Angular

Labs

2024-01-21#e6ff771

Table of contents

Lab 1: Getting started with Angular

Setting up your environment

Training on Local

You should install the following on your system:

- Node.js version LTS
- NPM (It will be installed at the same time as Node.js)
- Git
- IDE (e.g. Visual Studio Code)

Unzip the learning materials given by your trainer.

Training on Strigo

Strigo Lab provides a Windows VM with the following functional environment:

- Node.js
- NPM
- Git
- Visual Studio Code (`"C:\Programs Files\Microsoft VS Code"`)

Visual Studio Code Extensions

If you use VSCode as your IDE, install the following extensions in addition:

- Angular Language Service
- Auto Rename Tag (optional)
- Github Theme (optional)
- vscode-icons (optional)

Version control system

Note: to use "Git Credential Manager", you need to restart the Windows VM once all the programs have been installed.

- Open the browser and login 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 name and email:

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

Creating and running your Angular application

This app will be used along all labs.

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

```
npm i -g @angular/cli
ng new zenika-ng-website --standalone false
```

You will be displayed some options for your app. Choose "No routing" and "CSS" as CSS preprocessor.

If you can't install the Angular CLI globally, create your app with one of the following shell commands

```
npm init @angular zenika-ng-website
```

or:

```
npx @angular/cli new zenika-ng-website
```

In this case, to run an Angular CLI command, you will have 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 a placeholder content. 🚀

Taking control of your application

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

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

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

- Add some style in `src/app/app.component.css`:

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

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

```
class AppComponent {
  title = 'my first component';
}
```

- Check that the application has been updated correctly in the browser. 🚀

Now let's try running the application tests

```
ng test
```

or:

```
npm test
```

Because we've modified the application, the tests in `app.component.spec.ts` fail.

- Fix the test on property `title`
- Fix the test on tag `h1`

Finally let's build the application for production

```
ng build
```

- Open a shell window in `dist/zenika-ng-website/` folder and run the command:

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

- Open the browser at the URL specified in the console

Synchronize your repository

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
```

Lab 2: Components

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

The design team have been working hard, and the result is available in the `Exercises/resources/design` folder. You're going to integrate this design into your Angular application.

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

- Open a new shell window in the folder `design` and run the command:

```
npx serve .
```

- Open Chrome and visit: `http://127.0.0.1:3000/`. You should see the 4 products available in the catalog.
- Next, copy/paste the content of `design/assets` into `src/assets`
- Finally, open the file `design/index.html` in your editor and follow the detailed instructions below

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 tag `<body> <!-- ONLY WHAT'S INSIDE --> </body>` to `src/app/app.component.html`

- Before continuing, serve your app using `ng serve` to see if the result is equivalent to that of the designers

Creating the "menu" component

- Create a menu component with the shell command `ng generate component menu` and move the corresponding code into it

- Once done, add the component `<app-menu />` to `src/app/app.component.html`

Creating the "product" component

- Create a product component with the shell command `ng g c product` and move the corresponding code into it
- Add a file `product.types.ts` in the same folder (`src/app/product/`) and define the product interface

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

- The component should accept:
  - an input: `@Input() product!: Product;`
  - an output: `@Output() addToBasket = new EventEmitter<Product>();`
- Use the properties of the `product` object in the template to display the `title`, `description`, ...

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

- The event emitter should emit the product when the user clicks on the button "Ajoutez au panier"

Storing all products in the `AppComponent`

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

- In `src/app/app.component.ts`, define a `products: Product[] = [];` property
- Fill the array with the content of the file `Exercises/design/products.json`
- In `src/app/app.component.html`, use the component `<app-product />` instead of each hard-coded product (later in the training, we'll use a "for" loop to achieve this)

```
<app-product [product]="products[0]" />
```

- Define a `total = 0;` property that should be updated each time the user clicks on the button "Ajoutez au panier"

Lab 3: Unit testing

In this lab, you will implement the tests for the app you developed in the "**Lab 2: Components**".

The `MenuComponent` don't need to be tested, since it have no logic.

You're going to focus on the `ProductComponent` and the `AppComponent`.

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

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

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

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [AppComponent],
    });
    fixture = TestBed.createComponent(AppComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

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

- Run the tests using Angular CLI:

```
ng test
```

Some tests fail. Let's fix them!

`product.component.spec.ts`

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

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

- In the `beforeEach` section, define the required `product` property:

```
component.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 `EventEmitter` to check that it is called

`app.component.spec.ts`

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

This component depends on 2 other components:

- `MenuComponent`
- `ProductComponent`

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

- First approach - **with explicit dependency declaration**
- 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 4: Directives

In this lab, you'll use the `ngClass`, `ngFor` and `ngIf` directives to improve the application's logic.

`ProductComponent`

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

`AppComponent`

- Update `addToBasket` method to decrease the product stock when user clicks *"Ajouter au panier"*
- Add a getter `get hasProductsInStock(): boolean` that returns `true` when at least one product has a stock greater than 0
- Use `ngFor` directive to iterate over the `products` arrays to display each `<app-product />` component
- Use `ngIf` directive to display only the products with a `stock` greater than 0
  - Remember you can't have 2 strutural directives on the same element
  - Use `<ng-container>` to get around the problem
- Use `*ngIf="hasProductsInStock"; else...` statement to display the message *"Désolé, notre stock est vide !"* when there's no product left in the catalog

Tests

`product.component.spec.ts`

- 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

`app.component.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 5: Services

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

You need to create 2 services using Angular CLI:

- `src/app/catalog/catalog.service.ts` : to manage the products
- `src/app/basket/basket.service.ts` : to manage the basket items

`CatalogService`

The service should have:

- A `_products: Product[]` private property (move here the 4 products defined in `app.component.ts`)
- A `get products(): Product[]` getter that returns the `_products` property
- A `get hasProductsInStock(): boolean` getter that returns `true` if at least one product stock is greater than 0
- A `decreaseStock(productId: string): boolean` method to decrease the corresponding product stock if it is greater than 0

`BasketService`

- Define a new interface:

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

The service should have:

- A `_items: BasketItem[]` private property
- A `get items(): BasketItem[]` getter that returns the `_items` property
- A `get total(): number` getter that returns the basket total (derived from `_items`)
- A `addItem(item: BasketItem): void` method that add an item to the basket

Use of services

- `MenuComponent` : use the `BasketService` to display the number of items in the basket
- `AppComponent` : refactor the component to use both the `CatalogService` and `BasketService` services

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 `AppComponent` to display the app title

Tests

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

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

```
xdescribe('AppComponent', () => { /* ... */ });
```

`catalog.service.spec.ts`

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

`basket.service.spec.ts`

- It should update the items when a product is added
- It should update the total when a product is added

`menu.component.spec.ts`

The component now depends on the newly created `BasketService`.

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 BasketService {}
```

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

- Create a minimalist class called `BasketStubService` that will replace the `BasketService`

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

- Provide the stub in `menu.component.spec.ts`

Add test:

- It should display the number of items

`app.component.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 `AppComponent` 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 `CatalogStubService` that will replace the `CatalogService` (like you did above for the `BasketService`)
- Provide the 2 stubs in `app.component.spec.ts`
- Provide a value for `APP_TITLE` injection token
- Fix the remaining tests

Add new, more relevant tests:

- It should call "CatalogService.decreaseStock" and "BasketService.addItem" methods when a product is added to the basket
  - For that use `TestBed.inject` function (to get the services instances) and `spyOn` Jasmine function (to spy on these methods)
- It should display the app title

Lab 6: Pipes

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

`ProductComponent`

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!

- First, register the `"fr"` locale in your application

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

registerLocaleData(localeFr);
```

- Next, provide `LOCALE_ID` and `DEFAULT_CURRENCY_CODE` in `AppModule`

```
// src/app.app.module.ts
import { DEFAULT_CURRENCY_CODE, LOCALE_ID, NgModule } from '@angular/core';

@NgModule({
  providers: [
    { provide: LOCALE_ID, useValue: 'fr' },
    { provide: DEFAULT_CURRENCY_CODE, useValue: 'EUR' },
  ],
})
export class AppModule {}
```

The product price should now be displayed correctly.

`AppComponent`

- 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 your finished, use your pipe to sort the products in the `AppComponent` template

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

- Copy/paste the component `Exercises/resources/select-product-key` into your app at `src/app/select-product-key`
- Declare the component in `AppModule`

Use the component:

- Add `productKey` in `app.component.ts`

```
import { Component } from '@angular/core';
import { SelectProductKey } from './select-product-key/select-product-key.types';

@Component({ /* ... */ })
export class AppComponent {
  productKey: SelectProductKey = undefined;
}
```

- Use `<app-select-product-key>` in `app.component.html`

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

Tests

`app.component.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.component.spec.ts`

- It should display product title in uppercase
- It should display product price with currency

Lab 7: Http

PART 1

To begin with, you will learn how to send http requests from your Angular app to a web server.

Communicate with an http server

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 shell window in the `Exercises/resources/server` folder 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
- `GET /products/:productId` to fetch one product
- `GET /basket` to fetch the basket
- `POST /basket` with a request body of type `{ productId: string; }` to add a new item to the basket

`AppModule`

- Import the `HttpClientModule`

`CatalogService`

- Remove the default products in the `_products` property - it should now default to `[]`.
- Inject the `HttpClient` service
- Add a method `fetchProducts(): void` that queries the server for the products and stock the products response in `_products`

`AppComponent`

- Call the `catalogService.fetchProducts` method to fetch products in `ngOnInit`

Everything should be compiling at this point, verify

- your app still display the products
- you see a http request to `/products` in the devtools

`BasketService`

- Inject the `HttpClient` service
- Update the `addItem` method to save the added item on the server, it should have the following signature `addItem(productId: string): void`. Once saved, push the added product to `_items`.

`AppComponent`

- Update the method `addToBasket` to use the new signature of the `basketService.addItem` method

At this point your app should

- display the products again by fetching them from the server
- add a product to the basket kept in the server when clicking the 'add to basket' button (use the devtools again to verify the http call is working)

PART 2

You now have succeeded in requesting data from the server. But there is still room for improvements. The main problem in the current app is that the components have no idea when the http calls are finished.

Why is it a problem ?
In some cases (like when you fetch the product list) even if you don't know when the fetch finishes, it still works correctly as Angular detects it and updates your view. But what about when you have to explicitly trigger some code after a http call ?

For example, when you want to decrease the stock of the product after adding it to the basket ? Currently it is not working properly in the application: you decrease the stock before even knowing if the product was correctly added on the server.

This is one of the reason why it is considered a good practice to **always expose the Observable you create** (unless you have a very good reason not to do so). You never know if your caller (the part of the app that called the method) might need to wait for the http call (of whatever process represented through your Observable) to finish before doing something.

So let's improve the app.

Expose observables and use operators

`BasketService`

- Change the signature of the `addItem` method to:
  `addItem(productId: string): Observable<BasketItem>`
- In the `addItem` method, returns your Observable and use the `tap` operator to add the basket item to the property `_items` once the http called to our server is finished.

`CatalogService`

- Change the signature of the `fetchProducts` method to:
  `fetchProducts(): Observable<Product[]>`
- In the `fetchProducts` method, returns your Observable and use the `tap` operator to add the received products in the `_products` property.

`AppComponent`

- In the `addToBasket` method, subscribe to the Observable returned from `basketService.addItem` and decrease the stock of your product after it has been added to your basket.
- In the `ngOnInit` method, subscribe to the Observable returned from `fetchProducts`, do you understand why ?

Everything in your app should be working correctly at this point.

PART 3

For this last part, you will fix the last problem on your app with a bit less guidance to train on more realistic conditions 🙂

Before reading any further, you can try to found the remaining problem and fix it by yourself: it is a functional problem.

When you refresh your application, there is still two problems :

- you don't display the correct basket total
- you don't display the correct number of items in your basket

Hint : take a look at the `BasketService` and the `CatalogService`. You manage two data on your app: a catalog of products and a basket. When do you fetch the basket ?

Lab 8: Router

In this lab, you'll create a multi-page website (SPA) using the router provided by Angular.

`AppRoutingModule`

- Create the file `src/app/app-routing.module.ts` with the following content:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

- Declare the `AppRoutingModule` in your application
- Create the following components and declare a route for each one of them:
  - Component: `CatalogComponent` --> Route: `'catalog'`
  - Component: `BasketComponent` --> Route: `'basket'`
  - Component: `ProductDetailsComponent` --> Route: `'product/:id'`
- Add a route `**` that redirects to the `'catalog'`

`CatalogComponent`

- Move the main content you have developed in the `AppComponent` to this one, including:
  - the template
  - the class logic
  - the tests

`AppComponent`

- In `src/app/app.component.html`, put a `<router-outlet />` 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>
```

`RouterLink`

Add `routerLink` directives in the following templates:

- In `catalog.component.html`:
  - to visit the page *"Voir mon panier"*
- In `menu.component.html`:
  - to return the home page when clicking on *"Zenika Ecommerce"*
  - to visit the page *"Voir mon panier"*
- In `product.component.html`:
  - to visit the product details page

`BasketComponent`

- Use the following markup for the component template:

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

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

  <ul class="list-group list-group-flush">
    <!-- Use `*ngFor` directive 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 `*ngFor` directive 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 `BasketService` to implement the component logic
- Use `OnInit` lifecycle hook to dispatch the basket items
- 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

`BasketGuard`

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

- If there are items in the basket, the `BasketComponent` should be displayed
- It the basket is empty, an alternate `BasketEmptyComponent` should be displayed

Let's do this!

- Generate a `CanMatch` guard in `src/app/basket/basket.guard.ts` (from Angular 16, this will generate a functional guard)
- Use the `inject` function to inject your dependencies (instead of the `constructor` technique)

```
import { inject } from '@angular/core'; // <-- Warning: this is not `Inject`
import { CanMatchFn } from '@angular/router';
import { BasketService } from './basket.service';

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

- Declare the guard in the appropriate route
- Generate a new component `BasketEmptyComponent`
  - It simply displays *"Votre panier est vide."*
- Add the route `'basket'` to display the component
  - Yes, it's the same route as for the `BasketComponent`

`ProductDetailsComponent`

Remember the route for this component is `'product/:id'`.

- Retrieve the `:id` from the `ActivatedRoute` snapshot
- Fetch the product from the API using the `ApiService`:
  - `http://localhost:8080/api/product/:id`
- Use the following code snippet to display the product details:

```
<div class="row" *ngIf="product">
  <div class="col-12 col-md-4">
    <img
      [src]="product.photo"
      class="mb-3 mb-md-0 w-100 rounded-3" alt="Product image"
    />
  </div>

  <div class="col-12 col-md-8">
    <h1>{{ product.title | uppercase }}</h1>

    <div class="table-responsive-sm">
      <table class="table caption-top">
        <caption>Détails du produit</caption>
        <tbody>
          <tr>
            <th scope="row">Description</th>
            <td style="min-width: 280px">{{ product.description }}</td>
          </tr>

          <tr>
            <th scope="row">Stock disponible</th>
            <td>{{ product.stock }}</td>
          </tr>

          <tr>
            <th scope="row">Prix</th>
            <td>{{ product.price | currency }}</td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>
```

Bonus: Application performances (1/2)

At this point, take a look at the number of requests to the API in the Network tab of Chrome's developer tools.

- Each time you visit the `"catalog"` page, 2 requests are sent to fetch the catalog and basket items.
- Each time you visit the `"basket"` page, 1 request is sent to fetch the basket items

You can improve this behavior and ensure that requests are made only once. To do this, you need to update the `fetchProducts` and `fetchItems` methods.

- Here's an example for the `fetchProducts` method:

```
import { of } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class CatalogService {
  private _products?: Product[];

  get products(): Product[] | undefined {
    return this._products;
  }

  get hasProductsInStock(): boolean | undefined {
    return this._products?.some(({ stock }) => stock > 0);
  }

  /**
   *  @param refresh
   *    Fetch the products from the API server even if they
   *    have already been fetched and stored in the service
   */
  fetchProducts(refresh = false) {
    if (!refresh && this._products) {
      return of(this._products);
    }
    return this.httpClient
      .get<Product[]>('http://localhost:8080/api/products')
      .pipe(tap((products) => (this._products = products)));
  }
}
```

Bonus: Application performances (2/2)

Also, 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.component.html` template, wrap the content with `<ng-container *ngIf="products">` like this:

```
<ng-container *ngIf="products">
  <div *ngIf="hasProductsInStock; else stockEmpty">
    <ng-container *ngFor="let product of products">...</ng-container>
  </div>

  <ng-template #stockEmpty>
    <p>Désolé, notre stock est vide !</p>
  </ng-template>
</ng-container>
```

Lab 9: Forms

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

- You need first to import the forms module in your application

```
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [FormsModule],
})
export class AppModule {}
```

- Next, generate a new component using Angular CLI:
  - `src/app/checkout-form/checkout-form.component.ts`
- For the component template, copy/paste the design made with love by the UI/UX team:
  - `Exercises/resources/checkout-form/checkout-form.component.html`
- Insert the component at the end of the basket component template:
  - `<app-checkout-form />`
- Create a new interface that describes the shape of the checkout form:

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

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 appearence:
  - 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":
  - Use `*ngIf` directive to display only the relevant error

Form submission

- In the component class, create 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
  - Use the `checkoutForm.value` property as `checkout` method argument
- 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 this, add a new property `checkoutInProgress: boolean` in the component class)
- In the `ApiService`, add a new method:
  - `checkoutBasket(checkoutDetails: CheckoutDetails): Observable<{ orderNumber: number }>`
  - that should call the endpoint `POST /basket/checkout` with a request body of type `CheckoutDetails`
- In the `BasketService`, add a new method:
  - `clearBasket(): void` that should empty the basket items

We now have everything we need to implement the `checkout` method in the checkout form component class. Let's do it!

- Subscribe to the observable returned by the `ApiService.checkoutBasket` method and handle "next" and "error" events.
- On **"next"**:
  - Once the checkout is successful, call the method `BasketService.clearBasket`
  - Display the "success" message with the `orderNumber`
  - Add a `routerLink` directive to navigate when clicking on the link *"Retourner à l'accueil"*
  - In this case, the form fields must remain disabled
- On **"error"**:
  - Display the "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 folder to see the `ReactiveFormsModule` implementation:

- `Exercises/solutions/projects/09_forms/src/app/checkout-reactive-form/`