telerik.com

Building an Online Store Using ngrx/store and Angular

16-20 minutes

In this tutorial, we'll build a simple store where items can be added and removed from cart, and we'll manage the application's state using ngrx/store. As we'll see, it is easier to manage data flow in the application when side effects and data flow are abstracted from components.

Managing an application is tasking, as the application grows to a never ending maze that requires a makeshift map to navigate. When applications grow to be that complex, managing data throughout the application becomes a major headache. This is where the importance of state management libraries like Redux, MobX and ngrx/store arises.

An important advantage of state management libraries in large-scale applications, especially hierarchical ones, is the ability to abstract the state of the application from components into an application-wide state. This way, data can be passed around with ease and components can act independently of each other.

For Angular, a great state management library is ngrx/store. This is an RxJS-powered state management library. It uses a

similar syntax to Redux: actions, reducers, stores, effects, and RxJS's reactive API.

In this tutorial, we'll be building a fruit store using Angular. In our small store, a user will be able to add and remove fruits from the cart. We'll also look at how we can use Effects for handling network requests, reducers and actions for data management. We'll be setting up a minimal server using Express that will serve products to the Angular application.

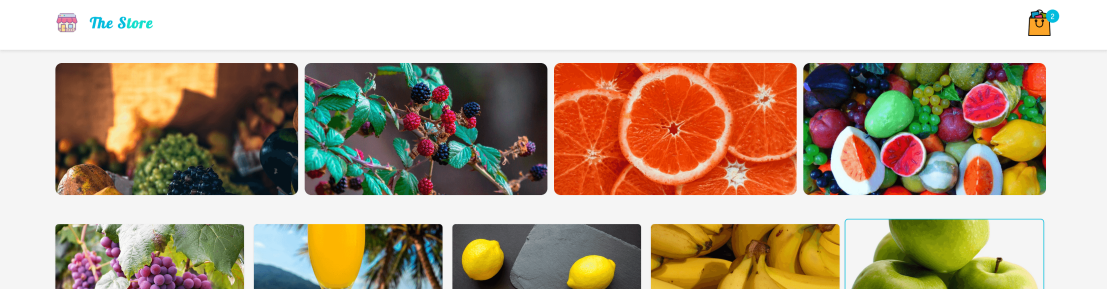
To follow this tutorial, a basic understanding of Angular and Node.js is required. Please ensure that you have Node and npm installed before you begin.

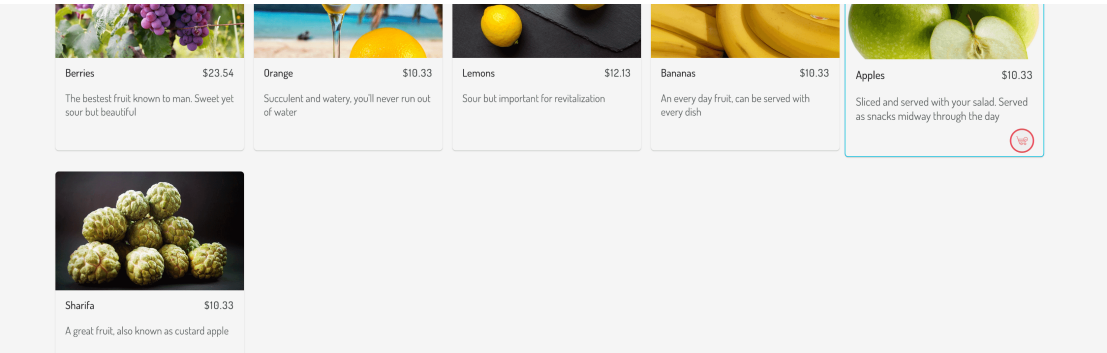
If you have no prior knowledge of Angular, kindly follow the tutorial here. Come back and finish this tutorial when you're done.

We'll be using these tools to build our application:

- Express
- Node
- Angular
- NgRx/store
- NgRx/effects

Here's a screenshot of the final product:





Initializing Application and Installing Dependencies

To get started, we will use the CLI (Command Line Interface) provided by the Angular team to initialize our project.

First, install the CLI by running `npm install -g @angular/cli`. npm is a package manager used for installing packages. It will be available on your PC if you have Node installed; if not, download Node here.

To create a new Angular project using the CLI, open a terminal and run:

```
ng new fruit-store --style=scss
```

This command is used to initialize a new Angular project; the project will be using SCSS as the pre-processor.

Next, run the following command in the root folder of the project to install dependencies.

Start the Angular development server by running `ng serve` in a terminal in the root folder of your project.

Building Our Server

We'll build our server using Express. Express is a fast,

unopinionated, minimalist web framework for Node.js.

Create a file called `server.js` in the root of the project and update it with the code snippet below

The calls to our endpoint will be coming in from a different origin. Therefore, we need to make sure we include the CORS headers (`Access-Control-Allow-Origin`). If you are unfamiliar with the concept of CORS headers, you can find more information here.

This is a standard Node application configuration, nothing specific to our app.

We're creating a server to feed data to our application so we can see how Effects can be used to fetch external resources to populate the store.

Create a file named `fruits.js` that will hold the products for our store. Open the file and populate it with the code below:

> **Note:** All image assets can be found in the GitHub repository here. Images were gotten from https://pexels.com.

Start the server by running the following command in a terminal within the project folder:

Home View

To get started, we'll define the views for the application, starting from the home page. The home page will house the products grid and the header. Using the CLI, we'll create a component named `home` within the `src/app` folder. Run the command below in the project folder to create the `home`

component:

Open the `home.component.html` file and replace it with the content below.

> You can find image assets used here.

In the snippet above, we've defined an area for the banners and products list. The banner area will house four banner images. We'll go about creating the product list component later in the tutorial.

Styling the Home Component

Next, we'll go about styling the home page, the banner area to be exact. We'll give the images a defined height and give the container a max width.

Since we'll be using external fonts, we'll update the `src/index.html` file with a `link` tag alongside the `src/styles.scss` file.

Then we'll select Dosis as our default font family. We'll also negate the default `padding` and `margin` on the `body` and `html` elements. Open the `styles.scss` file and update it with the following content:

The header component will display the the application logo and the cart total. The component will be subscribed to the store listening for changes to the cart array. More light on this when the NgRx/store library is introduced later in the article.

Run the following command to create the header component:

Next, open the `src/app/header`

`/header.component.html` file and update it to look like the code below:

> **Note:** Any image asset used can be found here in the GitHub repository.

Next, we'll style the header. Open the `header.component.scss` file and update it with the snippet below:

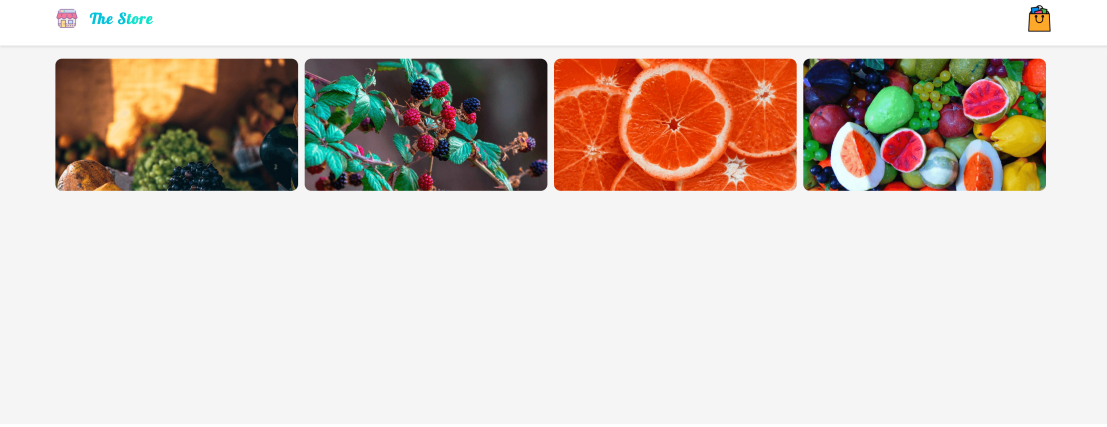
Open up the `header.component.ts` file and declare the `cart` variable used in the HTML file.

App Component

After creating the `home` and `header` components, the next step is to render the components in the root `App` component. Open the `app.component.html` file within the `src/app/` directory. Update it to render both `Header` and `Home` components.

Start the application server by running the following command: `npm start` or `ng serve`.

Then navigate to http://localhost:4200 on your browser. You should see the something similar to the screenshot below:



> Make sure to get the image assets from GitHub or use your preferred images.

Introducing NgRx/store

NgRx/store is a library for managing state in your Angular applications, it is a reactive state management library powered by RxJS. Similar to Redux, this library can be used to manage the flow of data throughout your application, when actions are dispatched, reducers act on them and mutate the store. Another library we'll be working with is NgRx/effects. Effects are commonly used to handle side effects in your application, like fetching data from an external resource.

The first step is to create and assign actions. The actions will be mapped to constants using an `enum`. Create a folder named `store` within the `src/app` directory, this folder will hold everything relating to our application's state management.

Within the `store` folder, create a file called `actions.ts`. Open the file and update it with the code below:

First, we declare an interface that defines the properties of the `Product` object. Then we go on to declare unique actions to be used.

Actions are typically used to describe events in the

application. When an event is triggered, a corresponding event is dispatched to handle the triggered events. An action is made up of a simple interface with a single property `type`, the `type` property is a unique identifier for the action.

An action `type` is commonly defined using the following pattern `[Source] event` — the `source` where the event originates and the event description.

You can create actions using as an `interface` or a `class`. Classes are easier to use if you need to extend the action with a `payload` property, so that's what we did.

After creating actions, a type `ActionsUnion` is exported. This export helps define all Actions in this feature area; it exposes the type information of the actions exported. You can read more on creating actions union here.

After creating actions, the next step is to create a reducer that handles transitions of state from the initial to the next based on the action dispatched. Create a file named `reducer.ts` in the `src/app/store` directory. Open the file and update it with the code below:

A reducer is simple pure function that transitions your application's state from one state to the next. A reducer doesn't handle side effects — it is a pure function because it returns an expected output for a given input.

First, we have to define the initial state of the application. Our application will display a list of `items` and also allow user add and remove items from the `cart`. So the `initialState` of our application will feature an empty array of `items` and an

empty `cart` array.

Next, we'll define the reducer which is a function featuring a switch statement that acts on the `type` of action dispatched.

- The first action type is the `LoadSuccess` action, which is called when products are successfully loaded from the server. When that happens, the items array is populated with that response.
- The next action type is `Add`. This action is dispatched when a user wishes to add an item to cart. The action features a `payload` property containing details of the item. The reducer takes the item and appends it to the cart array and returns the state.
- The final case is the `Remove` action. This is an event telling the reducer to remove an item from cart. The cart is filtered using the `name` of the item dispatched, and the item is left out of the next state.

You're probably thinking that the numbers don't add up. We created four actions but we're only acting on three of them. Well, actions can also be used for effects network requests; in our case, fetching items from the server. We'll look at creating a service to handle fetching the products from the server.

Registering the Reducer

After creating a reducer, it needs to registered in the `StoreModule`. Open the `app.module.ts` file and import the `StoreModule` from the ngrx/store library as well as the `ShopReducer` we just created.

When registering the `ShopReducer`, we assign it a unique identifier (`shop`). This is useful in case you need to register multiple reducers. This need will arise in a larger application where several reducers are created to handle different areas of the application.

Fetching Products from the Server

To handle fetching products from the server, we'll make use of the ngrx/effects library. The library can be used interact with services, abstracting them from components. Effects are used in collaboration with actions and reducers to transition state with the data returned after a network request.

First, we'll create a service that will handle fetching items from the server. To create this service using the CLI, run the command below:

Then open the file and update the content to be similar to the snippet below:

Import the `HttpClient`, create a method called `getAll` and return a call to the server to get fruits using the HttpClient. Next, we'll create an effects file that will make the network request using the `FruitService` when the appropriate action is triggered.

Create a file named `effects.ts` within the `src/app/store` directory. Open the file and copy the following code into the file:

An effect is simple a service with a `@Effect` decorator. There's a bit going on here so we'll explain each strange

keyword used here.

- `Actions` is an observable stream of all the actions dispatched after the application's state has been reduced.
- From the actions dispatched, we use the `ofType` operator provided by the library to filter the actions with the provided type (`LoadItems` in our case). One or more action types can be provided to the pipeable stream.
- The `mergeMap` operator by RxJS is for flattening and merging the actions into an Observable.
- The `getAll` method of the `FruitService` returns an observable that is mapped, and the response is then dispatched as an action, provided there was no error.
- The `catchError` operator handles any errors encountered during the process.

  After creating effects, we have to register it in the root module. Open the `app.module.ts` file and update it to fit the snippet below:

  In the `EffectsModule`, we can register our effects `ShopEffects`. Multiple effects can be registered by adding the effects to the array.

  Now that we've created actions to handle events in our application and reducers to transition state, let's populate the store with items from the server using the effects. Before we do that, let's define views for the product and products list.

Products List View

Run the following commands to generate components for the product item and product list:

And for the product list run:

Open the `product.component.html` file in the `src/app/product` directory and update with the code below:

Here we have two buttons for adding to and removing an item from the cart. A flag `inCart` is used to determine which of the button to display.

> **Note:** All image assets can be found in the GitHub repository here.

Let's style the component by updating the `product.component.scss` file with the styles below:

Open the `product.component.ts` file and update it with the variables and methods used in the HTML file.

First we import the `Store` observable from the `ngrx/store` library. The store property will be used to dispatch actions.

The `addToCart` method takes one parameter (`item`); the method dispatches an action to add an item to cart. After dispatching the action, the `inCart` property is set to `true`. This flag is for identifying which items are in cart.

Meanwhile, the `removeFromCart` method dispatches an action to remove an item from cart and updates the `inCart` property to `false`.

Next we'll render the `Product` component in the `product-list` component. Open the `product-`

`list.component.html` file and render the `Product` similar to the snippet below:

We'll add some styles to the component's stylesheet. Open the `product-list.component.scss` file and add the styles below:

The product list component will receive an `Input` from the `Home` component, so let's update the component to take an `Input` an array of `fruits`. Update the `product-list.component.ts` file to be similar to the snippet below:

After making this change, the final step is to render the product list component in the `home.component.html` file and dispatch an action to load the products from the server in the `OnInit` lifecycle of the component.

Open the `home.component.html` file and render the product list component within the element with the `product-area` class attribute:

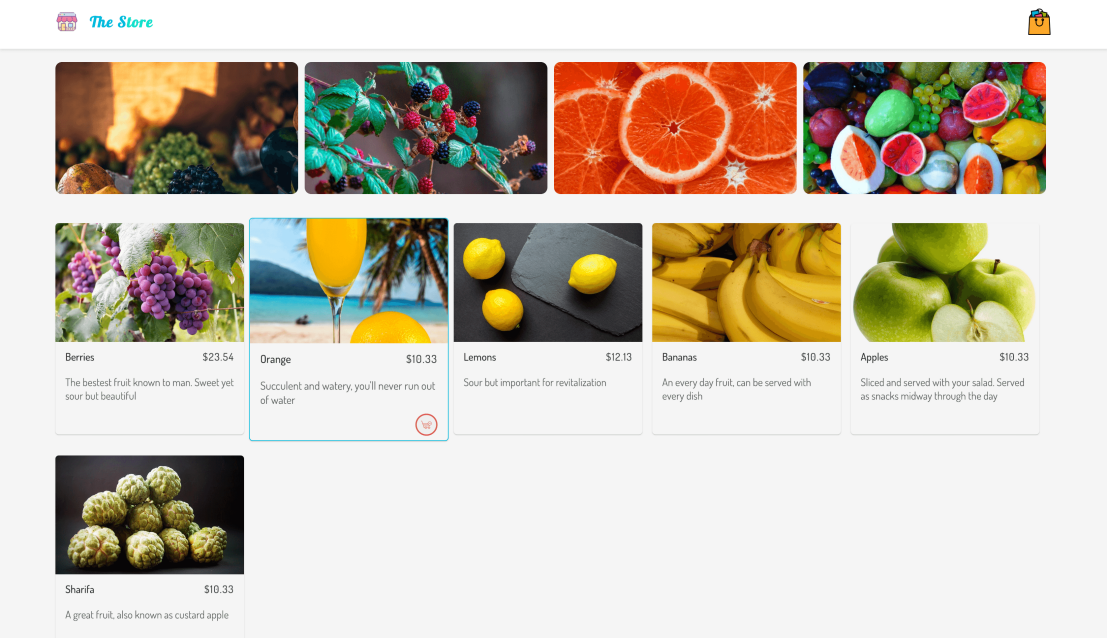
Then update the home component and make it similar to the snippet below:

First we dispatch a new action `GetItems`. The action type was registered in the effect that handled fetching products from the server. After dispatching the action, we use the `Store` observable and the `select` operator to select and subscribe to the store we registered in the `AppModule` file.

When subscribed to the store, the data returned is the current state of our store. If you remember, the initial state of our store had two properties, both of which are arrays. In the home

component, we need the array of items in the store, so using dot notation we'll get the current items from the state.

After this change, if you visit http://localhost:4200, you should see all the latest changes we've made, including the ability to add and remove an item from cart.

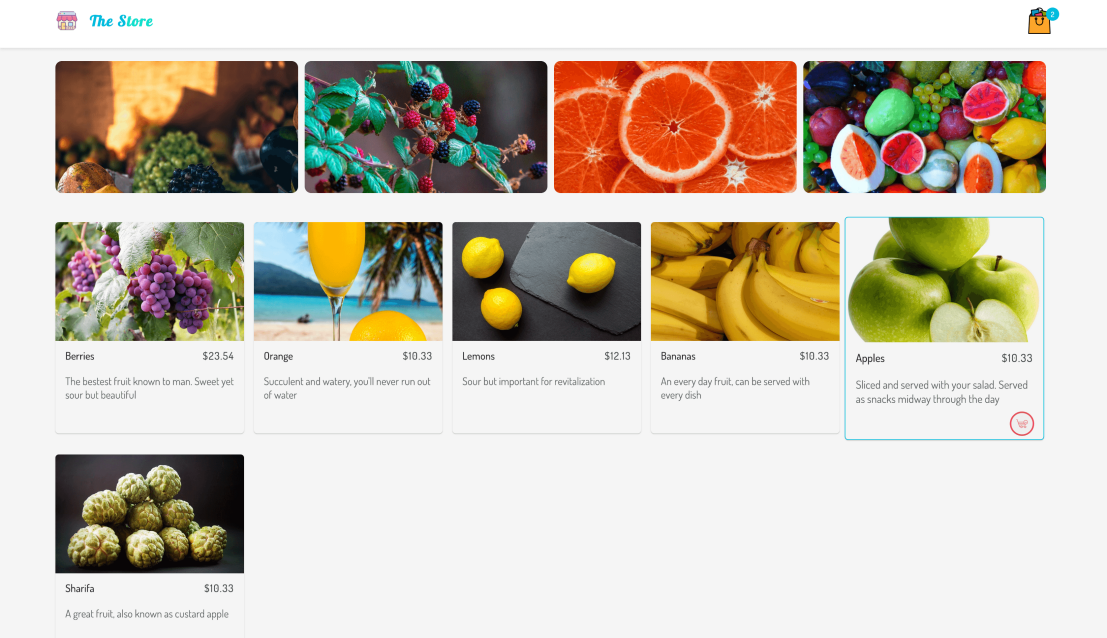


If you try adding an item to the cart, you'll notice it is successful, but our cart doesn't update with the number of items in the cart. Well, this is because we're not subscribed to the store, so we won't get the latest updates on the cart.

To fix this, open the `header.component.html` file and update the component to subscribe to the store in the component's constructor.

Similar to the `Home` component where we subscribed to the store and got the `items` array from the state, here we'll be subscribing to the `cart` property of the state.

After this update, you should see the amount of items in cart when an item is added or removed from the cart.



> **Note:** Ensure both that the Angular dev server is running on port **4200** and that the server is running on port **4000**.

Conclusion

In this tutorial, we've built a simple store where items can be added and removed from cart. We've been able to manage the application's state using NgRx/store. As we've seen, it is easier to manage data flow in the application when side effects and data flow are abstracted from components. The decision to pick a state management library is sometimes difficult. Some people introduce them too early, which adds another level of complexity to the application, and some people introduce them too late, but no matter what the case may be, state management libraries are helpful whenever they're introduced.

I remember a popular quote about Flux:

> You'll know when you need Flux. If you aren't sure if you need

it, you don't need it.

The same should be applied in this case. The choice rests on you. You can find the source code for this demo here.

"*We are our choices.*" - *Jean-Paul Sartre*

This blog has been brought to you by Kendo UI

Want to learn more about creating great web apps? It all starts out with Kendo UI - the complete UI component library that allows you to quickly build high-quality, responsive apps. It includes everything you need, from grids and charts to dropdowns and gauges.