Unit 5. Laravel Introduction. Routes

1. Laravel

1.1. The artisan command

When you create a Laravel project, a tool called 'artisan' is installed at the root of the project. It is a command line interface (CLI, Command Line Interface), which provides us a number of additional options to manage our Laravel projects. For example, create controllers, migrate data to a database, etc.

To verify that it is installed and the options it offers, we can write the following command in a terminal from the folder of the project that we have created:

```
docker exec -it laravel-myapp-1 bash
```

```
php artisan list
```

And we will get a list of the options offered by artisan. We will use later some options. To begin with, we can run this command to check the Laravel version of the project we are in:

```
php artisan --version
```

1.2. Structure of a Laravel project

When we create a Laravel project, a predefined folder and file structure is created. We now briefly explain the main folders and files that are generated:

- 'app':contains the source code of the application. Most of the classes we define will be in this folder. Initially, some subfolders are included within:
    - 'Models'
    - 'Http'- contains the drivers and *middleware*
    - 'Providers':contains the service providers of the application, plus those that we can define.
    - In addition, this includes, or can be included, additional folders for our application, such as the 'Events' folder to define the events that occur, or different folders to store the data model or classes of our application.
- 'bootstrap':contains the file 'app.php', which is the one that launches the application. In addition, it contains the 'cache' folder, where the files already uploaded by Laravel are stored to speed up their access in future requests.
- 'config':contains the configuration files of the application, where you have environment variables, or if our application is in development or production, or the connection parameters to the database, among other things. However, the configuration changes are preferable to do in the '.env' file, located at the root of the Laravel project, so that in this last file we will store the private data (user and password of

  the database, for example), and in the corresponding 'config' files we will access these environment variables defined in 'env'.
- 'database':stores the management elements of the database, such as object generators, migrations, etc.
- 'public':visible content of the web. It contains the file 'index.php',entry point of all requests to the web, as well as folders where to locate the static content of the client (images, CSS style sheets, JavaScript files ...).
- 'resources':contains, on the one hand, the views of our application. On the other hand, it also contains not compiled CSS (*sass* files) and JavaScript (unmiplayed files) files. In addition, it also stores the translation files, in case we want to make multi-language sites.
- 'routes':stores the routes of the application 'web.php'
- 'storage':contains the compiled views, and other files generated by Laravel, such as logs or sessions.
- 'tests':to store the tests on the components of our application
- 'vendor':where the dependencies or additional libraries that are required in our Laravel project are stored. Again, this folder should be ignored by Git, and regenerated every time the remote repository is cloned, to avoid storing too much unnecessary information.

Although some of the concepts seen here may not be clear yet, we will see them step by step during the course.

1.3. General project configuration

From the folder structure of a Laravel project seen above, we will now take a quick look at where the overall project settings are located.

On the one hand, we have a '.env' file at the root of the project, which basically contains a series of general environment variables. For example, you have the variable 'APP_NAME' with the name you want the application to have, or a set of variables that we will use later to connect to the database, among other things:

For example, for the project that we have created earlier (library):

```
APP_NAME=library
...
DB_CONNECTION=mysql
DB_HOST=127.21.0.2
DB_PORT=3306
DB_DATABASE=DB_myapp
DB_USERNAME=marta
DB_PASSWORD=
...
```

On the other hand, the 'config' folder contains some general configuration files.

Laravel 11 uses by default sqlite, but we are going to work with mysql. We must change in config/database.php from sqlite to mysql

```
'default' => env('DB_CONNECTION', 'mysql'),
```

2. Routes with Laravel

Routes are a mechanism that allows Laravel to establish what response to send to a request that tries to access a certain URL. These paths are specified in different files within the 'routes' folder of our Laravel project.

We could say that there are two main types of routes (stored in the 'routes/web.php' file of the application):

- The **web** paths, which will allow us to load different views depending on the URL indicated by the client.
- The **API** paths, through which we will define different REST services, as we will also see later.

We are going to focus during this unit on the first group. Let's see what types of routes we can define, and what characteristics they have.

In the file 'routes/web.php', initially there is already a predefined path to the root of the project, which loads the welcome page to it

```php
<?php


use Illuminate\Support\Facades\Route;


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

In addition to using the 'get' method, from the 'Route' class we can also access other useful static methods, such as 'Route::post' (useful for collecting data from forms, for example), or also 'Route::put', 'Route::delete'... We will look at them in more detail in later units.

2.1. Simple routes

Simple paths have a fixed path name, and a function that responds to that name by issuing a response. We can add, for example, a second route through which, if we access the URL *http://localhost:8010/hello* shows us "Hello world" message. We add for this the following route under the previous one that was already defined:

```php
Route::get('hello', function() {
    return "Hello world";
});
```

If we now access *http://localhost:8010/hello*,we should see the message, in plain text.

2.2. Routes with parameters

It is also possible to pass **parameters in the URL** of the path. To do this, we include the name of the parameter in curly braces, and we also pass it to the function of the second parameter. For example, if we define a path to greet the name that comes to us as a parameter, the code would look like this:

```
Route::get('greeting/{name}',  function($name){
return "Hello, "  . $name;
});
```

In this case, if the parameter is mandatory and we do not indicate it in the URL, it will redirect us to a 404 error page. To indicate that a parameter is not mandatory, its name is ended with a question mark, and it is also convenient to give it a default value in the PHP response function. This way we would modify the previous path so that the user's name is optional and, in case of not putting it, the name "Guest" is assigned.

```
Route::get('greeting/{name?}' ,  function($name  =  "Guest"){
return "Hello, "  . $name;
});
```

2.2.1. Parameter validation

Some parameters will need to follow a certain pattern. For example, a numeric identifier will only contain digits. To make sure of that, we can use the 'where' method when defining the route. To this method we pass two parameters: the name of the parameter to be validated, and the regular expression that it has to fulfill. In the case of the previous name, if we want it to contain only letters (uppercase or lowercase), we can do something like this:

```
Route::get('greeting/{name?}' ,  function($name  =  "Guest"){
return "Hello, "  . $name;
})->where('name', "[A-Za-z]+");
```

In case the route does not meet the pattern, an error page will be obtained. We will see how we can customize these error pages later.

2.3. Named routes

Sometimes it may be convenient to associate a name with a route. Especially, when that route is going to be part of a link on a page of our site, since in the future the route could change, and in this way we avoid having to update the links with the new name.

To do this, when defining the route, we associate with the function 'name' the name we want. For example:

```
Route::get('contact', function() {
return "Contact Page";
})->name('path_contact');
```

Now, if we want to define a link to this route anywhere, just use the 'route' function of Laravel, indicating the name that we have assigned to this route. So, instead of putting this:

```
echo '<a href="/contact">contact</a>';
```

We can do something like this, as we will see below when we use the Blade template engine:

```
<a href="{{ route('path_contact') }}">contact</a>
```

In this way, before future changes in the routes, we will only have to change the URL in 'Route::get'.

2.4. Combining items into paths

We can combine several 'where' clauses in a path to validate different parameters it may have. Also combine several 'where' with one 'name' clause. For example, the following path expects to receive a name with characters, and a numeric *id*, both with default values:

```
Route::get('greeting/{name?}/{id?}' ,
function($name="Guest",  $id=0)
{
return "Hello  $name,your code is the $id";
})->where('name', "[A-Za-z]+")
  ->where('id', "[0-9]+")
-> name('greeting');
```

If we access each of the following URLs, we will obtain each of the answers indicated:

URL	Answer
/greeting	*Hello Guest, your code is 0*
/greeting/Nacho	*Hello Nacho, your code is 0*
/greeting/Nacho/3	*Hello Nacho, your code is 3*
/greeting/3	Error 404 (incorrect URL)

Note that the last case is incorrect. We cannot specify an *id* without specifying a name in front, because it violates the URL pattern. An omitted parameter can be left, as long as the subsequent ones are also omitted.