

Cabify Challenge Frontend Alejandro Monteseirin

Configuration for development

- Node version 16.15.0 (At the moment of the development, is the last version of the node LTS branch). Installable files from <https://nodejs.org/es/download/>
- Run "npm install" in the folder where package.json is located. That will install all the required dependencies
- Run "ng serve" to start a development server in <http://localhost:4200/>

Package.json useful commands:

- ng serve : Run the development server in localhost:4200
- ng lint: run the linter
- ng build --prod: Compiles an Angular app into an output directory named dist/ using the production environment.
- ng test: run the unit test using karma jasmine
- ng e2e: run the end-to-end tests using cypress

Documentation

Analysis

Framework

The framework chosen for the development is Angular, in his last version 13.3.5 at the start of the development.

The reasons for choosing it are simply that Angular is the frontend framework I have the most experience with, and a result I was proud of would be delivered quicker.

Nevertheless, This doesn't mean I don't know how react.js or vue.js work, I'm just more fluent with Angular at the moment.

Ngrx-store

The Ngrx-store was the element that I doubt to add and just work with components interactions or a service for the state management, but seeing that the application have to be scalable and ready for production, I chose to analyzing the pros and cons.

- The cons of the store is the added layer of complexity, as you have to work with the actions and reducers, and take in account the immutability of the states. Also, it will have less performance than having the plane objects, because every state is a copy of the previous one.
 - The pros are a better scalability, more control of the states, the safety that every state is immutable, better debugging and more clean code. Also, as React is usually used with Redux, I think it will be a good point to have a more similar structure.
- After analyzing the pros and cons, I reach the conclusion to add it.

State properties of the store:

After doing a bit of analysis, the optimal solution for the values to save in the store was like this:

- indexedObjectProduct: { [id: string]: Product } :The Products indexed by code
- indexedObjectQuantity:{ [id: string]: number } : The quantity of product -indexed by Code
- indexedObjectRules:{ [id: string]: any } : The Rules indexed by Code of the product affected
- modalValue:Product |null : The value of the modal (Null dont show the modal, a Product Show a Modal with that product)

Mock api service:

Since we have data to input in the application that should not be hard-coded (products and promotions), we have created a small service that simulates a call to an API that returns the data. The service simply return a Promise that resolves after 800 milliseconds and return the data (to simulate a real API Call).

That allow us to easily connect the application to an API in a future, only replacing this service for a real one, as well this service can be use for testing proposes, as we have control of the data returned.

Models

Here below, we briefly explain each model created, they can be found in src\app\models

- Checkout Class and interface: Checkout class was implemented as a class that can be instantiated, it has all the properties needed to calculate the total cost and also the intermediate values (like discounted money, number of items...)
- The properties needed are: The Products, the discount rules, and the quantity of products.

You can instantiate the class with the values, or using the setters. As, the constructor has optional arguments, for example the OrderSummaryComponent instantiate the class empty and when the store returns the data the component use the setters, or if you want to use the scan() and total() methods, you can instantiate the class with the rules and products, and just start to use the scan method.

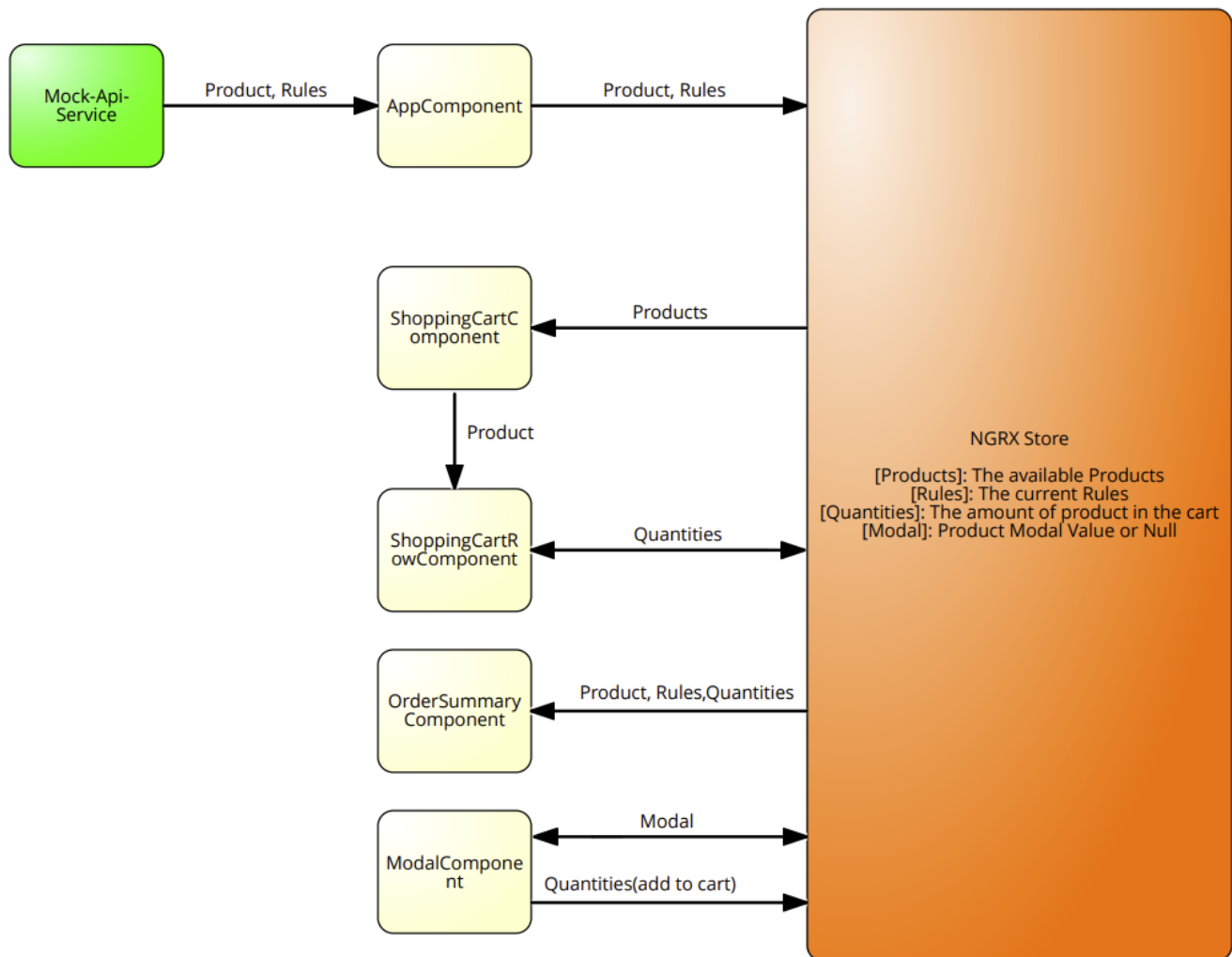
- Product class is a simple class that have the model of the product, with the properties "code", "name" and "price"
- OrderSummaryData is a simple class that have the model used for the OrderSummary Component, for readability and avoid have the properties disconnected. It has the properties showed in the component like "numberOfItems", "totalCost"...
- Discount class is a class with an only static method that receive the rules and the product code, price and quantity. It returns the discount applied or null if there is no discount. The discounts rules are configurable, and have a good performance (avoiding for loops for the rules and products). Moreover, you can create new rules like "5-for-3" or a new bulk discount without modifying the class (only adding the new rule to the Mock API). More info in the comment of the class.

Component Decomposition and Component Interaction Analysis

Here below, we briefly explain each component and his unique responsibility.

- AppComponent: The initialization of the Data, the mockApiService is called, and the products and rules are loaded in the store.
- ShoppingCartComponent: Is the "List" Component, Is subscribed to the store, and update the List when the products in the store change.
- ShoppingCartRowComponent: Show the data of a single product and its function is to manage the changes in quantity. Directly send the new quantity to the store.
- OrderSummaryComponent: Show the Order summary data, its changes when quantity of products are updated
- ModalComponent: Show the Modal view and allow the "add to cart" function, it changes when the modal value are updated

That's a simple image of all the component and they interaction

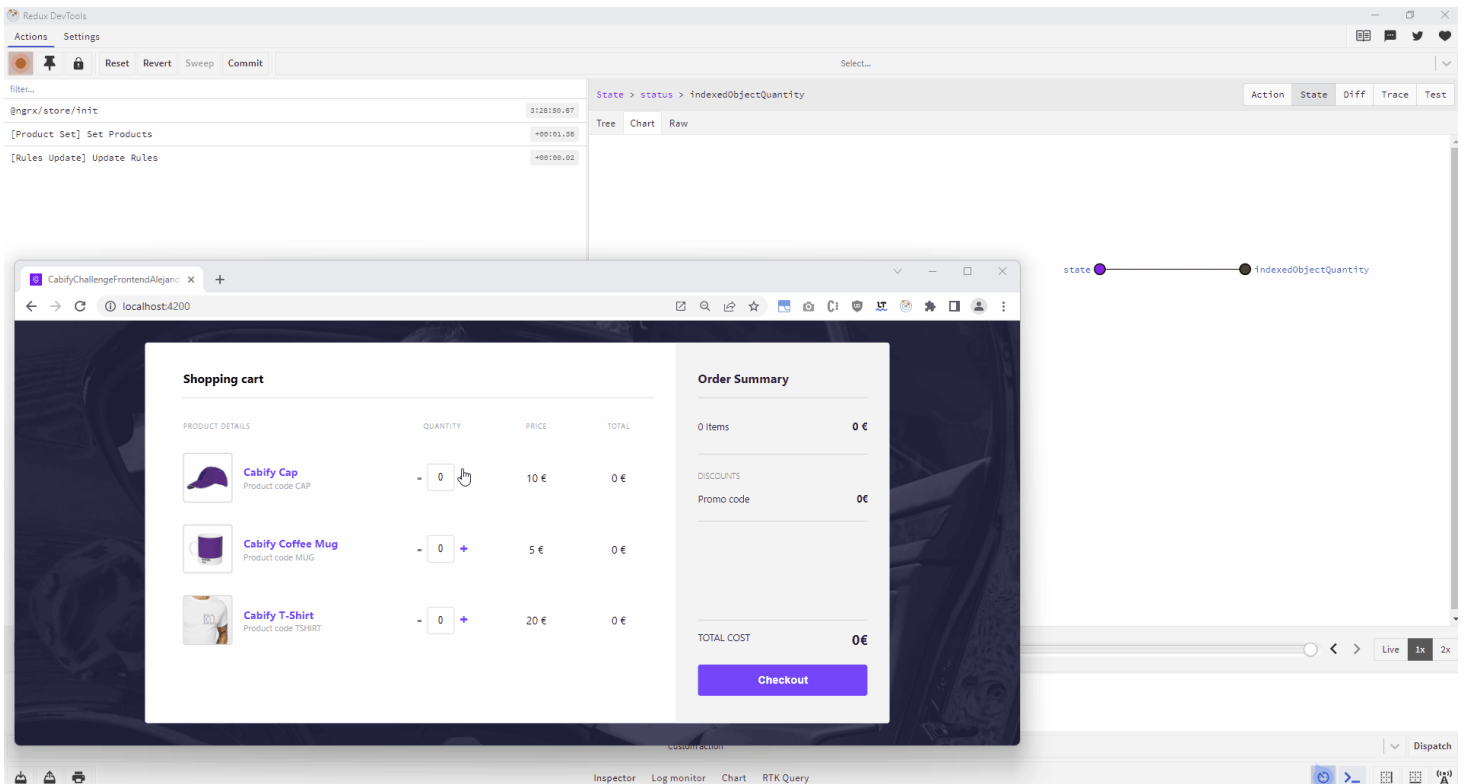


Extra Dependencies/Libraries:

A few extra dependencies/Libraries are configured in the project apart from the base ones. They will be automatically installed with the "npm install" command. The following is a brief list and explanation of each of them and why I have decided to add it:

- ESLint: For linting proposes "npm install eslint" <https://eslint.org/>
- @ngrx/store: for state management (Equivalent to Redux in React but for Angular)
- Karma Jasmine dependencies, for unit testing using the command "ng test"
- @cypress/schematic for end-to-end testing using the comand "ng e2e"
- @ngrx/store-devtools --save (for debugging and testing proposes) use <https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmбекрmknklioebfkpmmfbljd/related>

Below there are an example GIF of how the store devtools working



Testing

For Unit testing, we use Karma Jasmine and few test cases for covering the checkout class instantiation, the scan() function and the total() function. Also, We check that all components are correctly created.

The test can be found in the spec.ts file of every component and in the folder unit-test (src\app\unit-tests\checkout-class-tests.spec.ts)



For e2e Testing, we use Cypress and few tests for covering the most of the use cases, in this implementation we work with the example data that mock-api.service provides so it's not necessary to preload mock data. In the future the mock-api service will be use for cypress and a real API service for production, it can be configured using for example the environment variables.

The test can be found in CabifyChallengeFrontendAlejandroMonteseirin\cypress\integration\tests.ts

CabifyChallengeFrontendAlejandroMonteseirin

localhost:4200/.../tests/integration/tests.ts

Un software automatizado de pruebas está controlando Chrome.

< Tests 9 0 0 08:90

http://localhost:4200/ 1000 x 660 (100%)

cypress/integration/tests.ts

Check Page loaded

Visits the initial project page and check the title are loaded

Check Product loaded

Visits the initial project page and check the products are loaded

Check Order Summary loaded

Visits the initial project page and check the order summary component is loaded

Check Order Summary cart have 0 items

Visits the initial project page and check the order summary have 0 items

Add a cabify cap to the cart and check it is added to the order summary component

Visits the initial project page add a cup and check that is correctly added

Add 2 cabify Coffee mug and check if the promotion 2-in-1 is added

Visits the initial project page add 2 cabify coffe mug and check that is correctly added

Add the same items that in the example image and check if total is correct

Visits the initial project page, add 3 TSHIRT 4 MUG and 4 CUP, checkshould have a total of 107€




Check the modal open correctly

Visits the initial project page, click on cabify cap and check if modal opens

Check the modal add to cart button works

Visits the initial project page, click on cabify cap , open the modal and then push "add to cart", then check if quantity input changes and ordersummary is updated

Shopping cart

PRODUCT DETAILS	QUANTITY	PRICE	TOTAL
 Cabify Cap Product code CAP	- 1 +	10 €	10 €
 Cabify Coffee Mug Product code MUG	- 0 +	5 €	0 €
 Cabify T-Shirt Product code TSHIRT	- 0 +	20 €	0 €

Order Summary

1 Item 10 €

DISCOUNTS

Promo code 0€

TOTAL COST 10€

Checkout