Advanced Angular

Labs

2023-11-19#61f849d

Table of contents

Prerequisites

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
- ESLint
- Prettier - Code formatter
- Auto Rename Tag (optional)
- Github Theme (optional)
- vscode-icons (optional)

Version control system

Note: to use "Git Credential Manager", you might 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 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>
```

Lab 1: Setting up your environment

In this lab you will:

- Create and launch your Angular client app
- Launch an Express server exposing a Rest API that your Angular app will consume
- Synchronize your local repository with the remote one

Your Angular client app

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
```

You will be displayed some options for your app. Choose "Add routing" and "SCSS" 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>`.

In the generated app, replace the `src` directory by the one given by your trainer

Local:

- Copy `src` directory from "Exercises/resources" directory given by your trainer

Strigo:

- Copy `src` directory from: `C:\Users\Administrator\Desktop\Exercises\resources`

Add Boostrap v5.3.0 to the dependencies of your app

```
npm install bootstrap@5.3.0 --save-exact
```

Update the `angular.json` file by adding `bootstrap.min.css` to the `styles` array

Apply this change to the `build` and `test` sections. Example for the `build` section:

```
{
  "architect": {
    "build": {
      "options": {
        "styles": [
          "node_modules/bootstrap/dist/css/bootstrap.min.css",
          "src/styles.scss"
        ]
      }
    }
  }
}
```

Run the Angular dev server

Local / Strigo:

```
ng serve
```

or:

```
npm start
```

The Express server exposing a Rest API

Open the `server` directory in a separate terminal

Local:

- Open `server` directory from "Exercises/resources" directory given by your trainer

Strigo:

- Open `server` directory from: `C:\Users\Administrator\Desktop\Exercises\resources`

Run the following commands

```
npm install
npm start
```

Check that everything is up and running

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

You should see the catalog of the "Zenika Ecommerce" app. 🚀

Take some time to play with the app: add items to the basket and navigate between the catalog, the product and the basket pages. Also, take a tour at the source code to familiarize yourself with its structure. Ask the trainer questions if there are parts of the app that are not clear.

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: add ESLint and install Prettier

- Run the following commands:

```
ng add @angular-eslint/schematics
npm i -D prettier eslint-config-prettier prettier-plugin-organize-imports
```

- Add `"prettier"` to the `"extends"` array in your `.eslintrc.json` file

Make sure to put it last, so it gets the chance to override other configs:

```
{
  "root": true,
  "ignorePatterns": [
    "projects/**/*"
  ],
  "extends": [
    "prettier"
  ],
  "override": [...]
}
```

- Add `.prettierrc.json` file at the root of your workspace:

```
{
  "printWidth": 120,
  "singleQuote": true,
  "plugins": ["prettier-plugin-organize-imports"]
}
```

- Add `.prettierignore` file at the root of your workspace:

```
/.angular
/dist
```

- Add scripts in your `package.json` to run Prettier:

```
{
  "scripts": {
    "lint": "ng lint",
    "format": "prettier --write .",
    "format:check": "prettier --check ."
  }
}
```

- Run prettier:

```
npm run format
```

Lab 3: Best practices

As you may have noticed, the design of the app is very simple. Every component that needs a data, gets it from the server. Therefore, the `basket` endpoint is called twice: for the **basket total** and the **item counter** in the menu component!

Also note that when you add a product to the basket, the item counter **only updates when you refresh the page**!

To improve the design and performance of the app, you will create services to encapsulate the server data. Then you will gradually move all the operational code into these services.

Note: The app is in french but the currency is not well displayed: `€0.00` instead of `0,00 €`. This will be fixed in the Lab on i18n.

Here is the list of tasks to be performed:

- Use an `InjectionToken` to provide the `WELCOME_MSG`
- Move the product component inside the catalog folder because it is only used by the catalog component
- **Create a basket service with:**
  - an `items: BasketItem[]` property
  - a `total(): number` getter that derives from the `items` property to calculate the basket price
  - a `numberOfItems(): number` getter that derives from the `items` property
  - a `fetch(): Observable<BasketItem[]>` method that fetch the basket from the server and set the `items` property on the client
  - a `addItem(productId: string): Observable<BasketItem>` method that post a request to the server and update the `items` property on the client
  - a `checkout(customer: Customer): Observable<{ orderNumber: number }>` method that post a request to the server and empty the `items` property on the client
- Use the basket service in the the "basket", "catalog" and "menu" components
- **Create a catalog service with:**
  - a `products: Product[]` property
  - a `isStockEmpty(): boolean` getter
  - a `fetch(): Observable<Product[]>` method that fetch the products from the server and set the `products` property on the client

    - a `decreaseStock(productId: string): void` method that update the product stock on the client (note that the method signature differs from the original)
    - a `isAvailable(product: Product): boolean` method

- Finally, clean the operational code from the "basket", "catalog" and "menu" components and use the catalog services instead
- In the different components, implements the `OnInit` lifecycle hook to fetch the data (instead of doing this in the class `constructor`).
- Use the `inject` function to provide all the dependencies (instead of doing this in the class `constructor`)

Tips

- To generate a service, use the Angular CLI:

```
ng generate service <serviceName>
```

or:

```
ng g s <serviceName>
```

- To fetch data from the server and set a service property, use the `tap` operator from RxJS

Replace:

```
class MyComponent {
  myItems: Item[] = [];
  fetchMyItems(): Observable<Item[]> {
    return this.apiService.getMyItems().subscribe(
      (myItems) => (this.myItems = myItems)
    );
  }
}
```

With:

```
class MyItemsService {
  myItems: Item[] = [];
  fetch(): Observable<Item[]> {
    return this.apiService.getMyItems().pipe( // use `pipe()`...
      tap((myItems) => (this.myItems = myItems)) // ...and `tap()`
    );
  }
}
```

So, remember not to `.subscribe()` in your services. Instead use `.pipe()`.

Lab 4: RxJS

In this lab, you will make the app truly reactive.

- In the `CatalogService`, expose the products property using `BehaviorSubject`:
  - Use the JavaScript `#` private field instead of the TypeScript `private` scope
  - `#products$ = new BehaviorSubject<Product[]>([]);`
  - `products$ = this.#products$.asObservable();`
  - Expose the `isStockEmpty$` as observable that derive from the `#products$`
- Start using the `async` pipe in the templates to subscribe to the different observables
- In the `BasketService`, expose the items property using `BehaviorSubject`:
  - Use the JavaScript `#` private field instead of the TypeScript `private` scope
  - `#basket$ = new BehaviorSubject<BasketItem[]>([]);`
  - `basket$ = this.#basket$.asObservable();`
  - Expose the `total$` and `numberOfItems$` as observables that derive from the `#items$`
- Continue using the `async` pipe in the templates to subscribe to the different observables
- Switch all components to use the `OnPush` strategy for change detection
- In the `ProductDetailsComponent`
  - The page should no longer work properly
  - Use `ChangeDetectorRef` to refresh the UI when the HTTP request responds
- Copy the `alert` folder given by your trainer into `src/app/alert`, declare the `AlertComponent` in `AppModule` and add `<app-alert />` to the `AppComponent` template
- Catch errors and use the `AlertService` to display an error message to the user when:
  - fetching data in the `CatalogComponent`
  - adding an item to the basket
  - fetching data in the `BasketComponent`
  - the user checkouts the basket

Tips

To catch errors in observables, use the `error:` property when subscribing:

```
import { EMPTY } from "rxjs";

myObservable$.subscribe({
  error: () => {
    console.log("Oops!"); // Use the `AlertService` instead
    return EMPTY;
  },
});
```

Or use the `catchError()` operator in the observable pipeline:

```
import { catchError, EMPTY } from "rxjs";

myObservable$.pipe(
  catchError(() => {
    console.log("Oops!"); // Use the `AlertService` instead
    return EMPTY;
  })
);
```

Lab 5: Standalone Components

In this lab, you will convert all components of the app into standalone components directly. You will then remove the `AppModule` and bootstrap the `AppComponent`.

Components

- Convert one or two components manually. Try to import only the required functionality from `@angular/common` instead of the entire `CommonModule`

Here's an example where the template only uses the `NgIf` directive:

```
import { NgIf } from '@angular/common';
import { Component, Input } from '@angular/core';

@Component({
  standalone: true,
  selector: 'app-hi',
  imports: [NgIf], // <-- Instead of `imports: [CommonModule]`,
  template: '<h1 *ngIf="isVisible">Hi!</h1>',
})
export class HiComponent {
  @Input() isVisible = true;
}
```

- For the other components, run the schematic

```
ng generate @angular/core:standalone
```

You will be displayed some options. Choose "Convert all components, directives and pipes to standalone"

Bootstrapping

You can do it manually or run the schematic again and choose the other options to complete the migration.

If you use the schematic, you will notice that the result is not perfect. Finalize the migration manually.

- Bootstrap the app using standalone architecture
- Create a `src/app/app.routes.ts` file containing:
  `export const appRoutes: Routes = [...];`
- Use `provideRouter(appRoutes)` instead of
  `importProvidersFrom(BrowserModule, AppRoutingModule)`
- Create a `src/app/app.config.ts` file that exports the app configuration

Replace:

```
import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule)
```

With:

```
import { AppModule } from './app/app.module';
import { appConfig } from './app/app.config'; // <-- File to create

bootstrapApplication(AppComponent, appConfig);
```

Tips

The `app.config.ts` file should export something like this:

```
import { ApplicationConfig } from '@angular/core';

export const appConfig: ApplicationConfig = {
  providers: [],
};
```

Lab 6: Router

In this lab, you will address the following problems

Usually, e-commerce websites are composed of a catalog and a purchase section. Most of your visitors will only consume the catalog section. In this scenario, it doesn't make sense to make them load the purchase part of your app for no reason.

When a user visits an unknown route, the app displays a blank page without explaining what is happening.

Have you noticed that when you visit the catalog, it displays *"Sorry, our stock is empty"* for a short time? This is because the products cannot be displayed until we fetch them from the server.

Here is the list of tasks to be performed

- Lazy load components using `loadComponent` in the `Routes` configuration
- Create a `NotFoundComponent` and display it when the user visits an unknown route
- Create a `catalogResolver` for the catalog route and fetch the data from there (remove this logic from the `CatalogComponent`)
- Create a `basketGuard` for the basket route and prevent access to the page when the basket is empty
- Create a `BasketEmptyComponent` and use it as an alternative when access to the basket has been denied by the `basketGuard`
- Change the strategy to `PreloadAllModules` and see the changes in your netwook devtools

Bonus

- Add a resolver for the `ProductDetailsComponent` and fetch the data from there
- Then bind the resolved value to the `product: Product` property of the `ProductDetailsComponent`

To achieve this, use the feature `withComponentInputBinding` when providing the Router:

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter, withComponentInputBinding } from '@angular/router';
import { appRoutes } from './app.routes';
export const appConfig: ApplicationConfig = {
  providers: [provideRouter(appRoutes, withComponentInputBinding())],
};
```

Lab 7: Forms

The goal of this lab is to create a reactive form for a better user experience on the basket page.

- Create a `BasketFormComponent` inside the `./basket` folder and move all the form related code to it
- Use the `FormBuilder` to create the expected form fields:
  - `name` : required
  - `address` : required
  - `creditCard` : required and should match the pattern `/^\d{3}-\d{3}$/` (ex: 123-456)
- Display error messages below each field when touched and invalid
- Disable the submit button while the form is invalid and while the form submission

Lab 8: i18n

Currently, the app is only available in French. The goal of this lab is to create an additional version of the app in English.

- Run the schematic `@angular/localize`

```
ng add @angular/localize
```

- Update `angular.json` file:
  - define the `sourceLocale` property
  - add location of the translation file for the English version
  - tell Angular to build the app for every language
  - tell Angular to run the dev server using the English version

```
{
  "i18n": {
    "sourceLocale": "fr",
    "locales": {
      "en-US": {
        "translation": "src/locales/messages.en-US.json"
      }
    }
  },
  "architect": {
    "build": {
      "options": {
        "localize": true,
      },
      "configurations": {
        "development": {
          "localize": ["en-US"],
        }
      }
    }
  }
}
```

- Add `i18n` attributes in the catalog and product component templates to prepare them for translation
- Use `$localize` function if you need to prepare a string for translation in a component class

- Extract source translation file using **json** format

```
ng extract-i18n --format json --output-path src/locales --out-file
messages.fr.json
```

- Create a copy of the source translation file and name it `messages.en-US.json`
- Translate the French sentences in English
- Run the dev server to view the app in English
- Build the app

```
ng build
```

You should now have a generated varient of the app for english and french inside the folder: `dist/zenika-ng-website`

- Start a web server to serve the built apps

```
npx http-server dist/zenika-ng-website/
```

Bonus

Add an `index.html` for auto-language detection

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Redirection | Zenika Ecommerce</title>
    <script>
      const supportedLocales = ['en-US', 'fr'];
      const defaultLocale = supportedLocales[0];
      const locale = window.navigator.languages.find(
        (language) => supportedLocales.includes(language)
      );
      window.location.assign(`./${locale ?? defaultLocale}`);
    </script>
  </head>
  <body></body>
</html>
```

Lab 9: Universal

Follow the steps in the slides to have a node.js server that render your app on server side.

To add Server Side rendering to the app, run the command:

```
ng add @angular/ssr
```

Warning: At the time of writing, the schematics does not generate a proper content for the `server.ts` .

Everything should work as intended if you copy the file content from the following solution:

```
Exercises/solutions/projects/09_universal/server.ts
```

Lab 10: Animations

In this lab, you will add some cool animations to your amazing app using both CSS and Angular animations techniques

- In the `ProductComponent` add CSS transition when the product color changes

```
.app-product {
  transition:
    border-color ease 750ms,
    background-color ease 750ms,
    color ease 750ms;
}
```

- In the `CatalogComponent` add 2 Angular animations:
  - when the product becomes the last one in the stock
  - when the product stock is out (and the product is about to be removed from the UI)

Feel free to create beautiful animations of your choice 🍭

Lab 11: Material

In this lab, you will use some of the Angular Material components. Since you are using Bootstrap as your main CSS framework, you will configure Material to work well with it.

Making Bootstrap and Material work well together

Install Material

- Run the schematic `@angular/material`

```
ng add @angular/material
```

You will be displayed some options. Choose "Custom" for theme, "No" for typography and "Include and enable animations".

Note that `src/styles.scss` file has been updated by the schematic.

- Move the generated code into a new file: `src/styles/material.scss` and import this file in the main `styles.scss` :

```
@import './styles/material.scss';
```

- Remove the following lines from the generated code:

```
html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }
```

- Remove the added link tags related to Roboto font-family from the `index.html` file:

```
<link rel="preconnect" href="https://fonts.gstatic.com">
<link href="https://fonts.googleapis.com/css2?family=..." rel="stylesheet">
<link href="https://fonts.googleapis.com/icon?family=..." rel="stylesheet">
```

Before continuing with the configuration of Material, let's change the way Bootstrap is imported.

Import Bootstrap from Scss

Currently you import Bootstrap via the `styles` array in `angular.json` file. You are going to import it directly in your main SCSS file.

- Remove `"node_modules/bootstrap/dist/css/bootstrap.min.css"` from styles array in `angular.json` file
- Create a file `src/styles/bootstrap.scss` and copy/paste to it the content of this file: `node_modules/bootstrap/scss/bootstrap.scss`

As an optimization, feel free to remove the Bootstrap features that are not used in your app

- import this new file in the main `styles.scss` before the import of Material:

```
@import './styles/bootstrap.scss';
@import './styles/material.scss';
```

- Create a file `src/styles/_bootstrap-configuration.scss` with the following content:

```
@import '../../node_modules/bootstrap/scss/functions';
@import '../../node_modules/bootstrap/scss/variables';
@import '../../node_modules/bootstrap/scss/variables-dark';
@import '../../node_modules/bootstrap/scss/maps';
@import '../../node_modules/bootstrap/scss/mixins';
@import '../../node_modules/bootstrap/scss/utilities';
```

This file does not output any CSS. Its purpose is to make Bootstrap variables and mixins available to Material.

Configure Material

As you may have noticed, Material use "Roboto" as the main font-family. You are going to configure it to use the same font-family as Bootstrap.

- Modify the file `src/styles/material.scss` you created earlier:

```
// Use Bootstrap configuration on top of the file
@use './bootstrap-configuration.scss' as bs;

// Set Material typography configuration to use `bs.$font-family-base`
$zenika-ng-website-theme: mat.define-light-theme(
  (
    color: (
      primary: $zenika-ng-website-primary,
      accent: $zenika-ng-website-accent,
      warn: $zenika-ng-website-warn,
    ),
    typography:
      mat.define-typography-config($font-family: bs.$font-family-base)
  )
);
```

- As an optimization only import the Material components you are using in your app

Replace:

```
@include mat.all-component-themes($zenika-ng-website-theme);
```

With:

```
@include mat.core-theme($zenika-ng-website-theme);

@include mat.badge-theme($zenika-ng-website-theme);
@include mat.button-theme($zenika-ng-website-theme);
@include mat.form-field-theme($zenika-ng-website-theme);
@include mat.toolbar-theme($zenika-ng-website-theme);
```

Using Angular Material components

- Replace the Bootstrap buttons of your app with `MatButtonModule`.
- In the `MenuComponent`
  - replace the Bootstrap navbar with `MatToolbarModule`
  - use `MatButtonModule` and `MatBadgeModule` to render the button "Voir mon panier"
- If you have time, use `MatInputModule` to render the fields in the `BasketFormComponent`

Lab 12: Unit Testing

In this lab you will add unit tests to your application. Run your actual tests through the Angular CLI. Since you've developed your app without worrying about testing, it's normal that it won't pass.

```
ng test
```

Setup

You will focus on the following tests:

- `product.component.spec.ts`
- `catalog.service.spec.ts`
- `catalog.component.spec.ts`
- `alert.service.spec.ts`
- `api.service.spec.ts`

Delete all other test files. Now, let's fix the 3 remaining tests that fail!

product.component.spec.ts

- The component is now standalone, so we need to move the `ProductComponent` from `declarations` to `imports` in the `TestBed` setup
- The component uses the `RouterLink` directive, so we need to import the `RouterTestingModule`
- The component requires its `product` property to be defined, so set `component.product = { ... };` before the call of `fixture.detectChanges();`

catalog.service.spec.ts

- The service depends on the `ApiService`, so we need to mock it: `providers: [{ provide: ApiService, useValue: MockApiService }]`

catalog.component.spec.ts

- Provide the `InjectionToken` for the welcome message
- The component does no longer depends on the `ApiService`, so you can safely remove `{ provide: ApiService, useValue: MockApiService }`
- You can also safely remove the `CUSTOM_ELEMENTS_SCHEMA`
- The component now depends on `CatalogService` and `BasketService`, use the mocks given by your trainer to provide these 2 services

Local: Open `tests` directory from "Exercises/resources" directory given by your trainer

Strigo: Open `tests` directory from:
`C:\Users\Administrator\Desktop\Exercises\resources`

Copy/paste each mock into the appropriate folder. Finally, provides the 2 services in the `TestBed` setup:

```
providers: [
  { provide: CatalogService, useValue: MockCatalogService },
  { provide: BasketService, useValue: MockBasketService },
],
```

Adding meaningful tests

Now all tests should pass. Let's add some real-life test scenarios!

product.component.spec.ts

- It should display all product details
- It should NOT display warning when product stock is greater than 1
- It should display warning when product stock is equal to 1
- It should emit product when clicking on the button

catalog.service.spec.ts

- It should store the products after fetching them
- It should decrease the product stock
- It should know that stock is empty

catalog.component.spec.ts

- It should display welcome message
- It should display total price with currency
- It should navigate to the basket view when clicking on "Go to basket" button
- It should display the products
- It should add product to basket when product is clicked

Bonus

alert.service.spec.ts

Take a look at the tests implemented in the solutions, they use the `fakeAsync` function:

`Exercises/solutions/projects/12_tests/src/app/alert/alert.service.spec.ts`

Lab 13: NgRx

In this lab you will use NgRx to store the customer details. When the user proceeds to the checkout, you need to store the customer data. Then, when they return to the basket page for a new purchase, the customer form should already be pre-filled.

- Add NgRx to your project:

```
ng add @ngrx/store
ng add @ngrx/effects
ng add @ngrx/store-devtools
ng add @ngrx/schematics
```

- Install the Chrome Extension: *Redux DevTools*

More infos: https://github.com/reduxjs/redux-devtools

- Create the basket feature:

```
ng generate feature shared/store/basket --flat=false --skip-tests
```

- In the `basket.actions.ts` add the `'Fill Customer'` action:

```
export const basketActions = createActionGroup({
  source: 'Basket',
  events: {
    'Fill Customer': props<{ customer: Customer }>(),
  }
});
```

- In the `basket.reducer.ts` handle the action you just created:

```
export interface State {
  customer: Customer | undefined;
}

export const reducer = createReducer(
  initialState,
  on(basketActions.fillCustomer, (state, { customer }): State => ({ ...state,
customer })),
);
```

- In the `basket.selectors.ts` add a selector for the customer

- Add the entry point `shared/store/index.ts`:

```
import * as fromBasket from './basket/basket.reducer';

export interface AppState {
  [fromBasket.basketFeatureKey]: fromBasket.State,
}

export const appReducers: ActionReducerMap<AppState> = {
  [fromBasket.basketFeatureKey]: fromBasket.reducer,
};
```

- Add the reducers in `app.config.ts`:

```
import { appReducers } from './shared/store';

export const appConfig: ApplicationConfig = {
  providers: [
    provideStore(appReducers),
  ],
};
```

- In the `basket-form.component.ts`:
  - In the `checkout()` method, dispatch the action:
    `this.#store.dispatch(basketActions.fillCustomer(...))`
  - In the `constructor()` method, select the filled customer (if defined):
    `this.#store.select(...).subscribe(...)`

Tip: to fill the form with the stored customer, use something like this:

```
this.#store
  // Select the customer from the Store
  .select(...)
  .pipe(
    first(),
    // Be sure the customer is not undefined
    filter((customer): customer is Customer => customer !== undefined)
  )
  .subscribe((customer) => {
    // Update the form value
    this.yourFormGroup.setValue(customer);
    this.yourFormGroup.updateValueAndValidity();
  });
```

Bonus

Migrate the basket items to the store. This time, you will need to use Effets.

- In the `basket.actions.ts` add the `'Fetch'` and `'Fetch Success'` actions:

```
export const basketActions = createActionGroup({
  source: 'Basket',
  events: {
    'Fetch': emptyProps(),
    'Fetch Success': props<{ items: BasketItem[] }>(),
  }
});
```

- Handle the `'Fetch'` action in the `basket.effects.ts`
- Handle the `'Fetch Success'` action in the `basket.reducer.ts`