DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

HTTPS://WWW.IPLEIRIA.PT/ESTG-DEI/

APLICAÇÕES PARA A INTERNET

Engenharia Informática

Marco Monteiro

Laravel - 1

Objectives:

(1) Comprehend and use main concepts of Laravel Framework.

(2) Implement basic CRUD operations with Laravel.

(3) Debug Laravel application.

Note the following:

- Before starting the exercise, read the related content on Moodle;
- During the resolution of the exercises, consult the Laravel documentation (https://laravel.com) and other online resources.

Scenery

This worksheet will create a simple Web Application using Laravel Framework with some school related data, namely, courses and disciplines, available on a database that will be created and filled using the provided migrations and database seeds. The web application will support all CRUD operations for the courses and disciplines and will include a debugging tool (Laravel Telescope).

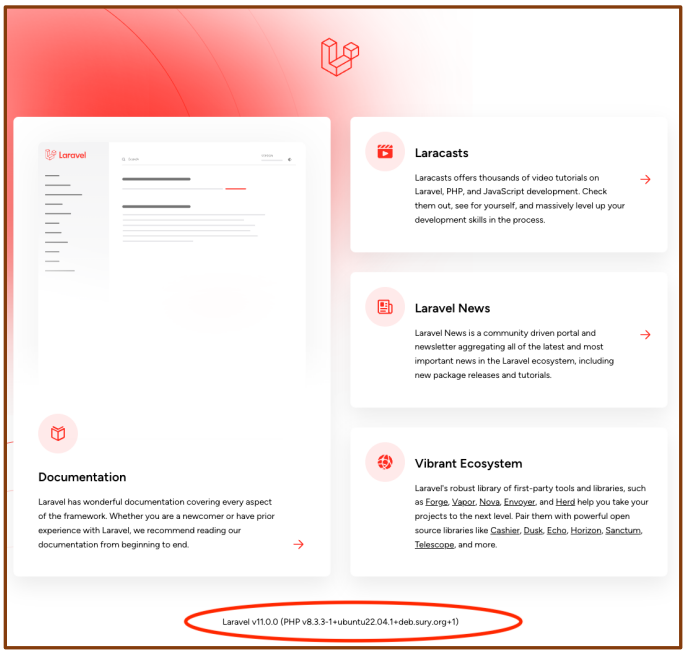
1. Preparation

To run the exercises of this worksheet and future worksheets, we will use Laragon (https://laragon.org), or Laravel Sail (https://laravel.com/docs/sail). For the database, we'll preferably use a MySQL server, or if that's not possible, a SQLite database.

Follow the instructions of the "**tutorial.laravel.01-laravel-install-configuration**", available on Moodle, to install and configure Laravel projects.

1.1. New Laravel Project

1. Create a new Laravel (**version 11 or newer**) project, named `ai-laravel-1`, using Laragon or Laravel Sail (Docker Containers).
   - The project can be installed and executed with **Laragon** or with **Laravel Sail** (that uses Docker Containers)
   - Follow the instructions of the "**tutorial.laravel.01-laravel-install-configuration**", available on Moodle.
2. To ensure that the Laravel application is correctly installed and configured, open the initial page on the URL: `http://ai-laravel-1.test` (*if using Laragon*) or `http://localhost` (*if using Laravel Sail*). It should present the following page:



   - Minimum version is Laravel 11.0
3. To ensure that the Laravel application can connect to a database and execute commands over that database, execute the following command on the root of the project:
   - Command version with **Laragon** terminal:

```
php artisan migrate:fresh
```

   - Command version with **Laravel Sail**:

```
sail art migrate:fresh
```

- If the command "`artisan migrate`" does not execute correctly, it means that the database configuration is not correct, or that the database does not exist or is not running correctly. Check the "**tutorial.laravel.01-laravel-install-configuration**", available on Moodle, to properly install and configure the database.
- Preferably we'll use a **MySQL** database. If that's not possible, we can use a SQLite database instead.

4. When using Laravel Sail, we recommend adjusting docker setting (file `docker-compose.yml`) to use only these containers: `laravel.test`, `mysql` and `adminer`. Check the "**tutorial.laravel.01-laravel-install-configuration**", available on Moodle, to learn how to do it.
   The provided file `docker-compose.yml` includes the required settings for these 3 containers.

1.2. Database

The database will be created and filled using the provided migrations and database seeds.

5. Delete the folder "database" from the root of the newly created project.
6. Copy the provided "database" folder to the root of the newly created project.
7. Execute the following command on the root of the project:
   - Command version with **Laragon** terminal:

```
php artisan migrate:fresh
```

   - Command version with **Laravel Sail**:

```
sail art migrate:fresh
```

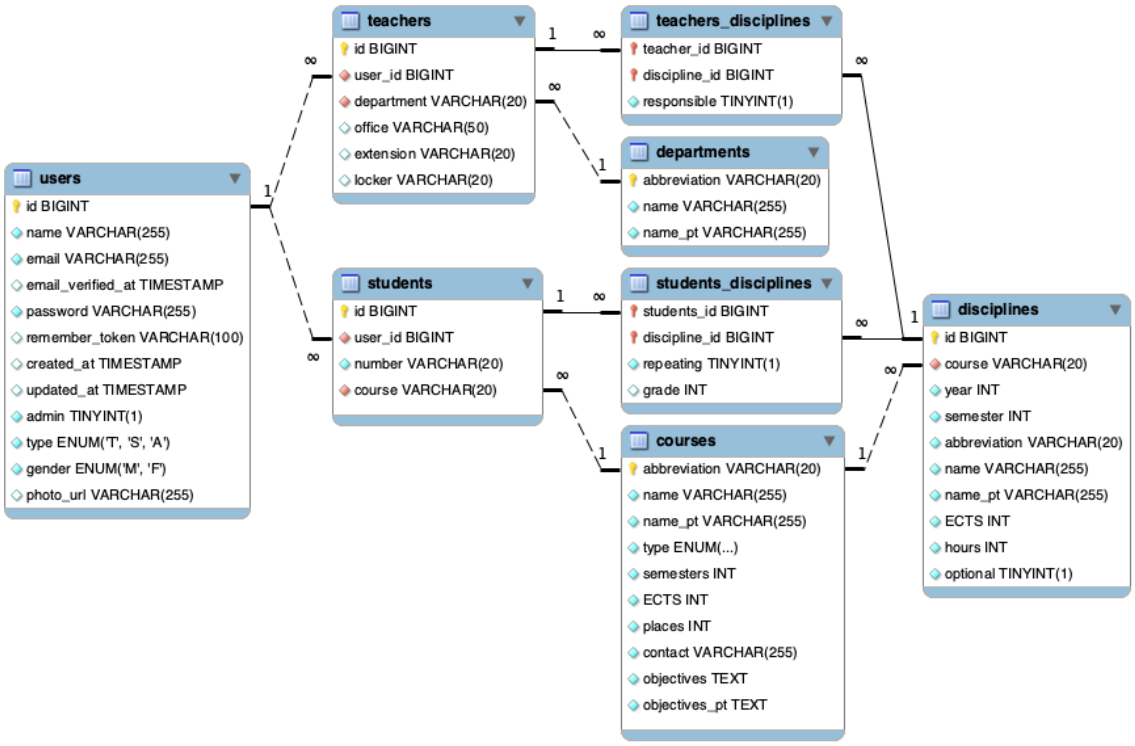
8. Execute the following command on the root of the project:
   - Command version with **Laragon** terminal:

```
php artisan db:seed
```

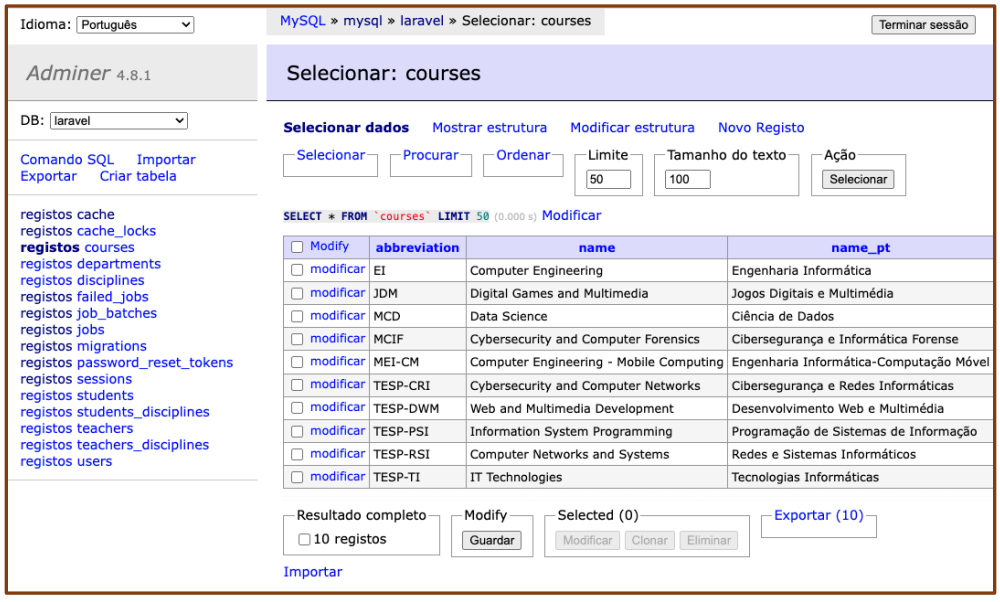
   - Command version with **Laravel Sail**:

```
sail art db:seed
```

9. Using a MySQL administration tool (HeidiSQL, Adminer, etc), ensure that the Laravel application is connected to the correct database. The structure of the database created with previous commands is:



10. Also, using a MySQL administration tool ensure that the tables have data. Check for instance the table "courses":

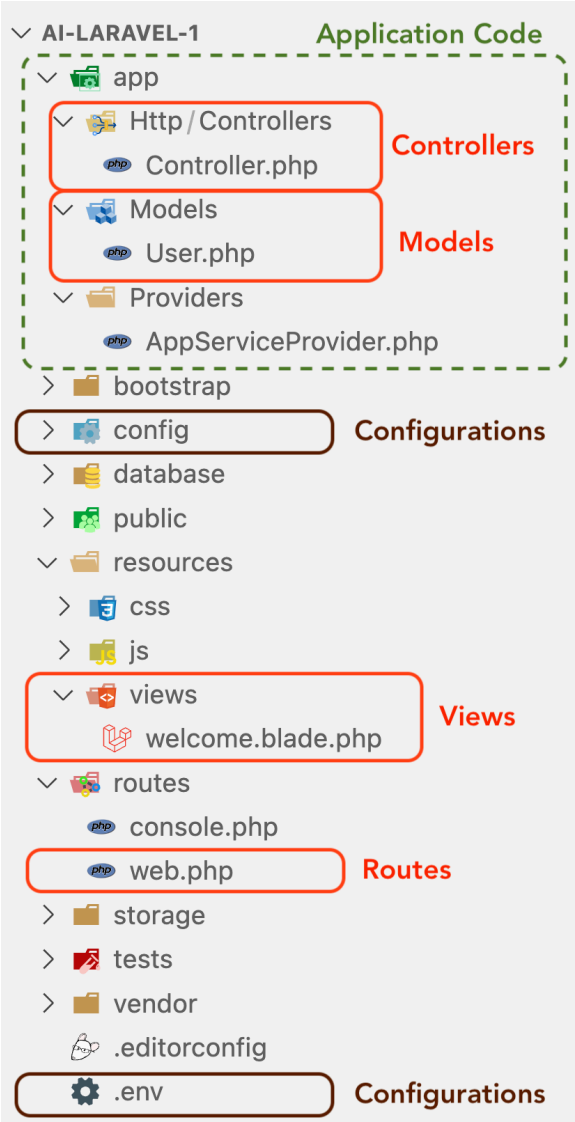


11. If the database was not correctly created by previous steps, use a MySQL administration tool to import the database from the provided file: "`laravel-database-backup`".

2. Project structure

On this section, we will analyze the structure of an empty Laravel project.

1. Using a code editor, open the newly created project. View the project folder's structure:



2. The most important folders and files, for now, are:

   - `app` – folder that includes all the user's code for the application. It includes the controllers and models, but also all other types of code structures that might be required during the development of the project.
   - `app/Models` – folder with the Models.
   - `app/Htpp/Controllers` – folder with the Controllers.
   - `resources/views` – folder with the Views.

- `routes/web.php` – file with the routes of the Web Application.
- `config` – folder with all configurations for the Application.
- `.env` – file with main configuration parameters for the Application. This file can be different on different environments (e.g. developer A, developer B, testing, preproduction, production).

3. Laravel basis: Route, Model, Controller, Views

After creating and preparing the Laravel web application and the database, we will implement our first page to show all courses, using the MVC (Model-View-Controller) pattern. For this we must create:

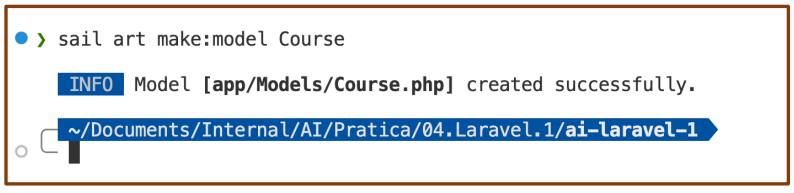
- 1 model – represents the courses from the database.
- 1 controller – it is responsible for loading the courses from the database and passing them to the view.
- 1 view – responsible for showing the list of courses (using a <table> element).
- 1 route – defines which method of the controller will handle the HTTP request (method `get`, url = "`courses`") – maps HTTP requests to methods of the controllers

1. Create the model "`Course`" by executing the following command on the root of the project:
   - Command version with **Laragon** terminal:

```
php artisan make:model Course
```

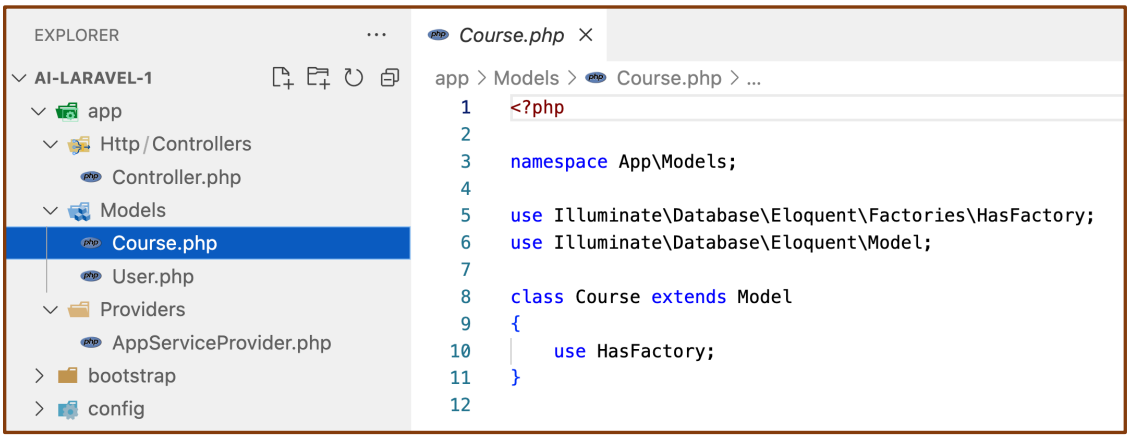
   - Command version with **Laravel Sail**:

```
sail art make:model Course
```



Note: from now on, we will only be providing one version of the artisan command.
It is the student's responsibility to adapt the command to their environment.

2. Previous command creates the Model file (`Course.php`) on the folder "**app/Models**":



- Laravel models use an **ORM** (Object Relational Mapping) library called **Eloquent**. Model classes (like the `Course` class) inherit from the class "`Illuminate\Database\`**`Eloquent`**`\Model`", which means that they inherit a large set of features to load and filter data from the associated table, create, update, or delete rows, represent data rows and columns, etc.
- One model class (e.g. "`Course`") maps/binds to one table of the database (e.g. "`courses`"), thus the name "Object Relational Mapping" – a mapping between relations (database tables) and objects.
- While the model class represents the table on the database, each model instance (each object) represents a row of the database table and is automatically filled by properties that represent columns of the database table. In this example, this means that instances of "`Course`" class will have the properties "`abbreviation`", "`name`", "`name_pt` ", "`type`", "`semesters`", "`ECTS`", "`places`", "`contact`", "`objectives`" and "`objectives_pt`".
- Multiple rows of the database table (sets of rows) are represented as collections of models (Eloquent Collections - `Illuminate\Database\Eloquent\Collection`), where each collection item is an instance of the model.
- Model classes are created by default on the "`app/Models`" folder - fully qualified class names: `App\Models\NameOfModelClass`.
    - Open the file "`composer.json`" and check the section `autoload`:

```
"autoload": {
    "psr-4": {
        "App\\": "app/",
        "Database\\Factories\\": "database/factories/",
        "Database\\Seeders\\": "database/seeders/"
    }
},
```

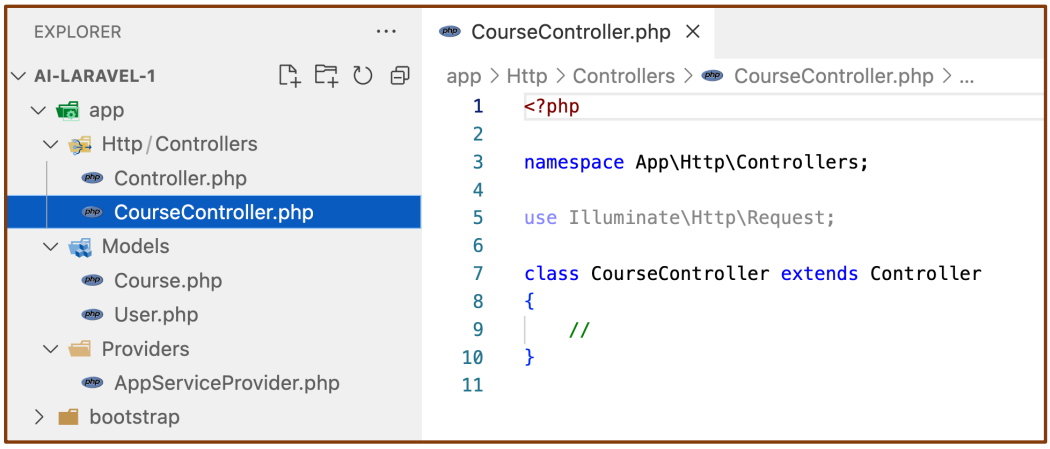
- The root folder for the namespace with the prefix "`App`" is "`app/`".

- The name of the model class is, by default, the singular in camel case of the associated table name (the name of the table is plural in lower case). This means that the model "`Course`" is associated to the table "`courses`" – this is the default behavior, which can be overridden.

3. Create the controller "`CourseController`" by executing the following command on the root of the project:

```
php artisan make:controller CourseController
```

4. Previous command creates the Controller file (`CourseController.php`) on the folder "**app/Http/Controllers**":



5. In the controller, create the **method** "`index`" that will load all `courses` from the database, (using the `Course` Eloquent model), and then it will show the view "`courses.index`" passing the list (collection) of all `courses` to the view.

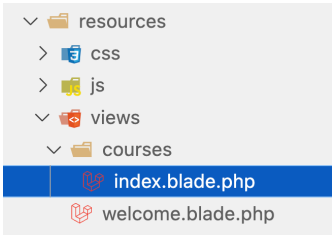
```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Models\Course;
use Illuminate\View\View;

class CourseController extends Controller
{
    public function index(): View
    {
        $allCourses = Course::all();
        return view('courses.index')->with('courses', $allCourses);
    }
}
```

- **Note** that we must add 2 “use” statements:

```php
use App\Models\Course;
use Illuminate\View\View;
```

- Eloquent models have a static method, named "`all`", that returns a collection of models that represents all rows of a database table. This method will execute the SQL command: "`select * from table`" – in this case, "`select * from courses`"
- Laravel naming conventions dictate that the `index` method of the controller refers to the method that will be responsible for showing the list of entities – in this case, the list of `courses`
- Laravel naming conventions dictate that the views relative to an entity are organized on folders (in this case, the folder would be `courses`), and the `index` view refers to the view that will be responsible for showing the list of entities – in this case, the list of `courses`

6. Next, we will create the view "`courses.index`". First, we'll create the folder `courses` on the "`resources/views`" – full name of the folder is "`resources/views/courses`". Then, we'll create the file "`index.blade.php`" on the newly created folder.



- Laravel views are defined on the folder "`resources/views`".
- Laravel view files will have the suffix `.blade.php`, because they will use the templating engine named **blade** (https://laravel.com/docs/blade).
- View names are organized by namespaces, where each namespace is a folder, and the root of the namespace is the folder `resources/views`. Example of a view name:
    - File: `resources/views/courses/index.blade.php`
    - Corresponds to the view name: `courses.index`
- Another example of a view name:
    - File: `resources/views/folder1/folder2/filename.blade.php`
    - Corresponds to the view name: `folder1.folder2.filename`

7. The first version of the "`courses.index`" view will have the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Courses</title>
</head>
<body>
    @dump($courses)
</body>
</html>
```

- The first example of a blade view uses the `@dump` directive, that "prints" a value on the page – it should be used for debugging purposes only.
- Note that, contrary to the Models and Controllers, there is no "`artisan`" command to create/generate a view. This is because Models and Controllers are specified with PHP classes, whose code follows a known pattern, and the view is specified by a blade template, whose content (mainly HTML content) does not follow a known pattern.

8. Finally, we have to create the route that will be associated to the incoming HTTP requests that have a `get` method and a relative URL "`courses`" (route will handle HTTP get requests with the following URL: `http://yourAppDomain/courses`). The route will

invoke the `index` method of the `CourseController` to handle the incoming HTTP request associated to the route. Add the following route to the "`routes/web.php`" file:

```php
<?php

use App\Http\Controllers\CourseController;
use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    return view('welcome');
});

Route::get('courses', [CourseController::class, 'index']);
```

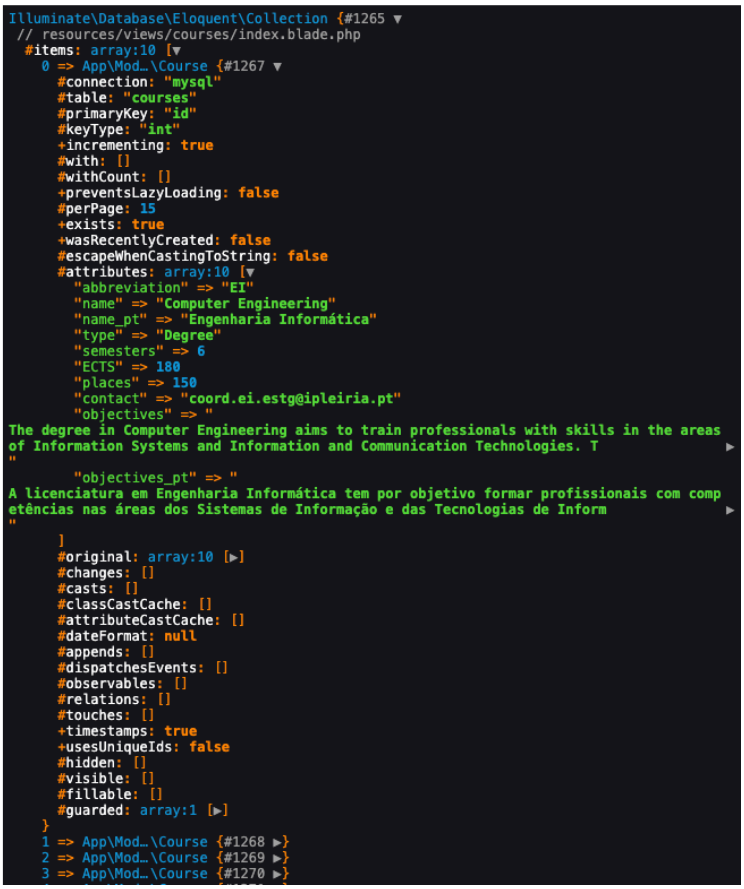
- Don't forget to add the "`use`" statement that refers to the controller class.

```php
use App\Http\Controllers\CourseController;
```

- When showing a page (returning a view), the route should always use the "**get**" method.

9. We can test our first example by opening the URL "`http://yourdomain/`**`courses`**".
   - Example with Laragon: `http://ai-laravel-1.test/courses`
   - Example with Laravel Sail: `http://localhost/courses`
   - The resulting page should be similar to:

   - Analyzing previous page, we can verify that the variable `$courses`, accessible in the view (passed on to the view by the controller) is a collection with 8 items (8 "courses").
   - When we debug the data with `@dump` blade directive, we can expand the properties of complex data. For our example, we can expand the property "items", then expand the first item (`items[0]`) and then expand the property "`attributes`":



10. So far, the project uses the MVC pattern to show a page that represents all courses. However, the representation for the data is only suitable for debugging – that is the purpose of `@dump` directive. We will change the view code (HTML + Blade directives) so that the representation of `courses` is appropriate for the end-user to read. Change the code of the view (`courses.index`) to:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Courses</title>
```

```
    <style>
        table,
        th,
        td {
            border: 1px solid black;
            border-collapse: collapse;
        }
    </style>
</head>
<body>
    <table>
        <thead>
            <tr>
                <th>Abbreviation</th>
                <th>Name</th>
                <th>Type</th>
                <th>Nº Semesters</th>
                <th>Nº Places</th>
            </tr>
        </thead>
        <tbody>
            @foreach ($courses as $course)
                <tr>
                    <td>{{ $course->abbreviation }}</td>
                    <td>{{ $course->name }}</td>
                    <td>{{ $course->type }}</td>
                    <td>{{ $course->semesters }}</td>
                    <td>{{ $course->places }}</td>
                </tr>
            @endforeach
        </tbody>
    </table>
</body>
</html>
```

- Note that there is some inline CSS (`<style>`) code that will be responsible for adding borders to the cells of the `<table>` element.
- **`@foreach`** / `@endforeach` – blade instruction will repeat a block of code. In this case, it will repeat the definition of a table row (`<tr>` element), which means that each course will have its own table row.
- **`{{ expression }}`** – blade instruction to display the content of an expression – similar to `<?php echo expression ?>`, but less verbose and cleans the output (using `htmlspecialchars` php function) to improve security.

- **`$course->abbreviation`** – `$course` is a model (an instance of the Course class), and therefore, has all columns of the associated table as properties – "`abbreviation`" is a property of the model, as well as all other columns of the table.

11. Try the URL "`http://yourdomain/courses`" again. Resulting page should be similar to:

Abbreviation	Name	Type	Nº Semesters	Nº Places
EI	Computer Engineering	Degree	6	150
JDM	Digital Games and Multimedia	Degree	6	50
MCD	Data Science	Master	4	50
MCIF	Cybersecurity and Computer Forensics	Master	4	20
MEI-CM	Computer Engineering - Mobile Computing	Master	4	40
TESP-CRI	Cybersecurity and Computer Networks	TESP	4	72
TESP-DWM	Web and Multimedia Development	TESP	4	52
TESP-PSI	Information System Programming	TESP	4	71
TESP-RSI	Computer Networks and Systems	TESP	4	10
TESP-TI	IT Technologies	TESP	4	10

AQUI - PL4

4. Debugging with Laravel Telescope

In the PHP / Laravel community there are several tools that help us debug our application. We will use "`Laravel Telescope`", that will help us analyze what happens from the moment the HTTP request "arrives" on the server up until the moment the HTTP response is sent back to the client. We can view the data included on the HTTP request, database commands generated by Eloquent ORM, custom user messages, etc.

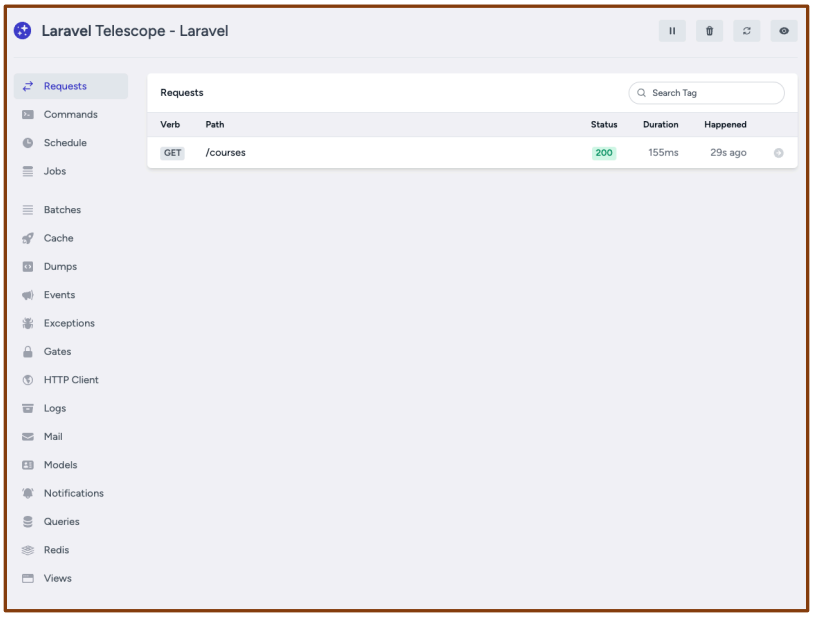
1. First, let us install Laravel Telescope. We will use the `--dev` option, so that the Telescope package is installed only locally (when publishing the application on a "proper" server, the package will not be installed). Execute the following commands:

```
composer require laravel/telescope --dev
```

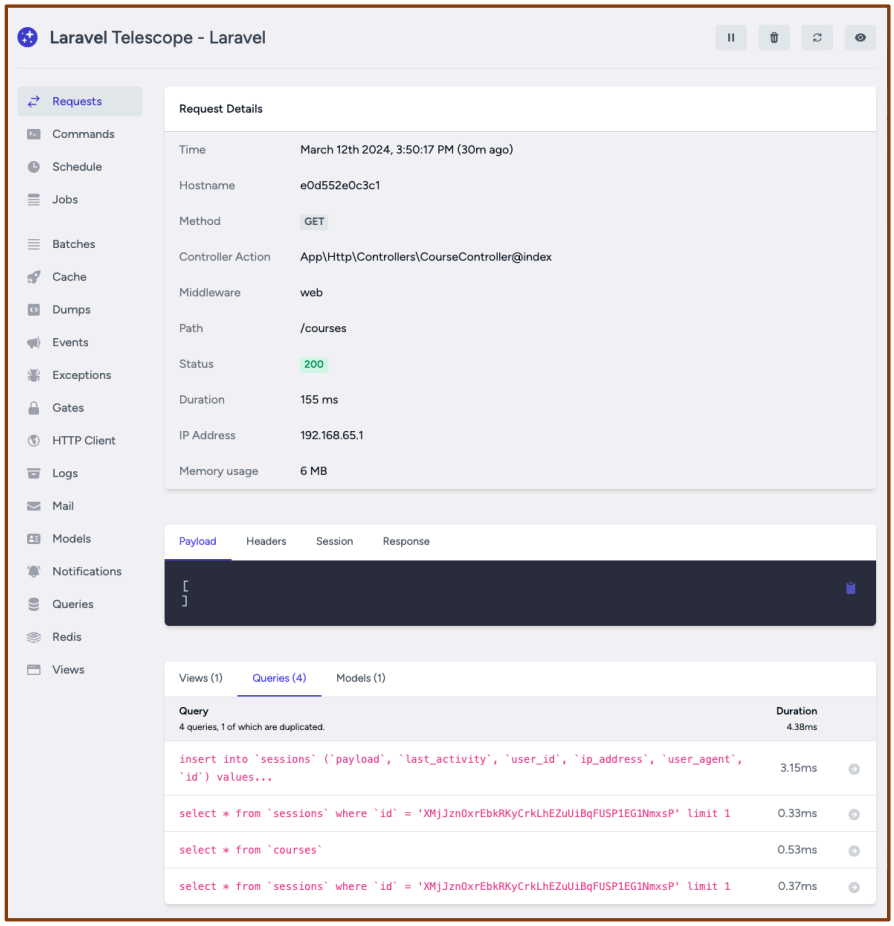
```
php artisan telescope:install
```

```
php artisan migrate
```

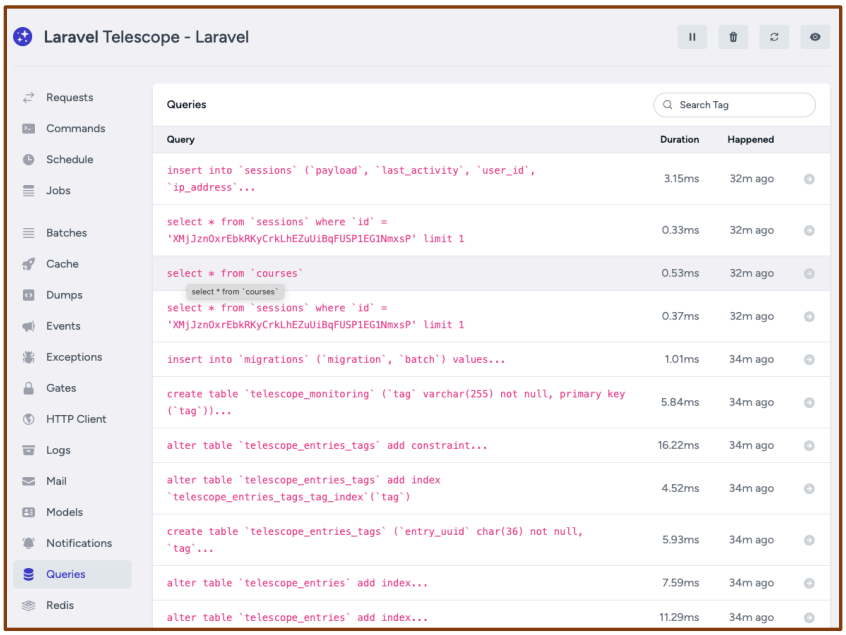
2. Try the URL "`http://yourdomain/courses`" again – just to ensure that the server receives one HTTP request.
3. Open the Telescope user interface with the URL "`http://yourdomain/telescope`". Telescope main dashboard is similar to:



4. Analyze the "`http://yourdomain/courses`". Click on the right arrow (right of the request row) of the HTTP request to analyze (`GET /courses`). This will open detailed information about how the web server handled the HTTP request. Several details about the HTTP Request and Response; models, views and controller used; database queries (SQL commands) executed, etc. Example:



- Note that we can access all database queries used by Eloquent ORM when handling one request (as viewed on previous image) or access all database queries executed on any context by selecting the menu option “Queries”:



5. In summary, to use Telescope we just continue to use the web application as usual on one browser (or tab of the browser) and view the telescope UI on another browser or tab, using the URL: `http://yourdomain/telescope`

6. Laravel Telescope is enabled by default. If you want to disable Laravel Telescope, just add the following line to the configuration "`.env`" file:

```
TELESCOPE_ENABLED=false
```

7. After changing the "`.env`" file, try to open the telescope page. It should return a “404 Not found” error.

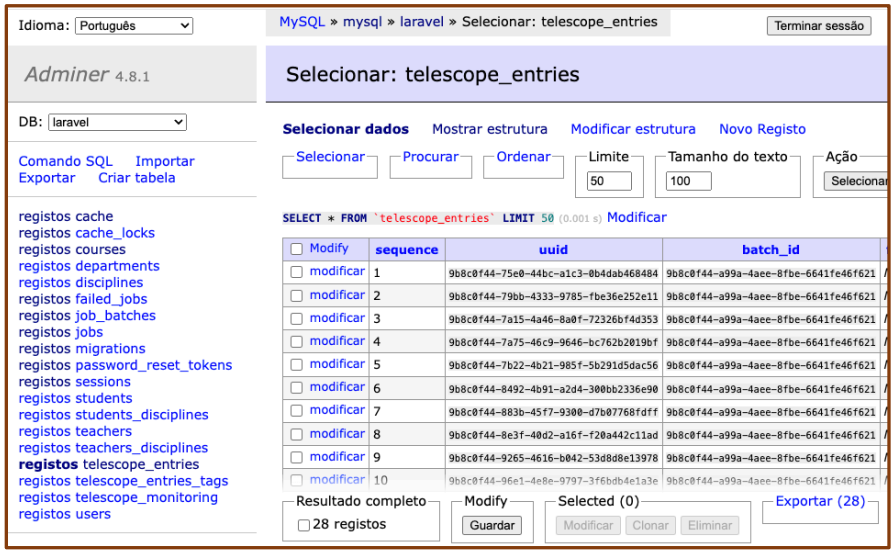
8. Remove the line from the “.env” file, or change it to:

```
TELESCOPE_ENABLED=true
```

9. When we installed Telescope, we have executed a migration (command: `php artisan migrate`) which created 3 telescope related tables on the database, namely:

   - `telescope_entries`
   - `telescope_entries_tags`
   - `telescope_monitoring`

10. Each HTTP request handled by the web application will create several log entries on these tables. Check the data on these three tables – there will be several entries already (particularly on the table `telescope_entries`)

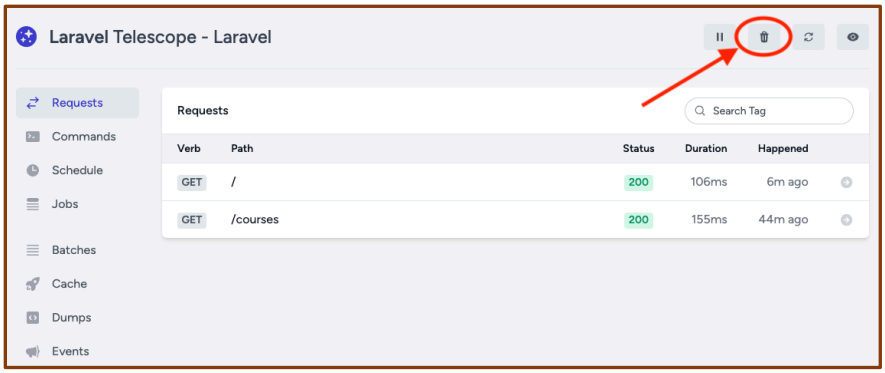


11. From time to time, we have to clear these tables to avoid creating a performance bottleneck on our application. To clear the entries, we can:

- Execute the following command:

```
php artisan telescope:clear
```

- Or, on the Telescope dashboard, click on the “trash” icon:

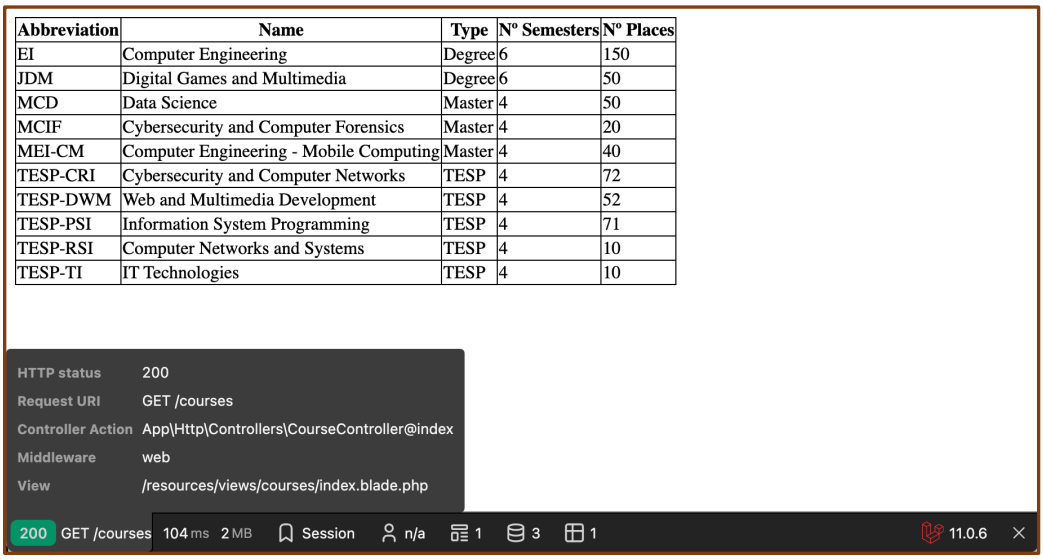


12. We will complement Laravel Telescope with the package "**`laravel telescope toolbar`**", that will show a small toolbar at the bottom of our pages with a summary of the information that Telescope provide us. Also, it will simplify the process of sending custom user messages to Telescope. To install the package "laravel telescope toolbar", execute these 2 commands on the terminal/shell:

```
composer require fruitcake/laravel-telescope-toolbar --dev
```

```
php artisan vendor:publish --
provider="Fruitcake\\TelescopeToolbar\\ToolbarServiceProvider"
```

13. Now, when we execute our web application all our pages have a toolbar at the bottom. Analyze what is available on the toolbar and try to click the options on the toolbar.



Abbreviation	Name	Type	Nº Semesters	Nº Places
EI	Computer Engineering	Degree	6	150
JDM	Digital Games and Multimedia	Degree	6	50
MCD	Data Science	Master	4	50
MCIF	Cybersecurity and Computer Forensics	Master	4	20
MEI-CM	Computer Engineering - Mobile Computing	Master	4	40
TESP-CRI	Cybersecurity and Computer Networks	TESP	4	72
TESP-DWM	Web and Multimedia Development	TESP	4	52
TESP-PSI	Information System Programming	TESP	4	71
TESP-RSI	Computer Networks and Systems	TESP	4	10
TESP-TI	IT Technologies	TESP	4	10

- Note that clicking on the bottom right will show or hide the toolbar.

14. The toolbar will only be visible when the application is in debug mode. Try to disable the debug mode by changing the parameter APP_DEBUG of the file "`.env`":

```
APP_DEBUG=false
```

15. Now, our pages will not have the toolbar available. Revert the "`.env`" configuration to :

```
APP_DEBUG=true
```

16. We can also control the visibility of the toolbar with the "`.env`" parameter TELESCOPE_TOOLBAR_ENABLED. Add this to the file ".env":

```
TELESCOPE_TOOLBAR_ENABLED=false
```

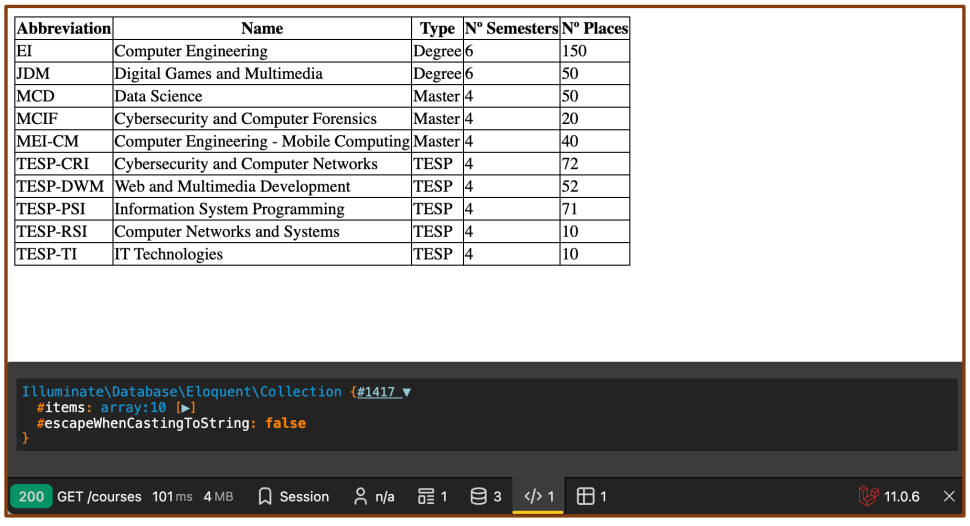
17. Once again, the toolbar is not visible. To make the toolbar visible again, change the "`.env`" configuration to:

```
TELESCOPE_TOOLBAR_ENABLED=true
```

18. Besides the visual toolbar, the "`laravel-telescope-toolbar`" package also adds the function `debug`, which is an excellent alternative to the dump() function.
    `debug(expression)` function will "dump" the expression value to the Telescope "Dump tab" without ever showing it in the application (even when Telescope is not available). Add the following debug() instruction to the class `CourseController` on the file: `app/Http/Controllers/CourseController.php`.

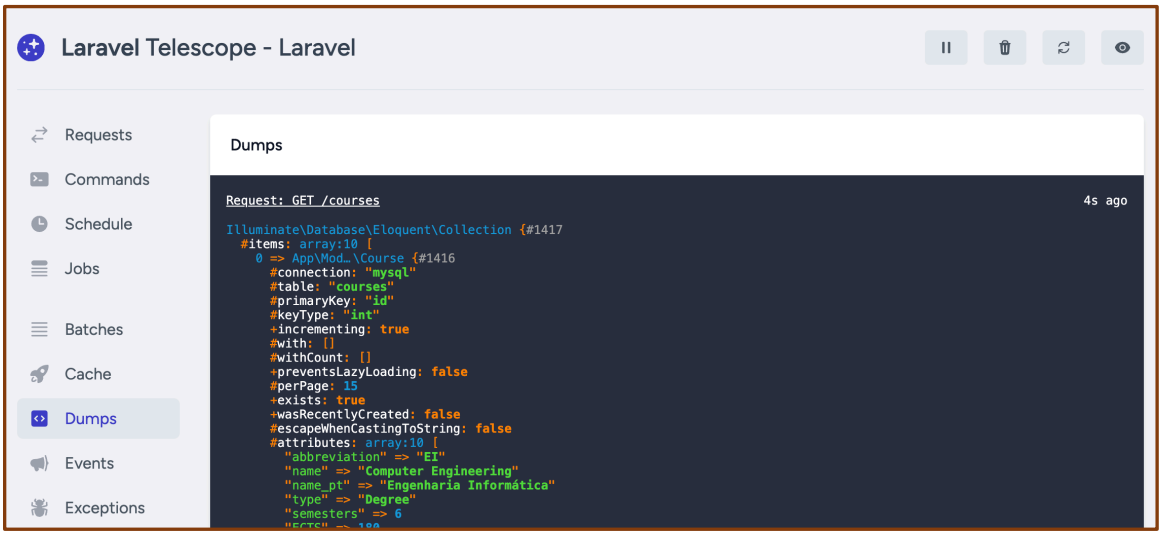
```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Models\Course;
use Illuminate\View\View;

class CourseController extends Controller
{
    public function index(): View
    {
        $allCourses = Course::all();
        debug($allCourses);
        return view('courses.index')->with('courses', $allCourses);
    }
}
```

19. Open the courses page again and verify that the custom debug message (with the value of `$allCourses` variable) is available on the "dumps" icon (`</>`) of the toolbar:



Abbreviation	Name	Type	Nº Semesters	Nº Places
EI	Computer Engineering	Degree	6	150
JDM	Digital Games and Multimedia	Degree	6	50
MCD	Data Science	Master	4	50
MCIF	Cybersecurity and Computer Forensics	Master	4	20
MEI-CM	Computer Engineering - Mobile Computing	Master	4	40
TESP-CRI	Cybersecurity and Computer Networks	TESP	4	72
TESP-DWM	Web and Multimedia Development	TESP	4	52
TESP-PSI	Information System Programming	TESP	4	71
TESP-RSI	Computer Networks and Systems	TESP	4	10
TESP-TI	IT Technologies	TESP	4	10

20. Open the tab "`Dumps`" of the telescope dashboard (`http://yourdomain/telescope`). The debug messages are also presented on this tab.



- Note that when multiple debug messages are visible, they are sorted from the newer to the oldest message (newer message will appear on top and oldest messages on the bottom)

5. CRUD – Part 1 (create and update)

Up until now, we've only added one GET route to our web application – to show all courses. Next, we will guarantee that "courses" entity support all CRUD operations – CRUD is acronym for: **C**reate **R**ead **U**pdate and **D**elete.
CRUD operations are the typical minimum operations associated to an entity, so that the end-user can view and manipulate the information about that entity. That does not mean that an entity is limited to the CRUD operations. CRUD operations usually include:

- Read
    1. A page that **lists** all entities (all courses) or shows a filtered list of entities (depends on the data size).
    2. A page that **shows** the details of **one** entity (one course) – this can correspond to a page with a read-only representation of the entity, or we can consider that the form to update the entity is also responsible for showing the detailed view.

- Create

    3. A page that shows the **form** for the user to fill the data necessary to **create** an entity (create a course)
    4. The **operation** to **create** (insert) the entity when the user submits data of the create form.

- Update

    5. A page that shows the **form** for the user to fill the data necessary to **update** an entity (update a course). This form should be preloaded with the existing entity data, so that the end-user updates only the information he wants (the columns/properties that have changed).
       This can also be used to show the detail of one entity.
    6. The **operation** to **update** one entity when the user submits data of the update form.

- Delete

    7. The **operation** to **delete** one entity.

As referred previously, an entity is not limited to the set of operations defined by CRUD. For instance, a student entity can have CRUD operations to show and manipulate basic data, but can also have operations like course registration; view students' grades; send notifications to student's email; etc.

1. Let start to implement the **create** operation for the courses, which will be comprised of 2 routes:
    1. Method **`GET`**, url "**`courses/create`**"– this route will be responsible for showing the form to create a "course"
    2. Method **`POST`**, url "**`courses`**" – this route will be responsible for the operation to create (insert) a "course"
2. Add the following code to the route's definition file ("`routes/web.php`")

```
Route::get('courses/create', [CourseController::class, 'create']);
Route::post('courses', [CourseController::class, 'store']);
```

- Note that both the methods and URLs follow the good practices / name conventions for the create operations on a REST application / Laravel application, namely:
    - Method **get** and url: `entities(plural)/create` to show the form to create
    - Method **post** and url: `entities(plural)` for the operation to create (insert)

- Also, note that Laravel naming conventions dictate that the "`create`" method of the controller refers to the method that will be responsible for showing the form to create, and "`store`" is the method of the controller that is responsible for the create operation.

3. Add the method "`create`" to the `CourseController` class - file `app/Http/Controllers/CourseController.php`:

```
. . .
class CourseController extends Controller
{
    . . .
    public function create(): View
    {
        return view('courses.create');
    }
}
```

- The controller method "`create`" only shows the form to create a new course (the view `courses.create` will have the HTML with form to create). It will not do any actual insert/create operation.

4. Add the view `courses.create` (file: `resources/views/courses/create.blade.php`) with the HTML page that "draws" the form to create a course.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Course</title>
</head>
<body>
    <h2>New Course</h2>
    <form method="POST" action="/courses">
        @csrf
        <div>
            <label for="inputAbbreviation">Abbreviation</label>
            <input type="text" name="abbreviation" id="inputAbbreviation" >
        </div>
        <div>
            <label for="inputName">Name</label>
            <input type="text" name="name" id="inputName" >
        </div>
```

```
        <div>
            <label for="inputName_pt">Name (PT)</label>
            <input type="text" name="name_pt" id="inputName_pt" >
        </div>
        <div>
            <label for="inputType">Type of course</label>
            <select name="type" id="inputType">
                <option>Degree</option>
                <option>Master</option>
                <option>TESP</option>
            </select>
        </div>
        <div>
            <label for="inputSemesters">Semesters</label>
            <input type="text" name="semesters" id="inputSemesters">
        </div>
        <div>
            <label for="inputECTS">ECTS</label>
            <input type="text" name="ECTS" id="inputECTS">
        </div>
        <div>
            <label for="inputPlaces">Places</label>
            <input type="text" name="places" id="inputPlaces">
        </div>
        <div>
            <label for="inputContact">Contact</label>
            <input type="text" name="contact" id="inputContact">
        </div>
        <div>
            <label for="inputObjectives">Objectives</label>
            <textarea name="objectives" id="inputObjectives" rows=10></textarea>
        </div>
        <div>
            <label for="inputObjectives_pt">Objectives (PT)</label>
            <textarea name="objectives_pt" id="inputObjectives_pt" rows=10></textarea>
        </div>
        <div>
            <button type="submit" name="ok">Save new course</button>
        </div>
    </form>
</body>
</html>
```

- Note that the method of the form is POST, and the action is `courses`. This means that when the user submits the form (clicks on the submit button – “Save new course”) the browser will send an HTTP POST request to the `http://yourdomain/courses`, which will be handled by the “post” route created previously.

- Note that the form includes the blade directive `@csrf`. By default, all Laravel "post" forms must include this directive `@csrf`, which will protect your application from cross site request forgery (CSRF) attacks. `@csrf` directive generates a CSRF "token" for each active user session managed by the application. This token is used to verify that the authenticated user is the person making the requests to the application.
- Note that all names of the fields (`<input>`; `<select>`; `<textarea>`) are the same as the names of the table columns / model properties. This will ensure that when inserting (or updating) data we don't need to convert between form data names and model data names.
- Note that the `<option>` elements of the `<select>` do not have the attribute value. This is because the value is the same as the option content.

5. To handle the HTTP POST request when the form is submitted, add the method "`store`" to the `CourseController` class - file `app/Http/Controllers/CourseController.php`:

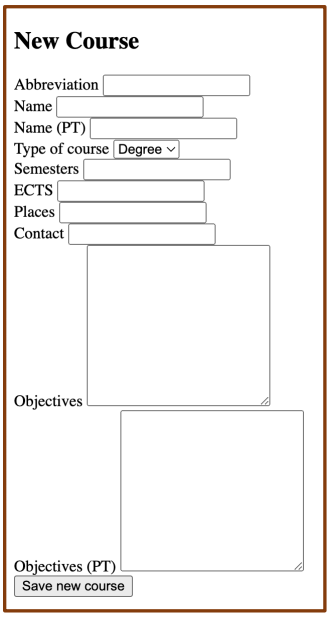
```
. . .
use Illuminate\Http\RedirectResponse;
. . .
class CourseController extends Controller
{
    . . .
    public function store(Request $request): RedirectResponse
    {
        Course::create($request->all());
        return redirect('/courses');
    }
}
```

- The controller method "`store`" is the responsible for the actual insert/create operation on the database.
- **`$request->all()`** is an array with all field values submitted by the user.
- **`Course::create`**`( … data_array …)` – `create` is a model (Course) static method that will insert a row on the database. The values (columns) to insert are passed on to the method as an array. This will execute a SQL `insert` command on the database.
- The store method does not return a view – it returns a **`RedirectResponse`**. This is the pattern for controller methods that insert, update, delete or execute some type of operation that changes data.

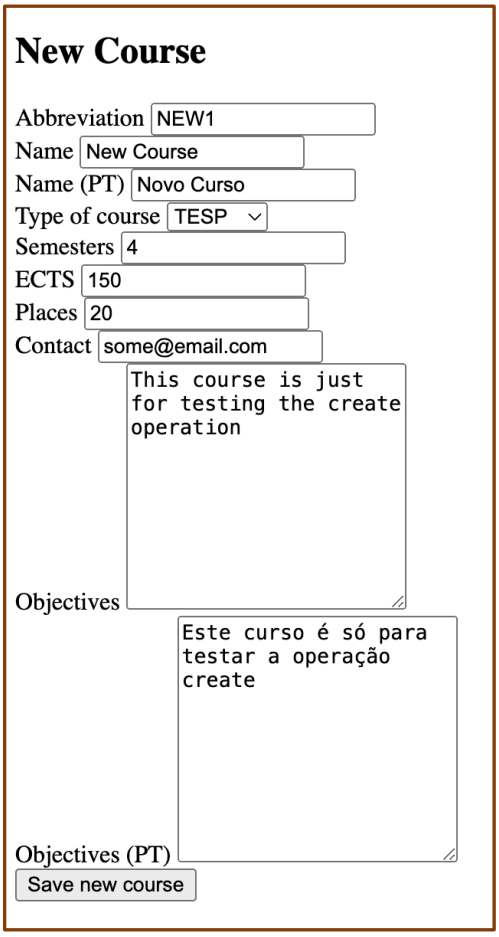
This means that after inserting a row on the database, the end-user will be redirected to the "`courses`" page – the browser will receive a response with information about where to redirect, and it will create a new HTTP GET request with the url `http://yourdomain/courses`

- Note that we've added the "use": `use Illuminate\Http\RedirectResponse;`

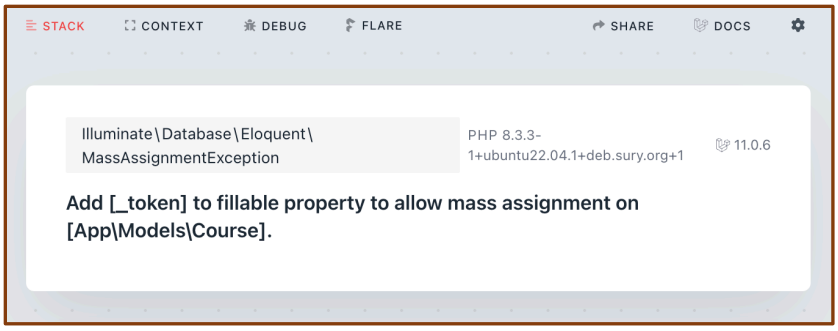
6. Open the page to insert new "course" (`http://yourdomain/courses/create`).



7. Fill out the "new course" form fields with some example data and submit the form by clicking on the "Save new course" button. Note that we haven't implemented any validations, so it is important to fill the form with correct (valid) values. Example:



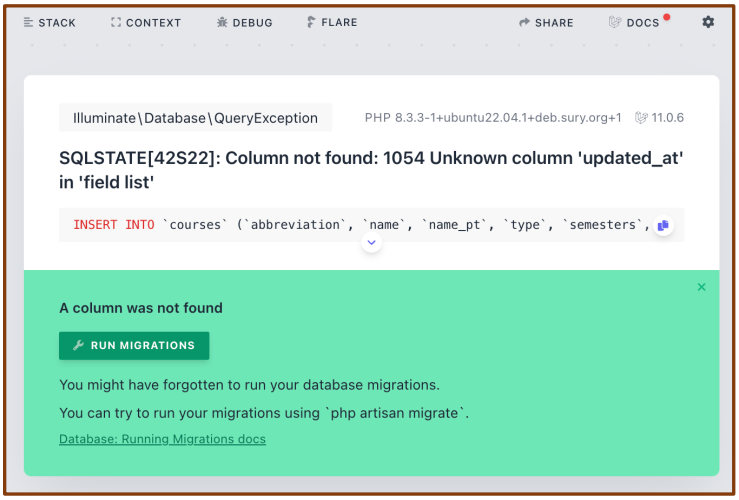
8. When the user submits the form (clicks on the button “Save new course”) an error occurs:



9. This error (“*mass assignment on ‘Model’*”) happens because, when executing a model method that inserts or updates all columns at once (as the method `create` does) we have to specify, on the model, which properties (columns) are fillable (which are included on the mass assignment). Add the `$fillable` property to the `Course` model (file `app/Models/Course.php`) to configure which properties are fillable:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Course extends Model
{
    use HasFactory;

    protected $fillable = [
        'abbreviation',
        'name',
        'name_pt',
        'type',
        'semesters',
        'ECTS',
        'places',
        'contact',
        'objectives',
        'objectives_pt',
    ];
}
```

10. Try to insert a new "`course`" again (`http://yourdomain/courses/create`). Use the same field values as were used on step 7.
11. Once again, an error occurs:



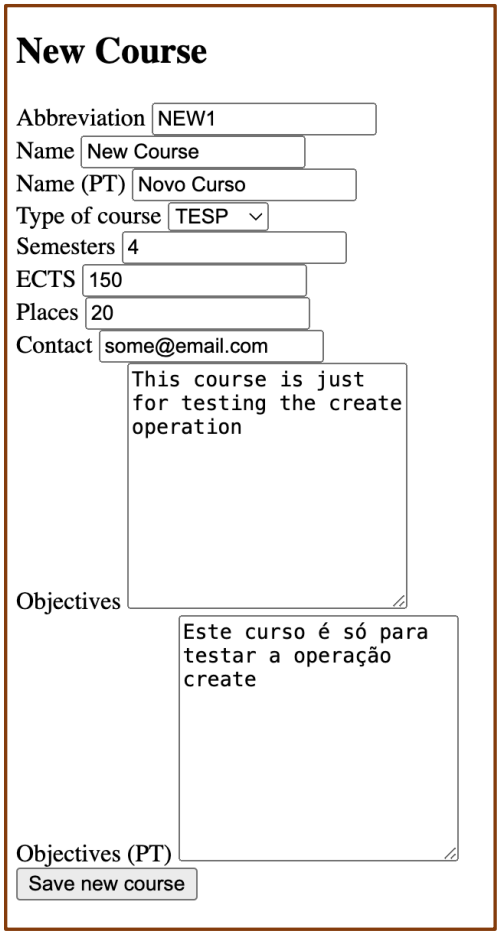
12. This error ("*SQLSTATE[42S22]: Column not found: 1054 Unknown column 'updated_at' in 'field list*") usually happens because the table structure on the database does not follow one of the Eloquent conventions, that dictates that the table associated to the model should have

2 timestamp columns: "`created_at`" and "`updated_at`". Since `courses` table does not have these 2 columns, we must configure the model to ignore this convention. Add the `$timestamps` property to the `Course` model (file `app/Models/Course.php`):

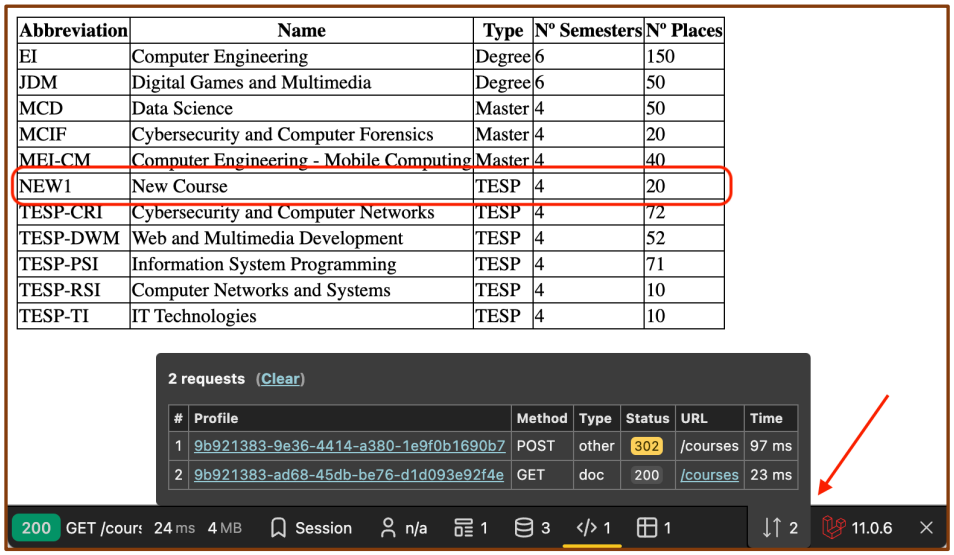
```
class Course extends Model
{
    . . .
    public $timestamps = false;
}
```

- `public $timestamps = false;` indicates that the table associated to the current model, does not have the timestamps columns `created_at` and `updated_at`.

13. Try to insert a new "`course`" again (`http://yourdomain/courses/create`). Use the same field values as were used on step 7:



14. This time, the create operation should work correctly. After the operation is concluded, the browser will redirect us to the "courses" page.



- Note that the table (`<table>`) has a new row with the values that were specified on the form – we can also open the table on the database and view the new row.
- If we analyze the requests using Laravel Telescope Toolbar, we can verify that from the moment the user submits the form until the moment the user views the table with all course, 2 requests were made:
    - First request was the **POST** (to store the data from the form) and the server returned a response with the status code 302 – that was the **Redirect** Response as specified on the controller method "`store`".
    - The second request was made by the browser (that is the responsible for redirection) and is a "simple" **GET** request to the url "`/courses`" – this request returns the list of courses.
- In summary, a **redirection** always involves **2 requests/responses** – the first request returns a response with redirection information (status code 302, and the "location" header with the url to redirect to) and the second request/response (made by the browser) is relative to the location specified on the redirect.

15. Use telescope to further analyze what has happened – view the 2 requests involved. For the post request, check the payload (data from the form sent to the server), the response, the model, the SQL queries used by the Eloquent ORM, etc.
16. Although the model `Course` is working perfectly for now, it is always a good practice to ensure that the model follows all **Eloquent conventions**, and if that is not the case, to configure the model accordingly. Read about eloquent conventions on https://laravel.com/docs/eloquent#eloquent-model-conventions

Laravel Eloquent assumes a set of conventions when mapping a Model to a table. If any of the conventions are not valid for the current model, then the model should be explicitly configured to ignore that convention (this analysis should be made for every new Model created on the project). Conventions:

- **Table name** is the plural name of the model, using "snake case". Valid examples:
    - Model = `Course` - Table Name = `courses`
    - Model= `AirTrafficController` Table name = `air_traffic_controllers`
    - If the table name does not follow this convention, configure the model with:

```
protected $table = 'table_name_on_db';
```

- The **primary key** of the table is the column named **id**.
    - If the table primary key is not called id, configure the model with:

```
protected $primaryKey = 'primary_key_column';
```

- The **primary key** of the table is an **incrementing** integer (autoinc).
    - If the table primary key is not an incrementing integer, configure the model:

```
public $incrementing = false;
```

- The **primary key** of the table is an **integer**.
    - If the table primary key is not an integer, configure the model with:

```
protected $keyType = 'string';
```

- The table must have 2 **timestamps** columns: **created_at** and **updated_at**.
    - If the table does not include these columns, configure the model with:

```
public $timestamps = false;
```

    - It is also possible to change these columns names and date format – check the documentation.

17. After reviewing the Eloquent conventions for model "`Course`", we can verify that the conventions are not followed for the timestamps fields (already handled) and for the primary

key name, autoincrement and type. Edit `Course` model (file "`app/Models/Course.php`") to configure the model correctly:

```
class Course extends Model
{
    . . .
    protected $fillable = . . .

    public $timestamps = false;
    protected $primaryKey = 'abbreviation';
    public $incrementing = false;
    protected $keyType = 'string';
}
```

- When creating a new model class, always **start by reviewing Eloquent conventions** and configure the model if necessary.

18. After the create operation, let us now implement the **update** operation for the courses which will be comprised of 2 routes:
    1. Method **`GET`**, url "**`courses/{course}/edit`**"– this route will be responsible for showing the form to update (edit) a specific course - `{course}` is the route parameter that specifies which course to edit.
    2. Method **`POST`**, url "**`courses/{course}`**" – this route will be responsible for the operation to update the specified (by the route parameter) course
19. Add the following code to the route's definition file ("`routes/web.php`")

```
Route::get('courses/{course}/edit', [CourseController::class, 'edit']);
Route::put('courses/{course}', [CourseController::class, 'update']);
```

- **`{course}`** is a **route parameter**. The parameter value will be passed on to the controller method.
    - For example, this route ("`courses/{course}/edit`") will accept URLs like "`courses/EI/edit`" or "`courses/JDM/edit`". In these examples, the parameter value would be `EI` and `JDM`.
    - When the route is relative to one model entity (e.g. edit one course), the parameter value should be the primary key of the associated table. For example, the "abbreviation" of the "course" (current example) or the "id" of the student

- The methods and URLs follow the good practices / name conventions for the update operations on a REST application / Laravel application, namely:
    - Method **get** and url: `entities(plural)/{entity}/edit` to show the form to update
    - Method **put** and url: `entities(plural)/{entity}` for the operation to update the entity.
- Also, Laravel naming conventions dictate that the "`edit`" method of the controller refers to the method that will be responsible for showing the form to update, and "`update`" is the method of the controller that is responsible for the update operation.

20. Add the methods "**`edit`**" and "**`update`**" to the `CourseController` class (file: `app/Http/Controllers/CourseController.php`):

```
class CourseController extends Controller
{
    . . .

    public function edit(Course $course): View
    {
        return view('courses.edit')->with('course', $course);
    }

    public function update(Request $request, Course $course): RedirectResponse
    {
        $course->update($request->all());
        return redirect('/courses');
    }
}
```

- The controller method "**`edit`**" is responsible for showing the view (the HTML with form to update). It is the method "**`update`**" that will update a row on the database – using the method "update" of the Course model instance.
- Note that both methods (`edit` and `update`) include a `Course $course` parameter, which is bonded to the route parameter with the same name. Using an implicit "Route-Model Binding" (https://laravel.com/docs/routing#route-model-binding) the route parameter accepts the model primary key (`abbreviation` for `Course` model) and Laravel automatically loads the model instance from the database and pass it on to the controller method. The model instance is injected automatically to the controller – the programmer only uses that value.

    - Laravel automatically resolves Eloquent models defined in routes or controller actions whose type-hinted variable names match a route segment name. If a matching model instance is not found in the database, a 404 HTTP response will automatically be generated.

- The model instance (`$course`) is passed on to the view "`courses.edit`" with the same name (`course`).
- `$course->update( … data_array …)` – updates the model `$course` with a new set of values (properties) passed on to the method as an array. This will execute a SQL `update` command on the database.

21. Add the view `courses.edit` (file: `resources/views/courses/edit.blade.php`) with the HTML page that "draws" the form to edit a course. Note that the variable `$course` is the model instance of the course to edit on the form.

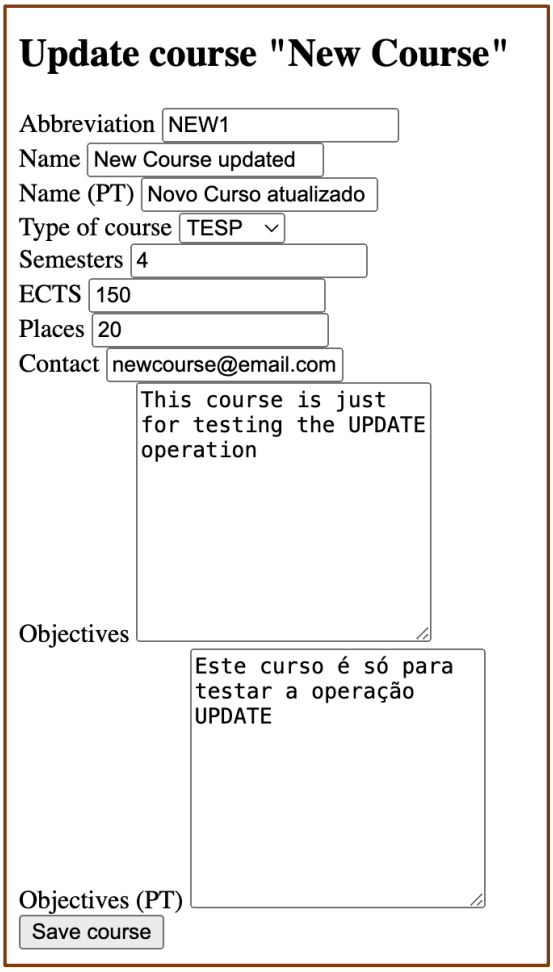
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Course</title>
</head>
<body>
    <h2>Update course "{{ $course->name }}"</h2>

    <form method="POST" action="/courses/{{$course->abbreviation}}">
        @csrf
        @method('PUT')
        <div>
            <label for="inputAbbreviation">Abbreviation</label>
            <input type="text" name="abbreviation" id="inputAbbreviation"
                   value="{{$course->abbreviation}}">
        </div>
        <div>
            <label for="inputName">Name</label>
            <input type="text" name="name" id="inputName" value="{{$course->name}}">
        </div>
        <div>
            <label for="inputName_pt">Name (PT)</label>
            <input type="text" name="name_pt" id="inputName_pt"
                   value="{{$course->name_pt}}">
        </div>
```

```
        <div>
            <label for="inputType">Type of course</label>
            <select name="type" id="inputType">
                <option {{$course->type == 'Degree' ? 'selected' : ''}}>Degree</option>
                <option {{$course->type == 'Master' ? 'selected' : ''}}>Master</option>
                <option {{$course->type == 'TESP' ? 'selected' : ''}}>TESP</option>
            </select>
        </div>
        <div>
            <label for="inputSemesters">Semesters</label>
            <input type="text" name="semesters" id="inputSemesters"
                   value="{{$course->semesters}}">
        </div>
        <div>
            <label for="inputECTS">ECTS</label>
            <input type="text" name="ECTS" id="inputECTS" value="{{$course->ECTS}}">
        </div>
        <div>
            <label for="inputPlaces">Places</label>
            <input type="text" name="places" id="inputPlaces" value="{{$course->places}}">
        </div>
        <div>
            <label for="inputContact">Contact</label>
            <input type="text" name="contact" id="inputContact"
                   value="{{$course->contact}}">
        </div>
        <div>
            <label for="inputObjectives">Objectives</label>
            <textarea name="objectives" id="inputObjectives" rows=10>
                 {{$course->objectives}}
            </textarea>
        </div>
        <div>
            <label for="inputObjectives_pt">Objectives (PT)</label>
            <textarea name="objectives_pt" id="inputObjectives_pt" rows=10>
                  {{$course->objectives_pt}}
            </textarea>
        </div>
        <div>
            <button type="submit" name="ok">Save course</button>
        </div>
    </form>
</body>
</html>
```

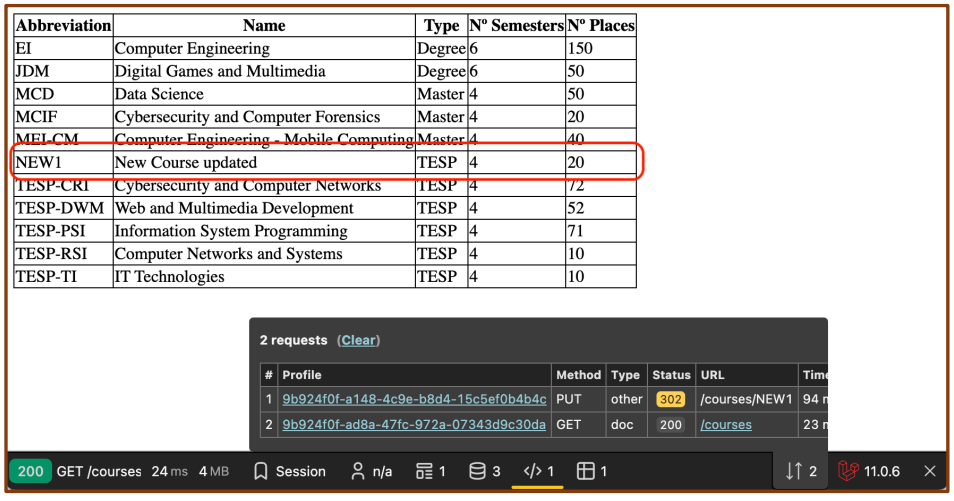
- This form also includes the blade directive `@csrf` (the same as the form to create)
- The view assumes that the variable `$course` has the course model to be updated.

- The action of the form (URL) is defined by `/courses/{{$course->abbrebviation}}`, which means that the URL format will be similar to: "`/courses/EI`". This allows us to specify which course will be updated ("abbreviation" is the "courses" primary key).
- The method of this form is POST – not PUT. This is because HTML forms only supports the methods GET or POST. HTTP requests with methods PUT, PATCH, DELETE or any other (except GET and POST) must be called using JavaScript code.
- The form includes the blade directive **`@method('PUT')`**. This blade directive adds a hidden field (name= `_method`, value = `PUT`) so that Laravel handles the POST request as if it is a PUT request.
    - When Laravel decides which route will handle the arriving HTTP request, it checks if the `_method` field is present. If that field is present, instead of considering the real HTTP method of the request, Laravel will consider the method specified on the `_method` field.
- The update form is initiated with data from the model. This is implemented by setting the value of the fields. Example: `. . . value="{{$course->name}}">`
    - Analyze all `<input>` elements.
    - Check how the `<select>` and `<option>` elements values are initiated.
    - Analyze `<textarea>` elements and verify how the value is initiated.

22. Try to update a `course` (open the page `http://yourdomain/courses/NEW1/edit`).



23. This should work correctly. After updating the row on the database, the end user is redirected to the “/course” page.



Abbreviation	Name	Type	Nº Semesters	Nº Places
EI	Computer Engineering	Degree	6	150
JDM	Digital Games and Multimedia	Degree	6	50
MCD	Data Science	Master	4	50
MCIF	Cybersecurity and Computer Forensics	Master	4	20
MEI-CM	Computer Engineering - Mobile Computing	Master	4	40
NEW1	New Course updated	TESP	4	20
TESP-CRI	Cybersecurity and Computer Networks	TESP	4	72
TESP-DWM	Web and Multimedia Development	TESP	4	52
TESP-PSI	Information System Programming	TESP	4	71
TESP-RSI	Computer Networks and Systems	TESP	4	10
TESP-TI	IT Technologies	TESP	4	10

- Analyze the 2 requests involved using Laravel Telescope. Check the requests, the payload (data from the form sent to the server), the response, the model, the SQL queries used by the Eloquent ORM, etc.

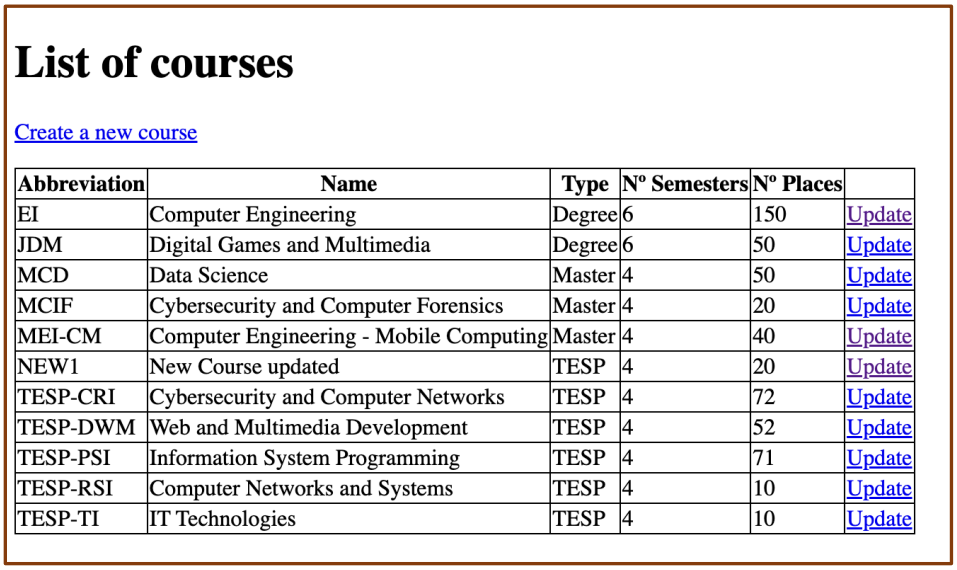
6. Named routes and URL generation

Now that we have 2 pages for updating and creating courses is time to add hyperlinks on the list of courses page (`http://yourdomain/courses`) to open these pages from there.

1. Change the view `courses.index` (file `resources/views/courses/index.blade.php`):

```
<!DOCTYPE html>
   . . .
<body>
    <h1>List of courses</h1>
    <p><a href="/courses/create">Create a new course</a></p>
    <table>
        <thead>
            <tr>
                <th>Abbreviation</th>
                 . . .
                <th>Nº Places</th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            @foreach ($courses as $course)
                <tr>
                    <td>{{ $course->abbreviation }}</td>
                    . . .
                    <td>{{ $course->places }}</td>
                    <td>
                        <a href="/courses/{{$course->abbreviation}}/edit">Update</a>
                    </td>
                </tr>
            @endforeach
        </tbody>
    </table>
</body>
</html>
```

2. Try to open the page `http://yourdomain/courses`



List of courses

Create a new course

Abbreviation	Name	Type	Nº Semesters	Nº Places	
EI	Computer Engineering	Degree	6	150	Update
JDM	Digital Games and Multimedia	Degree	6	50	Update
MCD	Data Science	Master	4	50	Update
MCIF	Cybersecurity and Computer Forensics	Master	4	20	Update
MEI-CM	Computer Engineering - Mobile Computing	Master	4	40	Update
NEW1	New Course updated	TESP	4	20	Update
TESP-CRI	Cybersecurity and Computer Networks	TESP	4	72	Update
TESP-DWM	Web and Multimedia Development	TESP	4	52	Update
TESP-PSI	Information System Programming	TESP	4	71	Update
TESP-RSI	Computer Networks and Systems	TESP	4	10	Update
TESP-TI	IT Technologies	TESP	4	10	Update

- By clicking on the link “Create a new course” we jump to the create form, and when clicking on one of the “Update” links, we jump to the corresponding “course”

3. Although it works perfectly fine, creating link references with a static URL is not a good approach. If we decide to change all “course” related URLs we would have to change all static URL on several views and controllers. This reduces the project maintainability.

4. Instead of using static URL for hyperlinks and form actions, we will use Laravel functions to generate the URL dynamically from the routes. But first, and for better organization, we will start by giving all routes a name – **named routes** will be all that we need to know when generating the URLs.

5. To add names to all routes. Edit the route file (`routes/web.php`) to:

```php
<?php

use App\Http\Controllers\CourseController;
use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    return view('welcome');
})->name('home');

Route::get('courses', [CourseController::class, 'index'])->name('courses.index');
Route::get('courses/create', [CourseController::class, 'create'])->name('courses.create');
Route::post('courses', [CourseController::class, 'store'])->name('courses.store');
Route::get('courses/{course}/edit', [CourseController::class, 'edit'])->name('courses.edit');
Route::put('courses/{course}', [CourseController::class, 'update'])->name('courses.update');
```

6. We can check the list of routes by executing the following shell command:

```
php artisan route:list
```

- Previous command gives us the complete list of routes supported by our application, and each route includes the method, the URL pattern, the route name and the controller's method that handles the route.

```
GET|HEAD  / ..... home
POST      _ignition/execute-solution ..... ignition.executeSolution › Spatie\LaravelIgnition › ExecuteSolutionController
GET|HEAD  _ignition/health-check ..... ignition.healthCheck › Spatie\LaravelIgnition › HealthCheckController
POST      _ignition/update-config ..... ignition.updateConfig › Spatie\LaravelIgnition › UpdateConfigController
GET|HEAD  _tt/assets/base.js ..... telescope-toolbar.baseJs › Fruitcake\TelescopeToolbar › ToolbarController@baseJs
GET|HEAD  _tt/assets/styling.css ..... telescope-toolbar.styling › Fruitcake\TelescopeToolbar › ToolbarController@styling
GET|HEAD  _tt/render/{token} ..... telescope-toolbar.render › Fruitcake\TelescopeToolbar › ToolbarController@render
GET|HEAD  _tt/show/{token}/{tab?} ..... telescope-toolbar.show › Fruitcake\TelescopeToolbar › ToolbarController@show
GET|HEAD  courses ..... courses.index › CourseController@index
POST      courses ..... courses.store › CourseController@store
GET|HEAD  courses/create ..... courses.create › CourseController@create
PUT       courses/{course} ..... courses.update › CourseController@update
GET|HEAD  courses/{course}/edit ..... courses.edit › CourseController@edit
```

- The list includes several routes that are used internally by Laravel or other packages (example: telescope)

7. Change the view `courses.index` (file `resources/views/courses/index.blade.php`) so that we generate URLs from the route name – instead of using static URLs:

```
. . .
<p><a href="{{ route('courses.create') }}">Create a new course</a></p>
. . .
<a href="{{ route('courses.edit', ['course' => $course]) }}">Update</a>
. . .
```

- Laravel function `route(route_name)` generates an URL (complete URL including the domain – example: "`http://yourdomain/courses`") from the route name.
- We can pass route parameters to the route function using the 2nd argument of the route function and passing all parameters on an array. Examples:

```
route('course.edit', ['course' => $course])

route('another.route', ['course' => $course, 'student' => $s, 'p3' => 25])
    // route with name 'another.route' has 3 parameters:
    // course, student and p3
```

8. Open the page `http://yourdomain/courses`. The behavior of the page is the same as before. However, analyze the HTML of the page:
   - On the browser, right click on the page and choose the option: "view source code" (or similar)
   - Find the code for the hyperlinks (elements <a>) – all link references use an absolute URL – example: `<a href="http://yourdomain/courses/EI/edit">`

9. Change the `CourseController` (file `app/Http/Controllers/CourseController.php`) so that all redirects use name routes (instead of static URL).

```
<?php
. . .
class CourseController extends Controller
{
    public function store(Request $request): RedirectResponse
    {
        . . .
        return redirect()->route('courses.index');
    }

    public function update(Request $request, Course $course): RedirectResponse
    {
        . . .
        return redirect()->route('courses.index');
    }
}
```

10. To complete our URL refactoring, we have to change the action of the form to create and form to update, so that it also uses route names instead of fixed URLs.
11. Change the view `courses.create` (`resources/views/courses/create.blade.php`) so that the redirect location uses named routes (instead of static URL).

```
<form method="POST" action="{{ route('courses.store') }}">
```

12. Change the view `courses.edit` (file `resources/views/courses/edit.blade.php`) so that the redirect location uses named routes (instead of static URL).

```
<form method="POST" action="{{ route('courses.update', ['course' => $course]) }}">
```

13. Open the page `http://yourdomain/courses.` , and from that page create and update courses. The behavior of all pages should be the same as before

7. Subviews

The form to create and the form to update are very similar – the HTML of all fields is almost the same. The only difference is that the update form fields have a value (the initial value when the

form is opened) and the create form has no initial value (but we could easily fill these initial values with null or an empty string).

To avoid code repetition (2 views with the same code) we will create a subview with the fields design and then use this subview in both views. For further information about including subviews, check the documentation: https://laravel.com/docs/blade#including-subviews

1. In the "`resource/views/courses`" folder create a new folder named "`shared`" (new folder complete name: `resource/views/courses/shared`). In this new folder, add the file "`fields.blade.php`". The content of this view (that will be used as a subview) is an exact copy of the fields section of the view "`courses.edit`":

```
<div>
    <label for="inputAbbreviation">Abbreviation</label>
    <input type="text" name="abbreviation" id="inputAbbreviation"
           value="{{$course->abbreviation}}">
</div>
<div>
    <label for="inputName">Name</label>
    <input type="text" name="name" id="inputName" value="{{$course->name}}">
</div>
<div>
    <label for="inputName_pt">Name (PT)</label>
    <input type="text" name="name_pt" id="inputName_pt" value="{{$course->name_pt}}">
</div>
<div>
    <label for="inputType">Type of course</label>
    <select name="type" id="inputType">
        <option {{$course->type == 'Degree' ? 'selected' : ''}}>Degree</option>
        <option {{$course->type == 'Master' ? 'selected' : ''}}>Master</option>
        <option {{$course->type == 'TESP' ? 'selected' : ''}}>TESP</option>
    </select>
</div>
<div>
    <label for="inputSemesters">Semesters</label>
    <input type="text" name="semesters" id="inputSemesters" value="{{$course->semesters}}">
</div>
<div>
    <label for="inputECTS">ECTS</label>
    <input type="text" name="ECTS" id="inputECTS" value="{{$course->ECTS}}">
</div>
<div>
    <label for="inputPlaces">Places</label>
    <input type="text" name="places" id="inputPlaces" value="{{$course->places}}">
</div>
```

```
<div>
    <label for="inputContact">Contact</label>
    <input type="text" name="contact" id="inputContact" value="{{$course->contact}}">
</div>
<div>
    <label for="inputObjectives">Objectives</label>
    <textarea name="objectives" id="inputObjectives" rows=10>
         {{$course->objectives}}
    </textarea>
</div>
<div>
    <label for="inputObjectives_pt">Objectives (PT)</label>
    <textarea name="objectives_pt" id="inputObjectives_pt" rows=10>
         {{$course->objectives_pt}}
   </textarea>
</div>
```

- The view we have created, named "`courses.shared.fields`" will be used as a **subview**.
- A subview is responsible only for a part of the HTML page, therefore, does not include the full HTML (`<html>`, `<body>`, etc.) – it only includes an HTML section.

2. Change the view `courses.edit` (`resource/views/courses/edit.blade.php`), so that it includes the subview `courses.shared.fields` instead of defining the full HTML for all fields:

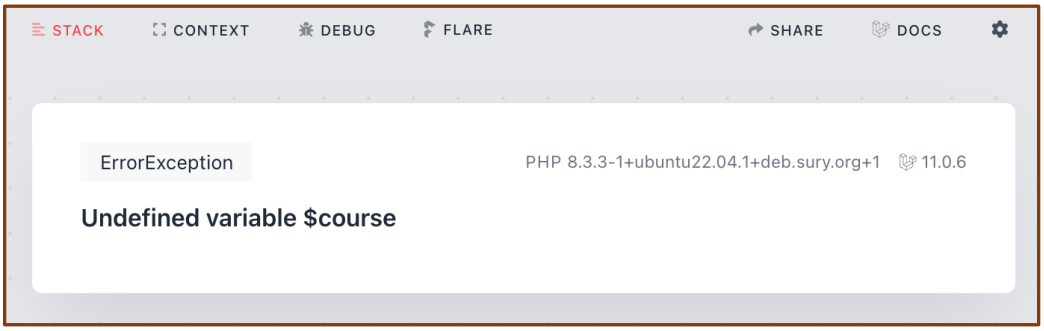
```
. . .
<form method="POST" action="{{ route('courses.update', ['course' => $course]) }}">
    @csrf
    @method('PUT')
    @include('courses.shared.fields')
    <div>
        <button type="submit" name="ok">Save course</button>
    </div>
</form>
. . .
```

3. Change the view `courses.create` (`resource/views/courses/create.blade.php`), so that it includes the subview `courses.shared.fields` instead of defining the full HTML for all fields:

```
. . .
<form method="POST" action="{{ route('courses.store') }}">
    @csrf
    @include('courses.shared.fields')
    <div>
        <button type="submit" name="ok">Save new course</button>
    </div>
</form>
. . .
```

- Compare the view to edit (`courses.edit`) and create (`courses.create`) and analyze the differences.

4. If we try to edit and to create a course, we can verify that the edit is working perfectly but when we open the form to create, we get the following error:



5. Previous error happens because the newly created subview ("`courses.shared.fields`") uses the variable `$course` to fill the fields values. The "`create`" method of the controller does not pass this variable to the view, so this error occurs. To solve this problem, we just pass that variable (`$course`) with an empty instance of the `Course` model. Open the `CourseController` (file `app/Http/Controllers/CourseController.php`) and change the method "`create`" to:

```
public function create(): View
{
    $newCourse = new Course();
    return view('courses.create')->withCourse($newCourse);
}
```

- The method `->withCourse($newCourse)` passes on to the view a variable named `$course` with the value `$newCourse`. The name of the method defines the name of the variable in the view.
- Name of method = `withNameOfVariable` will create a variable named `nameOfVariable` on the view.
- The argument of the method defines the value of the variable.
- The method `->withCourse($newCourse)` is the equivalent to the method: `->with(course', $newCourse)`

```
return view('courses.create')->withCourse($newCourse);
  // EQUIVALENT TO:
return view('courses.create')->with('course', $newCourse);
```

6. Try the create operation again. It should work correctly by now.

8. CRUD – Part 2 (delete and show)

Up until now, our CRUD for “courses” has 5 routes – one for listing all courses, 2 for handling the create operation and 2 for handling the update operation. For an entity to support all CRUD operations (resource full), it should support 2 more routes, one to delete the entity and one to show the detail of one entity.

1. Let us now implement the delete operation. Start by adding the route to delete one “course”, on the file “`routes/web.php`”.

```
Route::delete('courses/{course}', [CourseController::class, 'destroy'])->name('courses.destroy');
```

2. Next, implement the method “`destroy`” of the `CourseController` (file: `app/Http/Controllers/CourseController.php`)

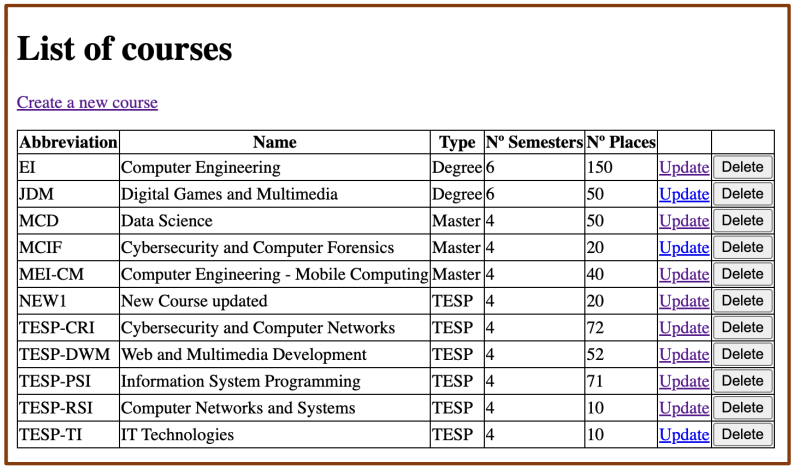
```php
public function destroy(Course $course): RedirectResponse
{
    $course->delete();
    return redirect()->route('courses.index');
}
```

3. Delete operations do not need UI (unless there is some kind of confirmation before delete), so there is no route to show a form to delete. However, to invoke the delete operation, we need to create a form so that we can call the HTTP request with the method DELETE – it is not possible to do this using hyperlinks. Edit the view "`courses.index`" (`resource/views/courses/index.blade.php`) and add a form to delete each "course" on the list:

```php
<!DOCTYPE html>
   . . .
    <table>
        <thead>
            <tr>
                . . .
                <th></th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            @foreach ($courses as $course)
                <tr>
                    . . .
                    <td>
                        <a href="{{ route('courses.edit',  ['course' => $course]) }}">Update</a>
                    </td>
                    <td>
                      <form method="POST"
                            action="{{ route('courses.destroy', ['course' => $course]) }}">
                            @csrf
                            @method('DELETE')
                            <button type="submit" name="delete">Delete</button>
                        </form>
                    </td>
                </tr>
            @endforeach
        </tbody>
    </table>
</body>
</html>
```

- To invoke an HTTP request with the method DELETE, we have to add a form with the blade directive `@method('DELETE')`. Using only a hyperlink (element `<a>`) it's impossible to send a HTTP request with a method different from the GET method.

4. Open the list of courses (`http://yourdomain/courses`) and try to delete some of the courses.

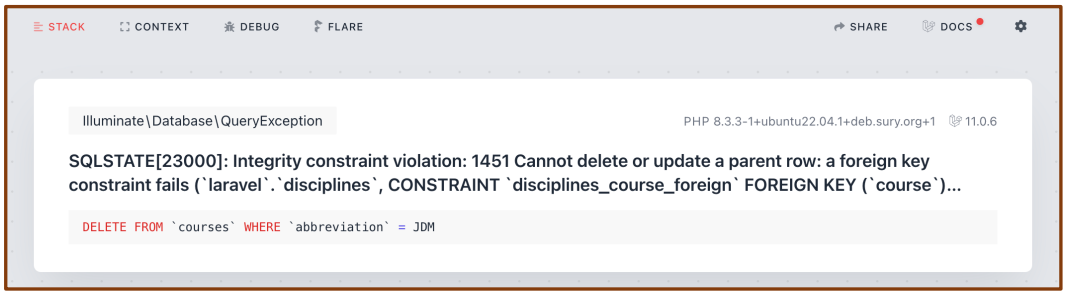


List of courses

Create a new course

Abbreviation	Name	Type	Nº Semesters	Nº Places		
EI	Computer Engineering	Degree	6	150	Update	Delete
JDM	Digital Games and Multimedia	Degree	6	50	Update	Delete
MCD	Data Science	Master	4	50	Update	Delete
MCIF	Cybersecurity and Computer Forensics	Master	4	20	Update	Delete
MEI-CM	Computer Engineering - Mobile Computing	Master	4	40	Update	Delete
NEW1	New Course updated	TESP	4	20	Update	Delete
TESP-CRI	Cybersecurity and Computer Networks	TESP	4	72	Update	Delete
TESP-DWM	Web and Multimedia Development	TESP	4	52	Update	Delete
TESP-PSI	Information System Programming	TESP	4	71	Update	Delete
TESP-RSI	Computer Networks and Systems	TESP	4	10	Update	Delete
TESP-TI	IT Technologies	TESP	4	10	Update	Delete

- You should only be capable of deleting newly created courses because the original courses included in the database, have related data ("disciplines", "students", etc.). When deleting these original courses we get the error: "`SQLSTATE[23000]: Integrity constraint violation: 1451 Cannot delete or update a parent row: `"



- Later, we will use CSS styles to change the visual aspect of the button "Delete" so that it looks like a hyperlink.

5. The latest CRUD operation to implement is the one responsible from showing the details of one course. As referred previously, the form to update one course could also be used to show the detail of one course.

   If we chose to implement an independent page just to show the detail, then it should be a read-only version of the entity – in this case a read-only version of the details of 1 course.

   We will implement a page just to show the detail of one course, but we will minimize code by reusing the code of the "`courses.share.fields`" subview.

6. Start by adding the route to show one course, on the file "`routes/web.php`".

```
Route::get('courses/{course}', [CourseController::class, 'show'])->name('courses.show');
```

7. Next, implement the method `show` of the `CourseController` (file: `app/Http/Controllers/CourseController.php`)

```
    public function show(Course $course): View
    {
        return view('courses.show')->with('course', $course);
    }
```

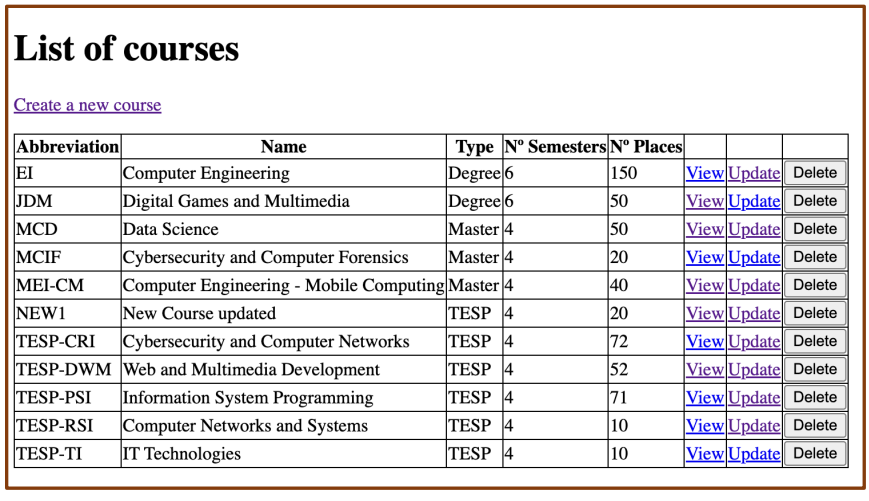
8. After the route and controller, we'll implement the view "`courses.show`". On the folder "`resources/views/courses`" create the file "`show.blade.php`" with the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Course</title>
</head>
<body>
    <h2>Course "{{ $course->name }}"</h2>
    <div>
        @include('courses.shared.fields')
    </div>
</body>
</html>
```

9. Add a hyperlink to the "show" page in the `courses.index` view (file `resources/views/courses/index.blade.php`):

```
<table>
    <thead>
        <tr>
            . . .
            <th>Nº Places</th>
            <th></th>
            <th></th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach ($courses as $course)
            <tr>
                . . .
                <td>{{ $course->places }}</td>
                <td>
                    <a href="{{ route('courses.show', ['course' => $course]) }}">View</a>
                </td>
                <td>
                    <a href="{{ route('courses.edit', ['course' => $course]) }}">Update</a>
                </td>
                . . .
            </tr>
        @endforeach
    </tbody>
</table>
```

10. Open the courses page (`http://yourdomain/courses`):

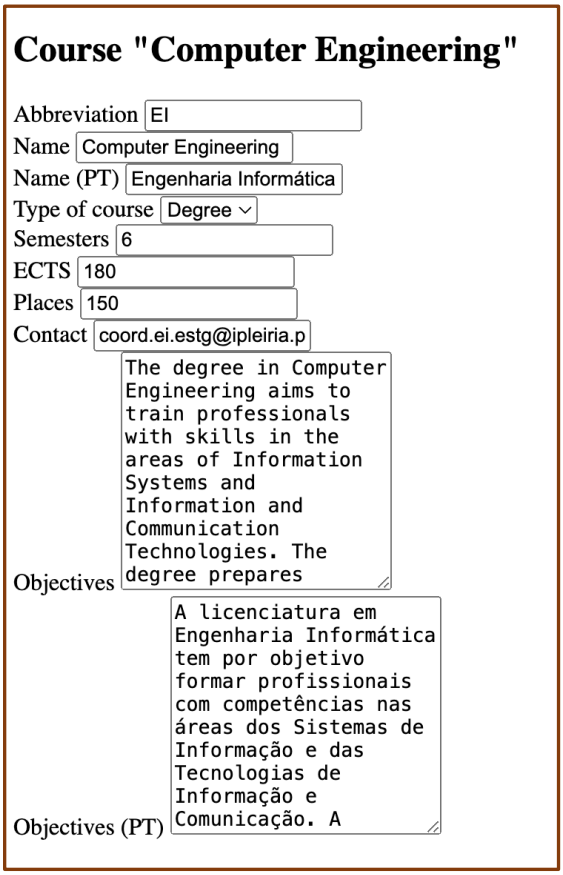


List of courses

Create a new course

Abbreviation	Name	Type	Nº Semesters	Nº Places			
EI	Computer Engineering	Degree	6	150	View	Update	Delete
JDM	Digital Games and Multimedia	Degree	6	50	View	Update	Delete
MCD	Data Science	Master	4	50	View	Update	Delete
MCIF	Cybersecurity and Computer Forensics	Master	4	20	View	Update	Delete
MEI-CM	Computer Engineering - Mobile Computing	Master	4	40	View	Update	Delete
NEW1	New Course updated	TESP	4	20	View	Update	Delete
TESP-CRI	Cybersecurity and Computer Networks	TESP	4	72	View	Update	Delete
TESP-DWM	Web and Multimedia Development	TESP	4	52	View	Update	Delete
TESP-PSI	Information System Programming	TESP	4	71	View	Update	Delete
TESP-RSI	Computer Networks and Systems	TESP	4	10	View	Update	Delete
TESP-TI	IT Technologies	TESP	4	10	View	Update	Delete

11. Click on “View” to open the details of one course.



12. Although previous page (show one course) is read-only, in the sense that it does not include a form, nor a submit button, so even if the end-user changes some field data, he could not save it in the database. However, it does not make sense to show the data using field inputs.

    We could replace the inputs with regular HTML elements (`<div>`, `<p>`, etc…), but we want to reuse the subview `"courses.shared.fields"` to optimize the developer's productivity.

    We'll add a new variable (called `$readonlyData`) and send it to the subview `"courses.shared.fields"`, so that when the variable value is true, the data will be represented as read-only data, and when the variable value is false, the data will be represented as writable data (with "normal" input fields). Change the view "`courses.show`" (file `resources/views/courses/show.blade.php`) code to:

```
<div>
    @include('courses.shared.fields', ['readonlyData' => true])
</div>
```

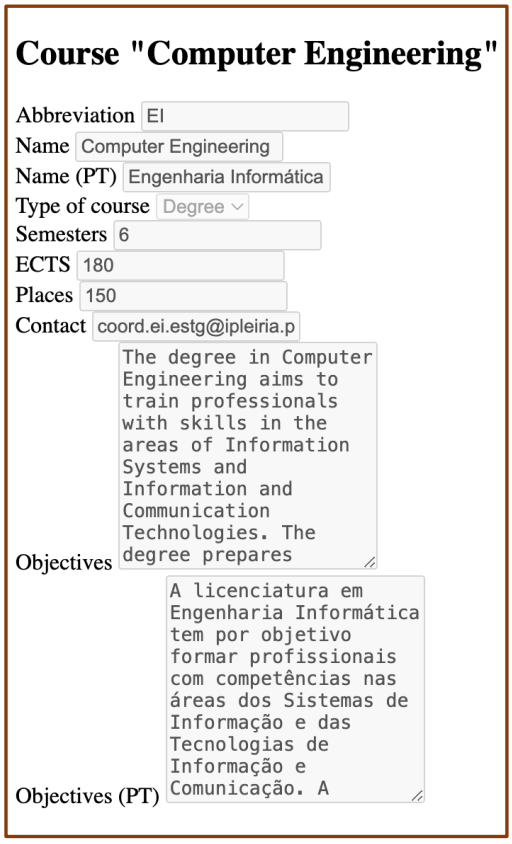
13. Finally, we will change the subview "`courses.shared.fields`" (file `resources/views/courses/shared/fields.blade.php`) so that if the `$readonlyData` variable exists and is true, all field inputs will be read-only (they will have the attribute

`disabled`), else they will all be “normal” fields (no attribute `disabled`). Add the following code to the `fields.blade.php` file:

```
@php
    $disabledStr = $readonlyData ?? false ? 'disabled' : '';
@endphp
. . .
<input type="text" name="abbreviation" id="inputAbbreviation" {{ $disabledStr }}
value="…">
. . .
<input type="text" name="name" id="inputName" {{ $disabledStr }} value="…">
. . .
<input type="text" name="name_pt" id="inputName_pt" {{ $disabledStr }} value="…">
. . .
<select name="type" id="inputType" {{ $disabledStr }}>
. . .
<input type="text" name="semesters" id="inputSemesters" {{ $disabledStr }} value="…">
. . .
<input type="text" name="ECTS" id="inputECTS" {{ $disabledStr }} value="…">
. . .
<input type="text" name="places" id="inputPlaces" {{ $disabledStr }} value="…">
. . .
<input type="text" name="contact" id="inputContact" {{ $disabledStr }} value="…">
. . .
<textarea name="objectives" id="inputObjectives" {{ $disabledStr }} rows=10>…
. . .
<textarea name="objectives_pt" id="inputObjectives_pt" {{ $disabledStr }} rows=10>…
. . .
```

- `@php` - `@endphp` block allows blade template engine to include “*normal*” PHP code. In this case, we are using this block just to guarantee that if the subview does not receive the variable `$readonlyData` it still works correctly.

14. Open one course detail page (example: `http://yourdomain/courses/EI`). Try to focus and edit any field – it should not be possible, as all fields are disabled.



- We can use CSS styles to modify the visual aspect of disabled fields (which we would do later), so that they do not appear to be inputs.

9. Resource controller

When implementing CRUD operations for an entity, we should follow the same pattern as we have used for courses, both in functionalities and naming conventions. Laravel has a concept, called resources, that applies this pattern to models (entities). We can think of each Eloquent model in our application as a "resource", with typical CRUD actions against each resource that follows the same pattern.

With Laravel we can create a resource controller, which is a controller that has the 7 methods required to implement all CRUD operations and have the same name as the ones we've created for courses entity.

1. Execute the following shell command:

```
php artisan make:controller DisciplineController --resource
```

2. This creates a controller (named `DisciplineController`) with all the 7 methods necessary for the CRUD operations. Analyze the controller created by previous command.

   - A resource controller implies a set of routes, actions (controller's methods) and route names that follow a specific convention, as we can observe on the next table (available on Laravel documentation - https://laravel.com/docs/controllers#resource-controllers) with an example for a "photo" resource.

Actions Handled by Resource Controllers

Verb	URI	Action	Route Name
GET	`/photos`	index	photos.index
GET	`/photos/create`	create	photos.create
POST	`/photos`	store	photos.store
GET	`/photos/{photo}`	show	photos.show
GET	`/photos/{photo}/edit`	edit	photos.edit
PUT/PATCH	`/photos/{photo}`	update	photos.update
DELETE	`/photos/{photo}`	destroy	photos.destroy

Our controller (`CourseController`) follows the same pattern as the resource controller, which means that it can be considered a resource controller. For these types of controllers, we can define all 7 routes at once, using the method `Route::resource(…)`. Open the "`routes/web.php`" file and replace all 7 routes for the course entity with a single line `Route::resource(…)`:

```
// REPLACE THESE 7 ROUTES:
// Route::get('courses', [CourseController::class, 'index'])->name('courses.index');
// Route::get('courses/create', [CourseController::class, 'create'])->name('courses.create');
// Route::post('courses', [CourseController::class, 'store'])->name('courses.store');
// Route::get('courses/{course}/edit', [CourseController::class, 'edit'])->name('courses.edit');
// Route::put('courses/{course}', [CourseController::class, 'update'])->name('courses.update');
// Route::delete('courses/{course}', [CourseController::class, 'destroy'])->name('courses.destroy');
// Route::get('courses/{course}', [CourseController::class, 'show'])->name('courses.show');

// WITH A SINGLE LINE OF CODE:
Route::resource('courses', CourseController::class);
```

   - The line `Route::resource('courses', CourseController::class);` creates 7 routes and associate them with 7 methods of the controller "`CourseController`". It is possible to define resources with a sub-set of actions (instead of 7 routes it can create

only a subpart of these), create nested resources or singleton resources, add additional routes to the resource (have more than the typical 7 routes), etc… Check the documentation https://laravel.com/docs/controllers#resource-controllers for further details.

3. Try the application again. It should work as before. This happens because we have followed all patterns and naming conventions, otherwise, we would have some broken features.

4. Also, confirm that `Routes::resource(…)` creates 7 routes with the command:

```
php artisan route:list
```

```
GET|HEAD        courses ..................................... courses.index › CourseController@index
POST            courses ..................................... courses.store › CourseController@store
GET|HEAD        courses/create ............................. courses.create › CourseController@create
GET|HEAD        courses/{course} ............................ courses.show › CourseController@show
PUT|PATCH       courses/{course} ........................... courses.update › CourseController@update
DELETE          courses/{course} ......................... courses.destroy › CourseController@destroy
GET|HEAD        courses/{course}/edit ....................... courses.edit › CourseController@edit
```

10. Disciplines *(autonomous work)*

1. Using the same pattern as the one used for courses entity, create all that is necessary for the application to support all CRUD operations for the entity disciplines – database table "disciplines".

Recommendations:

- Start by executing the shell command:

```
php artisan make:model Discipline --resource
```

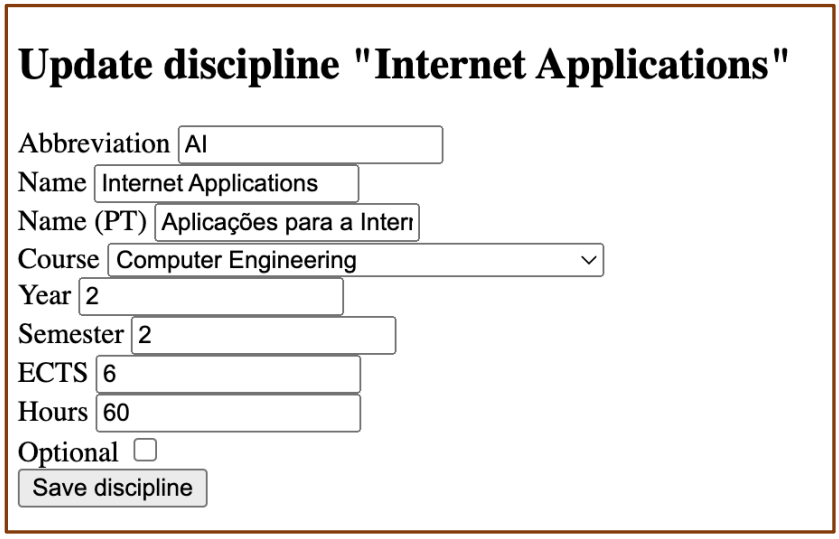
  - This will create the Discipline model and a resource controller for that model – if you have already created a resource controller for the Discipline (on step 9.1) it will only create the model.

- Verify if all eloquent conventions are correct for the disciplines table. If that's not the case, configure the model accordingly.
- Use the same pattern for the signature of the controllers' methods.
- Use "Route-Model Binding" when working with route parameters (the same as used for the courses route parameters)

Challenge:

- For selecting the list of courses of one discipline (create or update a discipline), create a `<select>` element with a dynamic (from the database) list of options.

2. Compare your implementation with the provided solution.



Summary

Summary of features, implementations, technologies, and concepts applied during the worksheet:

```
Create a Laravel Project

Database migration & seeding

Routes

      Simple URL signatures

      Http Methods

      Binding routes to controller actions (methods)

      Route-model binding

Routes and resources

      Named routes

      Route Parameters

Models

      Mapping to table and columns
```

- Eloquent conventions
- Fillable properties
- Model collections
- Model properties
- Static methods: all and create
- Methods: update and delete

Controllers

- Returning views
- Returning redirects
- Passing data to views
- Resource controllers (pattern for CRUD operations)
- Route parameters binding
- Request object
- $request->all()

Views

- View names and file structure
- Blade engine
- {{ expression }}
- @foreach
- forms
- @CSRF
- @method (method spoofing)
- @php
- URL generation (for actions and hyperlinks)
- URL generation with parameters
- Subviews
- Passing extra data to subviews

Resources and CRUD operations

- Resource naming conventions

- Resource controllers
- Route::resource
- Resource operations patterns
  - index
  - show
  - create
  - store
  - edit
  - update
  - destroy

Laravel Telescope (debugging)

- Installation and configuration
- Laravel Telescope UI usage
- Requests and Responses
- Database queries

Laravel Telescope Toolbar

- debug()