DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

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

APLICAÇÕES PARA A INTERNET

Engenharia Informática

Marco Monteiro

8 - Laravel - 5

Objectives:

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

(2) Authentication

(3) Authorization with gates and policies

(4) Send e-mail

Note the following:

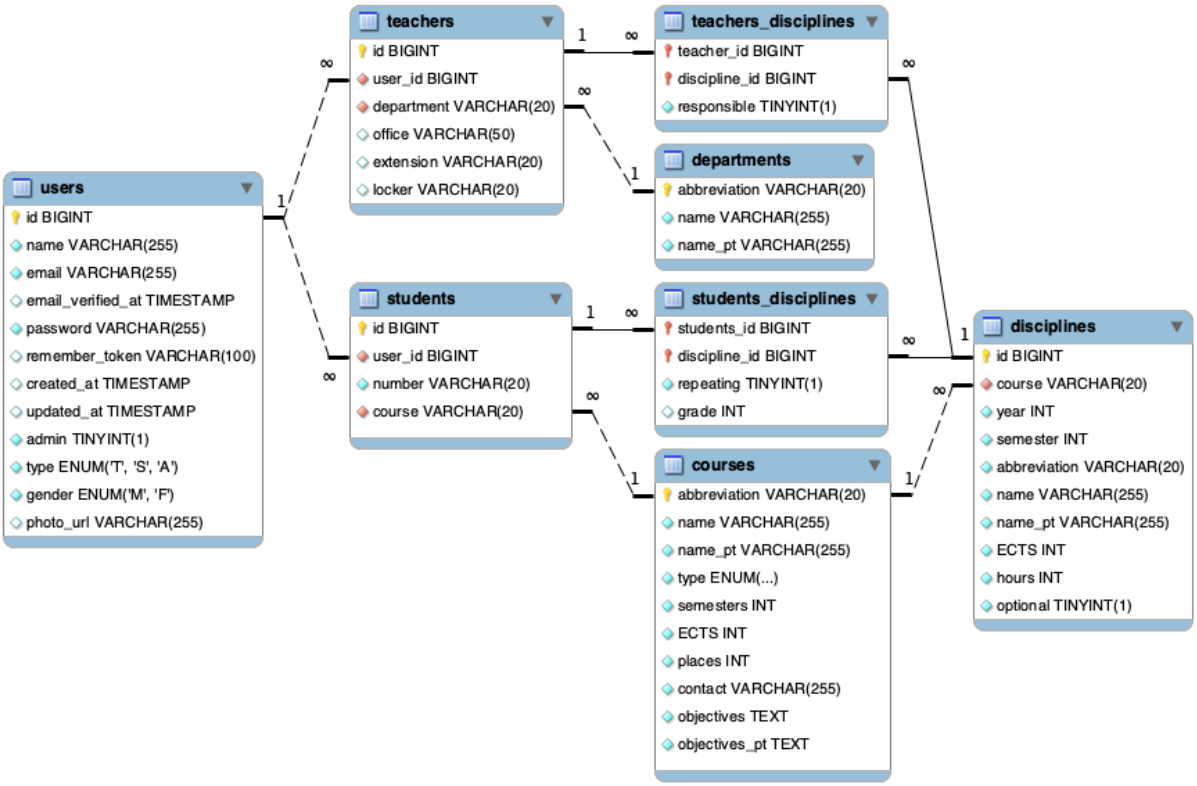
- Before starting the exercise, read the related content on Moodle;
- During the resolution of the exercises, consult the Laravel documentation (https://laravel.com) and other online resources.

Scenery

This worksheet will continue the development of the Web Application from the last worksheet where we've used the server's storage and implemented upload of files, and used sessions to create a shopping cart . In this worksheet we will delve into the authentication related features and implement authorization with Laravel middleware, gates and policies. We will also create a Mailtrap.io account (to send “fake” email) and configure Laravel to use an email server.

Database

This worksheet will use the same database as the one used on the last worksheet. The structure of the database is the following:



1. Preparation

To run the exercises of this worksheet 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.

To create the project for the current worksheet we have 3 options:

1. Copy the provided project and configure it as a new project, using Laragon.
2. Copy the provided project and configure it as a new project, using Laravel Sail.
3. Merge the provided project into the project that was implemented on previous worksheet (fastest option). Works with Laragon or Laravel Sail.

Consult the tutorial "**tutorial.laravel.01-laravel-install-configuration**", available on Moodle, to check for details on installation and configuration of Laravel projects.

1.1. New Laravel Project – with Laragon

1. Copy the provided zip file (`start.ai-laravel-5.zip`) into the Laragon root folder and decompress it on that folder.

2. Previous command will create the folder `ai-laravel-5`, that will be the worksheet project folder, inside the Laragon root folder. The worksheet project folder should be available as `C:\<laragon_www_root>\ai-laravel-5`. For example, `C:\laragon\www\ai-laravel-5` or `D:\ainet\ai-laravel-5` (it depends on the Laragon root folder)
3. Use previous database (from the first Laravel worksheet) or create a new database. Configure .env file accordingly. Typical database configuration for Laragon (with the database name "Laravel")

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

4. Run Laragon and start all services (in ESTG computers, before starting the services, it might be necessary to stop `vmware` services).
5. Open Laragon terminal and execute the following command on the project folder (`ai-laravel-5`), to rebuild the "`vendor`" folder:

```
composer update
```

6. To define the database structure and fill (seed) the data on the database, execute:

```
php artisan migrate:fresh
```

```
php artisan db:seed
```

7. Create a symbolic link for the public storage folder.

```
php artisan storage:link
```

8. Use the "`http://ai-laravel-5.test`" URL to access the content.
9. Test CRUD operations for courses (`http://ai-laravel-5.test/courses`) and disciplines (`http://ai-laravel-5.test/disciplines`)

1.2. New Laravel Project – with Laravel Sail

10. Copy the provided zip file (`start.ai-laravel-5.zip`) into any folder and decompress it.

11. Previous command will create the folder `ai-laravel-5`, that will be the current worksheet project folder.
12. Execute the following command on the project folder (`ai-laravel-5`), to rebuild the "`vendor`" folder – this will also install the required package Laravel Sail

```
composer update
```

- To execute previous command, it is necessary that the composer tool is installed on your local machine. Check https://getcomposer.org to install composer if necessary.
- If for some reason it is not possible to install composer on your machine, copy the provided zip file (`start.ai-laravel-5.all-folders.zip`) that already includes the vendor folder.

13. Ensure that Docker Desktop (or other similar application) is running.
14. On the `ai-laravel-5` folder execute the following command:

```
./vendor/bin/sail up -d
```

- If the sail alias is already configured, it is possible to execute the alternative command:

```
sail up -d
```

15. To define the database structure and fill (seed) the data on the database, execute:

```
sail php artisan migrate:fresh
```

```
sail php artisan db:seed
```

16. Create a symbolic link for the public storage folder.

```
sail php artisan storage:link
```

17. Use the "`http://localhost`" URL to access the content, and "`http://localhost:8080`" to access the `adminer` tool (for database administration)
18. Test CRUD operations for courses (`http://localhost/courses`) and disciplines (`http://localhost/disciplines`)

1.3. Merge Projects – with Laragon or Laravel Sail

19. Copy the provided zip file (`start.ai-laravel-5.zip`) into any folder and decompress it.
20. Previous command will create the folder `ai-laravel-5`, with the base project for the current worksheet. However, instead of using this new folder, we will continue to use the last worksheet project folder. With this approach we will reuse the folder "vendor" and "storage", as well as the database.
21. On the last worksheet project folder (that we want to continue using), **remove** the following folders:
    - `app`
    - `resources`
    - `routes`
22. Copy the 3 folders (`app`, `resources` and `routes`) from the provided folder (`ai-laravel-5`) to the last worksheet project folder (that we want to continue using).
23. If you are using Laragon, run Laragon and start all services (in ESTG computers, before starting the services, it might be necessary to stop `vmware` services).
    - Use the same URL as the last worksheet (probably `http://ai-laravel-1.test`, "`http://ai-laravel-2.test`", or similar) to access the content.
    - If courses images are not available on the `courses/showcase` page, execute the following command:

```
php artisan storage:link
```

    - Test CRUD operations for courses (`http://yourdomain/courses`) and disciplines (`http://yourdomain/disciplines`)
24. If you are using Laravel Sail, execute the following command on the root of the last worksheet project:

```
./vendor/bin/sail up -d
```

    - If the sail alias is already configured, it is possible to execute the alternative command:

```
sail up -d
```

- Use the same URL as the last worksheet (probably "`http://localhost`") to access the content.
- If courses images are not available on the `courses/showcase` page, execute the following command:

```
sail php artisan storage:link
```

- Test CRUD operations for courses (`http://localhost/courses`) and disciplines (`http://localhost/disciplines`)
- Test "`adminer`" tool for database administration: (`http://localhost:8080`)

2. Authentication Starter Kit

Laravel ecosystem includes several starter kits that automatically scaffold your application with the routes, controllers, and views you need to register and authenticate your application's users. In this section, we're going to use Laravel Breeze to implement authentication on our application.

Check https://laravel.com/docs/authentication and https://laravel.com/docs/starter-kits for more information about Laravel authentication and starter kits.

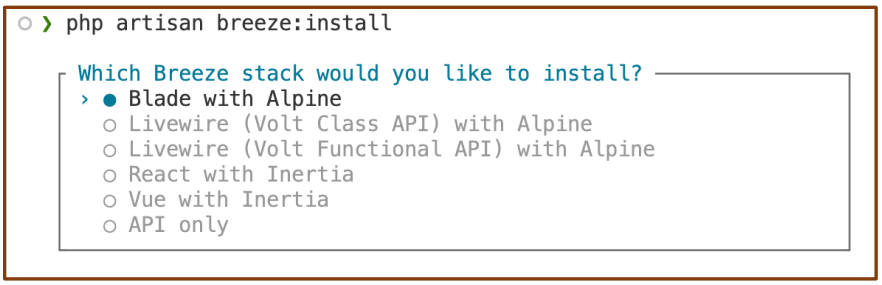
25. Let's start by installing **Laravel Breeze starter kit** on our project by executing the command:

```
composer require laravel/breeze --dev
```

26. Depending of the version of Laravel Breeze, there might be a small bug when generating the resources, that overwrites our current routes. To ensure we can handle that problem, let's copy the file "`routes/web.php`" as "`routes/web.bak.php`".
27. To generate the authentication scaffold resources, execute the command:

```
php artisan breeze:install
```

28. Choose the frontend stack "**Blade with Alpine**"



29. Then choose to support "dark mode" and the default testing framework.

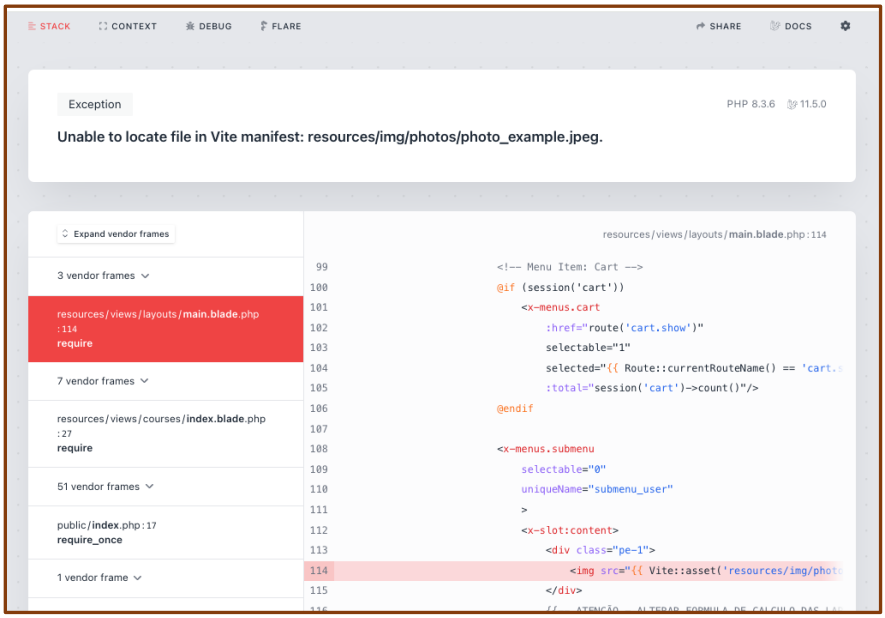
```
❯ php artisan breeze:install

  Which Breeze stack would you like to install?
  Blade with Alpine

  Would you like dark mode support?
  Yes

  Which testing framework do you prefer?
    ○ Pest
  › ● PHPUnit
```

30. Try to open the page `http://yourdomain/courses`. If the application returns a "404 not found" response, it might be due to the bug referred on step 26. If the page opens correctly, ignore the current step. If the page does not open correctly, edit the file "`routes/web.php`" and ensure that all pre-existing routes (on the file "`routes/web.bak.php`") are added to the current route file ("`routes/web.php`"), after the line of code:
    `require __DIR__.'/auth.php';`

```php
<?php

use App\Http\Controllers\ProfileController;
use App\Http\Controllers\AdministrativeController;
use App\Http\Controllers\CourseController;
use App\Http\Controllers\DisciplineController;
use App\Http\Controllers\DepartmentController;
use App\Http\Controllers\TeacherController;
use App\Http\Controllers\StudentController;
use App\Http\Controllers\CartController;
use Illuminate\Support\Facades\Route;

/* GENERATED BY LARAVEL BREEZE ------------- */
Route::get('/', function () {
    return view('welcome');
});

Route::get('/dashboard', function () {
    return view('dashboard');
})->middleware(['auth', 'verified'])->name('dashboard');

Route::middleware('auth')->group(function () {
    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');
    Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');
    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');
});

require __DIR__.'/auth.php';
```

```
/* ORIGINAL ROUTES ------------------------ */

Route::view('/', 'home')->name('home');

Route::get('courses/showcase', [CourseController::class, 'showCase'])->name('courses.showcase');
Route::get('courses/{course}/curriculum', [CourseController::class, 'showCurriculum'])-
>name('courses.curriculum');
Route::delete('courses/{course}/image', [CourseController::class, 'destroyImage'])
    ->name('courses.image.destroy');
Route::resource('courses', CourseController::class);

Route::resource('departments', DepartmentController::class);

Route::resource('disciplines', DisciplineController::class);

Route::delete('teachers/{teacher}/photo', [TeacherController::class, 'destroyPhoto'])
    ->name('teachers.photo.destroy');
Route::resource('teachers', TeacherController::class);

Route::delete('students/{student}/photo', [StudentController::class, 'destroyPhoto'])
    ->name('students.photo.destroy');
Route::resource('students', StudentController::class);

Route::delete('administratives/{administrative}/photo', [AdministrativeController::class,
'destroyPhoto'])
    ->name('administratives.photo.destroy');
Route::resource('administratives', AdministrativeController::class);

// Add a discipline to the cart:
Route::post('cart/{discipline}', [CartController::class, 'addToCart'])
    ->name('cart.add');
// Remove a discipline from the cart:
Route::delete('cart/{discipline}', [CartController::class, 'removeFromCart'])
    ->name('cart.remove');
// Show the cart:
Route::get('cart', [CartController::class, 'show'])->name('cart.show');
// Confirm (store) the cart and save disciplines registration on the database:
Route::post('cart', [CartController::class, 'confirm'])->name('cart.confirm');
// Clear the cart:
Route::delete('cart', [CartController::class, 'destroy'])->name('cart.destroy');
```

31. Try to open the page `http://yourdomain/courses` again. If the application returns an error similar to "*Unable to locate file in Vite manifest: ...some_resource_img…*" (check the image), it is probably because the code generator removed our custom JavaScript code.



32. Edit the file "`resources/js/app.js`" so that it includes or custom JavaScript code:

```
import './bootstrap';

import Alpine from 'alpinejs';

/* Begin - our custom JavaScript code */
import './menu'

import.meta.glob([
  '../img/**',
]);

/* End - our custom JavaScript code */

window.Alpine = Alpine;

Alpine.start();
```

33. Execute the commands:

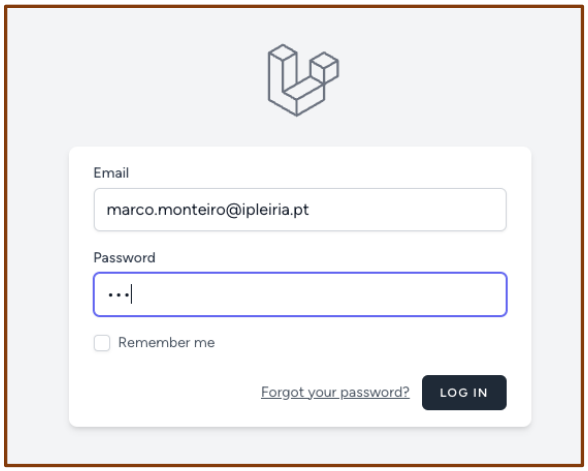
```
npm run build
npm run dev
```

34. Try to open the page `http://yourdomain/courses` again. The application should run as before. Navigate through the application's menus created previously (on previous worksheets) and confirm that everything is running correctly.

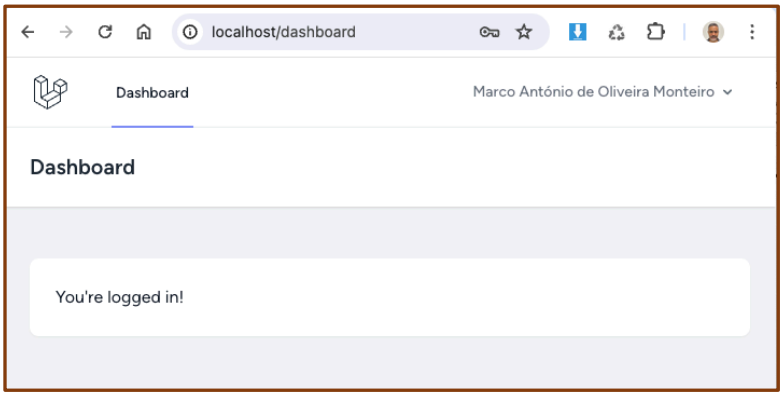
35. Open the page `http://yourdomain/`**`login`** and try to login with the following credentials:

    - **email** = marco.monteiro@ipleiria.pt
    - **password** = 123

    All users have the same password (123), so it is possible to test any credentials – just use the teacher/student/administrative email and the same password (123).



36. After a valid login, application redirects the user to the page `http://yourdomain/dashboard`:



37. Try several pages and features that were created by the Laravel Breeze: dashboard; profile; register; "Forget your password"; logout, etc.
38. Analyze the resources created by Laravel Breeze:

    - **controllers** – controllers on the folder `app/Http/Controllers/Auth`
    - **views** – authentication related views on the folder `resources/views/auth`
    - **profile and dashboard** – profile and dashboard related views on the folder `resources/views/profile`
    - **layouts** – "app" and "guest" layouts, and "navigation" partial view (used on the app layout) on the folder `resources/views/layouts`

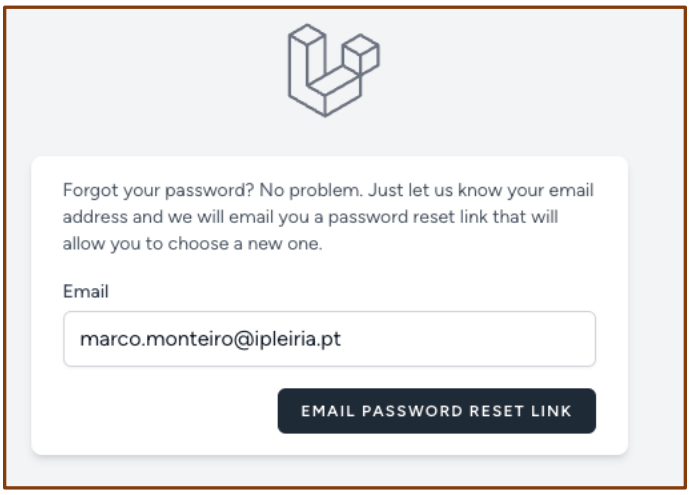
- **components** – several components used by the views and layouts on folder `resources/views/components`. These components are mixed with our custom components.

39. A partial resolution is available with the full project up until this exercise (file "`ai-laravel-5.partial.resolution.2.zip`").

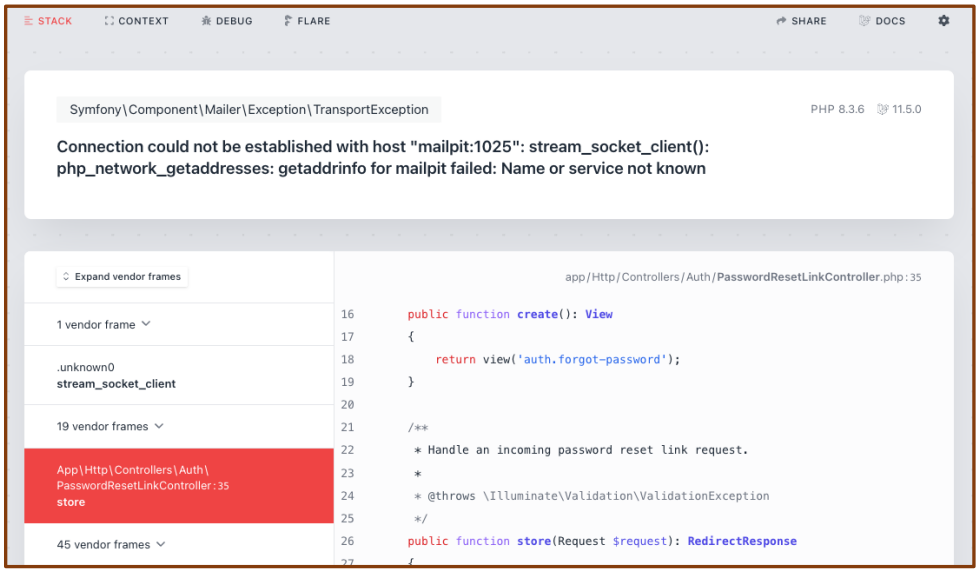
3. Laravel email sender

Laravel authentication includes features that requires sending emails. For instance, it includes the feature: "Forgot Your Password" that will send a reset password link to the email of the user. For these features to work, we have to configure an email account that will be used as for the application to send emails.

40. Logout and then open the page `http://yourdomain/login`. On the login page, click on "Forgot your password?" and then specify your email. Click on the "Email password reset link" to receive an email with a link for password resetting.

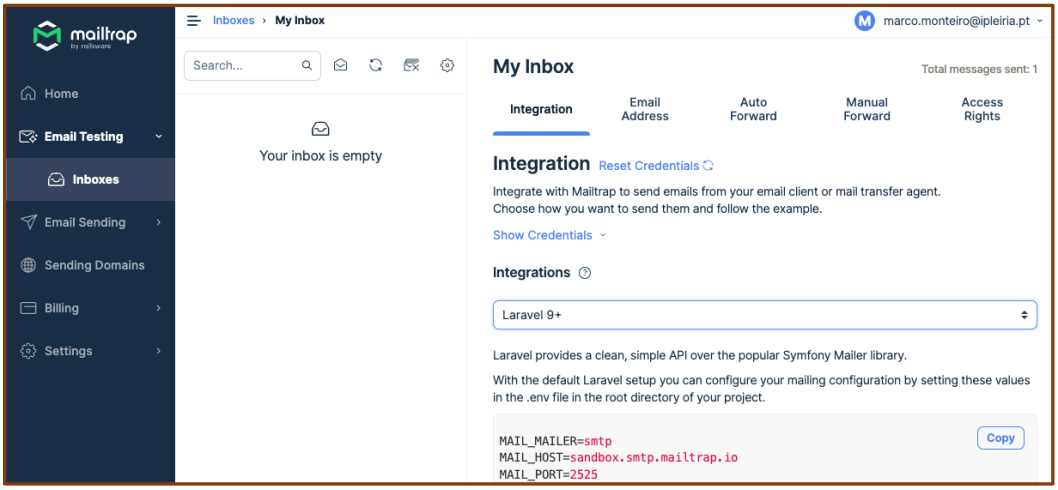


41. As the email sender is not configured yet, you'll get the following error (or similar, depending on the Laravel version):



42. For the feature “Forgot Your Password” (and any other feature that requires sending an email) work correctly, we must configure the email account that will be responsible for sending email messages on behalf of the application.
    During development **<u>we should not use a real email account</u>** – if we do this, then real email messages would be sent to the email addresses on the database. If the destination address does not exist, an error would occur. If the destination address exists, then we would be sending “fake/testing” email messages to that real destination address. Instead of a real email account, we are going to use a **<u>mailtrap.io</u>** account, that allow us to send “fake/test” messages to “fake/test” destinations – all email messages will be visible on the mailtrap.io.

    Note: there are other alternatives for sending email during development, but for consistency and simplification of the testing process, we'll use mailtrap.io.
43. Go to mailtrap.io (https://mailtrap.io/) and create a free account.
44. Go to mailtrap email “inbox” page and select "Laravel" integration:



45. Copy the configuration parameters from the “Laravel” integration – click on the “Copy” button
46. Past the parameters into the configuration .env file, so that these values replace the existing values (with the same parameter names). Also, you can change the MAIL_FROM_ADDRESS and MAIL_FROM_NAME parameters:

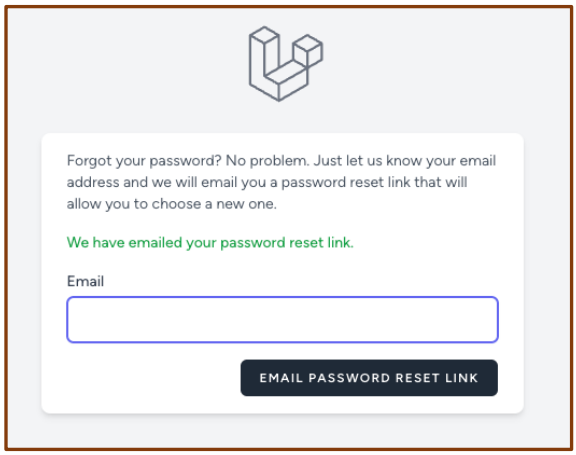
```
MAIL_MAILER=smtp
MAIL_HOST=sandbox.smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=your_user_name
MAIL_PASSWORD=your_password
MAIL_FROM_ADDRESS="noreply@example.com"
MAIL_FROM_NAME="${APP_NAME}"
```

47. That’s it. The application can now send emails.

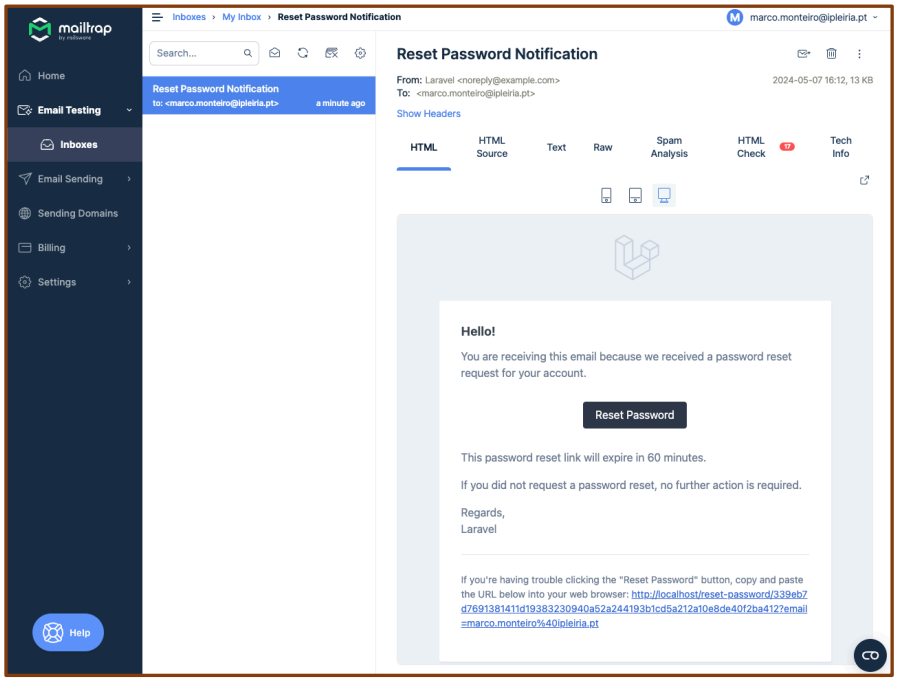
> Some networks (e.g. IPLeiria network) block the ports required to send emails. In these networks, sending an email with mailtrap.io (or any other external service) will not work.

48. Try again the "Forgot your password?" feature and specify an existing email (the email must exist on the database). Click on the "Email password reset link" to receive an email with a link for password resetting.

    The process to send an email can take a few seconds. After it is completed, the message "We have emailed your password reset link." Should appear.

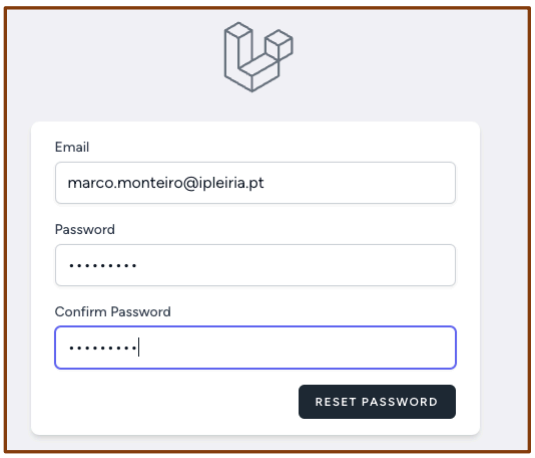


49. All messages sent by the application will be sent to the mailtrap inbox – we can view the messages as they would be seen by a real user, but no real message is sent by the application. Open the mailtrap inbox:



- The message on mailtrap inbox is the same message that the user would receive, but without sending a real message to the destination. This allows us to test all email messages sent by our application – all these messages will be shown on the mailtrap inbox.
- Note that “From” email field will have the MAIL_FROM_ADDRESS specified on the `.env` file, and “To” email field will have the address to where the message would be sent to.
- When the application is in production (published) we must configure a real email account that will be responsible for sending the emails.

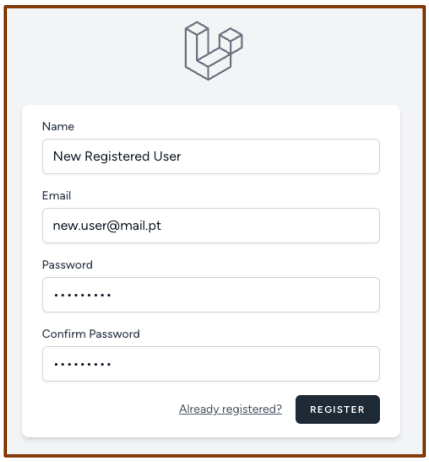
50. Click on the button "`Reset Password`" (or on the alternative link) in the email message (opened in the mailtrap.io inbox). This will open the “Reset Password” page on the application, where the user can change his password.



4. Registration and email verification

Laravel Breeze already includes a page for user registration – remove this feature if the application does not allow anonymous users to register independently. Also, if necessary, it is possible to adapt the register feature (controller, views, etc.) to your own application.

51. Open the register page: `http://yourdomain/register`, and create a new user. By default (for the worksheet database), this user will be a teacher.



52. Check the database ("users" table) to confirm that the user was created.
53. Try the login with the credentials for the new user. They should work as any other user previously created.
54. Another feature that Laravel supports is the **email verification**. This will send a verification email (to the user's email) after the user's registration is completed. Later we can restrict the access to the application only for users that have verified the email.

    Laravel provides convenient built-in services for sending and verifying email verification requests. Check https://laravel.com/docs/verification for more details about email verification.
55. First, we must prepare the User model by adding the interface `MustVerifyEmail`. Once this interface has been added to your model, newly registered users will automatically be sent an email (using the trait `Notifiable`) containing an email verification link. Edit the User model – file `app/Models/User.php`:

```php
<?php

namespace App\Models;

use Illuminate\Contracts\Auth\MustVerifyEmail;
. . .

class User extends Authenticatable implements MustVerifyEmail
{
    use HasFactory, Notifiable;
    . . .
```

56. To properly implement email verification, three routes will need to be defined.
    - First, a route will be needed to display a notice to the user that they should click the email verification link in the verification email that Laravel sent them after registration.
    - Second, a route will be needed to handle requests generated when the user clicks the email verification link in the email.
    - Third, a route will be needed to resend a verification link if the user accidentally loses the first verification link.
57. These 3 routes were created by Laravel Breeze. Open the file "`routes/auth.php`" and check that there are 3 routes related to the email verification:

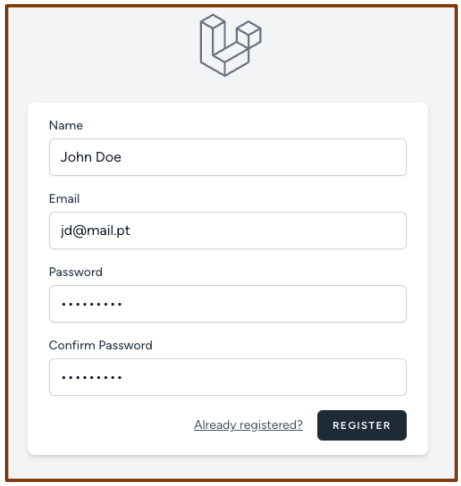
```php
Route::middleware('auth')->group(function () {
    Route::get('verify-email', EmailVerificationPromptController::class)
                ->name('verification.notice');

    Route::get('verify-email/{id}/{hash}', VerifyEmailController::class)
                ->middleware(['signed', 'throttle:6,1'])
                ->name('verification.verify');

    Route::post('email/verification-notification',
[EmailVerificationNotificationController::class, 'store'])
                ->middleware('throttle:6,1')
                ->name('verification.send');
```

58. Finally, to protect routes so that only verified users can access a given route, Laravel includes the middleware `verified`. Typically, this middleware is paired with the auth middleware. Open the file "`routes/web.php`" to view an example of a set of route protected by the `verified` middleware – search by "verified":

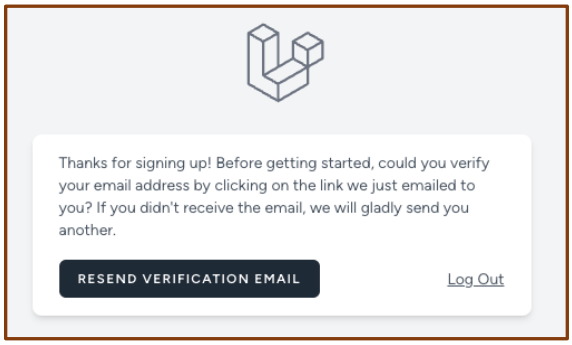
```
Route::get('/dashboard', function () {
    return view('dashboard');
})->middleware(['auth', 'verified'])->name('dashboard');

Route::middleware('auth')->group(function () {
    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');
    Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');
    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');
});
```

- Analyzing the routes, we can observe that the `/dashboard` is only accessible to users that have verified the email, and `/profile` is accessible to all authenticated users.

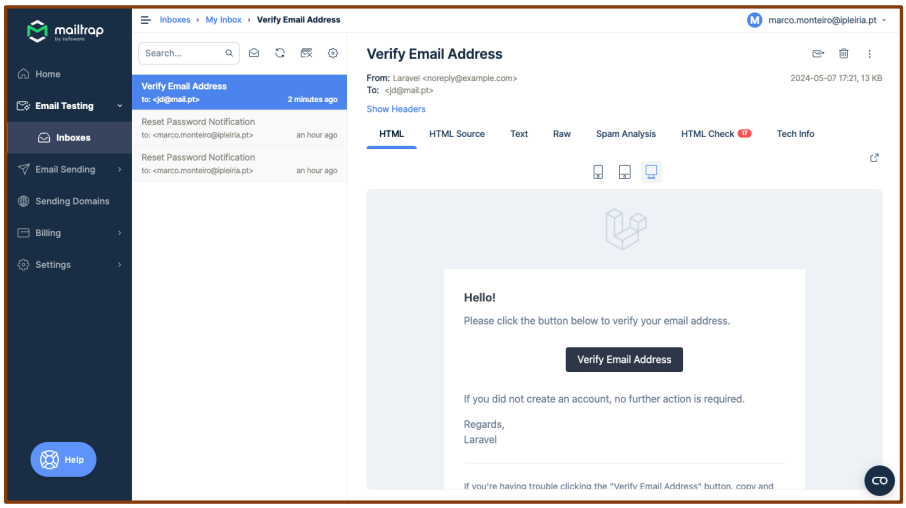
59. Let's test the email verification by registering a new user. The application will send an email to the new user, so that he can verify the email – do not open the email yet.
    - Note that the registering process is now significantly slower, because sending an email takes a few seconds to process.



60. Before validating the email, try to login with the new user credentials and try to open these pages:
    - http://yourdomain/**profile** - it must be accessible to the new user.
    - http://yourdomain/**dashboard** - it must **<u>not</u>** be accessible to the new user, as the associated route is protected by the verified middleware. This is the page that the user views while he does not verify the email (when we try to open the dashboard):



61. Go to the mailtrap inbox, open the email with the subject "Verify Email Address" and click on the "Verify Email Address" button.



62. After we click on the button to verify the email we are redirected to the dashboard - the dashboard is now fully accessible.
    - The button to "Verify Email Address" is a hyperlink to an URL with the following format:

```
http://yourdomain/verify-email/.../...token...?expires=...&signature=...token...
```

    - Tokens in the url ensure that the url is a legitimate url (has not been tampered with)
    - Url to verify the email expires on a specific timeframe and can only be used once.
63. A partial resolution is available with the full project up until this exercise (file "`ai-laravel-5.partial.resolution.4.zip`").

5. Layout adjustments

The controllers, views and other resources generated by Laravel Breeze can be adjusted for our own application. For instance, we can adjust the user registration to add custom fields to the

user. Also, we can adjust all views so that they are integrated on our layouts. These adjustments can vary from the very simple to the very complex.

Is this section, we will make very simple adjustments to the views generated by Laravel Breeze, to integrate them on our layout (`layouts.main`). Also, we will adjust the layout, so that the user's photo and name reflect the authenticated user and will be replaced with options to login and register when no user is authenticated.

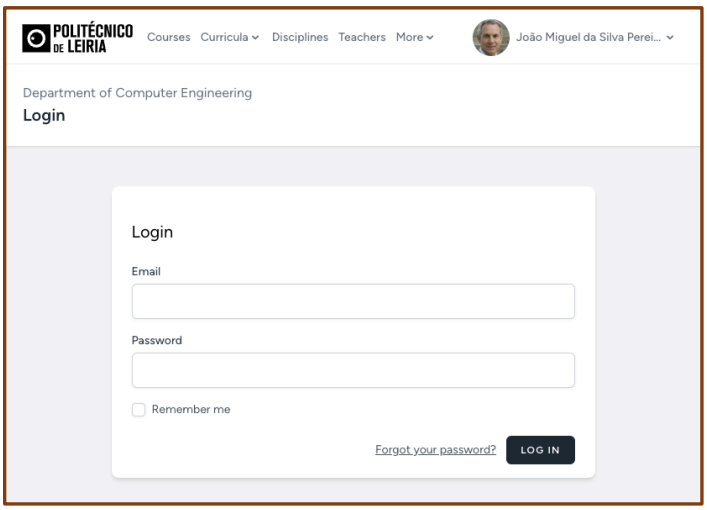
64. Edit the `auth.login` view – file `resources/views/auth/login.blade.php`. Replace:

```
<x-guest-layout>
    <!-- Session Status -->
    <x-auth-session-status class="mb-4" :status="session('status')" />
    . . .
</x-guest-layout>
```

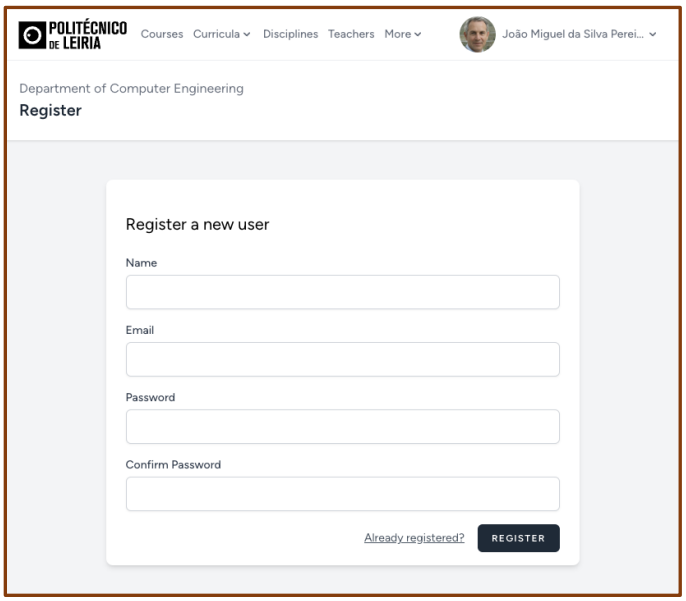
With:

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

@section('header-title', 'Login')

@section('main')
    <div class="min-h-screen flex flex-col justify-start items-center pt-6 sm:pt-0 bg-gray-100
                dark:bg-gray-900">
        <div class="w-full sm:max-w-xl mt-6 px-6 py-4 bg-white dark:bg-gray-800 shadow-md
                    overflow-hidden sm:rounded-lg">
            <h2 class="text-xl my-6">Login</h2>
            <!-- Session Status -->
            <x-auth-session-status class="mb-4" :status="session('status')" />
            . . .
        </div>
    </div>
@endsection
```

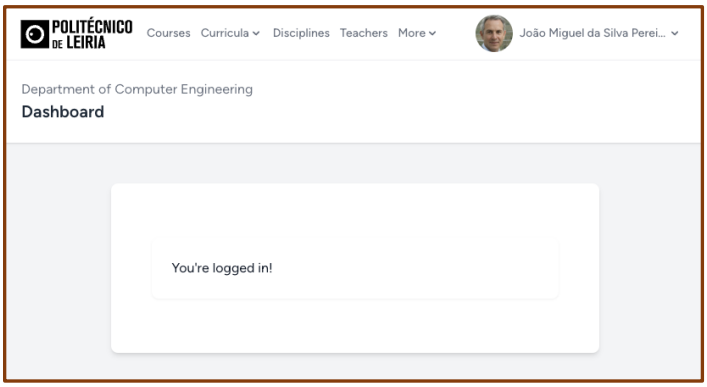
65. Check the visual aspect of the login page:



66. Apply the same pattern to the `auth.register` view.



67. Apply the same pattern to the `dashboard` view.

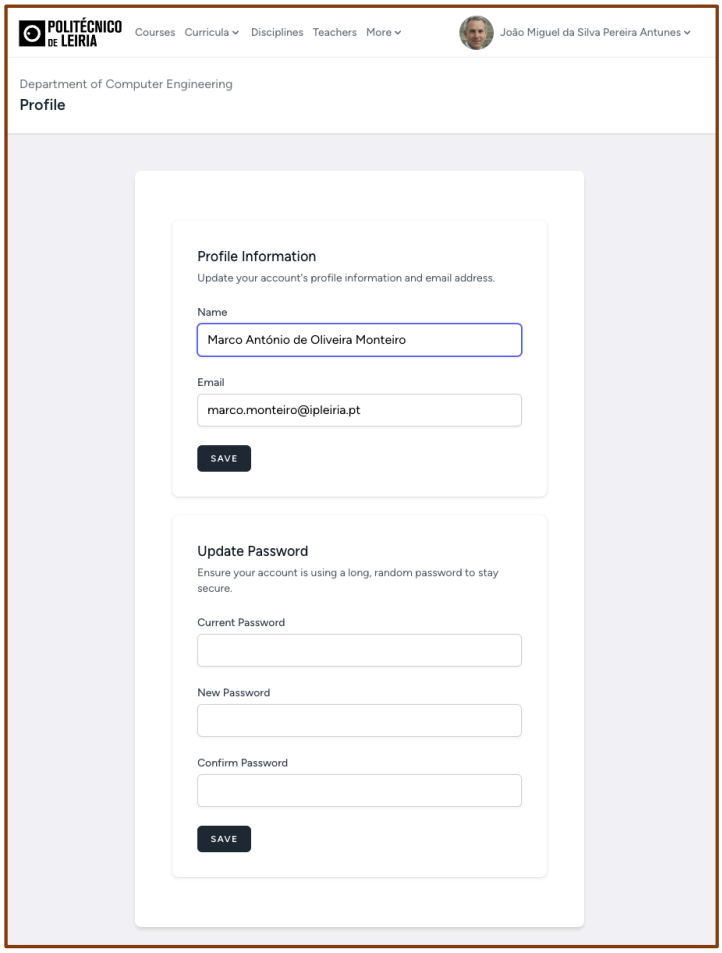


68. Change our layout (`layouts.main`) so that it supports the logout operation. Also, add the route for the menu option "Profile" and remove the menu option to change the password –

we will be able to change the password from the profile. File `resources/views/layouts/main.blade.php`:

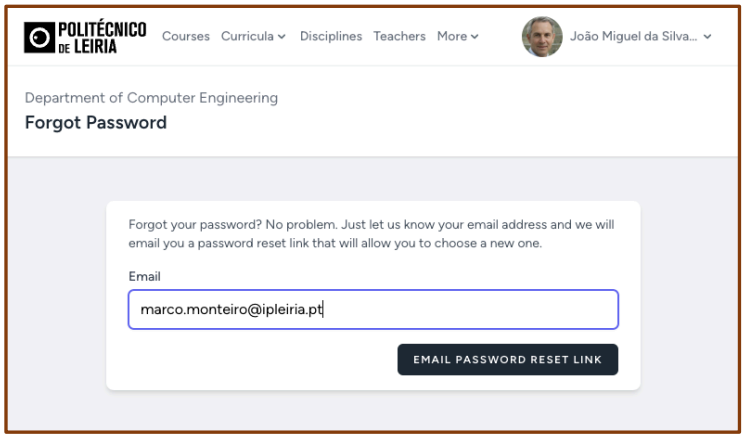
```
. . .
<x-menus.submenu-item
    content="Profile"
    selectable="0"
    href="{{ route('profile.edit') }}"/>
<hr>
<form id="form_to_logout_from_menu" method="POST" action="{{ route('logout') }}"
      class="hidden">
    @csrf
</form>
<a class="px-3 py-4 border-b-2 border-transparent
            text-sm font-medium leading-5 inline-flex h-auto
            text-gray-500 dark:text-gray-400
            hover:text-gray-700 dark:hover:text-gray-300
            hover:bg-gray-100 dark:hover:bg-gray-800
            focus:outline-none
            focus:text-gray-700 dark:focus:text-gray-300
            focus:bg-gray-100 dark:focus:bg-gray-800"
        href="#"
        onclick="event.preventDefault();
                 document.getElementById('form_to_logout_from_menu').submit();">
    Log Out
</a>
</x-menus.submenu>
. . .
```

69. Apply the same pattern to the `profile.edit` view.

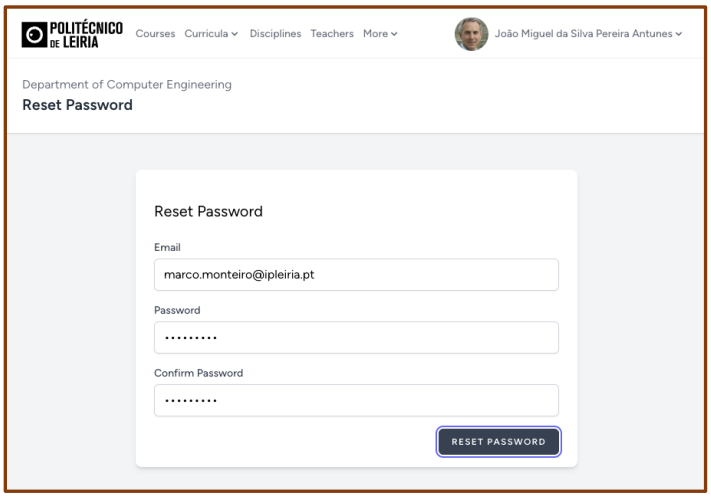


70. Apply the same pattern to the `auth.confirm-password` view.

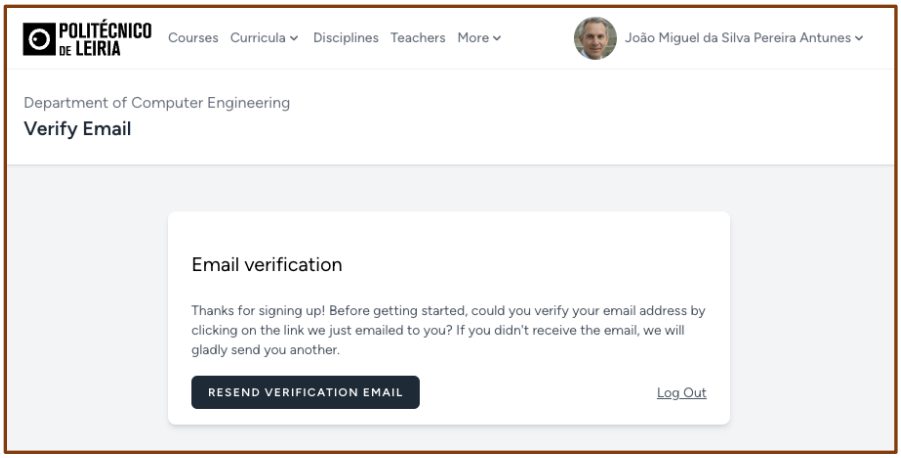
71. Apply the same pattern to the `auth.forgot-password` view.



72. Apply the same pattern to the `auth.reset-password` view.



73. Apply the same pattern to the `auth.verify-email` view.

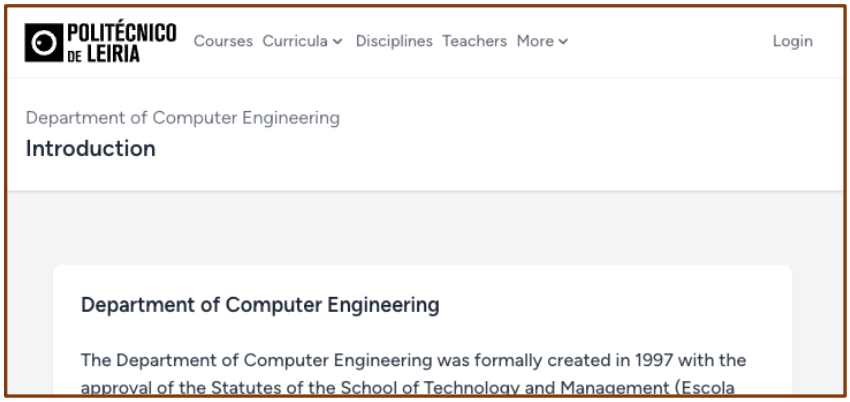


74. Finally, let's adapt the right section of menu in the main layout (`layouts.main`), so that the user's photo and name is relative to the currently authenticated user. When no user is authenticated, it will show a menu option for login.

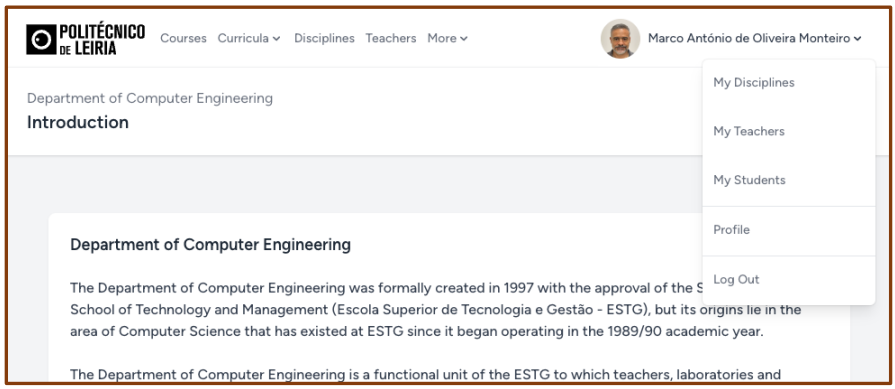
```
. . .
@auth
<x-menus.submenu
    selectable="0"
    uniqueName="submenu_user"
    >
    <x-slot:content>
        <div class="pe-1">
            <img src="{{ Auth::user()->photoFullUrl}}"
                 class=". . . ">
        </div>
```

```
        . . .
        <div class="ps-1  . . .  truncate">
            {{ Auth::user()->name }}
        </div>
    </x-slot>
    <x-menus.submenu-item
        content="My Disciplines"
        selectable="0"
        href="#"/>
    . . .
</x-menus.submenu>
@else
<!-- Menu Item: Login -->
<x-menus.menu-item
    content="Login"
    selectable="1"
    href="{{ route('login') }}"
    selected="{{ Route::currentRouteName() == 'login'}}"
    />
@endauth
```

75. Logout the user and jump to the home page. Top right menu will show the option for login:



76. Login a user. Top right menu will show the user's photo and name, and it will open a sub-menu with options associated to the authenticated user:



77. A partial resolution is available with the full project up until this exercise (file "`ai-laravel-5.partial.resolution.5.zip`").

6. Authorization

In this section we will implement some authorization examples using the `auth` and `verified` middleware, an example of a `gate` and an example of a `policy`.

78. Using the `auth` and `verified` middleware, ensure that all routes that are not accessible to the public (anonymous users) are protected – only authenticated users that have verified the email will have access to those routes. The public should have access to the following routes/pages:
    - All authentication routes required to login, forgot-password, reset-password (*the default setting generated by Laravel Breeze*).
    - The home page (/) – with the introduction to the DEI department.
    - The courses showcase.
    - The page to view the details of one course.
    - All curricula pages.
    - The page to view and filter all disciplines.
    - The page to view the detail of one discipline.

    Also, note that a non-verified user can have access to his profile and change the password (*the default setting generated by Laravel Breeze*).
79. Edit the `routes` (file `routes/web.php`) and make a group (with the middleware `auth` and `verified`) with all routes that are not public.

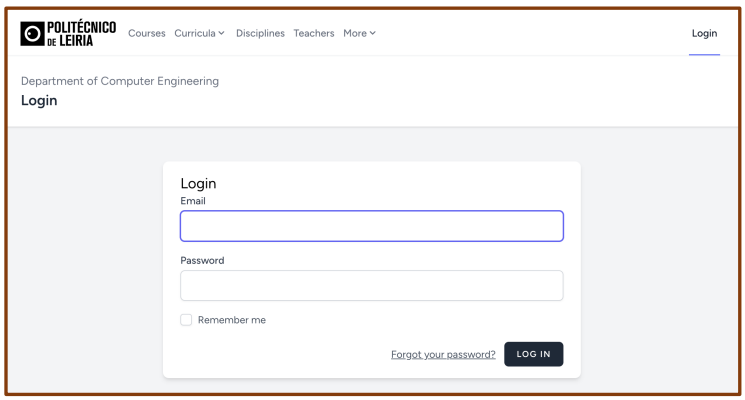
```php
<?php
. . . use . . .

/* ----- PUBLIC ROUTES ----- */
Route::view('/', 'home')->name('home');
Route::get('courses/showcase', [CourseController::class, 'showCase'])->name('courses.showcase');
Route::get('courses/{course}/curriculum', [CourseController::class, 'showCurriculum'])
        ->name('courses.curriculum');
```

```php
/* ----- Non-Verified users ----- */
Route::middleware('auth')->group(function () {
    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');
    Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');
    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');
});

/* ----- Verified users ----- */
Route::middleware('auth', 'verified')->group(function () {
    Route::view('/dashboard', 'dashboard')->name('dashboard');
    Route::delete('courses/{course}/image', [CourseController::class, 'destroyImage'])
        ->name('courses.image.destroy');

    //Course show is public, all other routes are not public
    Route::resource('courses', CourseController::class)->except(['show']);

    Route::resource('departments', DepartmentController::class);

    //Disciplines index and show are public
    Route::resource('disciplines', DisciplineController::class)->except(['index', 'show']);

    Route::delete('teachers/{teacher}/photo', [TeacherController::class, 'destroyPhoto'])
        ->name('teachers.photo.destroy');
    Route::resource('teachers', TeacherController::class);

    Route::delete('students/{student}/photo', [StudentController::class, 'destroyPhoto'])
        ->name('students.photo.destroy');
    Route::resource('students', StudentController::class);

    Route::delete('administratives/{administrative}/photo',
        [AdministrativeController::class, 'destroyPhoto'])
        ->name('administratives.photo.destroy');
    Route::resource('administratives', AdministrativeController::class);

    // Add a discipline to the cart:
    Route::post('cart/{discipline}', [CartController::class, 'addToCart'])
        ->name('cart.add');
    // Remove a discipline from the cart:
    Route::delete('cart/{discipline}', [CartController::class, 'removeFromCart'])
        ->name('cart.remove');
    // Show the cart:
    Route::get('cart', [CartController::class, 'show'])->name('cart.show');
    // Confirm (store) the cart and save disciplines registration on the database:
    Route::post('cart', [CartController::class, 'confirm'])->name('cart.confirm');
    // Clear the cart:
    Route::delete('cart', [CartController::class, 'destroy'])->name('cart.destroy');
});
```

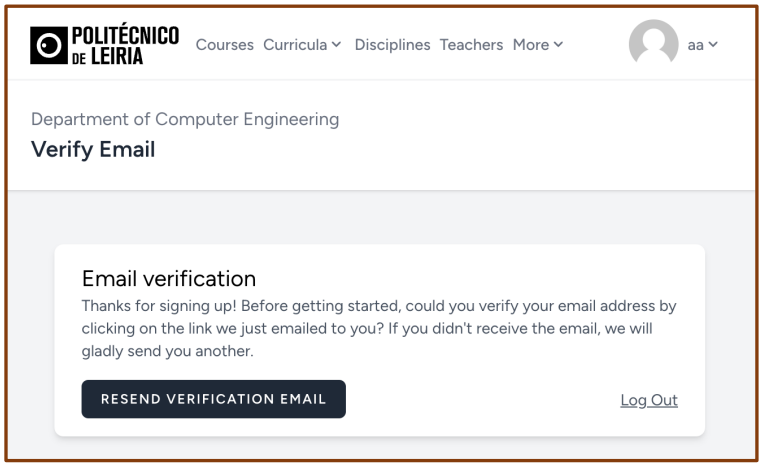
```
/* ----- OTHER PUBLIC ROUTES ----- */
//Course show is public.
Route::resource('courses', CourseController::class)->only(['show']);
//Disciplines index and show are public
Route::resource('disciplines', DisciplineController::class)->only(['index', 'show']);

require __DIR__ . '/auth.php';
```

- Note that the last 2 routes (`courses` and `disciplines` - before `require __DIR__` …) are public routes. These routes are defined after the group of routes, because the order of the routes is important – if `courses.show` had been defined before the `courses.create` route, then the url "`courses/create`" would always be handled by the `courses.show` route (it would consider `create` as the parameter of the show route)

80. Open the application as an anonymous user and try to navigate the application. Check that all public routes/pages referred previously are accessible, and that when we try to access a protected route/page we will be redirected to the login page:



81. Open the application as login as a user that has not validated the email – if no user exists without validated email, create a new user with the register page (`http://yourdomain/register`). This user should have access to the same pages as the anonymous user, with the exception of the profile page (that he has access to). When trying to access a protected page, he is redirected to the "Verify Email" page:



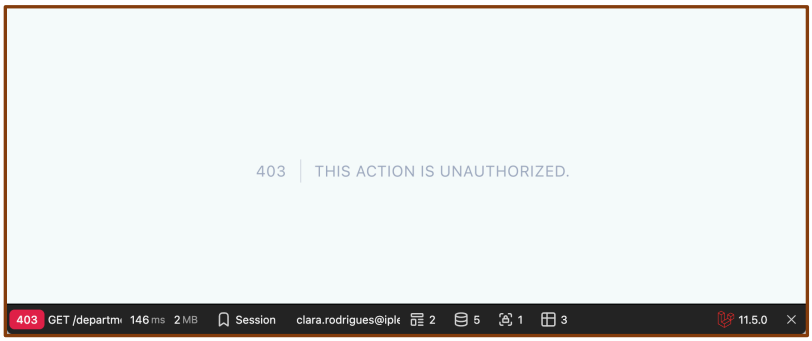
82. Next, we will ensure that the forms and operations to create, update and delete departments and courses are only accessible to admin users (users with the admin column value = 1). To implement this, we will create the gate "admin". Edit the `AppServiceProvider` (file: `app/Providers/AppServiceProvider.php`)

```php
<?php
. . .
use Illuminate\Support\Facades\Gate;
use App\Models\Course;
use App\Models\User;
. . .
class AppServiceProvider extends ServiceProvider
{
    public function boot(): void
    {
        Gate::define('admin', function (User $user) {
            // Only "administrator" users can "admin"
            return $user->admin;
        });
        try {
            View::share('courses', Course::orderBy('type')->orderBy('name')->get());
        } catch (\Exception $e) {
        }
    }
}
```

83. Now, we use the "`admin`" gate to protect the routes that are responsible for the insert, update and delete of courses and departments. Edit the `routes` (file `routes/web.php`):

```php
. . .
/* ----- Verified users ----- */
Route::middleware('auth', 'verified')->group(function () {
    . . .
    //Course show is public and index for any authenticated user
    Route::resource('courses', CourseController::class)->only(['index']);

    //Department show and index are accessible to any authenticated user
    Route::resource('departments', DepartmentController::class)->only(['index', 'show']);
    . . .
    Route::middleware('can:admin')->group(function () {
        //Course insert, update and delete related routes are for admin only
        Route::resource('courses', CourseController::class)->except(['index', 'show']);
        //Department insert, update and delete related routes are for admin only
        Route::resource('departments', DepartmentController::class)->except(['index', 'show']);
    });
});

require __DIR__ . '/auth.php';
```

84. Open the application with an admin user and with a non-admin user (check the database user's table). Compare both user's usage – the non-admin user will not be able to insert, update and delete departments and courses. When a non-admin user tries to access them, the server will respond with a `403 status` code response.



85. Next, we'll implement an example of a policy associated to the student's entity. This policy will ensure that:
    - The list of all (or filtered) students is accessible to all teachers and administrative,
    - The detail of one student is accessible to the student himself (each student only views his own details), all administrative and to the teachers that teach the disciplines the student is enrolled in.
    - The forms and operations to insert, update and delete a student are only available to the administrative

- The administrator (user with admin column = 1) has access to everything related to the student (all views, operations, etc.)

86. Create the policy with the name `StudentPolicy`. Execute the command:

```
php artisan make:policy StudentPolicy
```

87. Define the code of the `StudentPolicy` class (file "`app/Policies/StudentPolicy.php`"):

```php
<?php

namespace App\Policies;

use App\Models\Student;
use App\Models\User;

class StudentPolicy
{
    public function before(?User $user, string $ability): bool|null
    {
        if ($user?->admin) {
            return true;
        }
        // When "Before" returns null, other methods (eg. viewAny, view, etc...) will be
        // used to check the user authorizaiton
        return null;
    }

    public function viewAny(User $user): bool
    {
        return $user->type == 'T' || $user->type == 'A';
    }

    public function view(User $user, Student $student): bool
    {
        if ($user->type == 'A' || ($user->type == 'S' && $user->id == $student->user_id)) {
            return true;
        }
        // If user is teacher, then he can view the detail information of his students only
        if ($user->type == 'T') {
          // ID set of disciplines that user teaches:
          $disciplinesOfTeacherSet = $user->teacher->disciplines->pluck('id')->toArray();
          // ID set of disciplines that the student is enrolled:
          $disciplinesOfStudentSet = $student->disciplines->pluck('id')->toArray();
          return count(array_intersect($disciplinesOfTeacherSet, $disciplinesOfStudentSet)) >= 1;
        }
        return false;
    }
```

```
    public function create(User $user): bool
    {
        return $user->type == 'A';
    }

    public function update(User $user, Student $student): bool
    {
        return $user->type == 'A';
    }

    public function delete(User $user, Student $student): bool
    {
        return $user->type == 'A';
    }
}
```

- In the policy class we can add methods for each action it authorizes. For instance, if we add the method “update” on the StudentPolicy, we are defining whether the authenticated user can (method returns true) or cannot (method returns false) update a student.
- The phrase “can” “method/action” “entity” should make sense. Examples:
    - Can “update” a “student”
    - Can “view-any” “student”
    - Can “create” a “student”
    - etc.
- The method "**before**" is executed before any other action. If it returns true or false, the action is immediately authorized or not. If it returns null, then the associated action is executed to check if the user is authorized.

  For example, with the provided `StudentPolicy` class, when checking the update action ("can" the user "update" a "student") the policy class executes the "before" method. If the user is an administrator, the before method returns true and therefore, the user is immediately authorized to execute the action – no further code is executed. If the user is not an administrator, the "before" method returns null, and therefore the authorization "update" action is executed to check if the user is authorized or not. The same applies to all authorization actions – the "before" method is always executed.
- The first argument of the policy methods always refers to the current user – the authenticated user (or null for anonymous users)

- The second argument of the policy methods refers to the model instance we are protecting. For example, we can check if the user has permission to update (the method name) a specific "Student" (the second argument of the update method).
- The methods "`viewAny`" and "`create`" do not have a second argument, which means that they do not depend on a model instance – it makes no sense to check if the user has permission to "create" a specific model instance, because when creating a "student" the model instance does not exist yet.
- Summary of the policy methods of the `StudentPolicy`:
    - **before** – the "admin" has access to everything (is authorized for all actions). Other users will go through the "normal" authorization actions.
    - **viewAny** – (view all students) – returns true when user is a teacher or an administrative. Returns false otherwise.
    - **view** – (view the detail of one student) – returns true when user is an administrative or the student himself. Also returns true when the user is a teacher that teaches one of the disciplines the student is enrolled in. Returns false otherwise.
    - **create** – (view the form to create and operation – post – to create a student) – returns true when user is an administrative. Returns false otherwise.
    - **update** – (view the form to update and operation – put – to update a student) – returns true when user is an administrative. Returns false otherwise.
    - **delete** – (operation – delete – to delete a student) – returns true when user is an administrative. Returns false otherwise.
- Method names follow a convention so that it is easily mapped to the method names of a resourceful controller. If possible, we should follow that convention, but we can add extra methods – for instance, we could have a method called "view-disciplines" that would allow us to define which users can view the "disciplines" of a specific "student".
- There is an association between the convention names for methods of a policy and the convention names for methods of a controller:

Controller Method	Policy Method
index	viewAny
show	view
create	create
store	create
edit	update
update	update
destroy	delete

88. To apply the policy, we can replace the resource route line with 7 separated routes. On the routes file (`routes/web.php`) replace this:

```
Route::delete('students/{student}/photo', [StudentController::class, 'destroyPhoto'])
            ->name('students.photo.destroy');
Route::resource('students', StudentController::class);
```

With this:

```
    Route::delete('students/{student}/photo', [StudentController::class, 'destroyPhoto'])
        ->name('students.photo.destroy')
        ->can('update', 'student');
    Route::get('students', [StudentController::class, 'index'])->name('students.index')
        ->can('viewAny', Student::class);
    Route::get('students/{student}', [StudentController::class, 'show'])
        ->name('students.show')
        ->can('view', 'student');
    Route::get('students/create', [StudentController::class, 'create'])
        ->name('students.create')
        ->can('create', Student::class);
    Route::post('students', [StudentController::class, 'store'])
        ->name('students.store')
        ->can('create', Student::class);
    Route::get('students/{student}/edit', [StudentController::class, 'edit'])
        ->name('students.edit')
        ->can('update', 'student');
```

```
    Route::put('students/{student}', [StudentController::class, 'update'])
        ->name('students.update')
        ->can('update', 'student');
    Route::delete('students/{student}', [StudentController::class, 'destroy'])
        ->name('students.destroy')
        ->can('delete', 'student');
```

89. Try the application as an administrator, administrative, teacher and student. Check if the authorization rules work correctly. Consult the database to verify which students are enrolled to the teacher disciplines.
90. When we apply the conventions on the policy classes and controllers, we can simplify applying the policies. Revert the route file so that the 7 routes are replaced by the resource route. Comment the 8 routes that were created on previous exercise and replace them by these 2 routes:

```
Route::delete('students/{student}/photo', [StudentController::class, 'destroyPhoto'])
    ->name('students.photo.destroy')
    ->can('update', 'student');
Route::resource('students', StudentController::class);
```

91. Instead of protecting the routes we can protect the actions in the controller. Edit the `StudentController` (file `app/Http/Controllers/StudentController`):

```
. . .
use Illuminate\Foundation\Auth\Access\AuthorizesRequests;

class StudentController extends \Illuminate\Routing\Controller
{
    use AuthorizesRequests;

    public function __construct()
    {
        $this->authorizeResource(Student::class);
    }
    . . .
```

- This will configure the protection all 7 routes of a resource controller using a policy class. This will only work for policy and controllers that follow all Laravel conventions:

Controller Method	Policy Method
index	viewAny
show	view
create	create
store	create
edit	update
update	update
destroy	delete

92. Try the application again as an administrator, administrative, teacher and student. The authorization rules should work as previously.
93. A partial resolution is available with the full project up until this exercise (file "`ai-laravel-5.partial.resolution.6.zip`").

7. View adjustments

Up until now, we are using gates and policies to protect routes (endpoints). However, although that is the fundamental and most important part in the application authorization system, the content of the pages themselves should also be adjusted depending on the current user. Blade template language includes 2 directives - `@auth` and `@guest` – to adjust content for anonymous or authenticated user, and 3 directives - `@can`, `@cannot` and `@canany` – that are integrated with gates and policies and help us to make those adjustments without extra code to check users' permissions. Also, if required, in the view code we have access to the current user model (`Auth::user()`), from which we also have access to the can method.

In this section, we will use blade directives (`@auth`, `@guest`, `@can`, `@cannot` and `@canany`) to adjust the views, so that the current user only has access to UI elements that are associated to pages or operations that he has permission to access.

94. We've already protected our routes, so that an anonymous user (public) only has access to the following routes/pages:
    - All authentication routes required to login, forgot-password, reset-password (*the default setting generated by Laravel Breeze*).
    - The home page (/) – with the introduction to the DEI department.
    - The courses showcase.

- The page to view the details of one course.
- All curricula pages.
- The page to view and filter all disciplines.
- The page to view the detail of one discipline.

95. Try the application with an anonymous user and an authenticated user.
96. Let's change our layout (file `resource/views/layouts/main.blade.php`) so that all menu options that are not publicly accessible are hidden from the anonymous user. Currently, we just need to hide the "`Teachers`" and "`More`" menu options.

```
. . .
@auth
<!-- Menu Item: Teachers -->
<x-menus.menu-item
    content="Teachers"
    selectable="1"
    href="{{ route('teachers.index') }}"
    selected="{{ Route::currentRouteName() == 'teachers.index'}}"
    />

<!-- Menu Item: Others -->
<x-menus.submenu
    selectable="0"
    uniqueName="submenu_others"
    content="More">
        <x-menus.submenu-item
            content="Students"
            selectable="0"
            href="{{ route('students.index') }}" />
        <x-menus.submenu-item
            content="Administratives"
            selectable="0"
            href="{{ route('administratives.index') }}" />
        <hr>
        <x-menus.submenu-item
            content="Departments"
            selectable="0"
            href="{{ route('departments.index') }}"/>
        <x-menus.submenu-item
            content="Course Management"
            href="{{ route('courses.index') }}"/>
```

```
</x-menus.submenu>
@endauth

<div class="grow"></div>
. . .
```

- Check the file `08-Laravel.5.code.txt` for the complete code of the layout file.

97. Next, we'll adjust the student's related UI elements (menu options, pages, sections of pages, operations, etc.) according to the authorization rules defined by the policy class. We'll use the blade directive `@can` to check if current user has permission to view or execute something (related to the student).
98. Let's start by only showing the menu option "Students" – that will show the list of students - if the current user has permission to view any student (view the list of students).

```
. . .
<!-- Menu Item: Others -->
<x-menus.submenu
    selectable="0"
    uniqueName="submenu_others"
    content="More">
        @can('viewAny', App\Models\Student::class)
        <x-menus.submenu-item
            content="Students"
            selectable="0"
            href="{{ route('students.index') }}" />
        @endcan
. . .
```

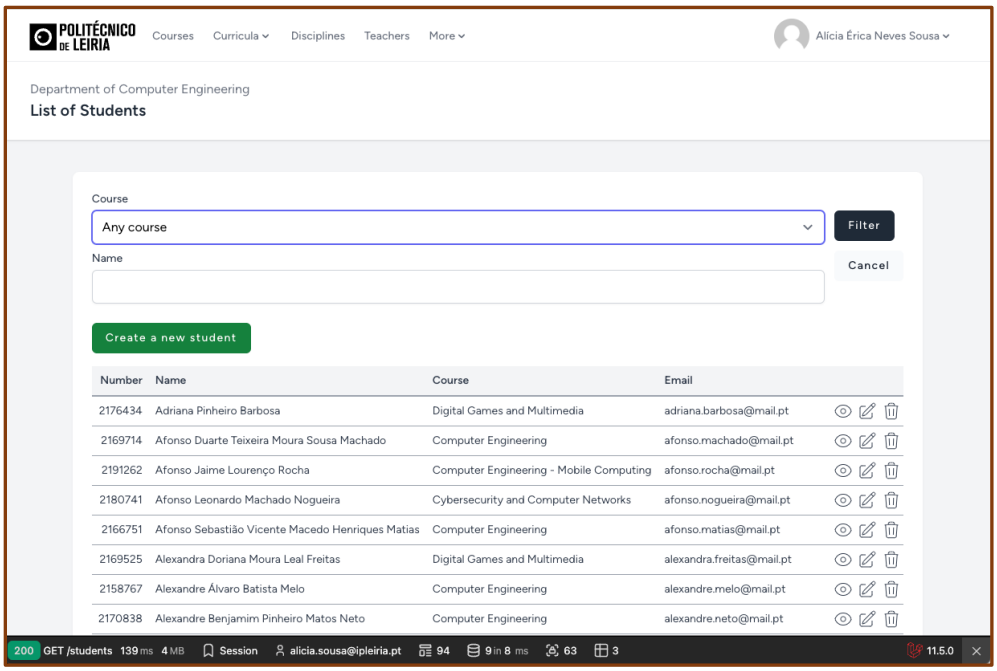
99. Try the application with a student (he should not have access to the list of students) and with a teacher or administrative.
100. On the `students.index` view, that shows (and filters) the list of students, we will hide the button to "create a new student" when the current user does not have permission to do it. On the file `resources/views/students/index.blade.php` change this section of code:

```
@can('create', App\Models\Student::class)
<div class="flex items-center gap-4 mb-4">
    <x-button
        href="{{ route('students.create') }}"
        text="Create a new student"
        type="success"/>
</div>
@endcan
```

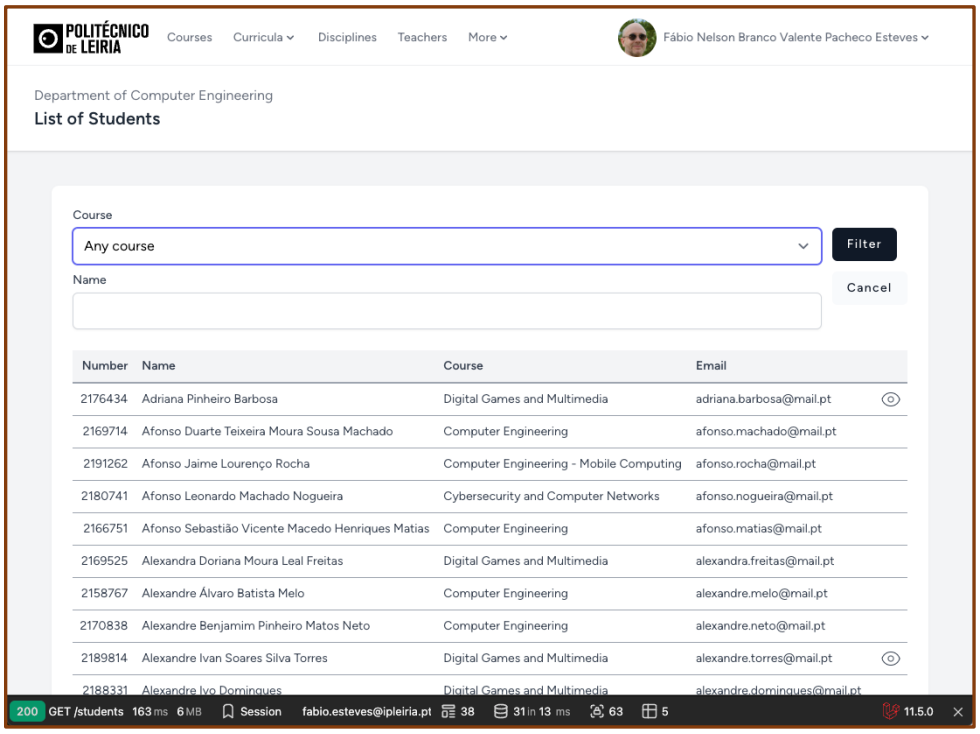
101. The next objective is to hide the student's table buttons to view, edit and delete a student, using the `@can` blade directive. We'll apply the `@can` directive directly in the "`students.table`" component's view. Edit the file `resources/views/components/students/table.blade.php`:

```
@if($showView)
    @can('view', $student)
        <td>
            <x-table.icon-show class="ps-3 px-0.5"
            href="{{ route('students.show', ['student' => $student]) }}"/>
        </td>
    @else
        <td></td>
    @endcan
@endif
@if($showEdit)
    @can('update', $student)
    <td>
        <x-table.icon-edit class="px-0.5"
        href="{{ route('students.edit', ['student' => $student]) }}"/>
    </td>
    @else
        <td></td>
    @endcan
@endif
@if($showDelete)
    @can('delete', $student)
    <td>
        <x-table.icon-delete class="px-0.5"
        action="{{ route('students.destroy', ['student' => $student]) }}"/>
    </td>
    @else
        <td></td>
    @endcan
@endif
```

102. If we try the application as an administrative, we'll have access to the "create a new student" button and all the icon buttons to view, edit or delete students:



103. If we try the application as a teacher (without "admin" privileges), we'll **<u>not have access</u>** to the "create a new student" or to any "edit" or "delete" button. We'll only have access to some "view" buttons, of the students that are enrolled on the disciplines of current teacher. This reflects current authorization rules specified by the `StudentPolicy`.



104. With the Laravel Telescope Toolbar we can verify that 31 database queries were executed to build the page. This is because when executing the policy method "view" for each student in the page (20 students) a query is executed to verify if the student's disciplines set includes any of the discipline of the authenticated user (teacher). To reduce the number of queries, will use an eager loading strategy for the disciplines of the student. Edit the method `index` of the `StudentController` so that when loading the students, also load the associated disciplines (file `app/Http/Controllers/StudentController.php`):

```
public function index(Request $request): View
{
    . . .
    $students = $studentsQuery
        ->with('user', 'courseRef', 'disciplines')
        ->paginate(20)
        ->withQueryString();

    return view(
        'students.index',
        compact('students', 'courseOptions', 'filterByCourse', 'filterByName')
    );
}
```

105. Try to open the same page as before (list of students) and verify that the number of database queries is now reduced to 12 queries (from 31 to 12 queries):

106. Student's detail (view and edit page for a student) also include buttons to create a new student, edit (or view) the student and delete the student. All these buttons should be hidden when the user does not have permission to the associated operation. Edit the `students.show` view (file `resources/views/students/show.blade.php`).

```
@can('create', App\Models\Student::class)
<x-button
    href="{{ route('students.create') }}"
    text="New"
    type="success"/>
@endcan
@can('update', $student)
<x-button
    href="{{ route('students.edit', ['student' => $student]) }}"
    text="Edit"
    type="primary"/>
@endcan
```

```
@can('delete', $student)
<form method="POST" action="{{ route('students.destroy', ['student' => $student]) }}">
    @csrf
    @method('DELETE')
    <x-button
        element="submit"
        text="Delete"
        type="danger"/>
</form>
@endcan
```

107. Do the same for the `students.edit` view (file `resources/views/students/edit.blade.php`).

```
@can('create', App\Models\Student::class)
<x-button
    href="{{ route('students.create') }}"
    text="New"
    type="success"/>
@endcan
@can('view', $student)
<x-button
    href="{{ route('students.show', ['student' => $student]) }}"
    text="View"
    type="info"/>
@endcan
@can('delete', $student)
<form method="POST" action="{{ route('students.destroy', ['student' => $student]) }}">
    @csrf
    @method('DELETE')
    <x-button
        element="submit"
        text="Delete"
        type="danger"/>
</form>
@endcan
```

108. If we try to open the detail (view) page of a student, as an administrative, we'll have access to all buttons of the student. If we open the same page as a teacher, we have no access to any of the buttons ("new"; "edit/view"; "delete").

109. A partial resolution is available with the full project up until this exercise (file "`ai-laravel-5.partial.resolution.7.zip`").

8. Autonomous Work

In this section students must implement everything autonomously, but taking into account the requirements, recommendations and suggestions. A solution for all the exercises is provided. Analyze the provided solution and compare it with your own solution.

8.1. Profile pages

Implement the pages "My Disciplines", "My Teachers" and "My Students".

- "**My Disciplines**" will show the list of disciplines the current user teaches (if the current user is a teacher); the list of disciplines the current user is enrolled in (if the current user is a student).
- "**My Teachers**" shows the list of teachers for the disciplines that the current user is enrolled in (if the current user is a student)
- "**My Students**" shows the list of students for the disciplines that the current user teaches (if the current user is a teacher)

Use the following URL patterns for the routes:

- "**My Disciplines**" – url = "disciplines/my"
- "**My teachers**" – url = "teachers/my"
- "**My Students** " – url = "students/my"

Refactor the profile page so that is used only to change the password. Use the pages to update the teacher (`teachers.edit`), student (`students.edit`) or administrative (`administratives.edit`) as the **profile**.

- Change the menu options to reflect previous refactoring. The "`Profile`" menu option will open the corresponding update page (the route depends on the user type). Add a menu option to change the password ("`Change Password`").

8.2. Policies and gates

Apply the same design pattern used on the `StudentPolicy` to create policy classes for the entities: department, course, discipline, teacher and administrative (associated to the User model). Create gates to handle cart features. Replace the authorization code (on the routes and views) relative to courses, curricula and disciplines that depended on authenticated and

non-authenticated user (middleware `auth` and blade directive `@auth`) with similar code that uses the new policies. Also, do the same for the "`admin`" gate – use course and department policies to authorize related resources (instead of an "`admin`" gate). Remove the "`admin`" gate.

Policy and Gates authorization rules

When creating or editing the **policy classes**, take into consideration the following:

- **Department** – any authenticated user can view (viewAny and view) the department information, but only an "admin" user can create, update or delete departments.
- **Course** – All users (including anonymous) can view the show case with all courses and the detail of one course (including the course curricula). Everything else (including viewing the table with the list of courses and executing any operation – create, update or delete) is restricted to "admin" users only.
- **Discipline** – All users (including anonymous) can view the list of disciplines (viewAny) and the detail of one discipline (except the list of teachers of that discipline, which should only be visible to an authenticated user). Only administrative, and "admin" users can create, update or delete disciplines.
  Teachers and Students may also view the list of their disciplines.
- **Teacher** – Anonymous users cannot view anything related to the teachers. All authenticated users (teachers, students and administrative) can view the list of teachers and the detail of one teacher. Only administrative, and "admin" users can create or delete teachers. The operation to update a teacher can be executed by an administrative, an "admin" user or by the teacher himself (each teacher can update his own information).
  Students may also view the list of their teachers.
- **Students** – Change the students' policy already defined on the worksheet, so that a student can update his own record (so that we can use the update student as the profile). Also, add a new authorization rule to allow Teachers to view the list of their students.
- **Administrative** – (this policy must be registered manually, so that it is mapped to the User model – an administrative is represented by the User model)
  All administrative and "admin" users can view (`viewAny` and `view`) any administrative, but only the "admin" user can create or delete administrative users (type = 'A'). The operation to update an administrative (user type = 'A') can be executed by an "admin" user or by the administrative himself (user type = 'A') - each administrative can update his own information.

    - Check https://laravel.com/docs/authorization#registering-policies to learn how to manually register policies.

Create **2 gates** to handle the cart (`use-cart` and `confirm-cart`) and take into consideration the following:

- **use-cart** gate – only unauthorized users, students and administrative users have access to the views, features and operations of the cart (except cart confirmation) – they can view the cart, add to cart, remove from cart and clear the cart.
  Note: teachers cannot use the cart.
- **confirm-cart** gate – only students and administrative users can execute the operation to confirm the cart (note: unauthorized users cannot confirm the cart).
    - Note that when a student confirms a cart, the only acceptable student number is his own number. Change the `CartConfirmationFormRequest` code to incorporate that validation.
    - Also, if a student opens the Cart page, the number of student should assume his number by default.

8.3. View adjustments

Considering all authorization rules applied by the `StudentPolicy` and the new policies created on previous section, adapt all views so that the current user only has access to the UI elements required for the features he has privileges.

- Use the `@can`, `@cannot` and `@canany` blade directives to hide UI elements.
- Ensure that if a user does not have access to the list of a specific entity, he should not be able to view that list through a sub-view or component. For instance, if a user (e.g. an anonymous user) cannot view the list of teachers, he should not be able to view the list of teachers of a discipline or the list of teachers of a department.

8.4. User creation

The business model for our application should not allow a free registration process – only administrative or "admin" users can create a student or a teacher, and only an "admin" user can create an administrative. Remove the "register" feature (created by Laravel Breeze) from the application (remove the routes – ensure that no routes are available for registering a user).

When creating a new user (student, administrative and teacher) send a verification email for the new user.

Check https://laravel.com/docs/verification#resending-the-verification-email. Use the following code to send the verification email manually:

```
$userModel->sendEmailVerificationNotification();
```

Note: to simplify the development, all new users will have the same password: "123"

8.5. Admin user

The "detail" and the "edit" page of teachers and administrative include an administrator flag (“admin” checkbox field). Implement all that is required so that only an administrator user ("admin" user) can change the value of that field, to grant or revoke administration privileges to other users. That field should be read-only for all other users.

When creating a new user (teacher or administrative) only the "admin" user can set him as an admin ("admin" checkbox is checked) – otherwise, new users are always "non admin". Also, students can never be "admin" users.

- Create the authorizations methods "`createAdmin`" (when creating a new teacher or administrative) and "`updateAdmin`" (when updating a teacher or administrative) in the teacher and administrative policy classes.
- An administrator user cannot revoke his own "admin" privileges – he can only change the "admin" value of other users.
    - This prevents a loophole from happening, where the last "admin" user would revoke his own "admin" privileges, and therefore no user would be able to grant "admin" privileges to anyone.
- Do not implement this protection on the UI only – don’t forget that an attacker can send HTTP requests without using the application user interface.

8.6. Final adjustments

Implement some final adjustments to the project.

Refactor submenu-item component.

The submenu "logout" was implemented directly in the main layout. Refactor the component `<x-menus.submenu-item>` so that it supports forms. Refactor the main layout so that the logout option uses the new version of the `<x-menus.submenu-item>` component.

Refactor DB class usage.

During the project implementation, we've used the DB class on several occasions because there was no ORM model available at that time – for instance, on the `destroy` method of several controllers, we have used the DB class to get the total of elements from the database. Currently the project already has all the required ORM models, so we can refactor all code that uses the DB class to use an ORM class.
Whenever possible replace the DB class with an ORM equivalent.

Correct errors.

During the project development, there is always the possibility of accidently introducing errors or incorrect code. For example, on some views to show the details of entities, we've added a button to create the associated entity with a code similar to the following:

```
href="{{ route('administratives.create', ['administrative' => $administrative]) }}"
```

Although it works fine, the second argument with the route parameter value is incorrect, because the associated route does not have any parameter. Code works fine because the parameter value is ignored – no error exception is thrown, but the code is not correct.

Correct the code to:

```
href="{{ route('administratives.create') }}"
```

If more errors or incorrect code is detected, correct them.

8.7. Comparation

Analise the provided solution. Compare it with your own solution.

Summary

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

```
Laravel Authentication
      Laravel/Ui – generation of authentication resources
      Adaptation of generated authentication resources
      Login
      Logout
      Forgot your password and reset password
      User registration
      Email verification
      Profile
      Change password
Sending email
      Mailtrap.io
      Configuring email server
Authorization
      Middleware based authorization
      Creating and registering custom middleware
      Gates and Policies
      Resource Policies – name conventions
      Class or instance-based action policies
      Before method on class policies
      Applying policy restrictions with:
            Middleware – can
            Controller method: authorizeResource
            User model instance – method can
            Blade directive @can, @cannot and @canany
      View adjustments depending on user privileges
```