APLICAÇÕES PARA A INTERNET

Engenharia Informática

Marco Monteiro

7 - Laravel - 4

Objectives:

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

(2) Upload files

(3) Sessions (cart)

Note the following:

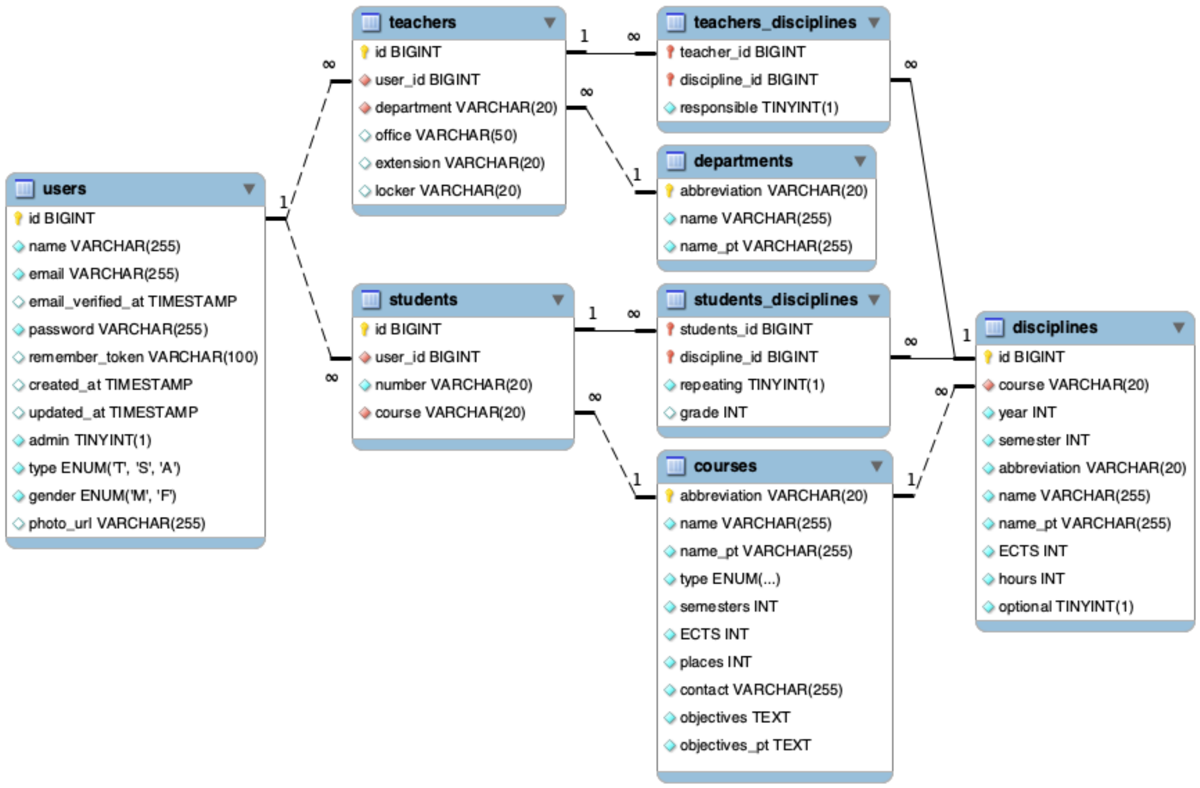
- Before starting the exercise, read the related content on Moodle;
- During the resolution of the exercises, consult the Laravel documentation (https://laravel.com) and other online resources.

Scenery

This worksheet will continue the development of the Web Application from the last worksheet where we've used eloquent queries and relationships, and other database related features and tools (Laravel Telescope and Laravel Tinker) to enhance the application functionalities. In the current worksheet we will learn how to upload files to the server's storage and how to use sessions, by creating a shopping cart.

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-4.zip`) into the Laragon root folder and decompress it on that folder.

2. Previous command will create the folder `ai-laravel-4`, 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-4`. For example, `C:\laragon\www\ai-laravel-4` or `D:\ainet\ai-laravel-4` (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-4`), 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-4.test`" URL to access the content.
9. Test CRUD operations for courses (`http://ai-laravel-4.test/courses`) and disciplines (`http://ai-laravel-4.test/disciplines`)

1.2. New Laravel Project – with Laravel Sail

10. Copy the provided zip file (`start.ai-laravel-4.zip`) into any folder and decompress it.

11. Previous command will create the folder `ai-laravel-4`, that will be the current worksheet project folder.
12. Execute the following command on the project folder (`ai-laravel-4`), 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-4.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-4` 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-4.zip`) into any folder and decompress it.
20. Previous command will create the folder `ai-laravel-4`, 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-4`) 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
```

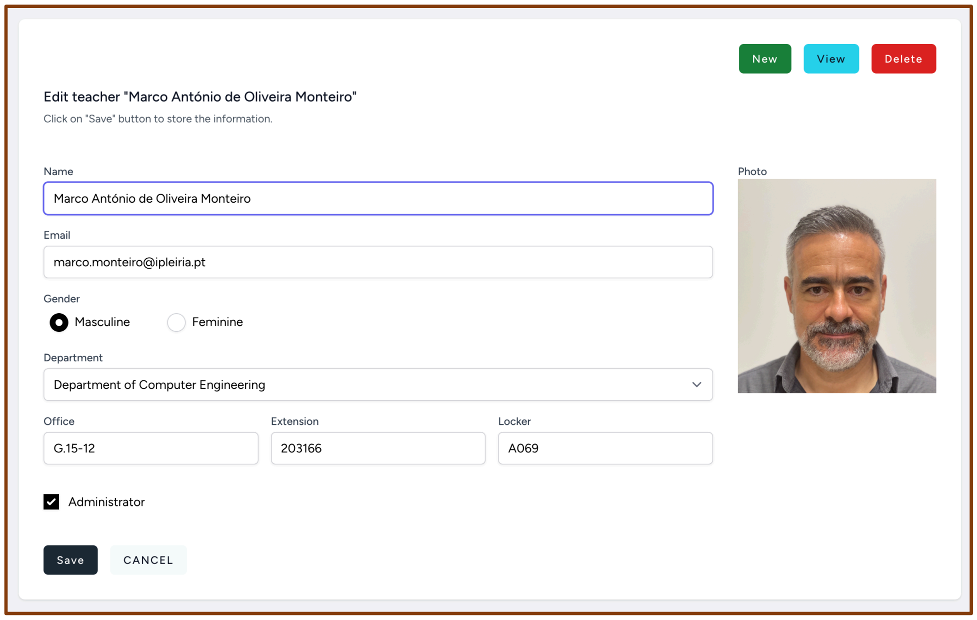
- Test CRUD operations for courses (`http://localhost/courses`) and disciplines (`http://localhost/disciplines`)
- Test "`adminer`" tool for database administration: (`http://localhost:8080`)

2. Storage

Laravel integrates the Flysystem PHP package which provides simple drivers for working with local filesystems, SFTP, and Amazon S3 for storing and accessing files. This package abstracts the storage, allowing us to use the same code for any type of storage and to change the type of storage just by modifying the configuration.

Check https://laravel.com/docs/filesystem for more information about Laravel file storage.

Currently, the photos of teachers, students and administrative are shown on the edit and view pages of these entities. Also, the image associated to a course is presented on the showcase page for the courses.



Edit teacher "Marco António de Oliveira Monteiro"
Click on "Save" button to store the information.

Name: Marco António de Oliveira Monteiro
Email: marco.monteiro@ipleiria.pt
Gender: [✓ Masculine] Feminine
Department: Department of Computer Engineering
Office: G.15-12
Extension: 203166
Locker: A069
Administrator ✓
Photo

25. Let's analyze the code to understand how these images are shown on the application.

26. Open the file "`resources/views/teachers/shared/fields.blade.php`". This file includes all the fields for the teacher's forms. The component responsible for showing the teacher's photo is the component `x-field.image`:

```
<x-field.image
    name="photo_file"
    label="Photo"
    width="md"
    readonly
    deleteTitle="Delete Photo"
    :deleteAllow="true"
    :imageUrl="$teacher->user->photoFullUrl"/>
```

27. The url for the teacher's photo is specified by the expression: `$teacher->user->photoFullUrl`. The `photoFullUrl` is a "calculated" field of the User model. Open the model file (`app/Models/User.php`) and check the code for the `photoFullUrl` field:

```
public function getPhotoFullUrlAttribute()
{
    if ($this->photo_url && Storage::exists("public/photos/{$this->photo_url}")) {
        return asset("storage/photos/{$this->photo_url}");
    } else {
        return asset("storage/photos/anonymous.png");
    }
}
```

- `$this->photo_url` - refers to the field `photo_url` on the users table of the database. If this value is null, it means that the `getPhotoFullUrlAttribute` function returns an image url for a generic user (`anonymous.png` file). Also, the same happens when the `photo_url` value does not match an existing file.
- `asset("storage/photos/file_name")` – if the `photo_url` value match an existing file, then `getPhotoFullUrlAttribute` function returns the url of that file using the `asset` function - `asset` function returns the url with the following format:

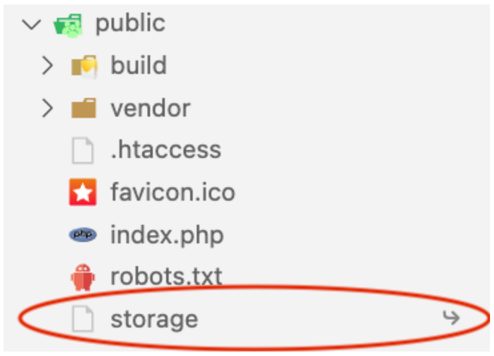
  `asset("`**`path/from/public/folder`**`")` returns the url:

  `http://yourdomain/`**`path/from/public/folder`**

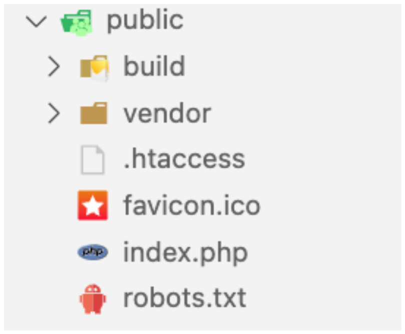
28. This means that the url of the photo (when the teacher has a photo) will be:

  `http://yourdomain/`**`storage/photos/random_file_name.jpg`**

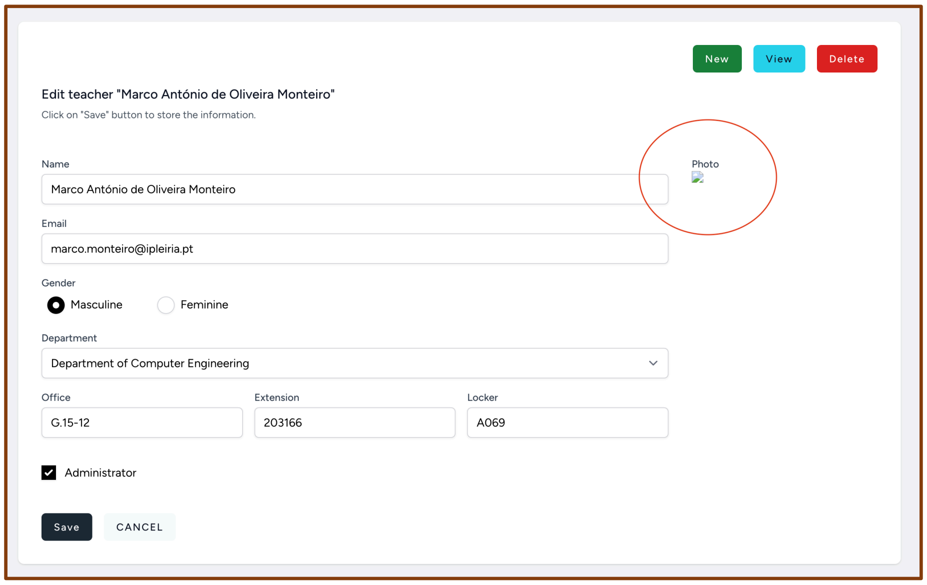
29. However, the photo file is not located on the public folder. Instead, all photo files are located on the folder: `storage/app/public/photos`. The url of the photo only works if we create a **symbolic link**, named "`storage`", on the public folder. This symbolic link is linked to the `storage/app/public` folder, which means that all content (files) located on the folder `storage/app/public` is available on the browser as if they are inside the "storage".



30. To better understand this, delete the symbolic link "storage" of the public folder.



- Now open the page to edit a teacher. The image is no longer available:



Edit teacher "Marco António de Oliveira Monteiro"
Click on "Save" button to store the information.

Name: Marco António de Oliveira Monteiro
Photo
Email: marco.monteiro@ipleiria.pt
Gender: [✓ Masculine] Feminine
Department: Department of Computer Engineering
Office: G.15-12
Extension: 203166
Locker: A069
[✓] Administrator

31. For the image to be available again, we have to recreate the symbolic link with the command:

```
php artisan storage:link
```

32. Now, if we open the edit page again, we can view the photo again. Right click on the photo image and select the option to "Copy the image address" / "Copiar Endereço da Imagem". The URL for the image will be something similar to: `http://yourdomain/storage/photos/random_name.jpg`

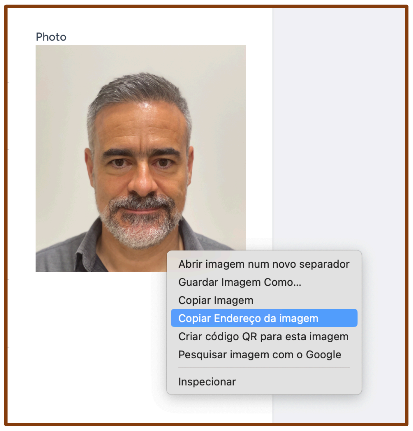
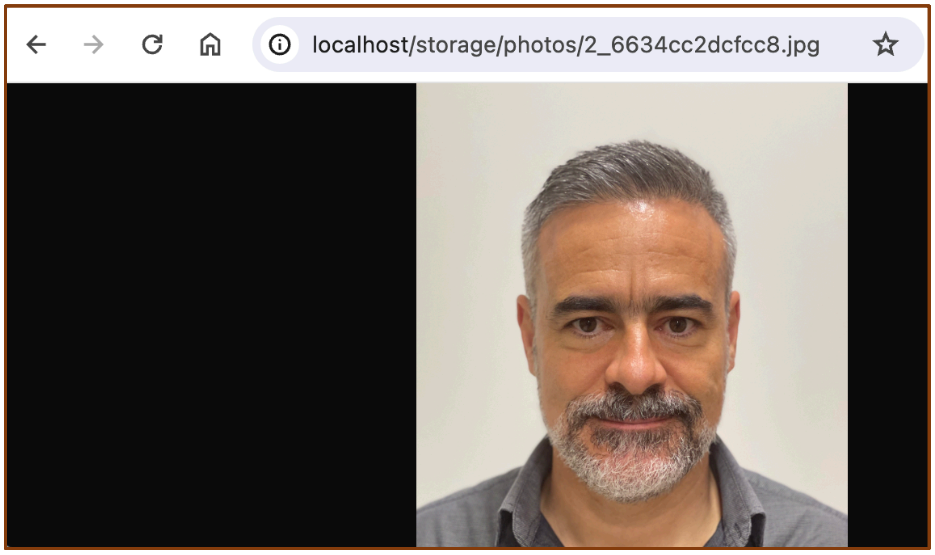


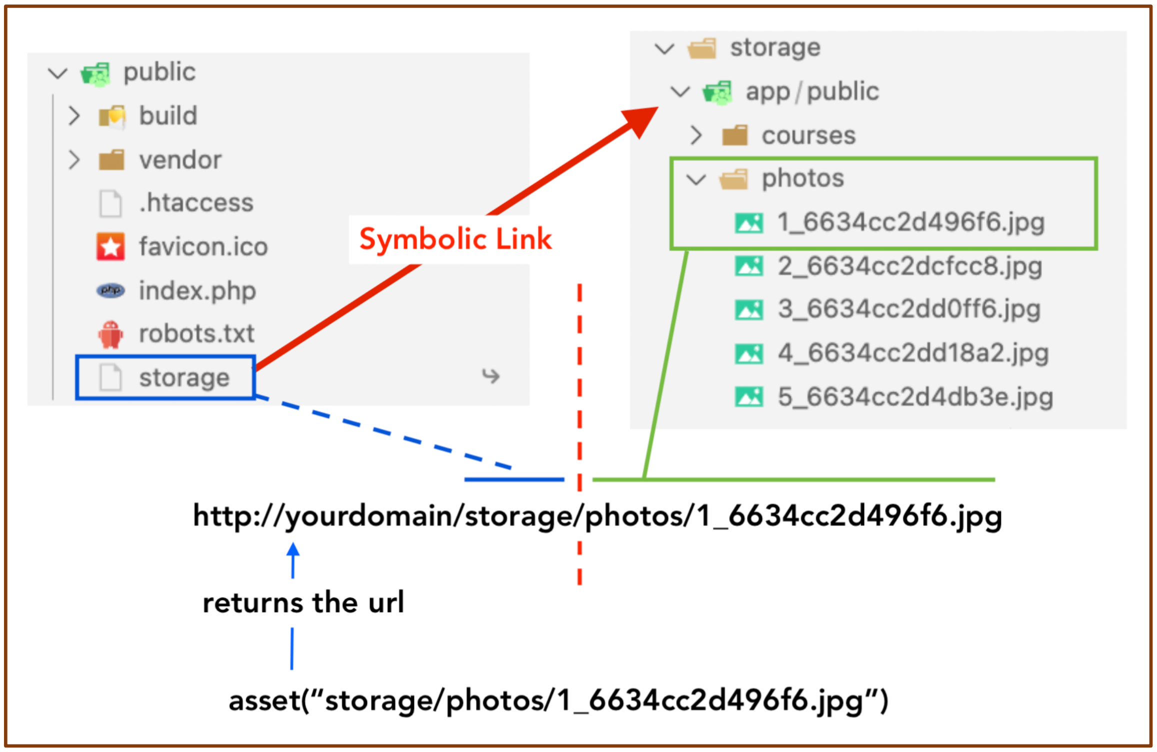
Photo
Abrir imagem num novo separador
Guardar Imagem Como...
Copiar Imagem
Copiar Endereço da imagem
Criar código QR para esta imagem
Pesquisar imagem com o Google
Inspecionar

33. If we open the image url directly on the browser, we have a direct access to the image file.



- In summary, after creating the "`storage`" **symbolic link**, all files and folders inside the folder `storage/app/public` will be available as if they were inside the "`storage`" on the public folder.
- The url of the files inside the folder `storage/app/public` should be obtained by:

```
asset("storage/path_of_file_inside_storage/app/public");
```



34. All folders and files inside the folder `storage/app` are storage files of the Flysystem, which are available to the application – using Laravel code we can create, manipulate, and read folders and files inside the folder `storage/app`. However, any folder or file on the folder `storage/app` that is not inside the folder `storage/app/public`, is not directly available to the browser. These files are private for the application – they can be accessed by Laravel code, but not by the browser, because they are not accessible from the `public` folder.

    Note: Browser can only directly access the content inside the public folder.

35. Check https://laravel.com/docs/filesystem for more information about Laravel Flysystem storage.

3. File Upload

Initial project of the current worksheet already provides access to public storage files (photos and course images) – as long as the storage symbolic link is correctly created.

On this section, we will implement the **file upload** feature (for the teacher) that will allow the end user to upload a photo (or any other type of file) to the server. The application will store the upload file on the storage Flysystem and associate the file with database data. The application will also support removing the file from the storage or replacing a file - the uploaded file will replace an existing older file.

36. First, we have to change the `enctype` attribute of the `<form>` element – to support file uploading the form has to change its `enctype` value to `multipart/form-data`. Edit the `<form>` element of the `teachers.edit` view (file `resources/views/teachers/edit.blade.php`):

```
. . .
<form method="POST" action="{{ route('teachers.update', ['teacher' => $teacher]) }}"
    enctype="multipart/form-data">
    @csrf
    @method('PUT')
    @include('teachers.shared.fields', ['mode' => 'edit'])
    <div class="flex mt-6">
        <x-button element="submit" type="dark" text="Save" class="uppercase"/>
        <x-button element="a" type="light" text="Cancel" class="uppercase ms-4"
                    href="{{ url()->full() }}"/>
    </div>
</form>
. . .
```

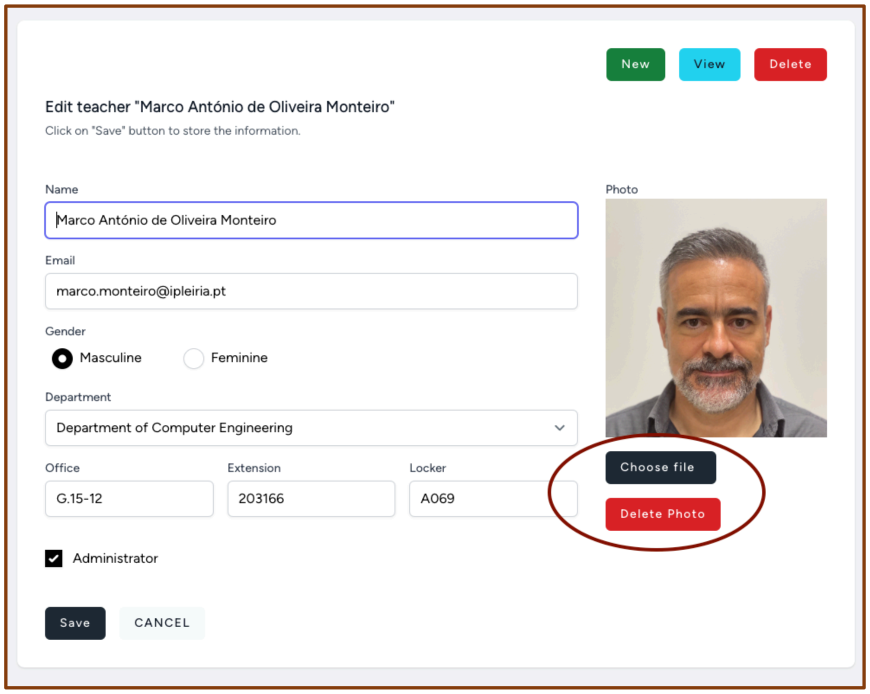
37. Make the same change (`enctype` attribute of the `<form>` element) to the `teachers.create` view (file `resources/views/teachers/create.blade.php`):

```
<form method="POST" action="{{ route('teachers.store') }}"
        enctype="multipart/form-data">
    @csrf
    @include('teachers.shared.fields', ['mode' => 'create'])
    <div class="flex mt-6">
    <x-button element="submit" type="dark" text="Save new teacher" class="uppercase"/>
    </div>
</form>
. . .
```

38. Now, we must ensure that the `<form>` includes at least one `<input type="file" …>` element that will be responsible for specifying which file (if any) from the client computer will be uploaded when submitting the form.
In our example, the `<x-field.image>` component already include that type of field. Change the property `readonly` value (the default value of that property is false) of the `<x-field.image>` component to reflect the value of existing `$readonly` variable. Change the code of the file `resources/views/teachers/shared/fields.blade.php`:

```
. . .
<x-field.image
    name="photo_file"
    label="Photo"
    width="md"
    :readonly="$readonly"
    deleteTitle="Delete Photo"
    :deleteAllow="true"
    :imageUrl="$teacher->user->photoFullUrl"/>
. . .
```

39. Open the teachers edit page (e.g. `http://yourdomain/teachers/1/edit`) and verify that 2 new buttons are available:



Edit teacher "Marco António de Oliveira Monteiro"

Click on "Save" button to store the information.

Name: Marco António de Oliveira Monteiro

Email: marco.monteiro@ipleiria.pt

Gender: [✓ Masculine] Feminine

Department: Department of Computer Engineering

Office: G.15-12

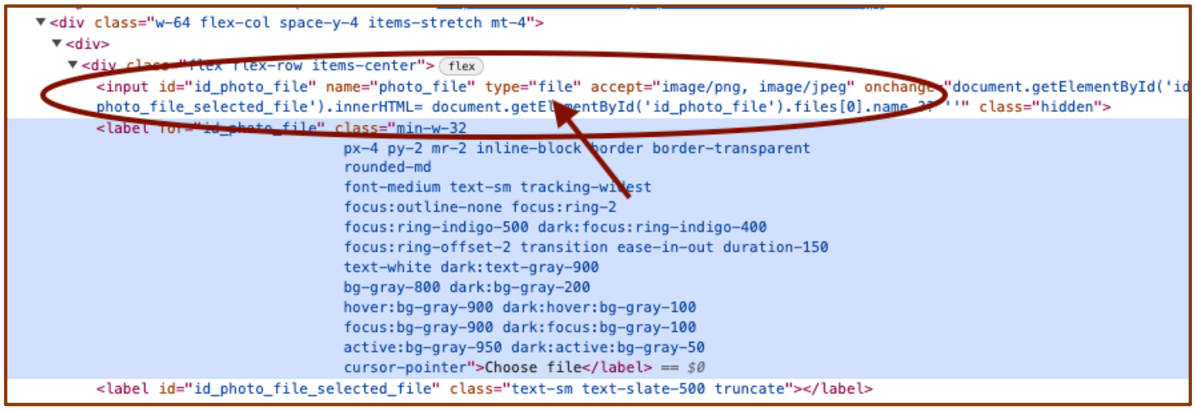
Extension: 203166

Locker: A069

[✓] Administrator

Photo

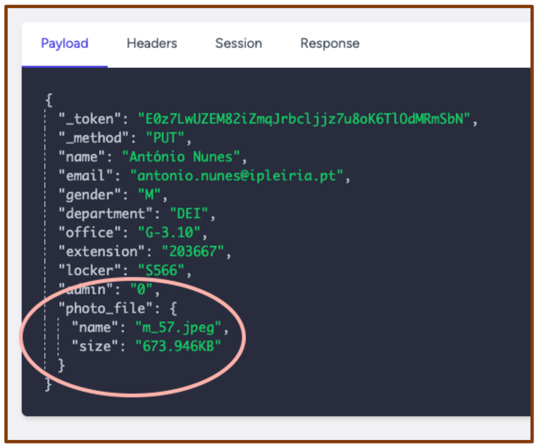
40. To better understand the structure of the `<x-field.image>` component, analyze the HTML produced by inspecting the "button" element "`Choose file`".



```
▼<div class="w-64 flex-col space-y-4 items-stretch mt-4">
  ▼<div>
    ▼<div class="flex flex-row items-center"> flex
      <input id="id_photo_file" name="photo_file" type="file" accept="image/png, image/jpeg" onchange="document.getElementById('id
      photo_file_selected_file').innerHTML= document.getElementById('id_photo_file').files[0].name ?? ''" class="hidden">
      <label for="id_photo_file" class="min-w-32
                            px-4 py-2 mr-2 inline-block border border-transparent
                            rounded-md
                            font-medium text-sm tracking-widest
                            focus:outline-none focus:ring-2
                            focus:ring-indigo-500 dark:focus:ring-indigo-400
                            focus:ring-offset-2 transition ease-in-out duration-150
                            text-white dark:text-gray-900
                            bg-gray-800 dark:bg-gray-200
                            hover:bg-gray-900 dark:hover:bg-gray-100
                            focus:bg-gray-900 dark:focus:bg-gray-100
                            active:bg-gray-950 dark:active:bg-gray-50
                            cursor-pointer">Choose file</label> == $0
      <label id="id_photo_file_selected_file" class="text-sm text-slate-500 truncate"></label>
```

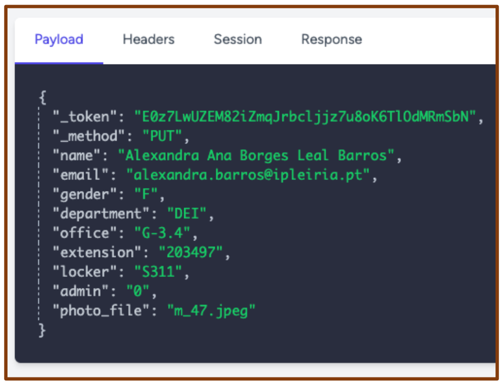
- The "button" "Choose file" is a label that has the visual aspect of a button and is associated to a hidden field.
- The hidden field associated to previous element, is the `<input type="file">` responsible for uploading a file.
- The name of the `<input type="file">` field is the value of the property "`name`" of the `<x-field.image>` component.
- The value of the attribute `accept` of `<input type="file">` field is "`image/png, image/jpeg`", which means that we can only select `png` or `jpeg` image files for that field. Change this attribute (`accept`) or remove it, for the update file to support other types of files.
- JavaScript and CSS guarantee that the `<input type="file">` field is hidden and replaced with a more visually appealing element. However, when submitting the form, it is the `<input type="file">` field value (upload file) that is sent to the server.
- The "Delete" button is a submit button to a form that will be responsible for deleting the photo – note that currently, that form does not exist yet, so the button to "Delete" will not work yet.

41. Analyze the code and design of the `<x-field.image>` component files: `app/View/Components/Field/Image.php` and `resources/Views/components/field/image.blade.php`.
42. On the teachers edit page (e.g. `http://yourdomain/teachers/20/edit`), choose an image file (`png` or `jpeg`) and click on the "`save`" button to submit the form. The image file will be sent to the server, but is not stored yet, as we didn't implement any code on the server to store it.
    Using telescope check the HTTP request and verify the value of payload:



```
{
  "_token": "E0z7LwUZEM82iZmqJrbcljjz7u8oK6TlOdMRmSbN",
  "_method": "PUT",
  "name": "António Nunes",
  "email": "antonio.nunes@ipleiria.pt",
  "gender": "M",
  "department": "DEI",
  "office": "G-3.10",
  "extension": "203667",
  "locker": "S566",
  "admin": "0",
  "photo_file": {
    "name": "m_57.jpeg",
    "size": "673.946KB"
  }
}
```

- The payload includes the field "`photo_file`" that is an object of type `Illuminate\Http\UploadedFile` that represents the uploaded file – telescope only shows the name and size properties.
- If you remove the `enctype` attribute from the `<form>` element, and try to upload a file again, the form payload includes the field "`photo_file`" but its value is only a string with the filename – the file content would not be submitted to the server:



```
{
  "_token": "E0z7LwUZEM82iZmqJrbcljjz7u8oK6Tl0dMRmSbN",
  "_method": "PUT",
  "name": "Alexandra Ana Borges Leal Barros",
  "email": "alexandra.barros@ipleiria.pt",
  "gender": "F",
  "department": "DEI",
  "office": "G-3.4",
  "extension": "203497",
  "locker": "S311",
  "admin": "0",
  "photo_file": "m_47.jpeg"
}
```

43. The form (only for forms with the method post) is prepared to upload the file through the HTTP request. Now, we must program the server to handle the uploaded file. First, let us change the associated Form Request (`TeacherFormRequest`) to include the field with the upload file (field name is `photo_file`). Add the rule for the field `photo_file` on the file `app/Http/Requests/TeacherFormRequest.php` to:

```
public function rules(): array
{
    return [
        . . .
        'photo_file' => 'sometimes|image|max:4096', // maxsize = 4Mb
    ];
}
```

- **sometimes** rule - defines that the field validation is only applied when the field is present on the request – otherwise it is ignored.
- **image** rule - defines that the field must be an image file.
- **max:X** rule - when applied to an upload field, defines the maximum size of the file in Kb – in this example, the maximum size is 4096 Kb (4 Mb).

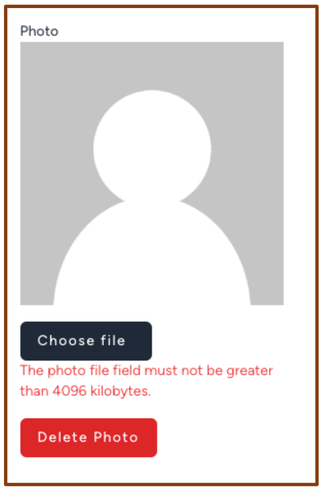
44. Finally, we just have to ensure that the action controller code will store the file correctly. Let's start by changing the `store` action of the `TeacherController` – handles the new teacher (file: `app/Http/Controllers/TeacherController`):

```
    public function store(TeacherFormRequest $request): RedirectResponse
    {
        $validatedData = $request->validated();
        $newTeacher = DB::transaction(function () use ($validatedData, $request) {
            $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'];
            $newTeacher->save();
            // File store is the last thing to execute!
            // files do not rollback, so the probability of having
            // a pending file (not referenced by any teacher)
            // is reduced by being the last operation
            if ($request->hasFile('photo_file')) {
                $path = $request->photo_file->store('public/photos');
                $newUser->photo_url = basename($path);
                $newUser->save();
            }
            return $newTeacher;
        });
        . . .
    }
```

- Note that the field name in the request (`photo_file`) is not the same as the field name on the database table (`photo_url`). This is by design (not an error), as they do not represent the same. The field of the request includes the content of the file, but the field on the database table only includes the name where the file was stored (inside the `storage/app/public/photos` folder)
- Storing the image is the last operation, after all other database code has been successfully completed. This is because the operation to store the file in the storage cannot be rollbacked automatically – if we store the image file in the initial part of the code, and then something goes wrong afterwards, then the database would rollback but the file on the storage would not be rollbacked (the file would be

left on the storage but not associated to any photo on the database). By making it the last operation, after we have guaranteed that all other database operations were successful, we are reducing the probability (although not to zero) of having a "lost"/"pending" file on the storage.

45. Try to create a new teacher with a photo. Everything should be working correctly.
46. Try to upload an image larger than 4Mb when creating a new Teacher. The operation will fail due to a validation error – the error message will be visible next to the "Choose file" button:



The photo file field must not be greater than 4096 kilobytes.

47. Change the `update` action of the `TeacherController` – (file: `app/Http/Controllers/TeacherController`) to handle the update of a teacher:

```
use Illuminate\Support\Facades\Storage;
. . .
public function update(TeacherFormRequest $request, Teacher $teacher):RedirectResponse
{
    $validatedData = $request->validated();
    $teacher = DB::transaction(function () use ($validatedData, $teacher, $request) {
        . . .
        $teacher->user->save();
        if ($request->hasFile('photo_file')) {
            // Delete previous file (if any)
            if ($teacher->user->photo_url &&
                Storage::fileExists('public/photos/' . $teacher->user->photo_url)) {
                    Storage::delete('public/photos/' . $teacher->user->photo_url);
            }
            $path = $request->photo_file->store('public/photos');
            $teacher->user->photo_url = basename($path);
            $teacher->user->save();
        }
        return $teacher;
    });
    . . .
}
```

- The photo of the teacher only changes if the user uploads a file. If no file is uploaded, then we consider that the photo is not to be changed.
- When changing the photo, if a previous file exists, it is deleted.

48. Try to update the photo of a teacher. Everything should be working correctly.
49. One final detail will allow us to delete a photo without having to change it - if a teacher has a photo and wants to remove the photo without adding a new photo. This will be implemented by the button "Delete" of the component `<x-field.image>`.
50. On the `resources/views/teachers/shared/fields.blade.php` change the properties of the component `<x-field.image>`:

```
<x-field.image
    name="photo_file"
    label="Photo"
    width="md"
    :readonly="$readonly"
    deleteTitle="Delete Photo"
    :deleteAllow="($mode == 'edit') && ($teacher->user->photo_url)"
    deleteForm="form_to_delete_photo"
    :imageUrl="$teacher->user->photoFullUrl"/>
```

- The `deleteAllow` property will be true only when the `$mode` is edit and when the teacher has a photo – this means that the `Delete` button will only be visible when we are editing a teacher that has a photo.
- When the user clicks on the `Delete` button, it will submit the form with the id = `form_to_delete_photo` (not yet created)

51. If we open the pages to create a new teacher, edit a teacher or view a teacher, we can observe that the `Delete` button is only visible when editing a teacher with a photo – this makes sense, as we cannot delete a photo of a new teacher (it does not have a photo yet) or when the teacher has no photo.
52. Next, we create a route to delete the photo, with the url "`teachers/{teacher}/photo`" and delete method. Edit the file "`routes/web.php`":

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

- The order of the routes is important. The new route must be defined before `Route::resource('teachers', TeacherController::class);`

53. Create the action `destroyPhoto` on the `TeacherController` (file: `app/Http/Controllers/TeacherController`):

```
public function destroyPhoto(Teacher $teacher): RedirectResponse
{
    if ($teacher->user->photo_url) {
        if (Storage::fileExists('public/photos/' . $teacher->user->photo_url)) {
            Storage::delete('public/photos/' . $teacher->user->photo_url);
        }
        $teacher->user->photo_url = null;
        $teacher->user->save();
    return redirect()->back()
        ->with('alert-type', 'success')
        ->with('alert-msg', "Photo of teacher {$teacher->user->name} has been deleted.");
    }
    return redirect()->back();
}
```

54. To execute the route "`Route::delete('teachers/{teacher}/photo', …)`" we must create a form and then submit it with the "delete" button of the component `<x-field.image>`. This form must have an id = `form_to_delete_photo`. Edit the view `teachers.edit` (file `resources/views/teachers/edit.blade.php`):

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

@section('header-title', 'Teacher "' . $teacher->user->name . '"')

@section('main')
<div class="flex flex-col space-y-6">
    . . .
</div>
```

```
<form class="hidden" id="form_to_delete_photo"
    method="POST"
    action="{{ route('teachers.photo.destroy', ['teacher' => $teacher]) }}">
    @csrf
    @method('DELETE')
</form>
@endsection
```

- The form to delete the photo is hidden and "outside" the remaining content – do not add a form element inside another existing form element.
- The id of the form is "`form_to_delete_photo`" – the id name matches the attribute `form` of the delete button. The HTML for the Delete button is:

```
<button type="submit" form="form_to_delete_photo" class="…" >
      Delete Photo
</button>
```

  This will ensure that when the "Delete" button is clicked, it will submit the form with the specified id.

- Previous technique is important when the submit button is not inside the form we want to invoke (as is the case for the Delete button).

55. Try to delete an existing photo of a teacher. Everything should be working correctly.
56. A partial resolution is available with the full project up until this exercise (file "`ai-laravel-4.partial.resolution.3.zip`").

4. Sessions and Shopping Carts

Most web applications technologies support the concept of sessions, that allows us to maintain data (usually in memory) associated to one session. The session is created (on the server) when the user accesses one of the pages of the web application for the first time, it is maintained (on the server) while the user is accessing pages of the web application and is terminated either explicitly or after a timeout – if the user does not access any page of the application for a predetermined amount of time.

When the session is created, the server creates a session ID and sends it in a cookie to the client (browser). On subsequent HTTP requests, the client (browser) send the session ID cookie to the server, which allows the server to identify the session for that specific request. This allows all requests from the same client (same browser instance) to be recognized as belonging to a specific session.

For each individual session, the web application server will maintain data (a set of session variables) that is specific for that session and is available throughout all pages of the web application – we can think of session variables like "global" variables for that specific session, in a sense that they are accessible on all pages and persist between multiple HTTP requests.

Laravel uses sessions to maintain the information about the authenticated user - Auth::user() -, to maintain the form state - function old(…) – or to implement the "flash data" mechanism. We can also add our own session variables. Check https://laravel.com/docs/session for more details about Laravel sessions.

One of the features that usually use sessions is the "shopping cart" – essential to online e commerce applications. The shopping cart includes a set (array/collection) of products to buy, and it must be maintained through all pages of the web application – that makes it a perfect match for a session variable.

In this section, we will use Laravel sessions to create a feature similar to a "shopping cart". Our project does not require a shopping cart, because it is not an e-commerce, and the database does not have products or orders. Instead, we will allow students to register "disciplinas" using a "shopping cart". Students ("alunos") will browse through the application and add "disciplinas" to the cart (as we would add products to a shopping cart). The cart will have multiple "disciplinas", and then the student can confirm all "disciplinas" registration at once (as we would confirm the order of the products in a shopping cart).

57. First, let's create the `CartController`. Execute the command:

```
php artisan make:controller CartController
```

58. Next, create 5 routes to:

- Add a discipline to the cart.
- Remove a discipline from the cart.
- View the cart.
- Confirm the cart (store the discipline registrations on the database).
- Clear the cart.

59. Add the following routes to the `routes/web.php` file:

```
use App\Http\Controllers\CartController;
. . .
// 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');
```

60. For now, add 2 methods to the `CartController` to handle 2 routes: add a discipline to the cart and show the cart. Edit the file `app/Http/Controllers/CartController.php`:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\View\View;
use Illuminate\Http\RedirectResponse;
use App\Models\Discipline;

class CartController extends Controller
{
    public function show(): View
    {
        $cart = session('cart', null);
        return view('cart.show', compact('cart'));
    }
    public function addToCart(Request $request, Discipline $discipline): RedirectResponse
    {
        $cart = session('cart', null);
        if (!$cart) {
            $cart = collect([$discipline]);
            $request->session()->put('cart', $cart);
        } else {
            if ($cart->firstWhere('id', $discipline->id)) {
                $alertType = 'warning';
                $url = route('disciplines.show', ['discipline' => $discipline]);
                $htmlMessage = "Discipline <a href='$url'>#{$discipline->id}</a>
                <strong>\"{$discipline->name}\"</strong> was not added to the cart
                because it is already there!";
                return back()
                    ->with('alert-msg', $htmlMessage)
                    ->with('alert-type', $alertType);
            } else {
                $cart->push($discipline);
            }
        }
        $alertType = 'success';
        $url = route('disciplines.show', ['discipline' => $discipline]);
        $htmlMessage = "Discipline <a href='$url'>#{$discipline->id}</a>
                <strong>\"{$discipline->name}\"</strong> was added to the cart.";
        return back()
            ->with('alert-msg', $htmlMessage)
            ->with('alert-type', $alertType);
    }
}
```

- To access the session "cart" variable we can use the global helper function `session()` or the `session()` request method.
    - `$cart = session('cart', []);`
    - `$request->session()->put('cart', $cart);`
- The session "cart" variable is a Collection.

61. Before creating the view to show the cart, we will modify the component `disciplines.table` so that it includes 2 new columns: one column to add the discipline to the cart and one column to remove the discipline from the cart. Also, we will create 2 new components to draw 2 new table icons (for the 2 new columns to create on the `disciplines.table`). Execute the following 2 commands:

```
php artisan make:component Table/IconMinus

php artisan make:component Table/IconAddCart
```

62. Both components will have two properties named `action` and `method`. Change the code of the `IconMinus` component class (file `app/View/Components/Table/IconMinus.php`):

```
. . .
class IconMinus extends Component
{
    public function __construct(
        public string $action = '#',
        public string $method = 'DELETE',
    )
    {
        //
    }
    . . .
}
```

63. Change the code of the `IconAddCart` component class (file `app/View/Components/Table/IconAddCart.php`):

```
. . .
class IconAddCart extends Component
{
    public function __construct(
        public string $action = '#',
        public string $method = 'POST',
    )
    {
        //
    }
    . . .
}
```

64. This is the code for the `IconMinus` component design (view) - file `resources/views/components/table/icon-minus.blade.php`:

```
<div {{ $attributes->merge(['class' => 'hover:text-red-600']) }}>
    <form method="POST" action="{{ $action }}"  class="w-6 h-6">
        @csrf
        @if(strtoupper($method) != 'POST')
            @method(strtoupper($method))
        @endif
        <button type="submit" name="minus" class="w-6 h-6">
            <svg  class="hover:stroke-2 w-6 h-6" xmlns="http://www.w3.org/2000/svg"
fill="none" viewBox="0 0 24 24" stroke-width="1" stroke="currentColor">
                <path stroke-linecap="round" stroke-linejoin="round" d="M15 12H9m12
0a9 9 0 1 1-18 0 9 9 0 0 1 18 0Z" />
            </svg>
        </button>
    </form>
</div>
```

65. This is the code for the `IconAddCart` component design (view) - file `resources/views/components/table/icon-add-cart.blade.php`:

```
<div {{ $attributes->merge(['class' => 'hover:text-blue-600']) }}>
    <form method="POST" action="{{ $action }}"  class="w-6 h-6">
        @csrf
        @if(strtoupper($method) != 'POST')
            @method(strtoupper($method))
        @endif
        <button type="submit" name="add_cart" class="w-6 h-6">
            <svg class="hover:stroke-2 w-6 h-6" xmlns="http://www.w3.org/2000/svg"
viewBox="0 0 24 24" fill="currentColor">
                <path d="M2.25 2.25a.75.75 0 0 0 0 1.5h1.386c.17 0
.318.114.362.278l2.558 9.592a3.752 3.752 0 0 0-2.806 3.63c0
.414.336.75.75.75h15.75a.75.75 0 0 0 0-1.5H5.378A2.25 2.25 0 0 1 7.5 15h11.218a.75.75
0 0 0 .674-.421 60.358 60.358 0 0 0 2.96-7.228.75.75 0 0 0-.525-.965A60.864 60.864 0 0
0 5.68 4.509l-.232-.867A1.875 1.875 0 0 0 3.636 2.25H2.25ZM3.75 20.25a1.5 1.5 0 1 1 3
0 1.5 1.5 0 0 1-3 0ZM16.5 20.25a1.5 1.5 0 1 1 3 0 1.5 1.5 0 0 1-3 0Z" />
            </svg>
        </button>
    </form>
</div>
```

66. Change the component `disciplines.table` class code (file: `app/View/Disciplines/Table.php`) to:

```
. . .
    public function __construct(
        public object $disciplines,
        public bool $showCourse = true,
        public bool $showView = true,
        public bool $showEdit = true,
        public bool $showDelete = true,
        public bool $showAddToCart = false,
        public bool $showRemoveFromCart = false,
    )
    {
        //
    }
. . .
```

67. Change the component `disciplines.table` view code (file: `resources/views/components/disciplines/table.blade.php`) to:

```
<div {{ $attributes }}>
    <table class="table-auto border-collapse">
        <thead>
        <tr class=" . . . ">
            . . .
            @if($showDelete)
                <th></th>
            @endif
            @if($showAddToCart)
                <th></th>
            @endif
            @if($showRemoveFromCart)
                <th></th>
            @endif
        </tr>
        </thead>
        <tbody>
        @foreach ($disciplines as $discipline)
            <tr class=" . . . ">
                . . .
                @if($showDelete)
                    . . .
                @endif
                @if($showAddToCart)
                    <td>
                        <x-table.icon-add-cart class="px-0.5"
                            method="post"
                            action="{{ route('cart.add',
                                       ['discipline' => $discipline]) }}"/>
                    </td>
                @endif
                @if($showRemoveFromCart)
                    <td>
                        <x-table.icon-minus class="px-0.5"
                            method="delete"
                            action="{{ route('cart.remove',
                                       ['discipline' => $discipline]) }}"/>
                    </td>
                @endif
            </tr>
        @endforeach
        </tbody>
    </table>
</div>
```

68. Next, we will add the column "addToCart" on the table of disciplines on 3 views: `disciplinas.index`, `teachers.show` and `students.show`. On these 3 views, change the `showAddToCart` property of the `<disciplines.table>` component. Start by changing the file `resources/views/disciplines/index.blade.php`:

```
<x-disciplines.table :disciplines="$disciplines"
    :showCourse="true"
    :showView="true"
    :showEdit="true"
    :showDelete="true"
    :showAddToCart="true"
    />
```

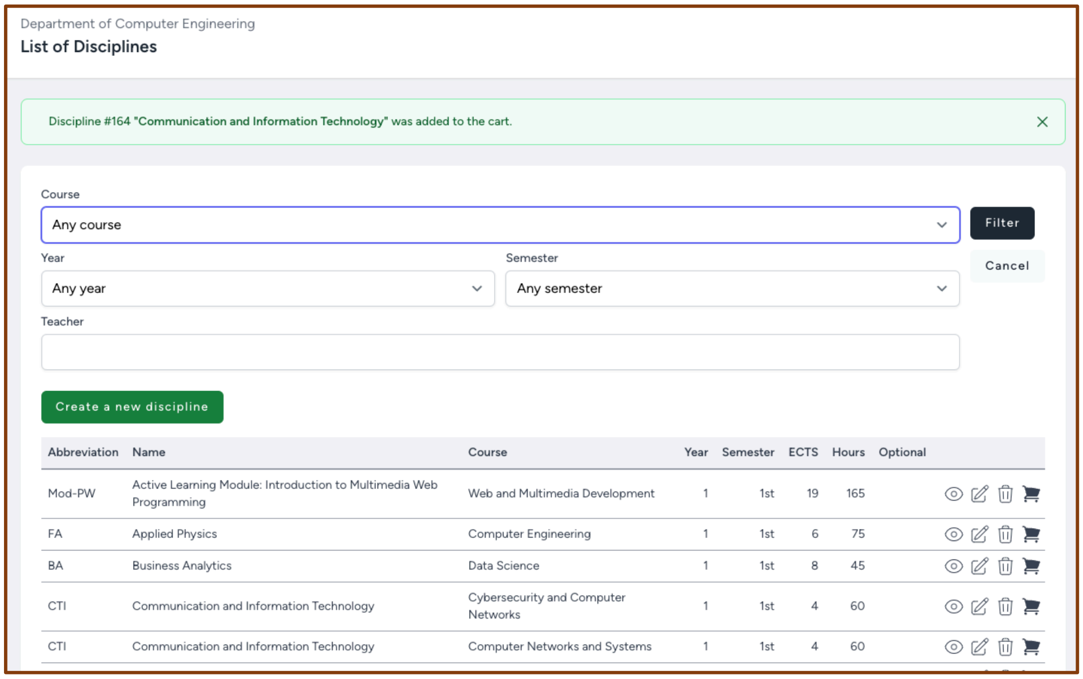
69. Change the file `resources/views/teachers/show.blade.php`:

```
<x-disciplines.table :disciplines="$teacher->disciplines"
    :showView="true"
    :showEdit="false"
    :showDelete="false"
    :showAddToCart="true"
    class="pt-4"
    />
```

70. Change the file `resources/views/students/show.blade.php`:

```
<x-disciplines.table :disciplines="$student->disciplines"
    :showView="true"
    :showEdit="false"
    :showDelete="false"
    :showAddToCart="true"
    class="pt-4"
    />
```

71. Try to open the disciplines page – it must show the icon to add a cart. Try to click on one of these icons (add to cart).



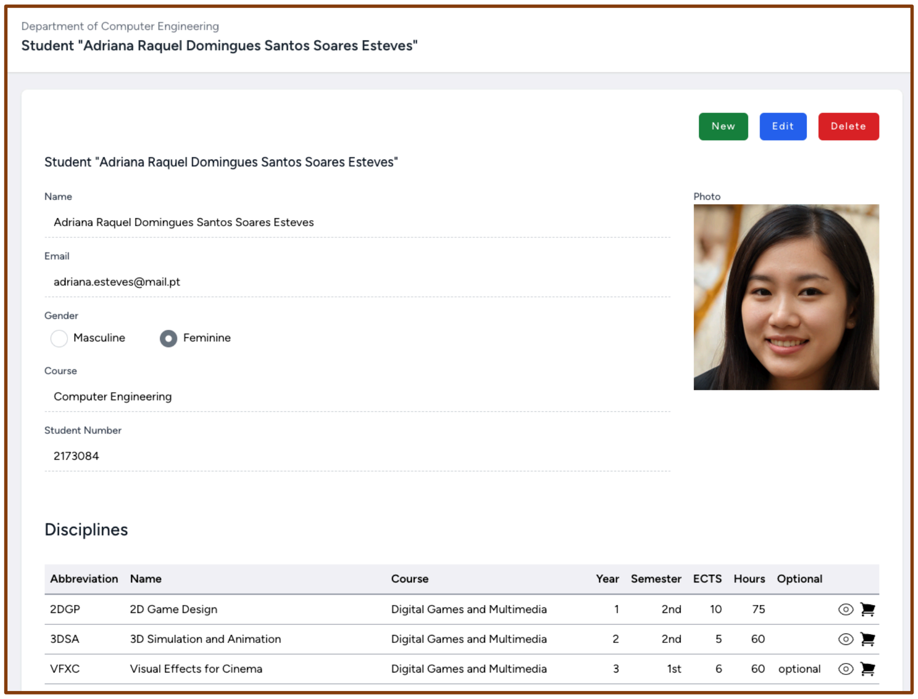
Department of Computer Engineering
List of Disciplines

Discipline #164 "Communication and Information Technology" was added to the cart.

Course: Any course
Year: Any year
Semester: Any semester
Teacher:

Abbreviation	Name	Course	Year	Semester	ECTS	Hours	Optional
Mod-PW	Active Learning Module: Introduction to Multimedia Web Programming	Web and Multimedia Development	1	1st	19	165	
FA	Applied Physics	Computer Engineering	1	1st	6	75	
BA	Business Analytics	Data Science	1	1st	8	45	
CTI	Communication and Information Technology	Cybersecurity and Computer Networks	1	1st	4	60	
CTI	Communication and Information Technology	Computer Networks and Systems	1	1st	4	60	

72. Also, try to open the view page of one teacher and one student. The disciplines associated to them also have a working "add to cart" icon:


Department of Computer Engineering
Student "Adriana Raquel Domingues Santos Soares Esteves"

Student "Adriana Raquel Domingues Santos Soares Esteves"

Name
Adriana Raquel Domingues Santos Soares Esteves

Email
adriana.esteves@mail.pt

Gender
Masculine [✓ Feminine]

Course
Computer Engineering

Student Number
2173084

Photo

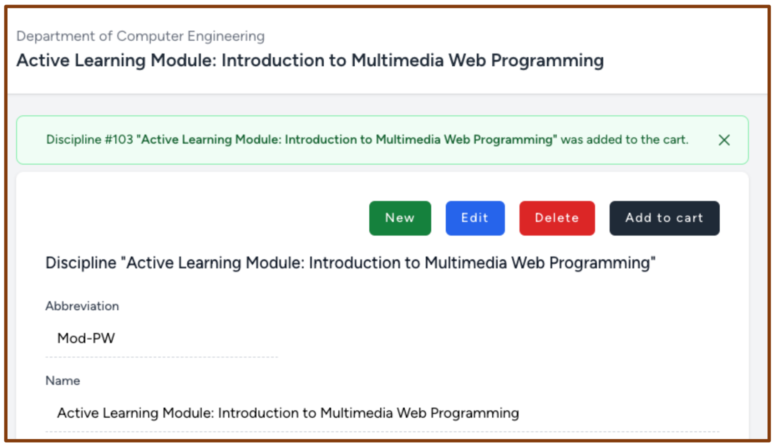
Disciplines

Abbreviation	Name	Course	Year	Semester	ECTS	Hours	Optional
2DGP	2D Game Design	Digital Games and Multimedia	1	2nd	10	75	
3DSA	3D Simulation and Animation	Digital Games and Multimedia	2	2nd	5	60	
VFXC	Visual Effects for Cinema	Digital Games and Multimedia	3	1st	6	60	optional

73. To complement the icon "add to cart" on the disciplines table, let's add a button with the same purpose on the view discipline page. Change the file `resources/views/disciplines/show.blade.php`:

```
  . . .
                <x-button
                    text="New"
                    . . . />
                <x-button
                    text="Edit"
                    . . . />
                <form method="POST"  . . . >
                    . . .
                    <x-button
                        text="Delete"
                        . . . />
                </form>
                <form method="POST"
                  action="{{ route('cart.add', ['discipline' => $discipline]) }}">
                    @csrf
                    <x-button
                        element="submit"
                        text="Add to cart"
                        type="dark"/>
                </form>
                </div>
                <header>
   . . .
@endsection
```

74. Open the view page of one discipline and try to add that discipline to the cart.



Department of Computer Engineering

Active Learning Module: Introduction to Multimedia Web Programming

Discipline #103 "Active Learning Module: Introduction to Multimedia Web Programming" was added to the cart.

Discipline "Active Learning Module: Introduction to Multimedia Web Programming"

Abbreviation

Mod-PW

Name

Active Learning Module: Introduction to Multimedia Web Programming

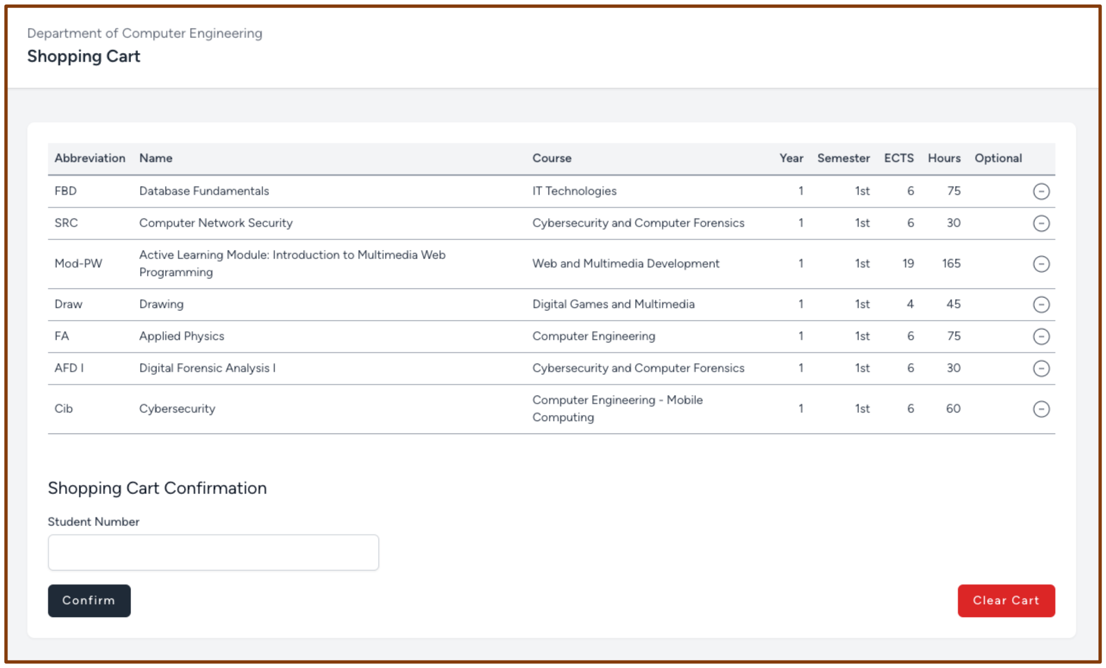
75. Currently, the application includes the action to add disciplines to the cart, from several places within the application. Next step is to draw the view of the cart. Add the folder `cart` to `resources/views` and then create the file `resources/views/cart/show.blade.php` with the following code:

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

@section('header-title', 'Shopping Cart')

@section('main')
    <div class="flex justify-center">
        <div class="my-4 p-6 bg-white dark:bg-gray-900 overflow-hidden
                    shadow-sm sm:rounded-lg text-gray-900 dark:text-gray-50">
            @empty($cart)
                <h3 class="text-xl w-96 text-center">Cart is Empty</h3>
            @else
            <div class="font-base text-sm text-gray-700 dark:text-gray-300">
                <x-disciplines.table :disciplines="$cart"
                    :showView="false"
                    :showEdit="false"
                    :showDelete="false"
                    :showAddCart="false"
                    :showRemoveFromCart="true"
                    />
            </div>
            <div class="mt-12">
                <div class="flex justify-between space-x-12 items-end">
                    <div>
                        <h3 class="mb-4 text-xl">Shopping Cart Confirmation </h3>
                        <form action="{{ route('cart.confirm') }}" method="post">
                            @csrf
                                <x-field.input name="student_number" label="Student Number"
                                               width="lg" :readonly="false"
                                               value="{{ old('student_number') }}"/>
                        <x-button element="submit" type="dark" text="Confirm" class="mt-4"/>
                        </form>
                    </div>
                    <div>
                        <form action="{{ route('cart.destroy') }}" method="post">
                            @csrf
                            @method('DELETE')
                        <x-button element="submit" type="danger" text="Clear Cart" class="mt-4"/>
                        </form>
                    </div>
                </div>
            </div>
            @endempty
        </div>
    </div>
@endsection
```

76. Test the cart page by opening the page `http://yourdomain/cart`:



Department of Computer Engineering

Shopping Cart

Abbreviation	Name	Course	Year	Semester	ECTS	Hours	Optional
FBD	Database Fundamentals	IT Technologies	1	1st	6	75	⊖
SRC	Computer Network Security	Cybersecurity and Computer Forensics	1	1st	6	30	⊖
Mod-PW	Active Learning Module: Introduction to Multimedia Web Programming	Web and Multimedia Development	1	1st	19	165	⊖
Draw	Drawing	Digital Games and Multimedia	1	1st	4	45	⊖
FA	Applied Physics	Computer Engineering	1	1st	6	75	⊖
AFD I	Digital Forensic Analysis I	Cybersecurity and Computer Forensics	1	1st	6	30	⊖
Cib	Cybersecurity	Computer Engineering - Mobile Computing	1	1st	6	60	⊖

Shopping Cart Confirmation

Student Number

- Example of the cart page after adding 7 disciplines to it.

77. The application layout includes, on the menu, a hyperlink with a cart image and the number of items in the cart. We'll ensure that the cart icon is only visible when the cart is not empty, and when it is visible, it show the correct number of items in the cart. Also, the cart image will be a hyperlink that will jump to the cart page.

    Change the code section relative to the **cart**, on the layout view "`layouts.main`" – file `resources/views/layouts/main.blade.php`:

```
<!-- Menu Item: Cart -->
@if (session('cart'))
    <x-menus.cart
        :href="route('cart.show')"
        selectable="1"
        selected="{{ Route::currentRouteName() == 'cart.show'}}"
        :total="session('cart')->count()"/>
@endif
```

78. Before adding the operation to confirm the cart, we'll implement 2 actions, the first will remove a discipline from the cart and the second will clear the cart. The routes for these 2 actions were already created, so we just add the controller's methods. Edit the `CartController` (file `app/Http/Controllers/CartController.php`) and add these 2 methods (`removeFromCart` and `destroy`):

```
public function removeFromCart(Request $request, Discipline $discipline): RedirectResponse
{
    $url = route('disciplines.show', ['discipline' => $discipline]);
    $cart = session('cart', null);
    if (!$cart) {
        $alertType = 'warning';
        $htmlMessage = "Discipline <a href='$url'>#{$discipline->id}</a>
            <strong>\"{$discipline->name}\"</strong> was not removed from the cart
            because cart is empty!";
        return back()
            ->with('alert-msg', $htmlMessage)
            ->with('alert-type', $alertType);
    } else {
        $element = $cart->firstWhere('id', $discipline->id);
        if ($element) {
            $cart->forget($cart->search($element));
            if ($cart->count() == 0) {
                $request->session()->forget('cart');
            }
            $alertType = 'success';
            $htmlMessage = "Discipline <a href='$url'>#{$discipline->id}</a>
            <strong>\"{$discipline->name}\"</strong> was removed from the cart.";
            return back()
                ->with('alert-msg', $htmlMessage)
                ->with('alert-type', $alertType);
        } else {
            $alertType = 'warning';
            $htmlMessage = "Discipline <a href='$url'>#{$discipline->id}</a>
            <strong>\"{$discipline->name}\"</strong> was not removed from the cart
            because cart does not include it!";
            return back()
                ->with('alert-msg', $htmlMessage)
                ->with('alert-type', $alertType);
        }
    }
}

public function destroy(Request $request): RedirectResponse
{
    $request->session()->forget('cart');
    return back()
        ->with('alert-type', 'success')
        ->with('alert-msg', 'Shopping Cart has been cleared');
}
```

79. Open the cart page and try to remove disciplines from the cart as well as clear the cart.

80. Now, the only cart feature missing is the confirmation, where we will add the code to create, for a given student (by inputting the student number), the disciplines registrations (where a student is registered / enrolled to a discipline).

    Later, when the authentication is implemented, we will not ask for the student number – the application will assume the student number of the authenticated student.

81. To guarantee that the confirmation request is valid, first we will create a Form Request to validate the confirmation form – it will guarantee that the student number exists.
    Create the Form Request with the following command:

```
php artisan make:request CartConfirmationFormRequest
```

82. Edit the `CartConfirmationFormRequest` (file `App/Http/Requests/CartConfirmationFormRequest`):

```
. . .
class CartConfirmationFormRequest extends FormRequest
{
    public function authorize(): bool
    {
        return true;
    }

    public function rules(): array
    {
        return [
            'student_number' => 'required|exists:students,number'
        ];
    }
}
```

83. Finally, we will create the action to add the disciplines registrations (where a student is registered / enrolled to a discipline).

    Note that a student can only register a discipline once, so we will ignore any discipline that the student is already registered to.

    Add the following method to the `CartController` (file `app/Http/Controllers/CartController.php`):

```
. . .
use App\Http\Requests\CartConfirmationFormRequest;
use Illuminate\Support\Facades\DB;
use App\Models\Discipline;
use App\Models\Student;
. . .
public function confirm(CartConfirmationFormRequest $request): RedirectResponse
{
    $cart = session('cart', null);
    if (!$cart || ($cart->count() == 0)) {
        return back()
            ->with('alert-type', 'danger')
            ->with('alert-msg', "Cart was not confirmed, because cart is empty!");
    } else {
        $student = Student::where('number', $request->validated()['student_number'])->first();
        if (!$student) {
            return back()
                ->with('alert-type', 'danger')
                ->with('alert-msg', "Student number does not exist on the database!");
        }
```

```php
        $insertDisciplines = [];
        $disciplinesOfStudent = $student->disciplines;
        $ignored = 0;
        foreach ($cart as $discipline) {
            $exist = $disciplinesOfStudent->where('id', $discipline->id)->count();
            if ($exist) {
                $ignored++;
            } else {
                $insertDisciplines[$discipline->id] = [
                    "discipline_id" => $discipline->id,
                    "repeating" => 0,
                    "grade" => null,
                ];
            }
        }
        $ignoredStr = match($ignored) {
            0 => "",
            1 => "<br>(1 discipline was ignored because student was already enrolled in it)",
            default => "<br>($ignored disciplines were ignored because student was already
                        enrolled on them)"
        };
        $totalInserted = count($insertDisciplines);
        $totalInsertedStr = match($totalInserted) {
            0 => "",
            1 => "1 discipline registration was added to the student",
            default => "$totalInserted disciplines registrations were added to the student",

        };
        if ($totalInserted == 0) {
            $request->session()->forget('cart');
            return back()
                ->with('alert-type', 'danger')
                ->with('alert-msg', "No registration was added to the student!$ignoredStr");
        } else {
            DB::transaction(function () use ($student, $insertDisciplines) {
                $student->disciplines()->attach($insertDisciplines);
            });
            $request->session()->forget('cart');
            if ($ignored == 0) {
                return redirect()->route('students.show', ['student' => $student])
                    ->with('alert-type', 'success')
                    ->with('alert-msg', "$totalInsertedStr.");
            } else {
                return redirect()->route('students.show', ['student' => $student])
                    ->with('alert-type', 'warning')
                    ->with('alert-msg', "$totalInsertedStr. $ignoredStr");
            }
        }
    }
}
```

- Analyze the code of the confirm method.
- `$student->disciplines()->attach($insertDisciplines);` code was used to insert multiple rows on the pivot table (`students_disciplines`) of the "`disciplines`" relationship. Check https://laravel.com/docs/eloquent-relationships#inserting-and-updating-related-models for more details.
- After completing the confirmation process (even if no disciplines registrations were added) the cart is cleared.

84. Try to confirm a cart with both a valid and an invalid student number. Everything should be working correctly.
85. A partial resolution is available with the full project up until this exercise (file “`ai-laravel-4.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. Upload

Use the same pattern as the one used to upload a photo of the teacher, to upload photos of students and administratives. Use the same pattern to upload the course image – used by the course showroom.

Also, ensure that when a teacher, student, administrative or course is deleted, the application also deletes all related files.

Suggestions

- Course images are stored on the folder `storage/app/public/courses`
- The name of the course images matches the course's abbreviation.

Summary

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

```
Eloquent models and database


Storage with Flysystem
      Storage folder
      Storage link
      Storage methods do delete and store files
asset() function
File upload
      <input type="file" …>
      enctype "multipart/form-data"
      File upload validation
      Storing files on Storage
basename() function
Sessions
      Shopping Cart
```