the database

During skills lab:

👥 Work in *programming pairs*: a team of two members sitting behind one computer programming as one.

Do the following assignments during the skills lab and bring the results to the Checkpoint. You can do these assignment either in your project or build upon the provided snapshot, which already includes all the features of yesterdays assignments:

laravel-taskiteasy-snapshot-2-1-0.zip 757.0KB

2.1.A - Setting up your working environment

Dive into this exercise to get comfortable with your workspace and tools for handling databases in Laravel apps with confidence. The basics about adding the database are already discussed during class, but you can find the related theory in this knowledge bank article:

Using Databases in Laravel

🛠️ Check your working environment. It should include some applications and tools installed and configured. Take the time to check if you can find and use them, and maybe a bit of configuration. Check for the following:

- **MySQL** - The database server application. It is either installed as part of the XAMPP package, installed manually or configured in a Docker container
- **DB manager application** - Software that serves as a graphical user interface to a MySQL database server. It lets you peek inside the database tables but also create new tables and change existing tables. Examples of these applications for MySQL are TablePlus, PHPMyAdmin, and SQLyog but there are also ones that are integrated in an IDE like PHPStorm (builtin) and VSCode (plugin)

The next step is to create a new database in MySQL.

🛠️ Open your DB manager, connect to your MySQL database server and create a new database called `taskiteasy`.

In order to experiment with MySQL and Laravel, you need some data

🛠️ Use your DB manager to run the following SQL script in the database you created before:

tasks-create-truncate-insert.sql 2.3KB

You should see a new table called 'tasks'. This table contains some rows of data

Then you need to configure your Laravel app to connect to the same database

🛠️ Update your `.env` file. The database connection settings need to match the ip address, user credentials and database name.

Finally, you need to test if everything works fine

🛠️ Open Tinker and run the following code:

```
DB::table('tasks')->first();
```

Check if the output corresponds to the snippet below. In the snippet, the first line shows opening tinker, which prints some information about the program on the second line. The command above is entered on the third line. Everything that follows is the corresponding output.

```
PS C:\...\taskiteasy> php artisan tinker Psy Shell v0.11.22 (PHP 8.3.0 — cli) by Justin Hileman > DB::table('tasks')->first(); = {#6296 +"id": 1, +"title": "Write a love letter to Laravel", +"description": "Compose a heartfelt love letter to Laravel, explaining how it has revolutionized your coding life. Express your deep affection for eloquent relationships and artisan commands.", +"priority": 3, +"state": "new", +"time_estimated": 15, +"time_spent": 0, +"created_at": "2023-10-15 09:00:00", +"updated_at": "2023-10-15 09:30:00", +"completed_at": null, }
```

2.1.B - Show individual tasks

Now you have filled the database with data, we can use this data to render the task views. We start by updating the `tasks.show` view.

🛠️ Inside the `TaskController@show` method, use the query builder to fetch data (the task where the value of `id` equals the value of the parameter `$id`) and pass this to the view. When done properly, the welcome view should not change that much. However, you might need to update statements like `$task['title']` into `$task->title`.

Now this is working, we have the opportunity to do some *negative testing*. What should happen when the user passes a non-existing id?

🛠️ Examine how the app responds when the user passes a non existing post id. Test with a url like: `/tasks/65535`

a Proper response to this should be a 404 page. We already practiced this in 1.1

🛠️ Update the `TaskController@show` method so it aborts to the 404 page when the given id is not present in the tasks table.

2.1.C - Introducing the Task model

Before you do the same for the `welcome` view, consider the following. During class we introduced the concept of *Models*. Each database table can have a corresponding model that is used to interact with that table. Let's use this model instead.

🛠️ Create an Eloquent model class for interacting with the tasks table. You may use the `make:model` Artisan command to generate this model. Decide on the name of the class. Keep in mind that convention over configuration dictates this name, so there is only one correct answer here (without adding some needless configuration).

Now you can start using this model to interact with the tasks table instead of the `DB::table('tasks')`. The first experiment is

🛠️ Inside the controller method that handles the `home` route, use the model to fetch the data to be passed to the view. This means that you should NOT use something like `DB::table('tasks')->get()`. It should be something like `Task::get()`. Note that you also need an extra `use` declaration on top of your controller.

Eloquent models offer more convenient ways to fetch specific rows:

1. What you did in the experiment above: a 'built-in query builder' for that specific table
2. The `::find()` method finds a row with the given id
3. The `::findOrFail()` method is like the previous, but aborts with a 404 when there is no record found

🛠️ Refactor the `TaskController@show` method, so it also uses this model instead of the query builder to fetch the right post from the database. Experiment with the 3 options mentioned above

2.1.D - The Task Migration

During class we also introduced the concept of *database migration*. as the Laravel way of managing the database structure. Let's experiment with this as well

🛠️ Use your DB manager to remove the tasks table. Find out how you can do this (tip: most DB managers call this process 'Dropping' or 'Deleting') or execute the SQL command `DROP TABLE tasks`.

If you check the DB manager the table should not be there (an empty table is not correct). Now let's try to recreate this with a migration.

🛠️ Create a migration for the tasks table. Check the database/migrations folder to see what the naming conventions are. Then decide on a proper name and use the `make:migration` Artisan command. When done properly, a `Schema::create` statement is already generated that creates a table called 'tasks'. If not, just remove the file and retry

When you check the generated migration, you will see that it has already chosen a table name for you (convention over configuration). Now let's fill the migration.

🛠️ Update the migration. Add the following declarations to the callback function that is passed to the `Schema::create`:

```
$table->string('title'); $table->text('description'); $table->integer('priority')->default(0); $table->string('state')->default('new'); $table->integer('time_estimated')->nullable(); $table->integer('time_spent')->default(0); $table->timestamp('completed_at')->nullable();
```

Note that these lines should be placed between the `$table->id();` and the `$table->timestamps()` declarations.

When you created the migration, the database is still not updated (check your DB manager if you don't believe this). In order to do this, you need to *run* the migration.

🛠️ Run the migration by executing the `php artisan migrate` command. Check your DB manager to see the new table

The new table should be present, with the correct attributes in the correct order, but the table is empty. The next step is to add data to that table.

🛠️ Open the 'tasks-create-truncate-insert.sql' script in your IDE. Remove the `CREATE TABLE` statement (lines 1-14) and run it in your DB manager again. It will insert the same tasks in the tasks table.

Again check your DB manager to see if the data is present

2.1.E - Route Model Binding (SHOULD)

Finally, we also mentioned *Route Model Binding* that uses dependency injection, see:
🗺️ **Routing - Route Model Binding**

🛠️ **Challenge!** Consider refactoring your `TaskController@show` such that it implements Route Model Binding. If you succeed, examine the code and marvel at the elegance and beauty of the solution.

2.1.F - Synchronize environments

👥 Work in *programming pairs*: a team of two members sitting behind one computer programming as one.

As a programming pair you have worked on one computer so far.

🛠️ Try to recreate the same results on the other computer within your programming pair as fast as you can. Try not to repeat each of the previous exercises, but think smartly about what to update on the other computer. Discuss similarities and differences

Checkpoint

Visit the checkpoint. Bring along the following:

- Your code, so we can discuss different variants of the solution
- Questions you got/things you got stuck on during the assignments

After the Checkpoint

You can experiment further with your own project. In the following assignments you will repeat everything once again to create a model, migration, controller, routes and views to display a blog, which is a collection of (blog) Posts. By following the next assignments you will discover that, when combining tasks, you might be a bit more efficient.

2.1.G - Catching up: Implementing previously missed features

👤 This is an individual assignment

In this part, you have the flexibility to either implement any missing features from yesterdays skill assignments in your project or build upon the provided snapshot, which already includes all the features of yesterdays assignments.

🛠️ Make your project ready for the following assignments. Check if all the features of the previous assignments, a.k.a. exercise 2.1.G, are present and bug free. If not, choose to either finish your project or use the provided snapshot.

laravel-taskiteasy-snapshot-2-1-1.zip 756.7KB

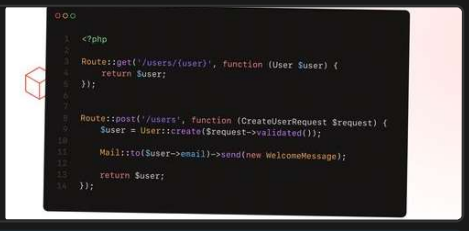
2.1.H - Boilerplating the Post model, migration and controller

We start of by creating the boilerplate code for this. The Artisan `make:model` command has options that generates other components like controllers and migrations. Check the following link:

Laravel - The PHP Framework For Web Artisans

Laravel is a PHP web application framework with expressive, elegant syntax. We've already laid the foundation — freeing you

https://laravel.com/docs/eloquent#generating-model-class...



🛠️ Create a Post model, migration and controller in one command. Find the correct option to do this without generating other components (so, DO NOT use the `--all` option).

The generated files implemented convention over configuration as much as possible.

2.1.I - Migration

In this section you will create the content of the posts table migration. This table will store small blog-like items. It is designed like so:

posts	
PK	id: bigInteger
	title : string
	excerpt: text
	body : longText
	created_at: timestamp
	updated_at: timestamp

Posts table design

🛠️ Update the migration. Add declarations to the callback function that is passed to the `Schema::create` : so it resembles the design as close as possible. You can copy and paste the following to start:

```
$table->string('title'); // TODO declare the excerpt and body colu
mns
```

We already added a declaration for the title column. You decide on how you to declare the excerpt and body columns. Think about name, data type and whether or not to use `nullable`

Next is to update the database structure

🛠️ Run the migration command again to create the table. Examine the output: it should only run the last migration and check the DB manager if the table is created correctly and is empty.

Now you can use an SQL script to fill the table with some initial data:

🛠️ Use your DB manager to run the following SQL script in the database you created before:

posts-insert.sql 5.8KB

You should see the 'posts' table contains some rows of data

2.1.J - Routes, controllers and views

Now, add the rest of the boilerplate code, which is specified in the following route list table:

URI	Handler	View name	Route name (optional)
`'/blog'`	`PostController@index`	`'posts.index'`	`posts.index`
`'/blog/{id}'`	`PostController@show`	`'posts.show'`	`posts.show`

Note that the `PostController@show` method should accept one parameter: the ID of the blog post the user wants to see.

🛠️ Create the required controller methods, routes and views for this to implement. The content of each view should hold a simple text (i.e. *'TODO: implement this'*) for now. So, no data needs to be fetched and passed to the views yet. Don't forget to keep each page nicely internally consistent within the application (a.k.a. use your layout) and perhaps add an item in your navigation bar

2.1.K - Update the views

Now you have all the boilerplating done, we can start to render some pages. We start by updating the `posts.index` view.

🛠️ Inside the `PostController@index` method, use the query builder to fetch sorted data for the view. The posts should be sorted by `created_at` where the latest post is first. Then use this to render a list of all the posts, showing each `title`, `excerpt` and `created_at` and a link to the appropriate `posts.show` route.

Now we can do the same for the `posts.show` view:

🛠️ Update the `PostController@show` method. Make it fetch the correct post or abort with a 404. You decide which method you use (see 2.1.C and 2.1.E). Pass the post data to the view and use this to render the `id`, `title`, `created_at` and `updated_at` and `body` somewhere on the page. Keep in mind that the `body` attribute contains *unescaped data* and should be rendered correctly.