Angular workbook

Table of Contents

Setting up your environment

The following software must be installed in your environment:

○ Node.js *(and NPM which is installed at the same time as Node.js)*

○ Git

○ Visual Studio Code

Following the course on your local system

Install the LTS versions of each software.

Following the course on Strigo Virtual Machine

The required software should already be installed.

For VSCode, be sure to use the one located at:

```
C:\Program Files\Microsoft VS Code
```

Application source code

To complete the labs, you'll be using the Angular application provided, for which **ESLint** and **Prettier** are already configured:

```
./lab/workspace/zenika-ng-website
```

The details of this configuration are beyond the scope of this course, but you'll find them clearly documented in the `README.md` file in the same directory.

Nevertheless, you will first create a raw Angular application to see for yourself how it works, even if you won't be using it for the rest of the course.

Creating and running an Angular application

Install the Angular CLI globally and create the app with these shell commands

```
npm i -g @angular/cli
ng new my-first-app
```

You will be shown some options for the app:

○ Choose **"CSS"** for style preprocessor

○ Choose **"No"** for SSR/SSG/Prerendering

○ Choose **"Yes"** for 'zoneless' application

If you can't install the Angular CLI globally, create the app with the following shell command

```
npx @angular/cli new my-first-app
```

However, in this case, to run an Angular CLI command, you will need to use NPM first: `npm run ng <command>` instead of just `ng <command>`.

Run the Angular dev server

```
ng serve # or: `npm start`
```

○ Open the Chrome browser and visit: http://localhost:4200.

You should see the app with placeholder content. 🚀

Taking control of the app

Even though we haven't yet studied the main concepts, let's modify the app right away!

○ Replace the content of `src/app/app.html` with:

```html
<h1>Welcome to {{ title }}!</h1>
```

○ Add some styling in `src/app/app.css`:

```css
h1 {
    color: blue;
}
```

○ Replace the property `title` in `src/app/app.ts` with:

```ts
class App {
    title = 'my first component';
}
```

○ Check that the app has been updated correctly in the browser. 🚀

Lab 1: Getting started

Now let's try running the app tests

```
ng test # or: `npm test`
```

Because we've modified the app, a test in `app.spec.ts` fails!

○ Fix the test *"should render title"* by guessing what's wrong

Finally, let's build the app for production

```
ng build # or: `npm run build`
```

○ Open a Terminal in the `dist/my-first-app/browser/` directory and run the command:

```
npx serve --single .
```

This command will download and run the NPM package named serve. Note that this package is not related to the `ng serve` command.

○ Open the browser at the URL specified in the console to see the production build in action. 🚀

Now, you are done with the raw application you've created! 🎉
From now on, you'll be using the Angular application provided:

```
./lab/workspace/zenika-ng-website
```

Version control system

○ Open the browser and log in to your favorite cloud-based version control system (Github, Gitlab, ...)

○ Remotely, create a new empty repository named `zenika-ng-website` in which to save your code

○ Locally, configure your Git user **name** and **email**:

```
git config --global user.name "<YOUR_NAME>"
git config --global user.email <YOUR_EMAIL>
```

○ Open the directory `./lab/workspace/zenika-ng-website` in VSCode

○ Run the following commands to create a new Git repository and add your first commit

```
git init
git add .
git commit -m "Initial commit"
```

○ Finally, push your local repository from the command line over *HTTPS* (not SSH).

Here's an example for Github:

```
git remote add origin https://github.com/[YOUR_USERNAME]/zenika-ng-website.git
git branch -M main
git push -u origin main
```

NPM packages

Install your app dependencies with NPM by running the following command:

```
npm i # or: npm install
```

Run the Angular dev server to verify that everything works fine.

```
ng serve # or: `npm start`
```

Visual Studio Code Extensions

If you use VSCode as your IDE, install the following extensions to improve your developer experience:

- Angular Language Service

- ESLint

- Prettier

- Auto Rename Tag (optional)

- Github Theme (optional)

- Material Icon Theme (optional)

Lab 2: Workspace 1/3

During the rest of the course, you will develop an e-commerce application.

The design team has been working hard, and the result is available in the following directory:

```
lab/resources/design
```

You're going to integrate this design into your Angular application.

First, let's start a local server to see what the app looks like.

○ Open a new Terminal in the **design** directory and run the command:

```
npx serve .
```

○ Open the browser at the URL specified in the console. You should see the 4 products available in the catalog.

○ Next, copy/paste the contents of **design/assets** into **public/assets**

○ Finally, open the file **design/index.html** in your code editor

→ It contains **brief information** about the layout of the design

→ Follow the **detailed instructions** provided below in this workbook to integrate the design into your Angular application

Lab 2: Workspace 2/3

Adding Bootstrap CSS

○ Install Bootstrap with NPM:

```
npm i bootstrap
```

○ In the **angular.json** file, add **bootstrap.min.css** to the **"styles"** array in both **"build"** and **"test"** sections:

```
{
  "projects": {
    "zenika-ng-website": {
      "architect": {
        "build": {
          "options": {
            "styles": [
              "node_modules/bootstrap/dist/css/bootstrap.min.css",
              "src/styles.css"
            ]
          }
        },
        "test": {
          "options": {
            "styles": [
              "node_modules/bootstrap/dist/css/bootstrap.min.css",
              "src/styles.css"
            ]
          }
        }
      }
    }
  }
}
```

Adding the HTML code

○ Copy/paste the inner content of the **body** tag to `src/app/app.html`

```
<body>
   <!-- ONLY WHAT'S INSIDE THE BODY TAG -->
</body>
```

○ Serve your app using `ng serve` to see if the result is equivalent to the design

Lab 3: Components 1/3

In this lab, you'll start creating Angular components to break down the giant **App** component template into smaller parts

Creating the "menu" component

○ Create a menu component with the following shell command and move the corresponding code into it

```
ng generate component menu
```

○ Once done, use the component `<app-menu />` in `src/app/app.html`

Lab 3: Components 2/3

Creating the "product-card" component

○ Create a product-card component with the following shell command and move the corresponding code into it

```
ng g c --flat true product/product-card
```

○ Add a file `product-types.ts` in the same directory and define the product interface

```typescript
export interface Product {
  id: string;
  title: string;
  description: string;
  photo: string;
  price: number;
  stock: number;
}
```

○ The `ProductCard` component should accept:

→ an input: `product = input.required<Product>();`

→ an output: `addToBasket = output<Product>();`

○ Use the properties of the `product` object in the template to display the `title`, `description`, ...

```html
... <a class="card-link">{{ product().title }}</a> ...
```

○ Use [`class.*`] syntax to add the CSS class `.text-bg-warning` to the element `<div class="card h-100 text-center">` but only when the product `stock` is equal to 1 (last chance to buy it!).

○ The output should emit the product when the user clicks on the button "Ajoutez au panier"

Lab 3: Components 3/3

Storing all products in the App component

Currently, the products are hard-coded in the template `src/app/app.html`. Let's give the **App** component class data ownership.

○ In `src/app/app.ts`, define a `products: Product[] = [];` property

○ Fill the array with the content of the file
`lab/resources/design/products.json`

○ In `src/app/app.html`, use the component `<app-product-card />` instead of each hard-coded product (later in the course, we'll use a "for" loop to achieve this)

```
<app-product-card [product]="products[0]" class="col" />
<app-product-card [product]="products[1]" class="col" />
...
```

😉 *Note that the tag added by the component selector* `<app-product-card class="col" />` *replaces the tag* `<div class="col">` *in the original HTML markup.*

○ In `src/app/app.ts`, define a `total = 0;` property for the total basket price, that should be updated each time the user clicks on the button "Ajoutez au panier"

→ To achieve this, add a method `addToBasket` in the **App** component class and use it in its template

→ Display the `total` in the component template

In this lab, you will implement the tests for the app you developed in the "**Lab 3: Components**".

The `Menu` component doesn't need to be tested, since it has no logic.

You're going to focus on the `ProductCard` and `App` components.

○ Before running the tests, replace the content of `app.spec.ts` with the following:

```typescript
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { App } from './app';

describe('App', () => {
  let component: App;
  let fixture: ComponentFixture<App>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [App],
    }).compileComponents();

    fixture = TestBed.createComponent(App);
    component = fixture.componentInstance;
    await fixture.whenStable();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

○ Run the tests using Angular CLI:

```
ng test
```

Some tests fail. Let's fix them!

product-card.spec.ts

○ First, let's focus on this test, disabling all the others:

```
// Add temporarily the prefix "f" ("focus") to the `describe` function
fdescribe('ProductCard', () => {
  /* ... */
});
```

○ In the **beforeEach** section, define the required **product** property:

```
fixture.componentRef.setInput('product', {
  title: 'TITLE',
  description: 'DESC',
  // ...
});
```

Now, the test setup should pass (but we're not testing anything useful at the moment).

Tests

○ It should display the product photo as image url

○ It should display the product description

○ It should display the product title

○ It should display the product price

○ It should emit addToBasket event with the given product when the button is clicked

→ Spy on the **emit** method of the **addToBasket** output to check that it is called

○ It should not add the "text-bg-warning" className when stock is greater than 1

○ It should add the "text-bg-warning" className when stock is equal to 1

Lab 4: Testing 3/3

`app.spec.ts`

- Now remove the "f" ("focus") prefix you previously added to the `describe` function of the `ProductCard` test.

The **App** component depends on 2 other components:

- `Menu`

- `ProductCard`

Choose one of the two approaches you learned about in the slides:

- First approach - **with implicit dependency import**

- Second approach - **allowing unknown HTML elements**

Tests

- It should display the products

- It should update the total when "addToBasket" class method is called (Class testing)

- It should update the total when a product emits the "addToBasket" event (DOM testing)

Lab 5: Control flow

In this lab, you'll use the `@for`, `@if` and `@else` blocks to improve the application's logic.

App **component**

○ Update **addToBasket** method to decrease the product stock when user clicks on *"Ajouter au panier"*

○ Add a getter `get productsInStock(): Product[]` which returns the list of products whose stock is greater than 0

○ Use a `@for` block to iterate over the **products** (in stock) array to display each `<app-product-card />` component

○ Use `@if (productsInStock.length) {} @else {}` to display the message *"Désolé, notre stock est vide !"* when all products have a stock equal to 0

Tests

`app.spec.ts`

○ It should decrease the stock of the product added to the basket

○ It should not display products whose stock is empty

○ It should display a message when stock is completely empty

Lab 6: Directives

HighlightPrice directive

○ Generate the directive with the following command:

```
ng generate directive highlight-price --flat false
```

○ Define a `price` property as directive input

→ with a default value equal to `0`

→ with an `alias` equal to the `'appHighlightPrice'`

○ Add a getter `highlightClass` that

→ should return `'text-body-tertiary fst-italic'` when price == 0

→ should return `'text-body-secondary'` when price < 50

→ should return `'text-body-emphasis'` when price >= 50

○ Use `host` directive metadata to bind `highlightClass` value to the `class` attribute

○ Apply the directive to the following element in `src/app/app.html`

```
<p>Votre panier s'élève à {{ total }} €</p>
```

Tests

highlight-price.spec.ts

○ should have class "text-body-tertiary fst-italic" when price == 0

○ should have class "text-body-secondary" when price < 50

○ should have class "text-body-emphasis" when the price >= 50

Lab 7: Signals

In this Lab, you'll convert the **App** component properties into signals.

`HighlightPrice` directive

○ Use a `computed<string>` signal for the `highlightClass` property

App **component**

○ Use a `signal<Product[]>` signal for the `products` property

○ Use a `computed<Product[]>` signal for the `productsInStock` property

○ Use a `signal<number>` signal for the `total` property

○ Fix the `addToBasket()` method to properly update the `products` and `total` signals

○ Update the component template to properly consume the different signals

Tests

`app.spec.ts`

○ Fix the tests to properly consume the different signals

In this lab, you'll move the data ownership from the **App** component to **services**.

You need to create 2 services using Angular CLI:

○ `src/app/catalog/catalog-resource.ts`: to manage the products

○ `src/app/basket/basket-resource.ts`: to manage the basket items

CatalogResource

The service should have:

○ A `_products = signal<Product[]>(...)` private property *(move here the 4 products defined in `app.ts`)*

○ A `products = this._products.asReadonly()` public property

○ A `productsInStock = computed<Product[]>(...)` computed signal that returns the list of products whose stock is greater than 0 *(move some code from `app.ts`)*

○ A `decreaseStock(productId: string)` method to decrease the corresponding product stock if it is greater than 0 *(move some code from `app.ts`)*

Usage

○ Refactor the **App** component to use the `CatalogResource` service

Lab 8: Dependency injection 2/5

BasketResource

○ Define a new interface:

```ts
// src/app/basket/basket-types.ts
export interface BasketItem {
  id: string;
  title: string;
  price: number;
}
```

The service should have:

○ A `_items = signal<BasketItem[]>(...)` private property

○ A `items = this._items.asReadonly()` public property

○ A `total = computed<number>(...)` computed signal that returns the basket total

○ A `addItem(item: BasketItem): void` method that adds an item to the basket

Usage

○ Refactor the `App` component to use the `BasketResource` service

○ In the `Menu` component, use the `BasketResource` to display the number of items in the basket. To achieve this, add a `numberOfItems = computed<number>(...)` property to the menu component.

Lab 8: Dependency injection 3/5

Bonus: use of injection token

○ Create an injection token `APP_TITLE` in `src/app/app.token.ts`

○ Provide the token using a `ValueProvider` with the value *"Bienvenue sur Zenika Ecommerce"*

○ Inject the token in the **App** component to display the app title

Tests

Since we've modified the application extensively, tests fail!

○ For now, let's disable the tests in `app.spec.ts` by adding an x before the main `describe()`:

```
xdescribe("App", () => { /* ... */ });
```

catalog-resource.spec.ts

○ It should decrease the product stock

○ It should not decrease the product stock when stock is empty

basket-resource.spec.ts

○ It should update the items when a product is added

○ It should update the total when a product is added

Lab 8: Dependency injection 4/5

`menu.spec.ts`

The component now depends on the newly created **BasketResource**. Note that, as this service is "provided in root", it is automatically provided in **TestBed** and used in our tests.

```
@Injectable({ providedIn: "root" })
export class BasketResource {}
```

But remember that the goal of unit testing is to test each unit in isolation. So, we need to use *Mocks* instead of real implementations.

○ Create a minimalist class called **BasketResourceMock** that will replace the **BasketResource**

```
// Note: do not use `{ providedIn: "root" }` metadata
// because the mock will be provided manually in our tests.
@Injectable()
export class BasketResourceMock implements Partial<BasketResource> {
  items = signal<BasketItem[]>([]);
  total = signal(0);
  addItem(item: BasketItem): void {
    this.items.update((items) => [...items, item]);
  }
}
```

○ Provide the mock in `menu.spec.ts`

Add test:

○ It should display the number of items

Lab 8: Dependency injection 5/5

`app.spec.ts`

Some tests currently performed in this component do not need to be fixed, but simply removed, as they are no longer relevant.

○ Remove the tests related to the computation of the **basket total** and **catalog stock update** (the `App` component is no longer responsible for these computations):

→ ~~It should update the total when a product emits the "addToBasket" event~~

→ ~~It should update the total when "addToBasket" class method is called~~

→ ~~It should decrease the stock of the product added to the basket~~

○ Remove the `x` from `xdescribe()` that you added previously to re-enable the tests

○ Create a minimalist class `CatalogResourceMock` that will replace the `CatalogResource` (such as you did above for the `BasketResource`)

○ Provide the 2 mocks in `app.spec.ts`

○ Provide a value for `APP_TITLE` injection token

○ Fix the remaining tests

Add new, more relevant tests:

○ It should call "CatalogResource.decreaseStock" and "BasketResource.addItem" methods when a product is added to the basket

→ For that use `TestBed.inject` function (to get the services instances) and `vi.spyOn` function (to spy on these methods)

○ It should display the app title

Lab 9: Pipes 1/3

In this lab, you'll use pipes to format the application content.

ProductCard component

Let's start by using pipes provided by the Angular framework:

○ Use the `uppercase` pipe to display the product title in uppercase

○ Use the `currency` pipe to display the product price with the currency

At the moment, notice that the price is in **$** and formatted for the **en-US** locale (example: **"$21"**). But we need to display it in **€** for the **fr** locale (example: **"21 €"**).

Let's fix this!

Application config

○ First, register the **"fr"** locale in your application

```
// src/app/app.config.ts
import { registerLocaleData } from "@angular/common";
import localeFr from "@angular/common/locales/fr";

registerLocaleData(localeFr);
```

○ Next, provide **LOCALE_ID** and **DEFAULT_CURRENCY_CODE** in the app config

```
// src/app/app.config.ts
import { DEFAULT_CURRENCY_CODE, LOCALE_ID } from "@angular/core";

export const appConfig: ApplicationConfig = {
  providers: [
    { provide: LOCALE_ID, useValue: "fr" },
    { provide: DEFAULT_CURRENCY_CODE, useValue: "EUR" },
  ],
};
```

The product price should now be displayed correctly. 🎉

App **component**

○ Use the `currency` pipe to display the basket total

SortProductsPipe

Now, let's create a custom pipe of our own!

We want to be able to sort the displayed products by `price` or `stock`.

○ Generate the pipe `src/app/sort-products/sort-products-pipe.ts` using Angular CLI

  → Implement the `transform` method that returns the sorted array of products

  → Add an optional parameter to the pipe to specify on which property (`price` or `stock`) to sort the products

○ Once you're finished, use your pipe to sort the products in the **App** component template

Finally, let's add a selector to choose between `price` and `stock` sorting. You'll find a component ready for use here: `lab/resources/select-product-key`.

○ Copy/paste the `lab/resources/select-product-key` directory into your app at `src/app/select-product-key`

Lab 9: Pipes 3/3

Use the component:

○ Add **productKey** in **app.ts** component

```typescript
import { Component } from "@angular/core";
import { ProductKey } from "./select-product-key/product-key-types";

@Component({
  /* ... */
})
export class App {
  productKey = signal<ProductKey>(undefined);
}
```

○ Use **<app-select-product-key />** in **app.html**

```html
<app-select-product-key [(productKey)]="productKey" />
```

Tests

app.spec.ts

○ It should display the products sorted by price

○ It should display the products sorted by stock

○ It should display the basket total with currency

sort-products-pipe.spec.ts

○ It should not sort products when key is undefined

○ It should sort products by price

○ It should sort products by title

product.spec.ts

○ It should display product title in uppercase

○ It should display product price with currency

Lab 10: HTTP client 1/3

In this lab, you'll communicate with a REST API server that will manage the products and the basket.

○ To run the server, open a new Terminal in the `lab/resources/server` directory and run the following commands:

```
npm install
npm start
```

The server is listening on: `http://localhost:8080/api/`

Here are the available endpoints:

○ `GET /products` to fetch all products

   → Response: `Product[]`

○ `GET /products/:productId` to fetch one product

   → Response: `Product`

○ `GET /basket` to fetch the basket

   → Response: `BasketItem[]`

○ `POST /basket` to add a new item to the basket

   → Request body: `{ productId: string; }`

   → Response: `BasketItem`

ApplicationConfig

○ Add the HTTP provider: `provideHttpClient(withFetch())`

CatalogResource

○ Inject the **HttpClient** service

○ Remove the hard coded products from the **_products** property

○ Add a **fetchProducts(): Observable<Product[]>** method that gets the products from the server and stores them in the **_products** signal.
To achieve this side effect, use the RxJS **tap** operator in the **.pipe()** transformation chain:

```typescript
import { Observable, tap } from 'rxjs';
import { Injectable } from '@angular/core';
import { Product } from '../product/product-types';

@Injectable({
  providedIn: 'root',
})
export class CatalogResource {
  fetchProducts(): Observable<Product[]> {
    return this.httpClient
      .get<Product[]>('http://localhost:8080/api/products')
      .pipe(tap((products) => this._products.set(products)));
  }
}
```

Updating the App component

○ Subscribe to **CatalogResource.fetchProducts** method in the **App** class constructor, to trigger data fetching

Lab 10: HTTP client 3/3

BasketResource

- Inject the `HttpClient` service

- Add a `fetchBasket(): Observable<BasketItem[]>` method (such as we did with `fetchProducts()` for the `CatalogResource`)

- Add a `addItem(productId: string): Observable<BasketItem>` method that posts (to the server) the item to be added and then update the `_items` property accordingly (in the service)

Updating the App component

- Subscribe to `basketResource.fetchBasket()` method in the **App** class constructor, to trigger data fetching

- Update the `addToBasket()` method so that it subscribes correctly to `BasketResource.addItem` method

Tests

At this point, a lot of tests fail! This is because the structure of the application has changed radically. Refactoring the tests would take too much time in the context of this course.

With the trainer, take a look at the new test implementation in the following directory:

- `lab/solutions/projects/10_http_client`

In this lab, you'll create a single-page application (SPA) using the Angular router.

app.config.ts

○ Create the `app.routes.ts` file with the following content:

```typescript
import { Routes } from '@angular/router';

export const routes: Routes = [];
```

○ Provide the router in the `app.config.ts` file:

```typescript
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';
import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
    // ...
    provideRouter(routes),
  ],
};
```

app.routes.ts

○ Create the following components and declare a route for each one:

→ Component: `Catalog` --> Route: `'catalog'`

→ Component: `Basket` --> Route: `'basket'`

○ Add a route `**` that redirects to `'/catalog'`

Catalog component

○ Move the main content you have developed in the **App** component to this one, including:

→ the template

→ the class logic

→ the tests (optional)

App component

○ In `src/app/app.ts`, add `RouterOutlet` to the component `imports`

○ In `src/app/app.html`, add a `<router-outlet />` directive, instead of the main content you just moved. The template should now look like this:

```
<app-menu />

<main class="py-4 container">
  <router-outlet />
</main>
```

Lab 11: Routing 3/8

RouterLink

Add `routerLink` directives in the following templates (don't forget to add `RouterLink` to the related component imports):

- In `catalog.html`:

  → to visit the page *"Voir mon panier"*

- In `menu.html`:

  → to return to the home page when clicking on *"Zenika Ecommerce"*

  → to visit the page *"Voir mon panier"*

- In `product-card.html`:

  → to visit the product details page at `['/product', product.id]` (below, you will create the `ProductDetails` component in the bonus section)

Basket **component**

○ Use the following markup for the component template:

```html
<h2 class="h4">Mon panier</h2>

<div class="card">
  <div class="card-header">2 articles</div>

  <ul class="list-group list-group-flush">
    <!-- Use `@for` block to loop over the basket items -->
    <li class="list-group-item d-flex justify-content-between">
      Coding the snow <span class="text-primary">19 €</span>
    </li>

    <li class="list-group-item d-flex justify-content-between">
      Coding the world <span class="text-primary">18 €</span>
    </li>
    <!-- End of: Use `@for` block to loop over the basket items -->

    <li class="list-group-item d-flex justify-content-between fw-bold">
      Total <span class="text-primary">37 €</span>
    </li>
  </ul>
</div>
```

○ Use the `BasketResource` service to implement the component logic

○ Subscribe to `BasketResource.fetchBasket` method in the class constructor, to trigger data fetching

○ To check that everything is working properly, you should be able to:

→ Visit the `http://localhost:4200/catalog` page, click on *"Voir mon panier"* and view the basket items

→ Reload the `http://localhost:4200/basket` page and view the basket items

Bonus: BasketGuard

When visiting the page `http://localhost:4200/basket`:

○ If there are items in the basket, the **Basket** component should be displayed

○ If the basket is empty, an alternate **BasketEmpty** component should be displayed

Let's do this!

○ Generate a **CanMatch** guard in `src/app/basket/basket-guard.ts`

```
import { inject } from "@angular/core";
import { CanMatchFn } from "@angular/router";
import { BasketResource } from "./basket-resource";

export const basketGuard: CanMatchFn = () => {
  const basketResource = inject(BasketResource);
  return /* to be continued... */;
};
```

○ Add the guard to the appropriate route

○ Generate a new **BasketEmpty** component

→ It simply displays *"Votre panier est vide."*

○ Add the route `'basket'` to display the component

→ Yes, it's the same route as for the **Basket** component

○ At this point, you can safely remove the **BasketResource.fetchBasket** subscription from the **Basket** component constructor, because data fetching is now triggered by the guard itself anyway

Bonus: `ProductDetails` component

○ Create the component and add a lazy-loaded route `'product/:id'`

○ Retrieve the `:id` from the `ActivatedRoute` snapshot

○ Fetch the product from the server using the `HttpClient` service:

  → `http://localhost:8080/api/product/:id`

○ Store the fetched product in a class property:

  → `product = signal<Product | undefined>(undefined);`

○ For the component template, copy/paste the following:

  → `lab/resources/product-details/product-details.html`

Bonus: application performance

Have you noticed that when loading the catalog, the message *"Désolé, notre stock est vide !"* appears briefly and is then replaced by the products once fetched?

You can improve this by not displaying anything as long as the **products** are undefined.

○ In the **/catalog/catalog-resource.ts** service, change the **_product** signature:

```
@Injectable({
  providedIn: 'root',
})
export class CatalogResource {
  private _products = signal<Product[] | undefined>(undefined);
  // Remember that previously, it was: `signal<Product[]>([])`
}
```

○ Fix the errors raised by this change

○ Finally, in the **/catalog/catalog.html** component template, use an **@if {}** statement like this:

```
@if (products()) {
  <!-- The "@for" loop that displays the products -->
}
```

Bonus: directories organization

○ The following directories can be moved within the `src/app/catalog/` directory:

    → `/highlight-price/`

    → `/product/`

    → `/select-product-key/`

    → `/sort-products/`

Lab 12: Forms 1/5

In this lab, you'll create an Angular form to checkout the basket.

- Generate a new component using Angular CLI:

    → `src/app/basket/checkout-form/checkout-form.ts`

- Add the `FormsModule` to the component `imports` metadata

- For the component template, copy/paste the design made with love by the UI/UX team:

    → `lab/resources/checkout-form/checkout-form.html`

- Insert the component selector at the end of the basket component template:

    → `<app-checkout-form />`

Handle form fields

○ For each field, add the `ngModel` directive and create a template variable to access it

   → For example `<input name="name" ngModel #nameModel="ngModel" />`

○ Fields validation:

   → All fields are required

   → Credit card field must match the pattern `^[0-9]{3}-[0-9]{3}$`

○ Fields appearance:

   → Add CSS class `.is-invalid` when the field's state is "touched" and "invalid"

   → Add CSS class `.is-valid` when the field's state is "valid"

○ Credit card field has 2 "invalid-feedback" messages:

   → Use an `@if` block to display only the relevant error

Lab 12: Forms 3/5

Handle form submission

○ In the component class, add a new method:

→ `checkout(checkoutDetails: CheckoutDetails): void`
   (leave the implementation empty for now…)

○ In the component template, on the `<form>` element:

→ Create a template variable `#checkoutForm` to access the `ngForm` directive

→ Handle the `ngSubmit` event to call the `checkout` method you just created

→ And use the `checkoutForm.value` property as the `checkout` method argument

○ Still in the component template:

→ The submit button should be disabled as long as the form is invalid

→ Form fields and the submit button should be disabled when the form is being submitted (to achieve these 2 points, add a new property `checkoutInProgress: signal<boolean>` in the component class)

Lab 12: Forms 4/5

Basket-related changes

○ In `src/app/basket/basket-types.ts`, add new interfaces:

```typescript
export interface CheckoutDetails {
  name: string;
  address: string;
  creditCard: string;
}

export interface CheckoutOrder {
  orderNumber: number;
}
```

○ In the `src/app/basket/basket-resource.ts` service, add a new method to checkout the basket:

```typescript
export class BasketResource {
  checkout(checkoutDetails: CheckoutDetails): Observable<CheckoutOrder> {
    return this.httpClient
      .post<CheckoutOrder>(
        'http://localhost:8080/api/basket/checkout',
        checkoutDetails
      )
      // Empty the basket items after checkout completes
      .pipe(tap(() => this._items.set([])));
  }
}
```

Lab 12: Forms 5/5

Back to `CheckoutForm` component

You now have everything you need to implement the `checkout()` method you created earlier

Subscribe to the `BasketResource.checkout()` method and handle "next" and "error" events:

- On **"next"**:

    → Display a "success" message with the `orderNumber`

    → Add a link to navigate to the home page using the `routerLink` directive

    → In this case, the form fields must remain disabled

- On **"error"**:

    → Display a "danger" message

    → The user should be able to hide the "danger" message when clicking on the "close" button

    → In this case, the form fields should be enabled again to allow the user to retry submitting the form

Bonus

Check the following directory to see the `ReactiveFormsModule` implementation:

- `lab/solutions/projects/12_forms/src/app/basket/checkout-reactive-form/`