DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

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

APLICAÇÕES PARA A INTERNET

Engenharia Informática

Marco Monteiro

6 - Laravel - 3

Objectives:

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

(2) Database access with DB and Eloquent ORM

(3) Laravel Tinker

(4) Model Relationships

Note the following:

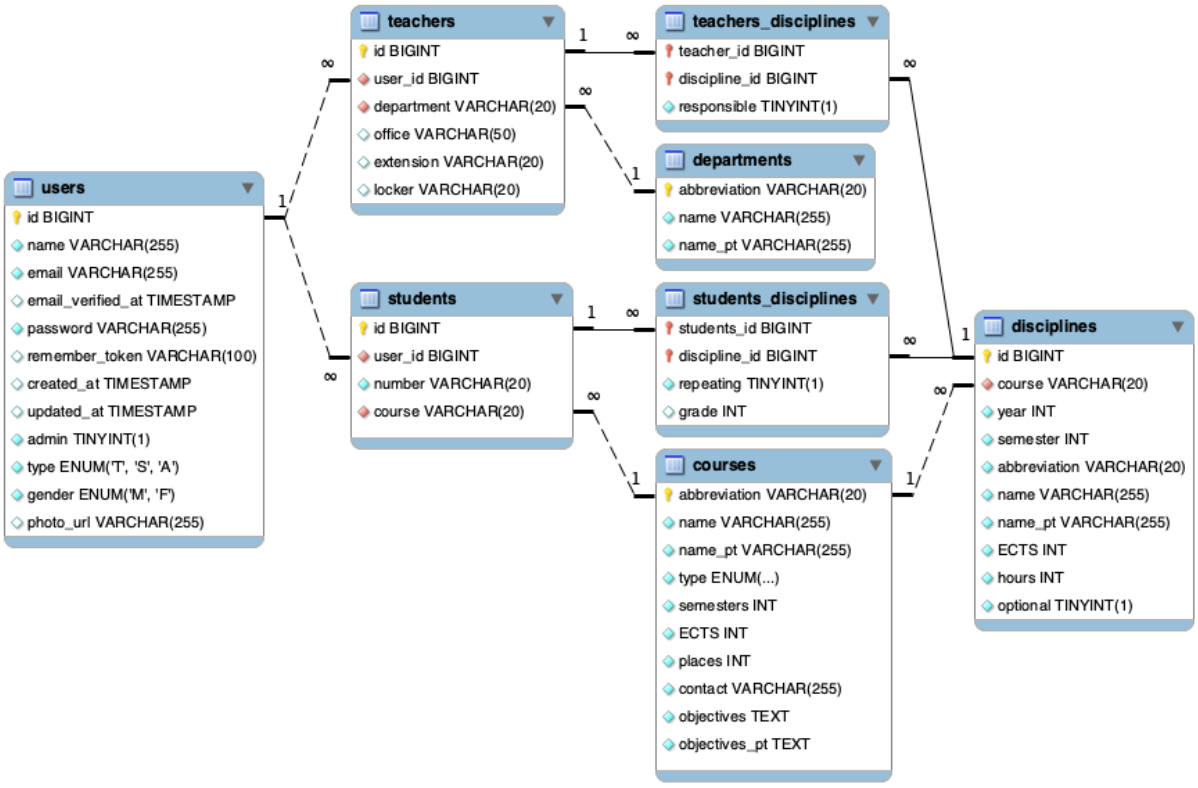
- Before starting the exercise, read the related content on Moodle;
- During the resolution of the exercises, consult the Laravel documentation (https://laravel.com) and other online resources.

Scenery

This worksheet will continue the development of the Web Application from the last worksheet where we've applied a template (Tailwind CSS based template) to define the web application layout and created several views and components to ensure a consistent "professional" looking. In the current worksheet we will use eloquent queries and relationships, and other database related features and tools (Laravel Telescope and Laravel Tinker) to enhance the application functionalities.

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-3.zip`) into the Laragon root folder and decompress it on that folder.

2. Previous command will create the folder `ai-laravel-3`, 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-3`. For example, `C:\laragon\www\ai-laravel-3` or `D:\ainet\ai-laravel-3` (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-3`), 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-3.test`" URL to access the content.
9. Test CRUD operations for courses (`http://ai-laravel-3.test/courses`) and disciplines (`http://ai-laravel-3.test/disciplines`)

1.2. New Laravel Project – with Laravel Sail

10. Copy the provided zip file (`start.ai-laravel-3.zip`) into any folder and decompress it.
11. Previous command will create the folder `ai-laravel-3`, that will be the current worksheet project folder.
12. Execute the following command on the project folder (`ai-laravel-3`), 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-3.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-3` 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-3.zip`) into any folder and decompress it.
20. Previous command will create the folder `ai-laravel-3`, 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-3`) 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` or "`http://ai-laravel-2.test`") 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://ai-laravel-1.test/courses`) and disciplines (`http://ai-laravel-1.test/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`)

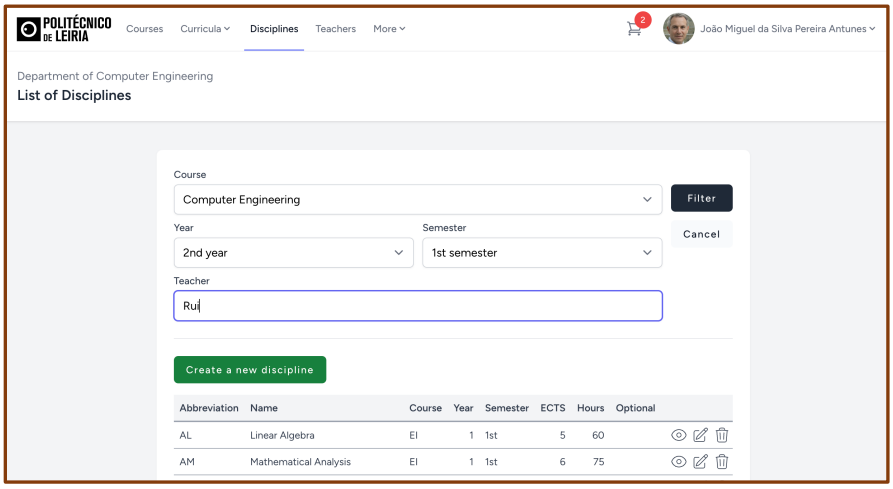
2. Eloquent queries

Up until now, our application either retrieves a single row (single model) from a table or retrieves all rows at once - `Model::all()` - or using Laravel pagination mechanism - `Model::paginate(20)`. However, one of the fundamental features of the relational databases is the ability to execute queries to filter the data of the database. These queries can vary from the extremely simple queries (e.g.: `select * from disciplines where year = 3`) that return a subset of table rows, to very complex queries that aggregate data from several tables.

Laravel Eloquent ORM (Object-Relational Mapping) models, as well as the DB class, support several methods that allow us to specify queries without using raw SQL command. Also, using the method select of the DB class - `DB::select(…)` – it is possible to execute a database query passing on the raw SQL command.

Let's use the Eloquent query features to change the disciplines page, so that the end-user can filter which disciplines to show.

25. Provided project already includes a form to define the disciplines filter. Open the page `http://yourdomain/disciplines` and try to add some values to the filter form:



26. Click on the button "Filter" to submit the form. Analyze the URL of the "submitted page". It should create a URL similar to the following:

```
http://yourdomain/disciplines?course=EI&year=2&semester=1&teacher=Rui
```

- Because the form uses the GET method, when data is submitted, the browser creates a HTTP request with GET method. Form data (input fields) are passed on to the HTTP request using **query string** parameters.

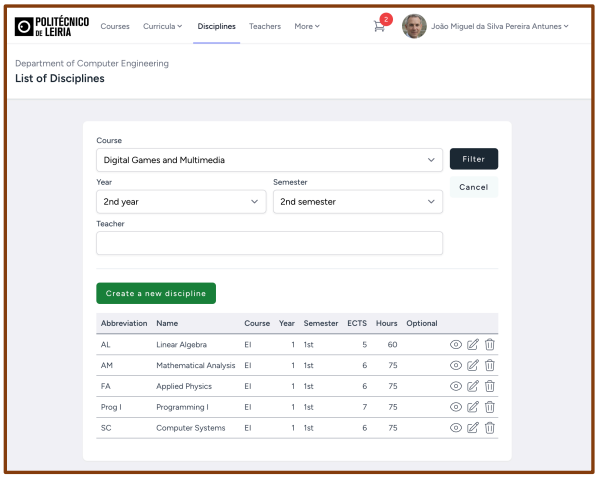
27. The form to filter the discipline is created by the component `disciplines.filter-card`. Analyze this components' code (`app/View/Components/Disciplines/FilterCard.php`) and design (`resources/views/components/disciplines/filter-card.blade.php`). Also, analyze the code of the `disciplines.index` view that uses this component.

28. Change the code of the method `index` of the controller `DisciplineController` (file `app/Http/Controllers/DisciplineController.php`) to handle the query string parameters `course`, `year` and `semester`. For now, we'll ignore the parameter `teacher`.

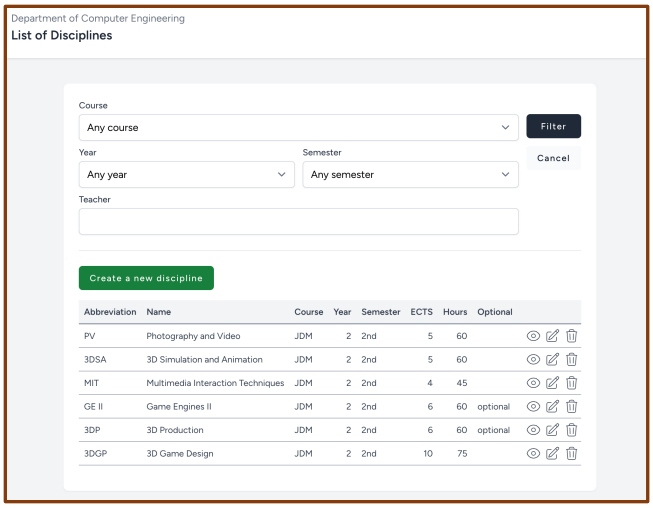
```php
public function index(Request $request): View
{
    $filterByCourse = $request->query('course', 'EI');
    $filterByYear = $request->year ?? 1;
    $filterBySemester = $request->input('semester') ?? 1;
    $disciplines = Discipline::where('course', $filterByCourse)
        ->where('year', $filterByYear)
        ->where('semester', $filterBySemester)
        ->paginate(20);
    return view('disciplines.index',
        compact('disciplines', 'filterByCourse', 'filterByYear', 'filterBySemester'));
}
```

- We've added the argument (`Request $request`) to the index method. When Laravel detects a `Request` argument it fills it with an instance of the current request.
- We've used 3 alternative approaches to read the value of a query string parameter:
    - `$request->query('course', 'EI');`
    - `$request->year ?? 1;`
    - `$request->input('semester') ?? 1;`

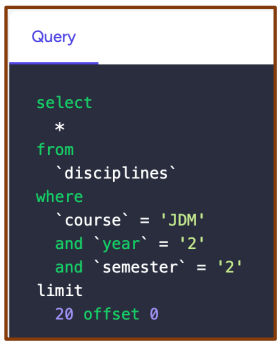
29. Open the disciplines page (`http://yourdomain/disciplines`). When opening the page, verify that the disciplines presented are the disciplines of Computer Engineering course (abbreviation = EI), first year and first semester, which correspond to the default values for `$filterByCourse`; `$filterByYear` and `$filterBySemester`. Change the filter values to Couse = "Digital Games and Multimedia"; Year = "2" and Semester = "2".



30. Click on "Filter" button (filter values: Couse = "Digital Games and Multimedia"; Year = "2" and Semester = "2"). It should present the disciplines of 2nd year and 2nd semester of the course "Digital Dames and Multimedia":



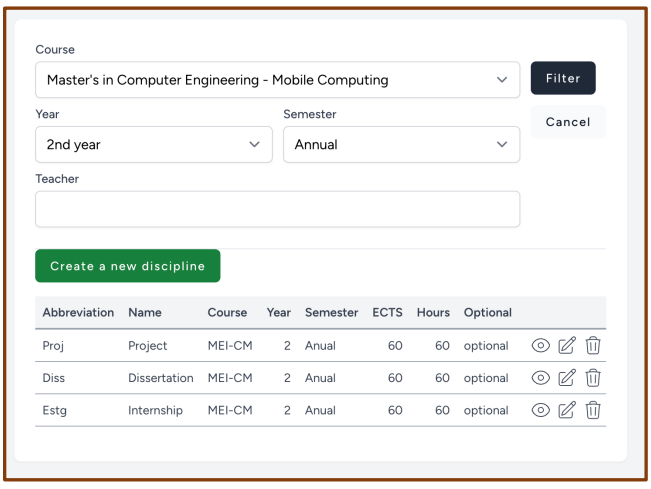
31. We can use “Laravel Telescope” to check which SQL command (Query) was sent to the database server. For the previous example:



32. Although the specified filter works fine, when showing the results the form fields do not maintain state. Change the section that uses the component `disciplines.filter-card` on the view `disciplines.index` view file (`resources/views/disciplines/index.blade.php`) to:

```
<x-disciplines.filter-card
    :filterAction="route('disciplines.index')"
    :resetUrl="route('disciplines.index')"
    :courses="$courses->pluck('fullName', 'abbreviation')->toArray()"
    :course="old('course', $filterByCourse)"
    :year="old('year', $filterByYear)"
    :semester="old('semester', $filterBySemester)"
    class="mb-6"
    />
```

33. Now, the filter form of the disciplines page maintains the state. For instance, try to filter all annual disciplines of the 2nd year of "Master's in Computer Engineering – Mobile Computing" course. When opening the page with the filter results, the form `<input>` elements are filled with the parameter values used on the filter:



34. Disciplines' filter still has a major problem – it doesn't work correctly when filtering by "Any course", "Any year" or "Any semester". Why?
    - These values ("Any course", "Any year" or "Any semester") are defined on the options of the select field as null values – check the component's class code (`app/View/Components/Disciplines/FilterCard.php`).
    - The null value is passed on to the query string parameter as an empty string, which then is converted again to null on the `$request` instance.
    - When the value is null, the controller code will use the default value instead (course = EI, year = 1 and semester = 1).
35. To solve this issue, we just have to change the controllers' code to consider the null value as a relevant value – it means that the associated filter parameter is ignored (e.g. if year is null, then we do not filter by year). Change the code of the method `index` of the controller `DisciplineController` (file `app/Http/Controllers/DisciplineController.php`)

```php
public function index(Request $request): View
{
    $filterByCourse = $request->query('course');
    $filterByYear = $request->year;
    $filterBySemester = $request->input('semester');
    $disciplinesQuery = Discipline::query();
    if ($filterByCourse !== null) {
        $disciplinesQuery->where('course', $filterByCourse);
    }
    if ($filterByYear !== null) {
        $disciplinesQuery->where('year', $filterByYear);
    }
    if ($filterBySemester !== null) {
        $disciplinesQuery->where('semester', $filterBySemester);
    }
    $disciplines = $disciplinesQuery->paginate(20);
    return view(
        'disciplines.index',
        compact('disciplines', 'filterByCourse', 'filterByYear', 'filterBySemester')
    );
}
```

- The static “query” method of the model - **Discipline::query()** - returns an empty **query builder** instance for the Discipline model. Using the query builder, we can fluently create a more elaborated query.
- Applying the method **paginate()** to the query builder, will generate and execute the SQL command and returns the data from the database as an Eloquent Collection.

- The expression Discipline::where(…)->where(…)->where(…)->paginate(…) also uses a query builder instance to create the SQL expression. The first “where” method returns the query builder instance, the second, third and all subsequent “where” methods use that query builder instance to manipulate the query. The expression ends with the method paginate(), that is applied to the same query builder instance and is responsible for executing the SQL command and returning the data from the database.
- The query builder has several methods, that mimic SQL expressions operators/functions. Examples of methods that can be applied to the query builder:
    - `where, whereNot, orWhere, orWhereNot,`
    - `whereIn, whereNotIn, orWhereIn, orWhereNotIn,`
    - `whereNull, whereNotNull, orWhereNull, orWhereNotNull,`
    - `orderBy,`
    - `groupBy, having,`
- The query builder includes several methods that generate and execute the SQL command and retrieve data from the database – these methods “terminate” the fluent sequence of methods that are applied to the query builder. Example of methods that execute the SQL command and retrieve data from the database:
    - **paginate** – returns one page of data as an Eloquent Collection. Essential method to implement the pagination mechanism.
    - **get** – returns data as an Eloquent Collection.
    - **first** – returns the first row of the query as an Eloquent Model (object).
    - **pluck** – returns data as a Collection (similar to array) with one column (value) or with 2 columns (key and value).
    - **chunk** – returns data as an Eloquent Collection, but small chunks at a time. Useful to cycle through very large datasets without breaking memory limitations. Loads a subset (a chunk) of the data to memory so that it can be handled with PHP code, then when the handling ends, loads the next chunk up until the full dataset is cycled through.
    - **count**, **min**, **max**, **sum**, **avg** – returns the aggregated value as a scalar value.
- Documentation about the query builder: https://laravel.com/docs/queries

36. Open the `http://yourdomain/disciplines` again and verify that the filter values "Any course", "Any year" or "Any semester" are considered when filtering. Also, they are the new

default values for the filter form. Try to click on the "Cancel" button – this will open the disciplines page without any query string parameter – the default values will apply.

37. Course, year and semester filter parameters are working perfectly. Next, we will also filter by a substring of the teacher's name. The objective is to filter by all 4 parameters (course, year, semester and substring of the teacher's name).

    The main problem with the teacher's name filter parameter is that the name of the teacher is not available on the "disciplines" table, instead it is available only on the "users" table (column "name" of the users table). The table "users" is related to the "disciplines" table through a chain of foreign key relations. Table "users" has a 1:1 relationship to the "teachers" table, and then table "teachers" has a N:M relationship with the table "disciplines" using a pivot table ("teachers_disciplines").

    This means that we cannot add a simple `where(…)` method to filter the disciplines by teacher's name. Instead, we have 2 alternatives:

    - Add "joins" to the query builder, so that we have access to the name of the teachers' that are related to the disciplines.
    - Before creating the command to query the disciplines, add the required query or queries that return a set (an array) of disciplines' ID that the teacher's names are associated to. Then, when querying the disciplines ensure that the filter is applied only to that subset of disciplines' ID – using the query method `whereIn` or `whereIntegerInRaw` (the latter has better performance but only works with integers)

38. For the first version of the teacher's name filter parameter solution, we will use joins. Change the code of the method `index` of the controller `DisciplineController` (file `app/Http/Controllers/DisciplineController.php`):

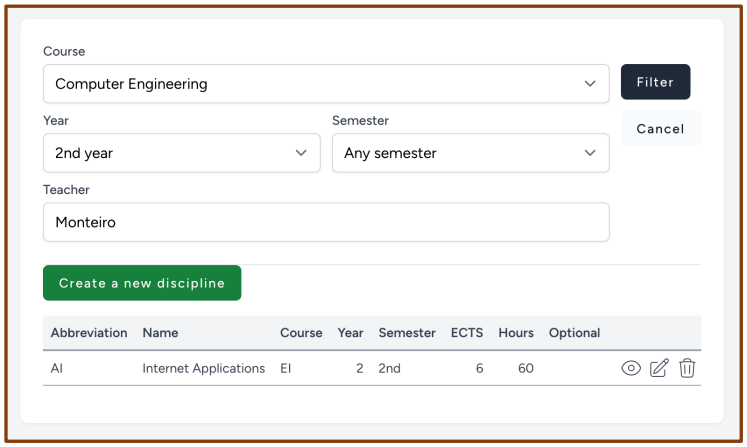
```
    public function index(Request $request): View
    {
        $filterByCourse = $request->query('course');
        $filterByYear = $request->year;
        $filterBySemester = $request->input('semester');
        $filterByTeacher = $request->teacher;
        $disciplinesQuery = Discipline::query();
        if ($filterByCourse !== null) {
            $disciplinesQuery->where('course', $filterByCourse);
        }
        if ($filterByYear !== null) {
            $disciplinesQuery->where('year', $filterByYear);
        }
```

```
        if ($filterBySemester !== null) {
            $disciplinesQuery->where('semester', $filterBySemester);
        }
        if ($filterByTeacher !== null) {
            $disciplinesQuery->join('teachers_disciplines',
                        'disciplines.id', '=', 'teachers_disciplines.discipline_id')
                ->join('teachers',
                        'teachers_disciplines.teacher_id', '=', 'teachers.id')
                ->join('users', 'teachers.user_id', '=', 'users.id')
                ->where('users.name', 'like', "%$filterByTeacher%")
                ->select('disciplines.*')
                ->distinct();
        }
        $disciplines = $disciplinesQuery->paginate(20);
        return view('disciplines.index', compact('disciplines', 'filterByCourse',
                    'filterByYear', 'filterBySemester', 'filterByTeacher')
        );
    }
```

39. Before testing the query, let's change the filter form so that is maintains the state of the "teacher" field. Change the section that uses the component `disciplines.filter-card` on the view `disciplines.index` view file (`resources/views/disciplines/index.blade.php`) to:

```
            <x-disciplines.filter-card
                :filterAction="route('disciplines.index')"
                :resetUrl="route('disciplines.index')"
                :courses="$courses->pluck('fullName', 'abbreviation')->toArray()"
                :course="old('course', $filterByCourse)"
                :year="old('year', $filterByYear)"
                :semester="old('semester', $filterBySemester)"
                :teacher="old('teacher', $filterByTeacher)"
                class="mb-6"
                />
```

40. Open the disciplines page and try to filter by: course = "Computer Engineering"; year = "2" and teacher = "Monteiro".



- The results may vary because most of the data on the database is random.

41. Use telescope to analyze the SQL command used by last filter.

Query

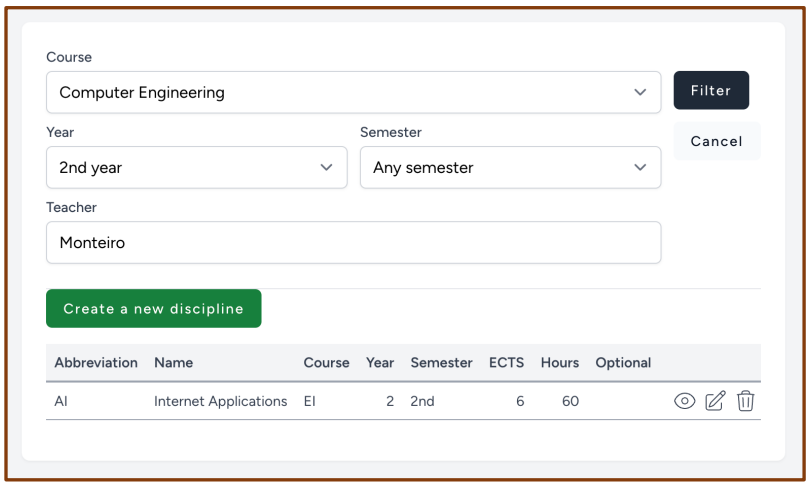
```
select
  distinct `disciplines`.*
from
  `disciplines`
  inner join `teachers_disciplines` on `disciplines`.`id` = `teachers_disciplines`.`discipline_id`
  inner join `teachers` on `teachers_disciplines`.`teacher_id` = `teachers`.`id`
  inner join `users` on `teachers`.`user_id` = `users`.`id`
where
  `course` = 'EI'
  and `year` = '2'
  and `users`.`name` like '%Monteiro%'
limit
  20 offset 0
```

42. Next, we'll implement the second version of the teacher's name filter parameter solution. Instead of a relative complex query with multiple joins, we will add several simple queries, that return sets of IDs. Change the code of the method `index` of the controller `DisciplineController` (file `app/Http/Controllers/DisciplineController.php`):

```
    public function index(Request $request): View
    {
        $filterByCourse = $request->query('course');
        $filterByYear = $request->year;
        $filterBySemester = $request->input('semester');
        $filterByTeacher = $request->teacher;
        $disciplinesQuery = Discipline::query();
        if ($filterByCourse !== null) {
            $disciplinesQuery->where('course', $filterByCourse);
        }
        if ($filterByYear !== null) {
            $disciplinesQuery->where('year', $filterByYear);
        }
```

```
        if ($filterBySemester !== null) {
            $disciplinesQuery->where('semester', $filterBySemester);
        }
        if ($filterByTeacher !== null) {
            $usersIds = DB::table('users')
                ->where('type', 'T')
                ->where('name', 'like', "%$filterByTeacher%")
                ->pluck('id')
                ->toArray();
            $teachersIds = DB::table('teachers')
                ->whereIntegerInRaw('user_id', $usersIds)
                ->pluck('id')
                ->toArray();
            $disciplinesIds = DB::table('teachers_disciplines')
                ->whereIntegerInRaw('teacher_id', $teachersIds)
                ->pluck('discipline_id')
                ->toArray();
            $disciplinesQuery->whereIntegerInRaw('id', $disciplinesIds);
        }
        $disciplines = $disciplinesQuery->paginate(20);
        return view('disciplines.index', compact('disciplines', 'filterByCourse',
                    'filterByYear', 'filterBySemester', 'filterByTeacher')
        );
    }
```

- Try the filter with course = "Computer Engineering"; year = "2" and teacher = "Monteiro". It should return the same result as before.



- Analyze the code. First, we will get the subset of users that are teachers and whose name includes the substring ("Monteiro") – this subset is returned as an array of IDs (`$usersIds`). Then, we get the set of teachers Ids (`$teachersIds`) whose `user_id` is the previous subset. Afterwards, we get the set of disciplines

(`$disciplinesIds`) associated to that set of teachers – the list of disciplines (disciplines Ids) that the original set of users are associated to.

Having the set of disciplines associated to the users that are teachers and whose name includes the substring ("Monteiro"), we just ensure that the remaining query parameters are applied to that subset of disciplines.

- Instead of the method `whereIntegerInRaw` we could have used the method `whereIn`. We've chosen the method `whereIntegerInRaw` because the performance of this method is much better that the method `whereIn`, particularly for large and very large arrays. The method `whereIntegerInRaw` is restricted to using only integer arrays – if that is the case, we recommend to always use `whereIntegerInRaw` instead of `whereIn`.

43. Using Laravel Telescope, analyze all the SQL commands used to achieve the filter by course, year and teacher:
    - First SQL command - returns the set of users (users ids) that are teachers and whose name includes "Monteiro"

```
select `id` from `users`
where `type` = 'T' and `name` like '%Monteiro%'
```

        - Returns the set of users' ids: (2, 22, 23, 30, 113)
    - Second SQL command - returns the set of teachers from the previous set of users.

```
select `id` from `teachers`
where `user_id` in (2, 22, 23, 30, 113)
```

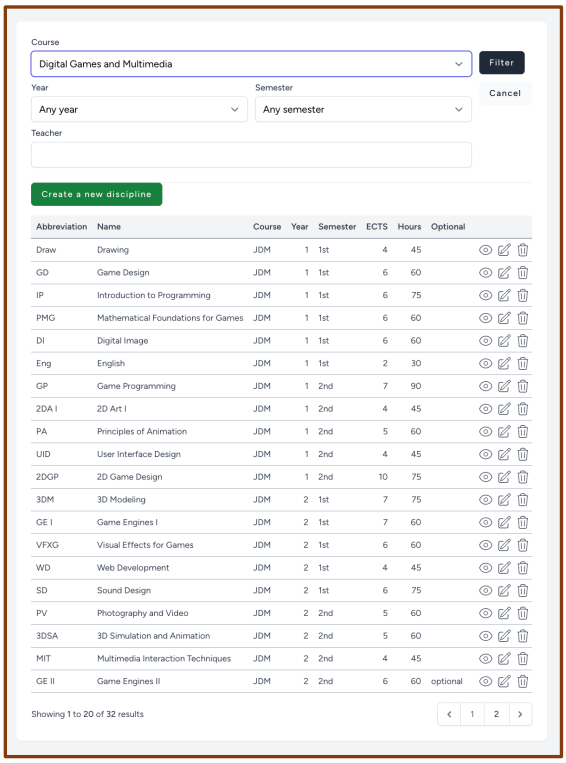
        - returns the set of teachers' ids: (1, 21, 22, 29, 112)
    - Third SQL command – returns the set of disciplines that are associated to previous set of teachers.

```
select `discipline_id` from `teachers_disciplines`
where `teacher_id` in (1, 21, 22, 29, 112)
```

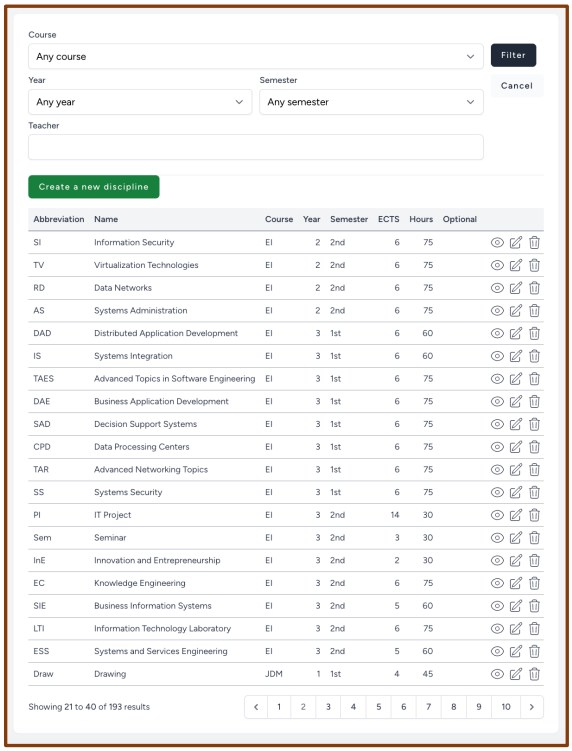
        - returns the set of disciplines' ids: (1, 17, 106, 143, 7, 64, 128, 66, 83, 10, 26, 115, 117, 162, 9)
    - Fourth SQL command – applies the filter parameters (e.g. course = 'EI' and year = 2) to the subset of disciplines returned previously.

```
select * from `disciplines`
where `course` = 'EI' and `year` = '2' and
      `id` in (1, 17, 106, 143, 7, 64, 128, 66, 83, 10, 26, 115, 117, 162, 9)
Limit 20 offset 0
```

44. Disciplines filter by course, year, semester, and teacher work correctly, but one details is missing from the final solution. Try to filter by course only (course = "Digital Games and Multimedia")



45. Previous filter returns 2 pages of data, because " Digital Games and Multimedia " has more than 20 disciplines. Try to jump to the second page.

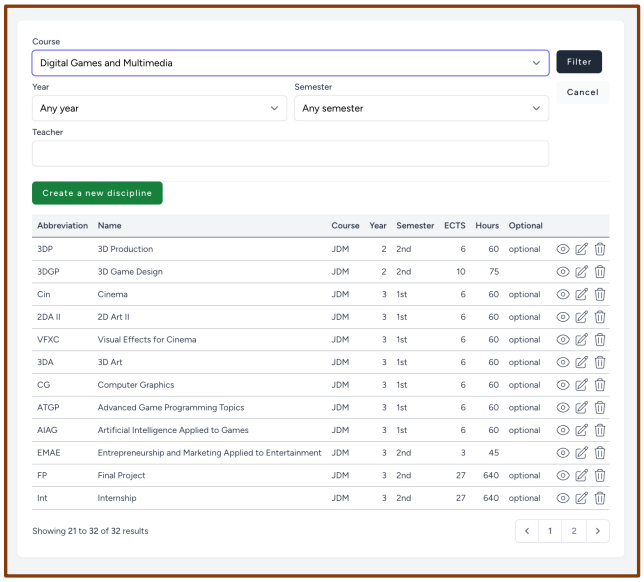


- When we click on the second page, the filter parameters are lost.
- This happens because the link to the second page, does not include the current (when we click on the second page) query string parameters – it only includes the query string parameter "page" – the URL for the second page is "`http://yourdomain/disciplines?page=2`" but it should be: "`http://yourdomain/disciplines?page=2&course=JDM`" – preexisting query string parameters should be maintained.

46. To solve this issue, just add the method "`withQueryString()`" to the `paginate()` method on the controller. Change the code of the "`pagination`" on the method `index` of the controller `DisciplineController` (`app/Http/Controllers/DisciplineController.php`):

```
public function index(Request $request): View
{
    . . .
    $disciplines = $disciplinesQuery->paginate(20)->withQueryString();
    return view( . . . );
}
```

47. Try again. Filter by course only (course = "Digital Games and Multimedia") and jump to the second page. Now, it maintains the filter state while navigating through the pagination navigation bar.



48. A partial resolution is available with the full project up until this exercise (file "`ai-laravel-3.partial.resolution.2.zip`").

3. Tinker

When working with the database, it is important to analyze the SQL commands sent to the database with Laravel Telescope, or similar tool. Another useful tool in this context, and other contexts, is Laravel **Tinker**, a REPL (Read, Evaluate, Print, and Loop) tool that allows us to test Laravel code on the shell without having to implement a route or controller.

49. On the shell (on the root folder of the project), execute the following command:

```
php artisan tinker
```

50. Previous command opens the **tinker shell**. On the tinker shell execute the following commands:

```
use App\Models\Discipline;
$testData = Discipline::where('course', 'EI')->where('year', 2)
            ->where('semester', 2)->get();
```

```
> $testData = Discipline::where('course', 'EI')->where('year', 2)->where('semester', 2)->get();
= Illuminate\Database\Eloquent\Collection {#6210
    all: [
      App\Models\Discipline {#6365
        id: 17,
        course: "EI",
        year: 2,
        semester: 2,
        abbreviation: "AI",
        name: "Internet Applications",
        name_pt: "Aplicações para a Internet",
        ECTS: 6,
        hours: 60,
        optional: 0,
      },
      App\Models\Discipline {#6380
        id: 18,
        course: "EI",
        year: 2,
```

- Executing the command on Laravel Tinker we can confirm that the syntax of the command is valid, and we can visually verify the data returned by the expression.

51. After previous command, and without closing tinker shell, execute the following commands:

```
$arrayIds = $testData->pluck('id')->toArray();

$finalResult = Discipline::whereIntegerInRaw('id', $arrayIds)
                ->where('name', 'like', '%Internet%')->get();
```

```
> $arrayIds = $testData->pluck('id')->toArray();
= [
    17,
    18,
    19,
    20,
    21,
    22,
    23,
    24,
  ]

> $finalResult = Discipline::whereIntegerInRaw('id', $arrayIds)->where('name', 'like', '%Internet%')->get();
= Illuminate\Database\Eloquent\Collection {#6363
    all: [
      App\Models\Discipline {#6347
        id: 17,
        course: "EI",
        year: 2,
        semester: 2,
        abbreviation: "AI",
        name: "Internet Applications",
        name_pt: "Aplicações para a Internet",
        ECTS: 6,
        hours: 60,
        optional: 0,
      },
    ],
  }
```

- Laravel Tinker maintains the state (variables) between commands. The first command defined the value of `$testData` variable which was used by the second command, which defined the value of `$arrayIds` that was used by the third command, and so on.

52. **Leave** the **tinker shell** by pressing **CTRL + C** or with the command **exit**

4. Relationships

In this section we will create two new Model classes, one for the Teachers and another for the Students. These models will be related (through relationships) to other models. Teacher's model will be related to User, Department and Discipline. Student's model will be related to User, Course and Discipline. Because we're adding relationships, we must modify the associated models also.

53. Create the model Teacher with an artisan command:

```
php artisan make:model Teacher
```

54. Create the model Student with an artisan command:

```
php artisan make:model Student
```

55. Both the Teacher and Student models follow most of Eloquent conventions – they are associated to the "*plural*" tables "`teachers`" and "`students`", the primary key is "id" – an automatic integer. The only convention that both fail to comply is that the table does not include the timestamp fields (`created_at` and `updated_at`). Therefore, the first version of Teacher model (file `app/Models/Teacher.php`) code is:

```php
<?php

namespace App\Models;

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

class Teacher extends Model
{
    use HasFactory;

    public $timestamps = false;

    protected $fillable = ['user_id', 'department', 'office', 'extension' , 'locker'];
}
```

56. The first version of Student model (file `app/Models/Student.php`) code is:

```
Models
<?php

namespace App\Models;

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

class Student extends Model
{
    use HasFactory;

    public $timestamps = false;

    protected $fillable = ['user_id', 'number', 'course'];
}
```

57. The second version of Teacher model will have relationships to the User and Department models. Teacher model (file `app/Models/Teacher.php`) code is:

```
<?php
. . .
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Teacher extends Model
{
    . . .
    public function departmentRef(): BelongsTo
    {
        return $this->belongsTo(Department::class, 'department', 'abbreviation');
    }

    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }
}
```

- The relationship between Department and Teacher (`departmentRef`) is a 1:N relationship. Department has many teachers, and the teacher belongs to a Department.
- The relationship between User and Teacher is a 1:1 relationship. User has one teacher (is a teacher or a student) and the teacher belongs to a User (is a user).

- The user relationship definition (`belongsTo`) only has one argument (instead of 3 arguments) because the primary and foreign key follow the Eloquent Relationships naming conventions – the primary key name is "`id`" and the foreign key name is "`model_id`". Is this particular case, the user primary key is "`id`" and the foreign key is "`user_id`".

58. The second version of the Student model will have relationships to the User and Course models. Student model (file `app/Models/Student.php`) code is:

```
<?php
. . .
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Student extends Model
{
    . . .
    public function courseRef(): BelongsTo
    {
        return $this->belongsTo(Course::class, 'course', 'abbreviation');
    }
    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }
}
```

59. Next, we will add the inverse relationship on the Department model (department has many teachers) - file "`app/models/Department.php`":

```
<?php
. . .
use Illuminate\Database\Eloquent\Relations\HasMany;

class Department extends Model
{
    . . .
    public function teachers(): HasMany
    {
        return $this->hasMany(Teacher::class, 'department', 'abbreviation');
    }
}
```

60. Add the inverse relationship on the Course model (course has many students) - file "`app/Models/Course.php`". Also, add a relationship to the course to get all the disciplines of that course (one course has many disciplines):

```php
<?php
. . .
use Illuminate\Database\Eloquent\Relations\HasMany;

class Course extends Model
{
    . . .
    public function students(): HasMany
    {
        return $this->hasMany(Student::class, 'course', 'abbreviation');
    }

    public function disciplines(): HasMany
    {
        return $this->hasMany(Discipline::class, 'course', 'abbreviation');
    }
}
```

61. Add the inverse relationships on the User model (user has one student and user has one teacher). Also, configure the fillable properties of the user model - file "`app/Models/User.php`":

```php
<?php
. . .
use Illuminate\Database\Eloquent\Relations\HasOne;

class User extends Authenticatable
{
    . . .
    protected $fillable = [
        'name',
        'email',
        'password',
        'admin',
        'type',
        'gender',
        'photo_url'
    ];
```

```
    . . .
    public function teacher(): HasOne
    {
        return $this->hasOne(Teacher::class);
    }

    public function student(): HasOne
    {
        return $this->hasOne(Student::class);
    }
}
```

62. Teacher and Student models are related to the Discipline model through N:M relations, which means that there is a third table on the database that establishes the relation between Teacher and Discipline (table `teachers_disciplines`), and between Student and Discipline (table `students_disciplines`). These tables are called **pivot tables**.
    Let's start by defining the N:M relationship on the Teacher model (teachers have several disciplines, and vice-versa - disciplines have several teachers) - file `app/Models/Teacher.php`:

```
<?php
. . .
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class Teacher extends Model
{
    . . .
    public function disciplines(): BelongsToMany
    {
        return $this->belongsToMany(
            Discipline::class,
            'teachers_disciplines'
        );
    }
}
```

- The second argument of `belongsToMany` method is the name of the pivot table, because the name of that table does not follow the conventions.
- The method `belongsToMany` does not have the third and fourth arguments, because the foreign keys on the pivot table follow the relationships naming conventions.

63. Define the N:M relationship on the Student model (file `app/Models/Student.php`):

```php
<?php
. . .
use Illuminate\Database\Eloquent\Relations\BelongsToMany;

class Student extends Model
{
    . . .
    public function disciplines(): BelongsToMany
    {
        return $this->belongsToMany(
            Discipline::class,
            'students_disciplines',
            'students_id',
            'discipline_id'
        );
    }
}
```

- The second argument of `belongsToMany` method is the name of the **pivot table**.
- The third argument of `belongsToMany` method is the name of the foreign key column on the pivot table, that is related to the current table – by mistake, the name of that column did not follow the relationship naming conventions – it should have been `student_id` and not `students_id`
- The fourth argument of `belongsToMany` method is the name of the foreign key column on the pivot table, that defines the relation to the model you are joining to.
- Check https://laravel.com/docs/eloquent-relationships for more information about naming convention and other details about Eloquent Relationships

64. Define the inverse relationships on the Discipline model (`app/Models/Discipline.php`):

```php
<?php
. . .
use Illuminate\Database\Eloquent\Relations\BelongsToMany;
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Discipline extends Model
{
    . . .
    public function courseRef(): BelongsTo
    {
        return $this->belongsTo(Course::class, 'course', 'abbreviation');
    }
```

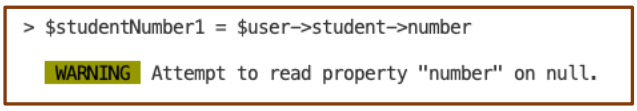
```
    public function teachers(): BelongsToMany
    {
        return $this->belongsToMany(Teacher::class, 'teachers_disciplines');
    }

    public function students(): BelongsToMany
    {
        return $this->belongsToMany(
            Student::class,
            'students_disciplines',
            'discipline_id',
            'students_id'
        );
    }
}
```

65. At this point we have defined all relationships required for our exercises. Let's use Laravel Tinker to verify if the relationships are correctly defined. Here is an example of a Laravel Tinker session (a set of consecutive commands).

```
$department = \App\Models\Department::findOrFail('DEI')

$teachersDei = $department->teachers

$course = \App\Models\Course::find('EI')

$disciplinasEI = $course->disciplines

$studentsEI = $course->students

$user = \App\Models\User::where('name', 'like', '%Marco%Monteiro%')->first()

$teacher = $user->teacher

$teacherName = $teacher->user->name

$deptName = $teacher->departmentRef->name

$teachersOfMyDepartment = $teacher->departmentRef->teachers

$myDisciplines = $teacher->disciplines

$teacher->disciplines->count()

$myFirstDiscipline = $myDisciplines[0]
```

```
$allTeacherOfFirstDiscipline = $myDisciplines[0]->teachers

$myStudentsOfFirstDiscipline = $myDisciplines[0]->students

$myStudentsOfFirstDiscipline->count()

$student = $myDisciplines[0]->students[0]

$student->user->name

$studentDisciplines = $student->disciplines

$arrayStudentDisciplines = $student->disciplines->pluck('name')

$studentColeguesFirstDisc = $student->disciplines[0]->students

$user = \App\Models\User::where('name', 'like', '%Marco%Monteiro%')->first()

$user->name

$student = $user->student

$studentNumber1 = $user->student->number

$studentNumber2 = $user->student?->number
```

- Beware that `$student = $user->student` returns null, because $user is a teacher
- Because `$user->student` is null, the instruction `$user->student->number` is invalid.



- Because `$user->student` is null, we should use the following syntax to read the student number (returns null if student is also null).

```
$user->student?->number
```

- Beware that **some relationships** can return **null values**.

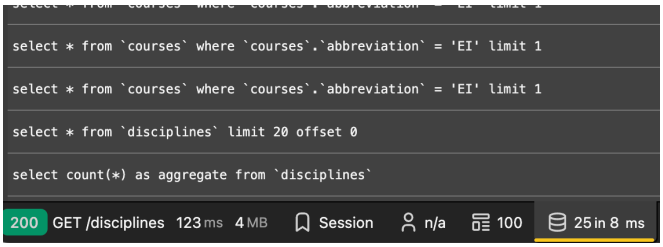
66. Now that we have all relationships established, we can use them easily. Change the table with disciplines so that the "course" column shows the name of the course instead of the

course abbreviation. Edit the code of the file:

`resources/views/components/disciplines/table.blade.pbp`

```
@if($showCourse)
    <td class="px-2 py-2 text-left hidden md:table-cell">
        {{ $discipline->courseRef->name }}
    </td>
@endif
```

67. Open the disciplines page (without applying any filter), and analyze the queries required to build the page (with Laravel Telescope).

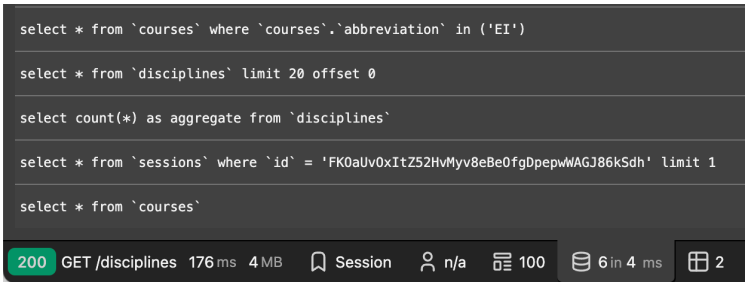


- On our example, 25 SQL commands are executed (some of them are internal Laravel commands to handle sessions). For each row in the table, a SQL command is executed to read the value of the course – every time the relationship `courseRef` is used, Laravel Eloquent generates a SQL command to read the value of the course – this is because by default Eloquent Relationships use a **Lazy Loading** strategy.

68. We'll adapt our controller code so that disciplines will use an **Eager Loading** strategy (instead of a Lazy Loading strategy). Edit the `DisciplineController` controller - file: `app/Http/Controllers/DisciplineController.php`:

```
. . .
$disciplines = $disciplinesQuery->with('courseRef')->paginate(20)->withQueryString();
. . .
```

69. Open the disciplines page again (without applying any filter), and analyze the queries required to build the page (with Laravel Telescope).



- Now, only 6 queries are executed, and only 2 of them are related to the loading of the disciplines:
- `select * from `disciplines`limit 20 offfset 0`
- `Select * from `courses` where `courses`.`abbreviation` in ('EI')`

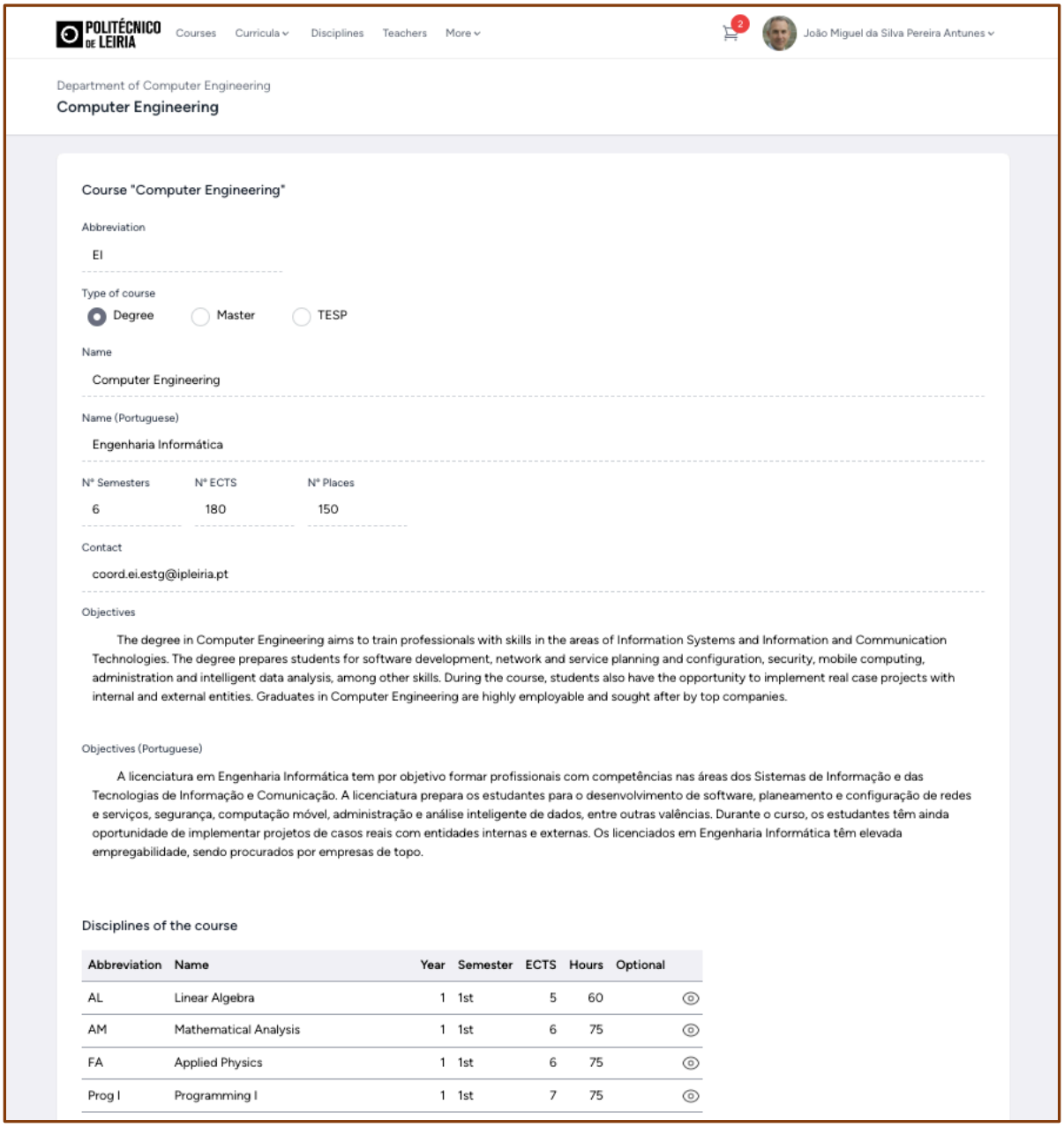
70. Next, we will add the list of disciplines associated to a course, on the page that shows (readonly) a course. Since we have already defined the relationship between the course and disciplines (one course has many disciplines), and we have already created a component to show a table of disciplines, the solution is now extremely simple. Change the code of the view that shows the course (`courses.show`) – file `resources/views/courses/show.blade.php`:

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

@section('header-title', $course->name)

@section('main')
<div class="flex flex-col space-y-6">
    <div class="p-4 sm:p-8 bg-white dark:bg-gray-900 shadow sm:rounded-lg">
        <div class="max-full">
            <section>
                <header>
                    <h2 class="text-lg font-medium text-gray-900 dark:text-gray-100">
                        Course "{{ $course->name }}"
                    </h2>
                </header>
                <div class="mt-6 space-y-4">
                    @include('courses.shared.fields', ['mode' => 'show'])
                </div>
                <h3 class="pt-16 pb-4 text-lg font-medium text-gray-900
                           dark:text-gray-100">
                    Disciplines of the course
                </h3>
                <x-disciplines.table :disciplines="$course->disciplines"
                    :showCourse="false"
                    :showView="true"
                    :showEdit="false"
                    :showDelete="false"
                    />
            </section>
        </div>
    </div>
</div>
@endsection
```

- We've only added a `<h3>` element with a sub-title for the disciplines, and a component `<x-disciplines.table …>`.
- The disciplines passed on to the component are obtained from the "`disciplines`" relationship of the `$course` model:

  `<x-disciplines.table :disciplines="$course->disciplines"`

71. Open the "view" page for the course "Computer Engineering":
    `http://yourdomain/courses/ei`



72. After defining relationships, we can also integrate them on our queries. As an example, we are doing to change the query that filters the disciplines by the name of the teacher. Instead

of using one or more `whereIntegerInRaw` (or `whereIn`) we are integrating the relationship in the query. Edit the `index` method of the `DisciplineController` and **replace the code**:

```
. . .
if ($filterByTeacher !== null) {
    $usersIds = DB::table('users')
        ->where('type', 'T')
        ->where('name', 'like', "%$filterByTeacher%")
        ->pluck('id')
        ->toArray();
    $teachersIds = DB::table('teachers')
        ->whereIntegerInRaw('user_id', $usersIds)
        ->pluck('id')
        ->toArray();
    $disciplinesIds = DB::table('teachers_disciplines')
        ->whereIntegerInRaw('teacher_id', $teachersIds)
        ->pluck('discipline_id')
        ->toArray();
    $disciplinesQuery->whereIntegerInRaw('id', $disciplinesIds);
}
. . .
```

With the following:

```
. . .
if ($filterByTeacher !== null) {
    $disciplinesQuery->with('teachers.user')->whereHas(
        'teachers.user',
        function ($userQuery) use ($filterByTeacher) {
            $userQuery->where('name', 'LIKE', '%' . $filterByTeacher . '%');
        }
    );
}
. . .
```

73. Using Laravel Telescope, analyze the SQL command generated by previous code, by querying the disciplines by the name of the teacher:

Query

```
select
  *
from
  `disciplines`
where
  exists (
    select
      *
    from
      `teachers`
      inner join `teachers_disciplines` on `teachers`.`id` = `teachers_disciplines`.`teacher_id`
    where
      `disciplines`.`id` = `teachers_disciplines`.`discipline_id`
      and exists (
        select
          *
        from
          `users`
        where
          `teachers`.`user_id` = `users`.`id`
          and `name` LIKE '%Monteiro%'
      )
  )
order by
  `year` asc,
  `semester` asc,
  `name` asc
limit
  20 offset 0
```

74. A partial resolution is available with the full project up until this exercise (file "`ai-laravel-3.partial.resolution.4.zip`").

5. 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

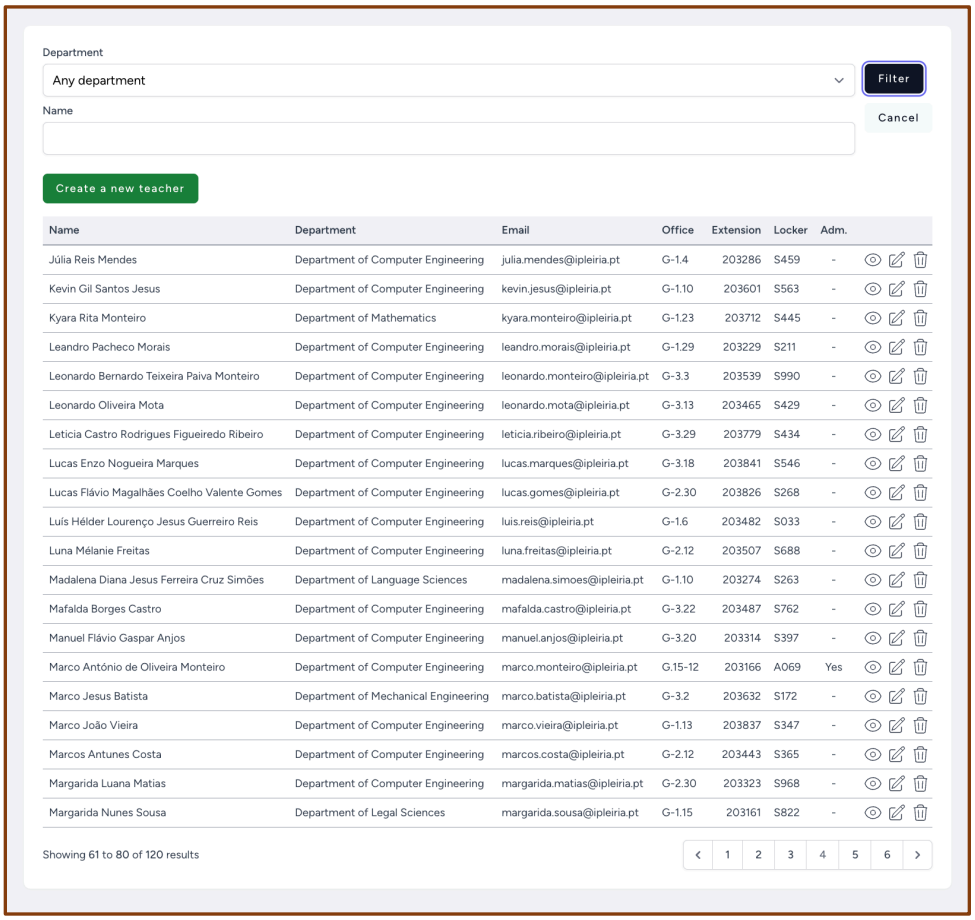
5.1. Teachers, students and administrative

Implement the complete CRUD for the teachers, students and administrative, except the uploading of photos or avatar (to be implemented on next worksheet). Also, ensure that the menu options "`Teachers`", "`More/Students`" and "`More/Administratives`" are linked to list of teachers, students and administrative. Teachers should be filtered by department and substring of name, students by course and substring of name and administrative by a substring of the name.

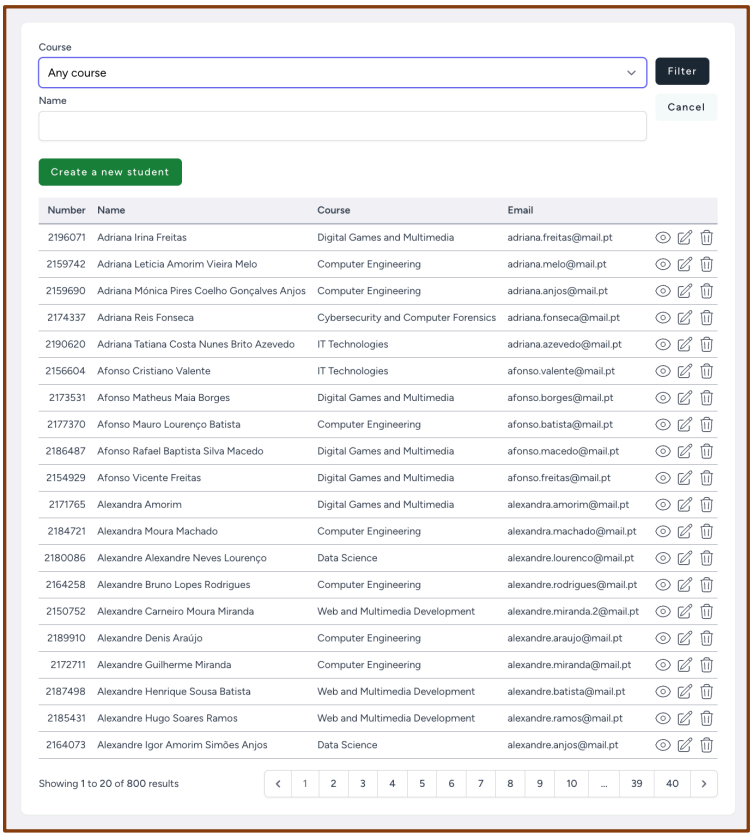
Resulting pages

The final aspect of the list of teachers, students and administrative is very similar to the list of disciplines. The differences are on the filter fields and the columns on the table.

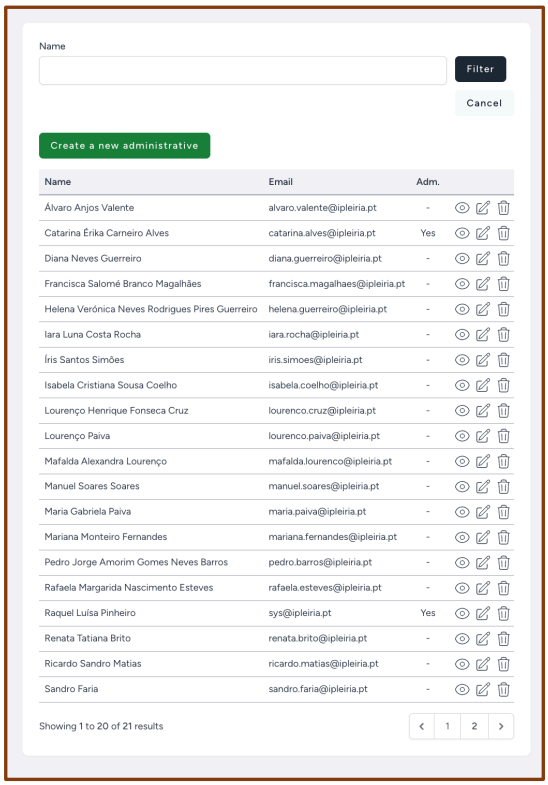
Teachers:



Students:

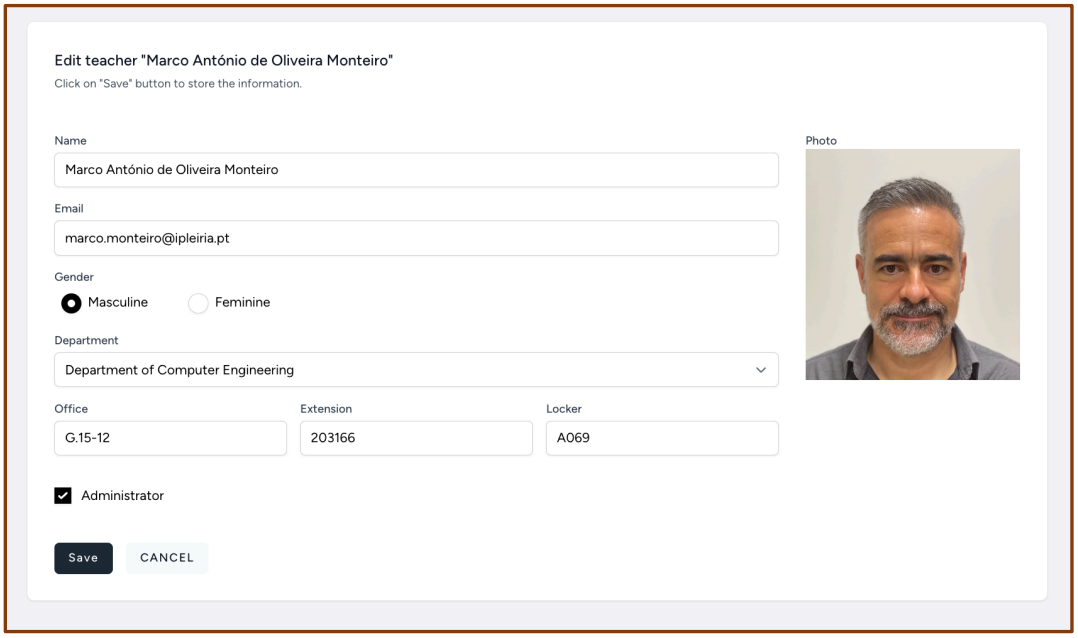


Administrative:

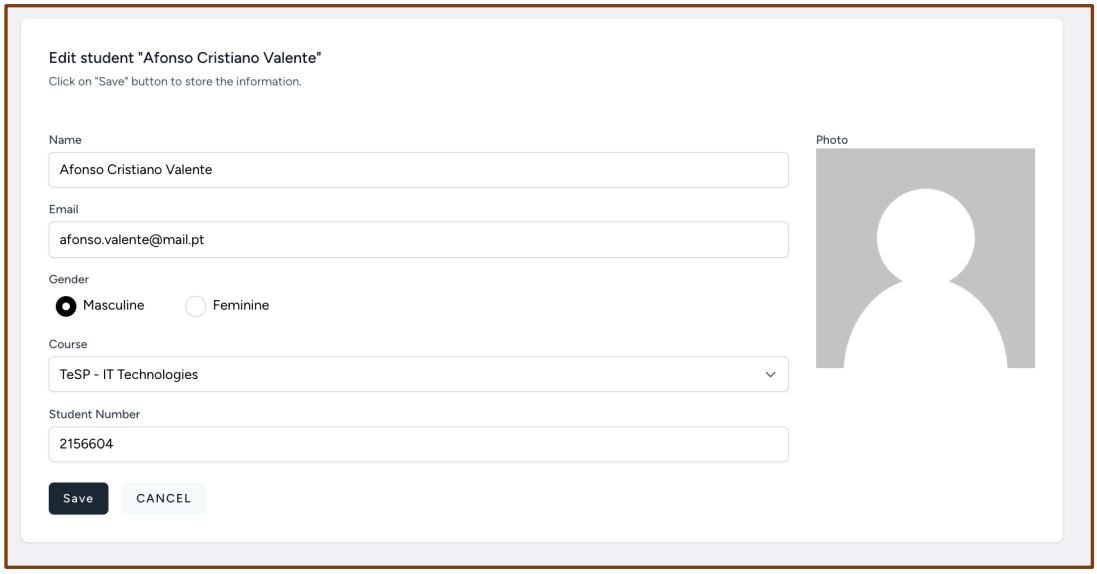


The final aspect of the form to edit or create, as well as view teachers, students and administrative is very similar to the equivalent pages of the disciplines. One major difference is the inclusion of a photo, which can be implemented as a component.

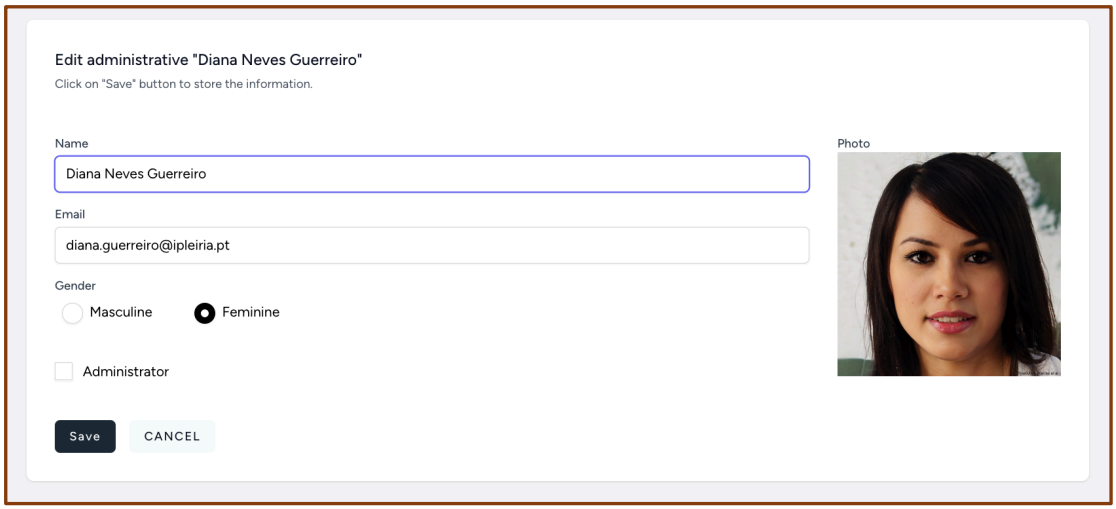
Edit a teacher:



Edit a student:



Edit an administrative:



Validation rules

When validating teachers, students and administrative insert (post request) or update (put request) the following validation rules should be implemented:

Teacher:

- **name** – mandatory string with a maximum size of 255 characters.
- **email** – mandatory, email format, and email must be unique. Note that the email of the teacher is stored on the users table (not on the teachers table)
- **gender** – mandatory, with only 2 valid values: "F" or "M"
- **department** – mandatory, maximum size of 20 characters, must exist on the departments table (column abbreviation)
- **office**– mandatory string with a maximum size of 50 characters.
- **extension**– mandatory string with a maximum size of 20 characters.
- **locker**– mandatory string with a maximum size of 20 characters.
- **admin**– mandatory boolean (teacher is either a administrator or not)

Student:

- **name** – mandatory string with a maximum size of 255 characters.
- **email** – mandatory, email format, and email must be unique. Note that the email of the teacher is stored on the users table (not on the teachers table)
- **gender** – mandatory, with only 2 valid values: "F" or "M"

- **course** – mandatory, maximum size of 20 characters, must exist on the courses table (column abbreviation)
- **number**– mandatory string with a maximum size of 20 characters.

Administrative:

- **name** – mandatory string with a maximum size of 255 characters.
- **email** – mandatory, email format, and email must be unique. Note that the email of the teacher is stored on the users table (not on the teachers table)
- **gender** – mandatory, with only 2 valid values: "F" or "M"
- **admin**– mandatory boolean (teacher is either a administrator or not)

Transactions

A Teacher record (data about one teacher) is stored on 2 tables: "users" table and "teachers" table. This means that the "teachers" table represents a subclass entity of the "users" – a teacher is also a user. When we insert, update or delete a teacher, we have to insert, update or delete rows on the 2 tables: "users" and "teachers".

Any of these operations involve at least 2 SQL commands that affect database data, which means that these operations should use **transactions**.

As an example, we provide the complete code to insert a teacher (action "Store" of the TeacherController):

```
public function store(TeacherFormRequest $request): RedirectResponse
{
    $validatedData = $request->validated();
    $newTeacher = DB::transaction(function () use ($validatedData) {
        $newUser = new User();
        $newUser->type = 'T';
        $newUser->name = $validatedData['name'];
        $newUser->email = $validatedData['email'];
        $newUser->admin = $validatedData['admin'];
        $newUser->gender = $validatedData['gender'];
        // Initial password is always 123
        $newUser->password =bcrypt('123');
        $newUser->save();
        $newTeacher = new Teacher();
        $newTeacher->user_id = $newUser->id;
        $newTeacher->department = $validatedData['department'];
        $newTeacher->office = $validatedData['office'];
        $newTeacher->extension = $validatedData['extension'];
        $newTeacher->locker = $validatedData['locker'];
        $newTeacher->save();
        return $newTeacher;
    });
    $url = route('teachers.show', ['teacher' => $newTeacher]);
    $htmlMessage = "Teacher <a href='$url'><u>{$newTeacher->user->name}</u></a>
                    has been created successfully!";
    return redirect()->route('teachers.index')
        ->with('alert-type', 'success')
        ->with('alert-msg', $htmlMessage);
}
```

Apply the same principle when updating and deleting a Teacher, as well as inserting, updating and deleting a student. Administrative use a single table (table "users"), so we don't need to apply the same principle – only one SQL command is used to insert, update or delete a administrative, which means that we don't have to create a transaction – transactions are automatic when a single SQL command is executed.

Suggestions

- Apply the same design patterns as the ones used by the discipline's entity.
- Create a component called "`Field/Image`" to represent the image field (that shows the photo of the teacher, student or administrative). This component shows an image, and later (on the next worksheet), it will be used to upload the image to the server. Currently, this component can have the following properties (later we will add more properties to support uploading and deleting the file to/from the server):

    - **label** – name of the field label.
    - **imageUrl** – the URL of the image to show.
    - **width** = (default = 'full') – the width of the image – use an approach similar to the other field components.
- When the user (teacher, student or administrative) has no photo, use the image file `storage/app/public/photos/anonymous.png`. Note that the `storage/app` folder and files, were created during the database seeder process.
- User's photos are also available on the same folder: `storage/app/public/photos`. The name of these files (photo image files) is randomized and should be the same as the name on the users table (column `photo_url`). To create a valid URL for these photos, consult the following documentation: https://laravel.com/docs/filesystem#the-public-disk

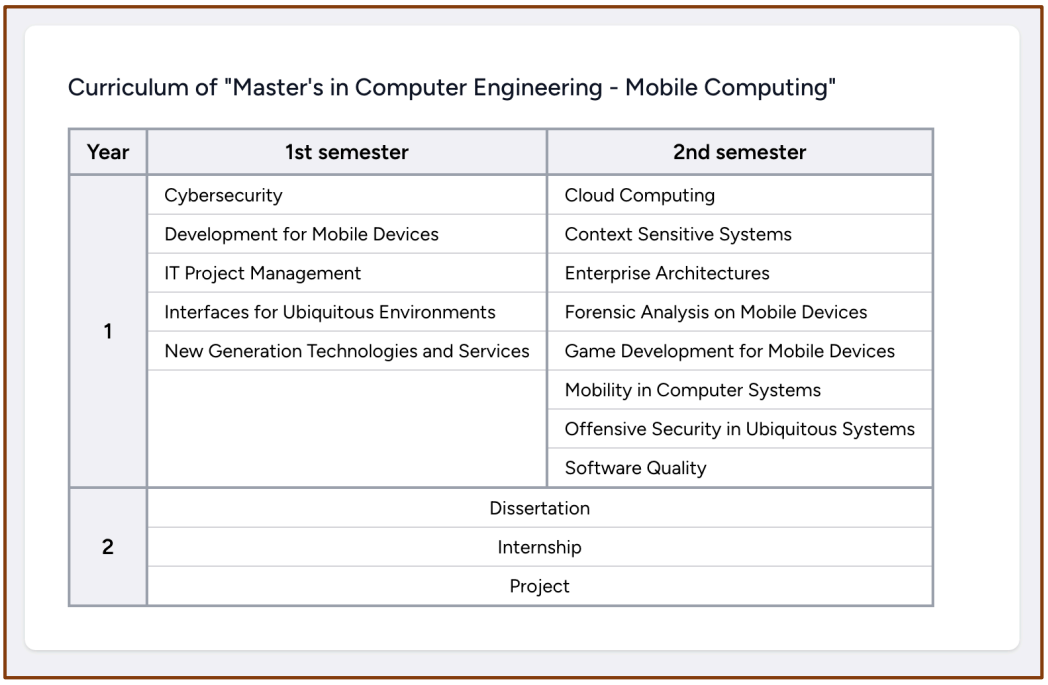
5.2. Curricula

Show the curricula of the courses with a display format that is similar to the one used by worksheet2 (TailwindCSS) and worksheet3 (PHP). Ensure that all sub-menu options of the "`curricula`" main menu are linked to the associated curriculum. Also, the disciplines presented on the curriculum page should be hyperlinks that jump to the page that shows the detail of the corresponding discipline.

Resulting page

The final aspect of each curriculum is similar to the one used by worksheet2 (TailwindCSS) and worksheet3 (PHP). As an example, we can view the curriculum of the course "Master's in Computer Engineering - Mobile Computing". Note that the curriculum integrates annual disciplines, and asymmetric semesters – when a semester has less disciplines than the other one, it adds a cell with rowspan, so that the final vertical space is aligned.

Curriculum of the course "*Master's in Computer Engineering - Mobile Computing*".



Curriculum of "Master's in Computer Engineering - Mobile Computing"

Year	1st semester	2nd semester
1	Cybersecurity	Cloud Computing
	Development for Mobile Devices	Context Sensitive Systems
	IT Project Management	Enterprise Architectures
	Interfaces for Ubiquitous Environments	Forensic Analysis on Mobile Devices
	New Generation Technologies and Services	Game Development for Mobile Devices
		Mobility in Computer Systems
		Offensive Security in Ubiquitous Systems
		Software Quality
2	Dissertation	
	Internship	
	Project	

Suggestions

- Add an action (e.g. "`showCurriculum`") to the existing Course Controller (`CourseController` class)
- Add a route to show the curriculum of a course:

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

- Add a component called "`Courses/Curriculum`", that shows one curriculum. This component has the following property on the constructor (available as an attribute of the component's tag element):
    - **disciplines** – a collection with all disciplines of the course.
- Considering that "disciplines" property is a collection (eloquent collection) already loaded from the database will all disciplines of one course, we can use collection's methods (example: `where`, `sortBy`, `values`, `pluck`, `unique`, etc) to filter/manipulate that data in memory.
- The algorithm to draw a curriculum is not simple, as it requires us to detect and handle asymmetric semesters and disciplines (vertically – the number of disciplines is different on the first and second semester – we must add a rowspan; horizontally – an annual discipline occupies the space of 2 semesters – we must add a

colspan). We could implement that algorithm mainly on the view (replicating the solution for a similar problem on the worksheet 3 (PHP). Instead, we recommend handling the complexity of the algorithm on the component's code and therefore, simplifying the component's view. Convert the given collection of disciplines, to a property called "curriculum" that:

- Is an array of years (key values are 1, 2, 3, etc..).
- Within each year, instead of having an array with 2 semesters or an array with all disciplines, create an array with all rows of a specific year.
- Each of these rows replicates the structure of the HTML table – each row is also an array with 2 elements (the first and second semester) and each of these 2 elements have the following properties: `colspan`; `rowspan` and `discipline` (discipline can be null)

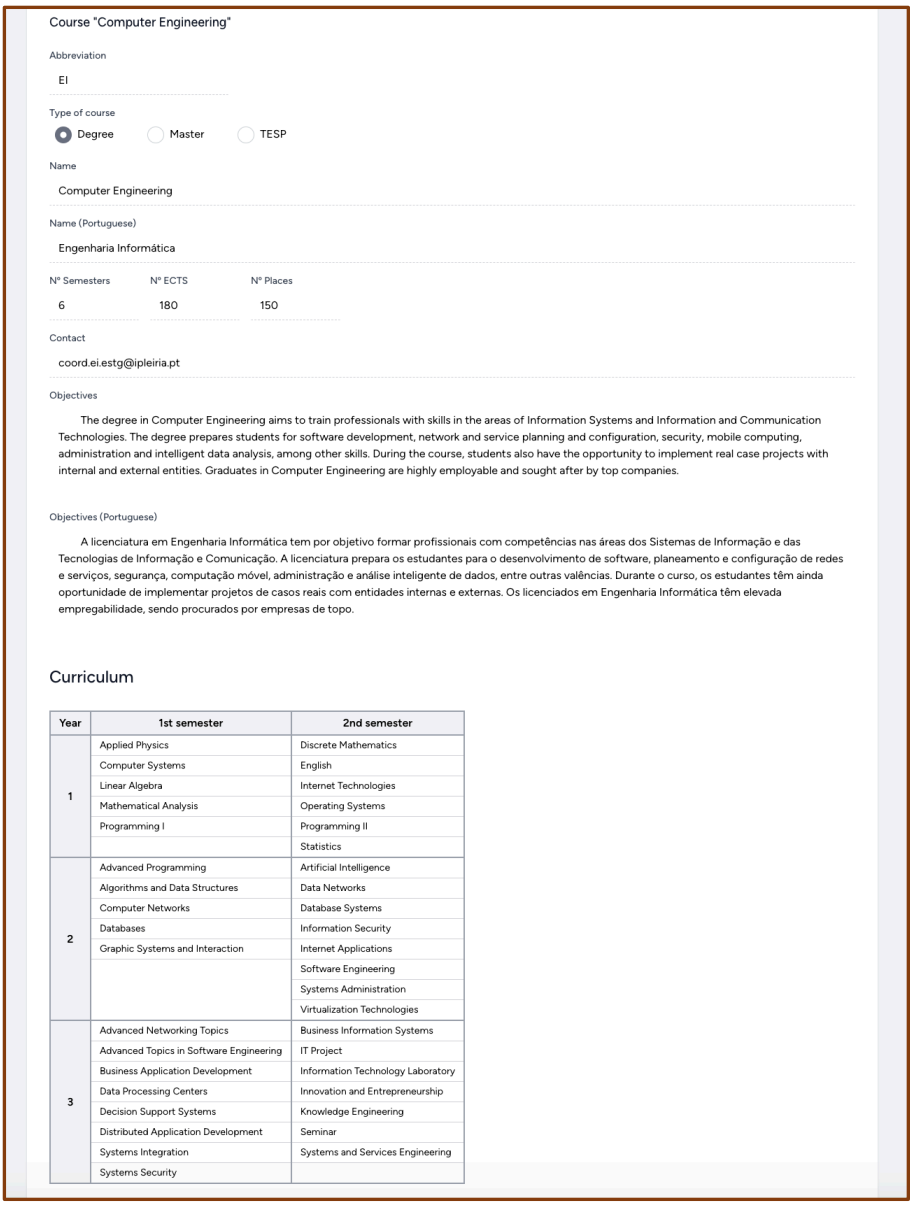
5.3. Related data

After defining model relationships and creating components to represent sets of data (e.g. components that shows tables with disciplines, teachers, students, administrative, courses, curriculum, etc.), it is very simple to represent the same data with multiple variations. For instance, we can view all disciplines (filtered or not), or the disciplines of a given course, or the disciplines taught by a teacher, or the disciplines that a given student is enrolled in, etc.

Implement the following data relations on the UI:

- On the **departments view** page, show all teachers of the department.
- On the **disciplines view** page, show all teachers of the discipline.
- On the **teachers view** page, show all disciplines that he/she teaches.
- On the **students view** page, show all disciplines that he/she is enrolled in.

- On the **courses view** page, replace the list of disciplines, with the curriculum for that course:



Course "Computer Engineering"

Abbreviation

EI

Type of course

Degree Master TESP

Name

Computer Engineering

Name (Portuguese)

Engenharia Informática

Nº Semesters	Nº ECTS	Nº Places
6	180	150

Contact

coord.ei.estg@ipleiria.pt

Objectives

The degree in Computer Engineering aims to train professionals with skills in the areas of Information Systems and Information and Communication Technologies. The degree prepares students for software development, network and service planning and configuration, security, mobile computing, administration and intelligent data analysis, among other skills. During the course, students also have the opportunity to implement real case projects with internal and external entities. Graduates in Computer Engineering are highly employable and sought after by top companies.

Objectives (Portuguese)

A licenciatura em Engenharia Informática tem por objetivo formar profissionais com competências nas áreas dos Sistemas de Informação e das Tecnologias de Informação e Comunicação. A licenciatura prepara os estudantes para o desenvolvimento de software, planeamento e configuração de redes e serviços, segurança, computação móvel, administração e análise inteligente de dados, entre outras valências. Durante o curso, os estudantes têm ainda oportunidade de implementar projetos de casos reais com entidades internas e externas. Os licenciados em Engenharia Informática têm elevada empregabilidade, sendo procurados por empresas de topo.

Curriculum

Year	1st semester	2nd semester
1	Applied Physics	Discrete Mathematics
	Computer Systems	English
	Linear Algebra	Internet Technologies
	Mathematical Analysis	Operating Systems
	Programming I	Programming II
		Statistics
2	Advanced Programming	Artificial Intelligence
	Algorithms and Data Structures	Data Networks
	Computer Networks	Database Systems
	Databases	Information Security
	Graphic Systems and Interaction	Internet Applications
		Software Engineering
		Systems Administration
		Virtualization Technologies
3	Advanced Networking Topics	Business Information Systems
	Advanced Topics in Software Engineering	IT Project
	Business Application Development	Information Technology Laboratory
	Data Processing Centers	Innovation and Entrepreneurship
	Decision Support Systems	Knowledge Engineering
	Distributed Application Development	Seminar
	Systems Integration	Systems and Services Engineering
	Systems Security	

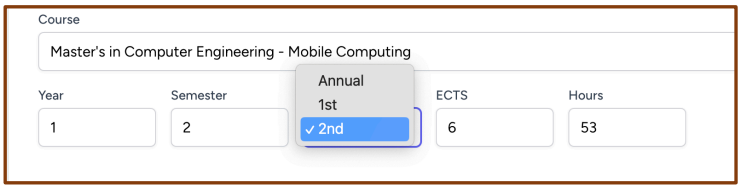
Suggestions

- Use previously built components (e.g. Disciplines/Table and similar) to show the data on the UI
- Use previously built relationships as the source of data for the components (e.g. $teacher->disciplines)

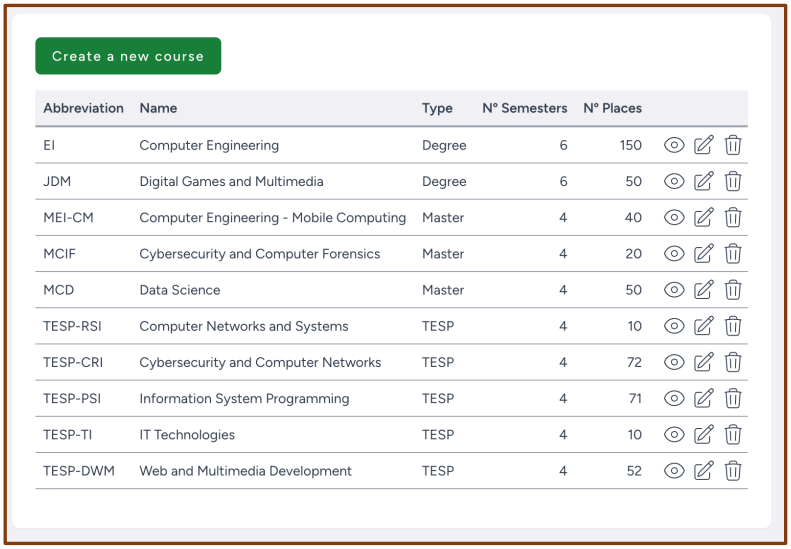
5.4. Final adjustments

To finalize the solution for this worksheet, implement the following adjustments:

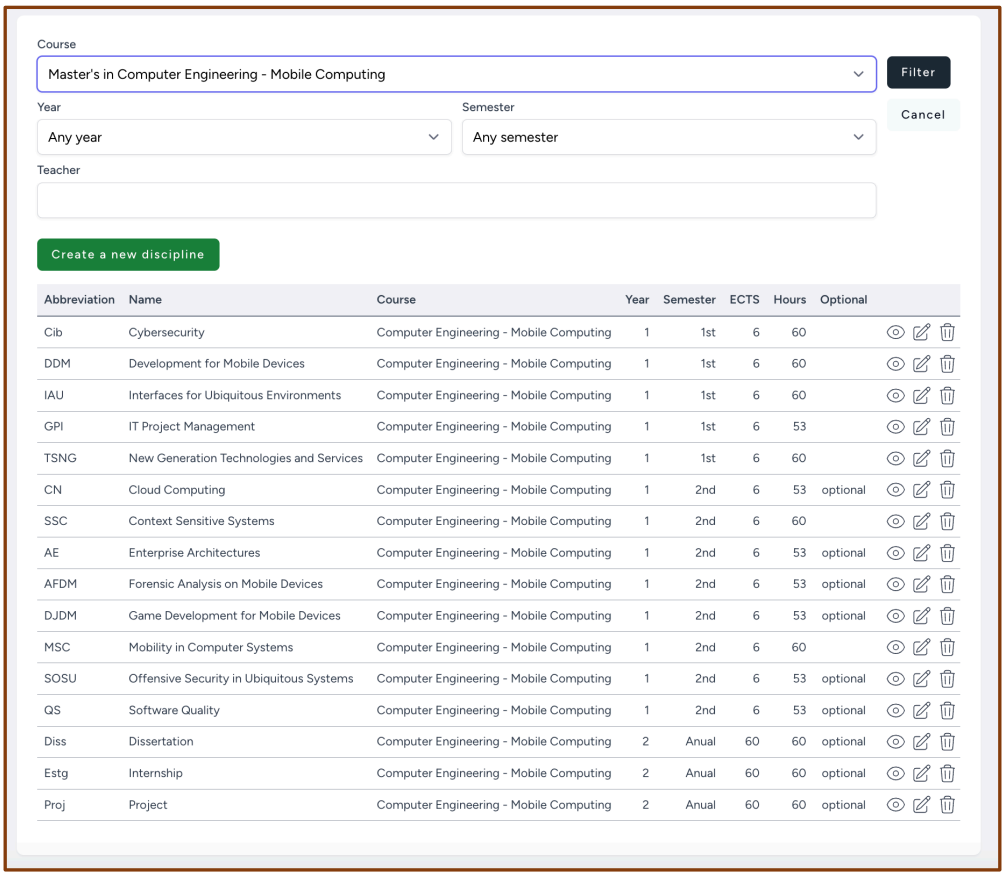
- On the disciplines form (to create, update or view a discipline), add a dropdown select (component `x-select`) for the semester (possible values are 0 => Annual; 1 => 1st; 2 = >2nd.



- Ensure that all list of courses use the full name (e.g. "master's in data science")
- Order all lists (on tables, dropdown select elements, etc) of courses by type and then by name.



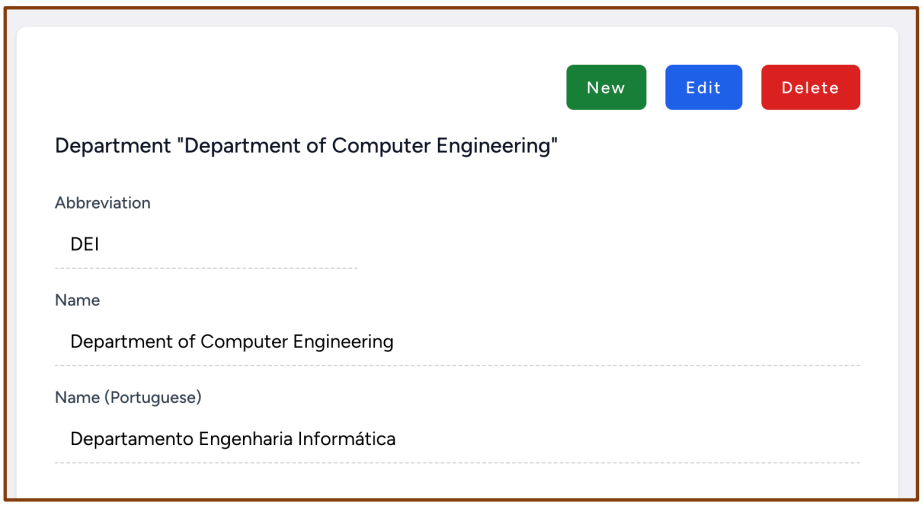
- Order all lists of disciplines (on tables, dropdown select elements, etc) by year, semester and name:



Abbreviation	Name	Course	Year	Semester	ECTS	Hours	Optional
Cib	Cybersecurity	Computer Engineering - Mobile Computing	1	1st	6	60	
DDM	Development for Mobile Devices	Computer Engineering - Mobile Computing	1	1st	6	60	
IAU	Interfaces for Ubiquitous Environments	Computer Engineering - Mobile Computing	1	1st	6	60	
GPI	IT Project Management	Computer Engineering - Mobile Computing	1	1st	6	53	
TSNG	New Generation Technologies and Services	Computer Engineering - Mobile Computing	1	1st	6	60	
CN	Cloud Computing	Computer Engineering - Mobile Computing	1	2nd	6	53	optional
SSC	Context Sensitive Systems	Computer Engineering - Mobile Computing	1	2nd	6	60	
AE	Enterprise Architectures	Computer Engineering - Mobile Computing	1	2nd	6	53	optional
AFDM	Forensic Analysis on Mobile Devices	Computer Engineering - Mobile Computing	1	2nd	6	53	optional
DJDM	Game Development for Mobile Devices	Computer Engineering - Mobile Computing	1	2nd	6	53	optional
MSC	Mobility in Computer Systems	Computer Engineering - Mobile Computing	1	2nd	6	60	
SOSU	Offensive Security in Ubiquitous Systems	Computer Engineering - Mobile Computing	1	2nd	6	53	optional
QS	Software Quality	Computer Engineering - Mobile Computing	1	2nd	6	53	optional
Diss	Dissertation	Computer Engineering - Mobile Computing	2	Anual	60	60	optional
Estg	Internship	Computer Engineering - Mobile Computing	2	Anual	60	60	optional
Proj	Project	Computer Engineering - Mobile Computing	2	Anual	60	60	optional

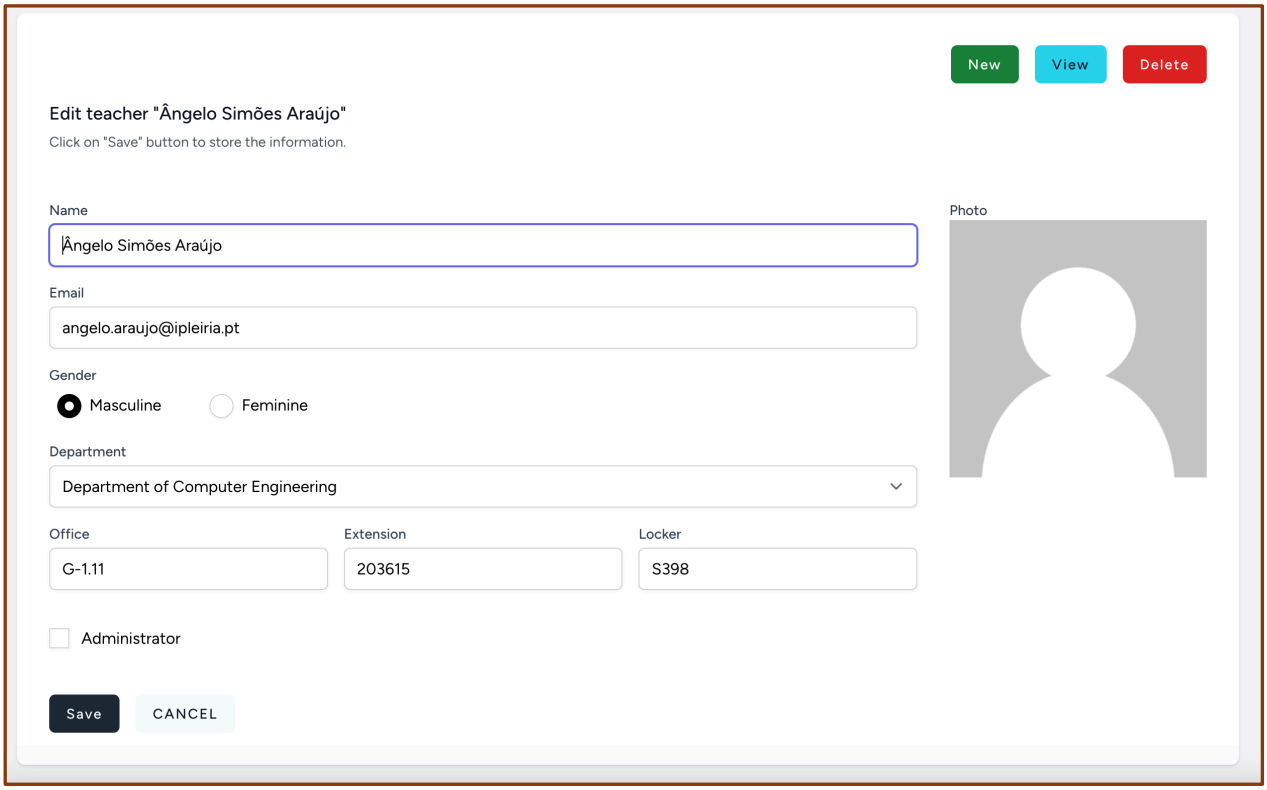
- Order all lists of departments (on tables, dropdown select elements, etc) by name:
- Order all lists of teachers, students and administrative (on tables, dropdown select elements, etc) by name
- On **all view** pages (view discipline, view course, etc...) add 3 buttons to the top right of the "card"
    - **New** – hyperlink to the page to insert a new model/row.
    - **Edit** – hyperlink to the page to update the current model/row.
    - **Delete**– a submit button that deletes the current model/row.

Example for the Department view page:



- On **all edit** pages (edit discipline, view course, etc...) add 3 buttons to the top right of the "card"
    - **New** – hyperlink to the page to insert a new model/row.
    - **View** – hyperlink to the page to view (read-only) the current model/row.
    - **Delete**– a submit button that deletes the current model/row.

Example for the Teacher edit page:



Suggestions

- To implement orders for the courses, disciplines, departments and administrative, add one or more `orderBy` method to the query builder. Example:

```
$allCourses = Course::orderBy('type')->orderBy('name')->paginate(20);
```

- The implementation of orders for the teacher and students is more complex, because we are ordering the results using a field (name) that is not directly available on the table entity – e.g. when ordering teachers by name, the table associated to the model is the table "teachers" and the column we want to order from is on the table "users".
  For these cases, a simple solution is to add a `join` method so that we have access to all fields of teachers and users table. Example:

```
$teachersQuery
    ->join('users', 'users.id', '=', 'teachers.user_id')
    ->select('teachers.*')
    ->orderBy('users.name');
```

- The method `select` of previous code (`select ('teacher.*')`), ensures that only the fields of the teachers table are returned by the query – this avoids potential problems with conflicting column names.

Summary

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

Eloquent models and database

- Eloquent queries with "query builder"
- Methods to manipulate the query builder
- Methods to retrieve data (paginate, get, first, pluck and chunck)
- Joins and column selection
- Filtering with WhereIn or WhereIntegerRaw
- Eloquent relationships (1:N, 1:1 and N:M)
- Relationships "lazy loading" versus "eager loading"
- Eloquent queries with relationships
- Database transactions (autonomous work)
- Insert and update with model instances (autonomous work)
- Analyzing database queries with Laravel Telescope

Request parameters with Query String

Pagination and URL query Strings

Related detailed information – reusage of components and partial views

Eloquent collections and base collections

Tinker tool