Laravel workshop

In this workshop, you will develop a Laravel website that allows vendors to present their products. Each vendor must be able to register and then login to the website. Once logged in, the vendor can add, modify or delete products in his collection. A product has the following attributes:

products
id
name
price
image
description
category_id
user_id

A published product belongs to a category and to a user/vendor.

categories
id
name

A guest can explore the products presented by the vendors. He can filter the products by vendor, category, price, etc. He can also search for a product by name.

Installing Laravel

If you already have installed Herd, just go to the Herd directory in a terminal (cd ~/Herd).

If you are using a linux OS and you already have PHP, you should get composer: https://getcomposer.org/download/

Then install the Laravel installer via Composer by executing the following instruction in a terminal: composer global require laravel/installer

Creating a new Laravel project

Execute the following instruction in a terminal: laravel new ecommerce.
ecommerce is the name of the project. You can choose another name.

When creating a new project Laravel will ask you if you want to install a starter kit. Choose Laravel Breeze authentication system. It will automatically add an authentication system to your project without having to create all the views and logic required for such system.
Moreover, choose Blade with Alpine as the Breeze stack, no for dark mode and Pest as the testing framework.

Choose MySQL as your database management system then accept to run the default database migrations. You will get an error because the application is not yet configured to connect to the MySQL database.

When the installation is done, go to the project directory (cd ecommerce) and edit the .env file. You must set the following variables to the right values in your system:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=[port_number]
DB_DATABASE=[database_name]
DB_USERNAME=[user_name]
DB_PASSWORD=[user_password]
```

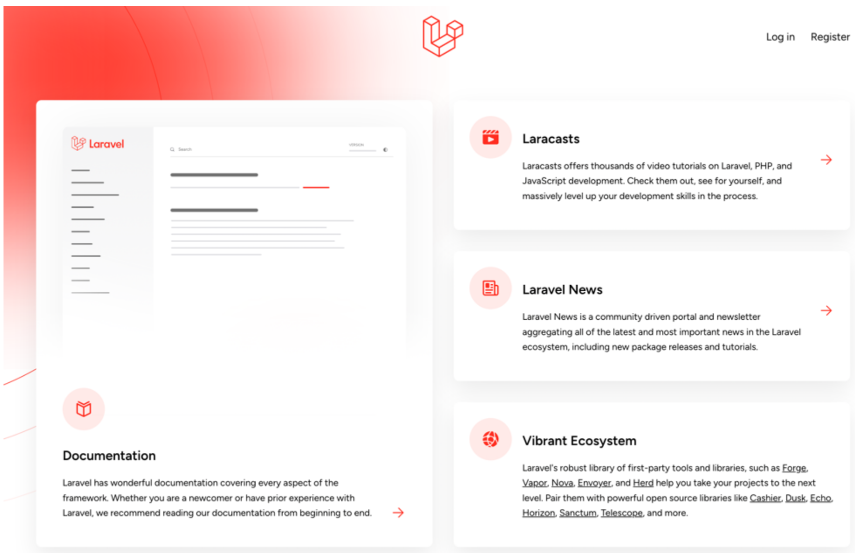
For this project, you must create a database in MySQL where Laravel will create the tables required for authentication.

Once you have well configured Laravel to be able to connect to the database, run the default migrations again: php artisan migrate

You can notice that Laravel has created many tables in your database, such as users, jobs, sessions, etc.

Display your web application in the browser

- With Herd, you can display the welcome page in the web browser at the address ecommerce.test
- Without Herd, from the directory containing your project, execute the instruction: "php artisan serve" and access it in the web browser at the address http://localhost:8000



Laracasts

Laracasts offers thousands of video tutorials on Laravel, PHP, and JavaScript development. Check them out, see for yourself, and massively level up your development skills in the process.

Laravel News

Laravel News is a community driven portal and newsletter aggregating all of the latest and most important news in the Laravel ecosystem, including new package releases and tutorials.

Documentation

Laravel has wonderful documentation covering every aspect of the framework. Whether you are a newcomer or have prior experience with Laravel, we recommend reading our documentation from beginning to end.

Vibrant Ecosystem

Laravel's robust library of first-party tools and libraries, such as Forge, Vapor, Nova, Envoyer, and Herd help you take your projects to the next level. Pair them with powerful open source libraries like Cashier, Dusk, Echo, Horizon, Sanctum, Telescope, and more.

Figure 1: the default welcome view

You can see that the authentication system is already available. You can register or login through the links in the upper right side of the website.

If the links are not working, run in the terminal: **npm install && npm run build**
If you don't have npm, run: nvm install, then retry.

Preparing the database

Before starting the development of the web pages, you will prepare the database.
Create the migrations for the products and categories tables with artisan.
For example: php artisan make:migration createCategories

Edit the created migrations to add all the table's columns in the up() method.

Create the models for both tables: php artisan make:model Category

Since the plural of category is categories and not categorys, you must give the name of the table in the DB to the category model by adding the "table" attribute to the Category model: protected $table = 'categories';

Now you can add the foreign keys in the createProducts migration.
For example, $table->foreignIdFor(\App\Models\Category::class, "category_id");

Execute the newly created migrations with: php artisan migrate.

Populating the tables

Create two users using the authentication system.

To create a lot of categories and products, use factories. First add the trait "use HasFactory;" to the Category and Product models. Second, create a factory for each table: php artisan make:factory CategoryFactory

Edit the definition() method of the created factories. For the name of the products or the categories, generate random values in the pattern "product1", "product3", etc.
Ex: "name" => 'product'.rand(1,100)

For the user and category foreign id, retreive a random one from the corresponding model. Example: "user_id" => User::*all*()->random()

To run the created factories, add the following instructions in the run() method in the file database/seeders/DatabaseSeeder.php:
Category::*factory*(5)->create();
Product::*factory*(30)->create();

Then execute the DatabaseSeeder as follows: php artisan db:seed

If you want to put real data in your database, you can add the data given in the first workshop using phpMyAdmin.

Relations

To let the model know that a product belongs to a category and to a user, you should add the relations in the models. For example in the Product model add the following to say that a product belongs to a category:

```
public  function category(){
   return $this->belongsTo(\App\Models\Category::class);
}
```

On the other hand, to let the Category model know that a category might have many products, add in the Category Model the following:

```
public function product(){
   return $this->hasMany(\App\Models\Product::class);
}
```

Show products

To handle the request related to products, create a product controller:

```
php artisan make:controller ProductController
```

The show products request will be sent to the "/products" route and will be handled by the index() method of the ProductController. Therefore, define in routes/web.php, the following route:

```
Route::get('/products', [\App\Http\Controllers\ProductController::class, 'index']);
```

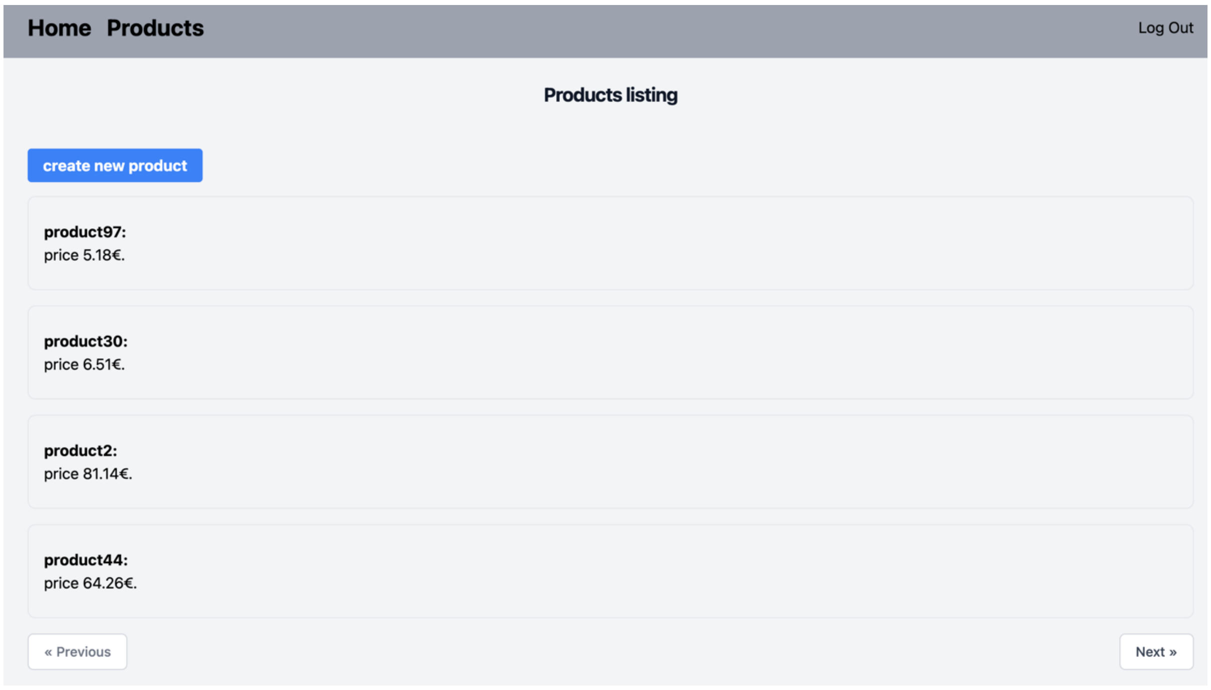
Define then the index() method in ProductController. It will retreives all the products with an eloquent query and show the products.blade.php view:

```
$products=Product::all();
return view('products.index',['products'=>$products]);
```

In resources/views, create a directory called "products". It will contain all the views to display and edit products. Create a view called "index.blade.php" in this directory. It will displays all the products.

```
@foreach ($products as $product)
   <a href="/products/{{ $product['id'] }}">
      <div>
         <strong>{{ $product['name'] }}:</strong> <br/>
         price: {{ $product['price'] }}€.
      </div>
   </a>
@endforeach
```

In order not to show all the products in one page and to avoid overloading the server, you can use laravel pagination. Replace in the index method of the controller, all() with "simplePaginate(nbOfRows)" and add to the products.index view, {{$products->links()}} to show the pagination links.



Products listing

product97:
price 5.18€.

product30:
price 6.51€.

product2:
price 81.14€.

product44:
price 64.26€.

Figure 2: index view that shows the stored products

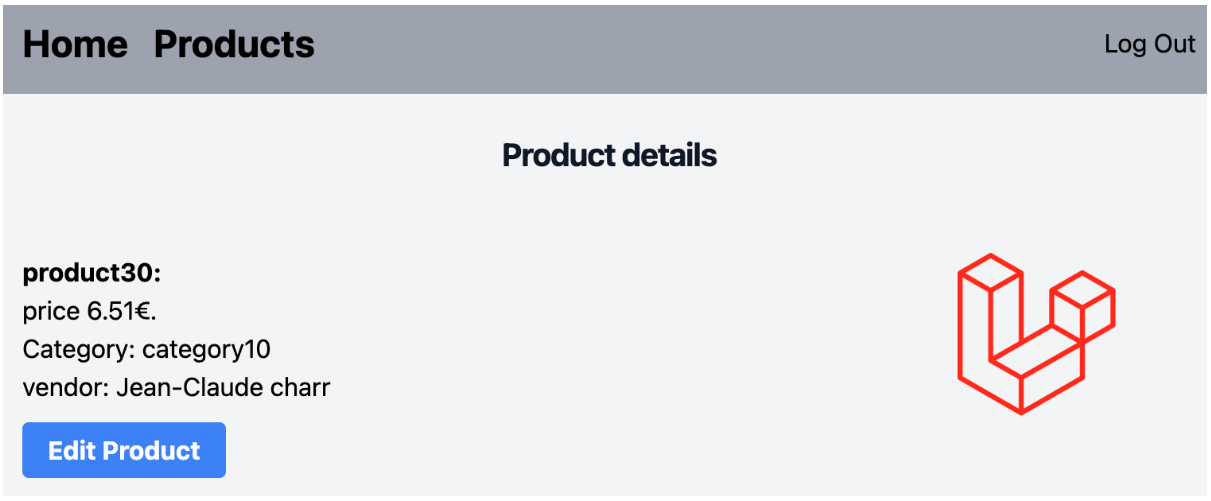
Show a product

Each product in the products.index view has a link to "/products/{{ $product['id'] }}" where the id of the selected product is given to the request to display more details about that product.

As you did in the previous section, define a route to show a product with a given id:
Route::get('/products/{product}', [\App\Http\Controllers\ProductController::class, 'show']);

Then define the show() method in the controller. It should return the view to show the details of the product such as the category and the vendor name.

You can also add an image of the product. The images should be stored in the "public" directory and can be added to the view as follows:
<img src="{{ asset('img/imageName.png') }}" alt="description of image">



Product details

product30:
price 6.51€.
Category: category10
vendor: Jean-Claude charr

Figure 3: show view that displays the details of a product

Modify a product

A user can only edit its own products. Therefore, the edit link should only appear in the "show" view if the product belongs to the connected user. To be able to verify if the shown product belongs to the connected user, you should create a Gate in the boot() method of the app/provider/AppServiceProvider class as follows:

```
Gate::define('update-product', function (User $user, Product $product) {
   return $user->id === $product->user_id; });
```

Then the gate can be used in the view as follows to show the edit link if the product belongs to the connected user:

```
@can('update-product', $product)
   <a href="/products/{{ $product->id }}/edit">Edit Product</a>
@endcan
```

You can notice that the edit link points to a new route. Add a new route entry in the routing web.php file to handle this edit route:

```
Route::get('/products/{product}/edit', [\App\Http\Controllers\ProductController::class, 'edit']);
```

Afterwards, define the edit method in ProductController. It should show a view to edit the selected product's properties:

```
public function edit(Product $product){
   return view('products.edit', ['product' => $product]);
}
```

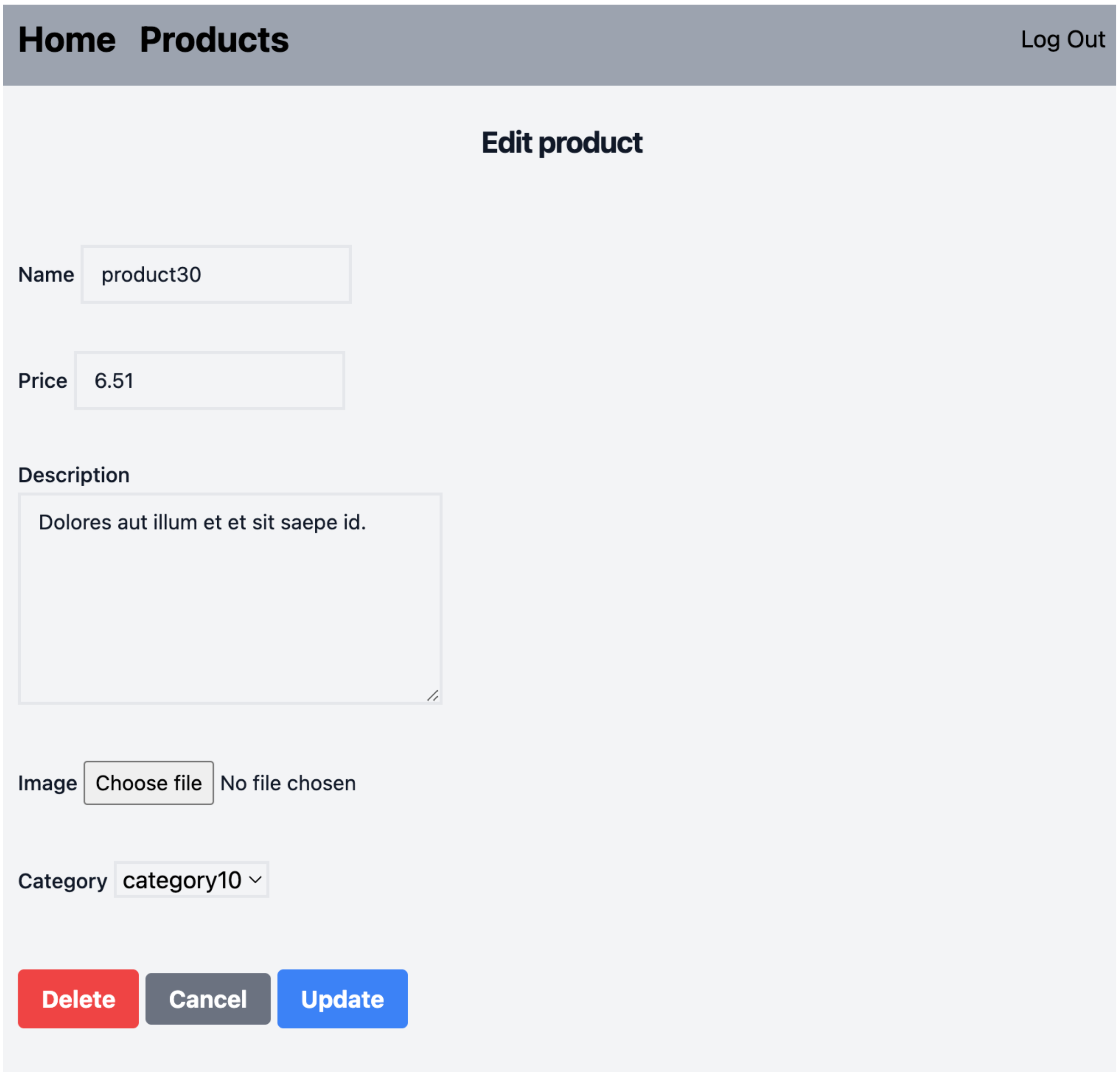
Implement the edit view which contains a form prefilled with the product's data that the user can modify. The form also contains two buttons, one to submit the modifications (update), and the other to cancel the modifications.

In the form, the user should see the product's current values and he should be able to modify the name, price, description, image and category of the product. There will be inputs for the name, price and image properties, a text area for the description property and a select (dropdown list) for the category property. The options of the select should show all the possible categories stored in the category table. Therefore, the edit method of the ProductController should get all the categories and pass them as an aditional parameter to the edit view.

The update button sends a patch request to /products/{product}. Most browsers have not implemeted yet the patch method for the form tag. Therefore, keep the form method equal to post and add in the form the following directive: @method("PATCH"). It will tell laravel that it is a patch request.

Now add a new route in the web.php file to handle this path request:

```
Route::patch('/products/{product}', [\App\Http\Controllers\ProductController::class,
'update']);
```



Name: product30

Price: 6.51

Description: Dolores aut illum et et sit saepe id.

Image: Choose file No file chosen

Category: category10

Figure 4: edit view that allows to modify the product's details

Define the update method of the ProductController that gets the new data from the request object and updates the database:

```
//save the uploaded file to public/img
$file = request()->file('image');
$path = $file->store('img', 'public_uploads');

//update the product details
$product->update(['name' => request('name'),
    'price' => request('price'),
    'description' => request('description'),
    'category_id' => request('category'),
    'image'=>$path
]);

return view('products.show', ['product' => $product]);
```

If you didn’t add the Cross-Site Request Forgery directive (@csrf) in the form you will get the error ‘expired’. This token is necessary to verify that the update request is not coming from another website.

To specify that the image file must be stored in the public directory, add to the config/filesystems.php the following instructions to define the public_uploads storage:

```
'public_uploads' => [
   'driver' => 'local',
   'root'   => public_path(),
],
```

You may notice when trying to update a product you will get the exception **Illuminate\Database\Eloquent\MassAssignmentException**. Laravel by default does not allow mass assignment for security reasons. To modify this restriction add the following to the Product model: protected $fillable = ['name', 'price', 'description', 'category_id', 'image']; Now you can mass assign all the columns in the $fillable array;

Validation

For security reasons such as SQL injection and to avoid execution errors, It is very important to validate the request data before inserting them in the database. The request method can be used to apply various validation rules on each field. For example, the following instruction check if the “name” field has a value consisting of at least 3 alpha-numeric characters:

```
request()->validate('name' => ['required','alpha_num:ascii','min:3']);
```

Add validations rules to check if the 'price' field has a numeric value between 0 and 999.99 and if the ‘image’ field has a MIME type equal to jpg or png.

Don’t forget to verify that the connected user has the right to uopdate this product:

```
if(Gate::authorize('update-product',$product)) {
        //update
}
```

Delete a product

Add to the edit view a button to delete the product. A delete request must use the ‘delete’ method. However, as for the ‘patch’ method, most browser do not recongnize the ‘delete’ method. Therefore, you have to create a new form that uses the ‘delete’ method and call the delete route. The delete button must refer to this new form:

```
<button form="delete-form">Delete</button>
<form id="delete-form" method="POST" action="/products/{{$product->id}}" class
="hidden">
   @csrf
   @method("DELETE")
</form>
```

Define in the routes file (web.php), the route to delete the selected product:

```
Route::delete('/products/{product}', [\App\Http\Controllers\ProductController::class,
'destroy']);
```

It calls the destroy method defined in the ProductController:

```
public function destroy(Product $product){
   $product->delete();
   return redirect("/products");
}
```

Of course, you have to verify that the connected user has the right to delete this product:

```
if(Gate::authorize('update-product',$product)) {
         //delete product
}
```

Add a new product

Add an ‘add’ button in the products.index view to add a new product. This button should be only shown if the user is connected. This button will send a GET request to show a view containing a form to create a new product:

```
@can('create-product')
    <a href="/products/create">create new product</a>
@endcan
```

A ‘create-product’ gate should be defined to verify that the user is connected.

Now add the route "/products/create" to the routing file:

```
Route::get('/products/create', [\App\Http\Controllers\ProductController::class, 'create']);
```

Be carefull to add this route before the route to show a product with a given id. Otherwise, Laravel will think that “create” is an id and will search for a product with the id equal to “create”.

Afterwards, define the create() method in ProductController. It should show a new products.create view. This view contains a form to insert the data of the new product and submit them to the server. It is very similar to the ‘edit’ view. The form in this view will submit a post request tp ‘/products’.

Add This new route to web.php. It will be handled by the “store()” method in the ProductController. Finally, the store() method is similar to the “update()” method. The request data is validated and then a new Product is created and saved in the DB. The “store()” method redirects to “/products” after inserting the new product.

Additional tasks:

- Filter products

- Search for a product
- Show last five products on the home page
- Show the user’s products in its dashboard when connected