State management in Angular

Author: Luka Mihić

In the world of software engineering, the main thing in every aspect is data. The backend uses databases to store this data, but the frontend part also requires a way of storing that data temporarily. How is it managed? This is where state management comes in.

State management is a way of reducing API requests on the client side by storing the initial response in a **store**. A store is a place where data is accumulated. In other words, a store is a place where different states are stored. States are classes which represent a container of specified data types.

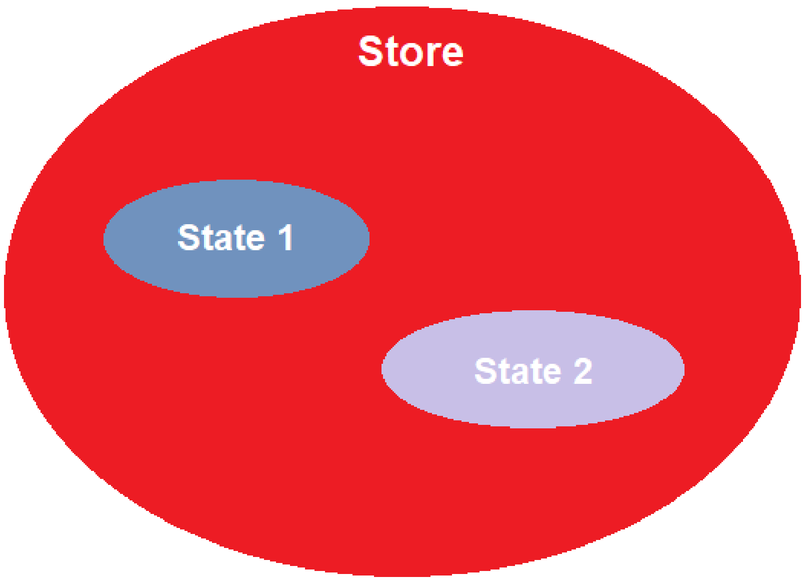

Image 1. A store is a union of various states

Frontend frameworks have different ways of managing the state, for example React apps can use e.g. Mobx, while Vue apps can use e.g. Vuex.

In applications using Angular as a Javascript frontend framework, state management is used as well. The main focus of this post is *NGXS*, as our company uses it most of the time, but in this text, we are also gonna refer to *NGRX*. NGRX has a bit of a different code structure, and I am not saying people shouldn't be using it, but we found NGXS as a perfect fit for our solutions using Angular on the frontend part.

The reason NGXS is more chosen by our developers is the simplicity of it, but great power at the same time. It allows us to do more things with less code, but also lets us maximize the usage of received observables. While the NGRX has Reducers and Effects, in NGXS everything comes down to state. The Effects in NGRX make both writing and understanding the code more complex, while in NGXS one state is capable of handling all the actions.

State management with NGXS is based on CQRS (Command and Query Responsibility Segregation). Redux is formed in the same way, but NGXS reduces the usage of boilerplates (including repeated parts of code with minimum variations) by using strongly typed code (Typescript). This is another good side of this state management, because NGRX still uses boilerplate code.

NGXS consists of four major concepts: store, actions, states and selects, while NGRX also uses effects.
We have already talked about the store and states, but what are actions and selects? Actions are commands which trigger specific changes to the state, while selects are functions in which we use specific state data. Selects are most commonly declared to variables using decorators in components (the naming convention is adding a dollar sign at the end of variable name (e.g. *businessUnits$*).

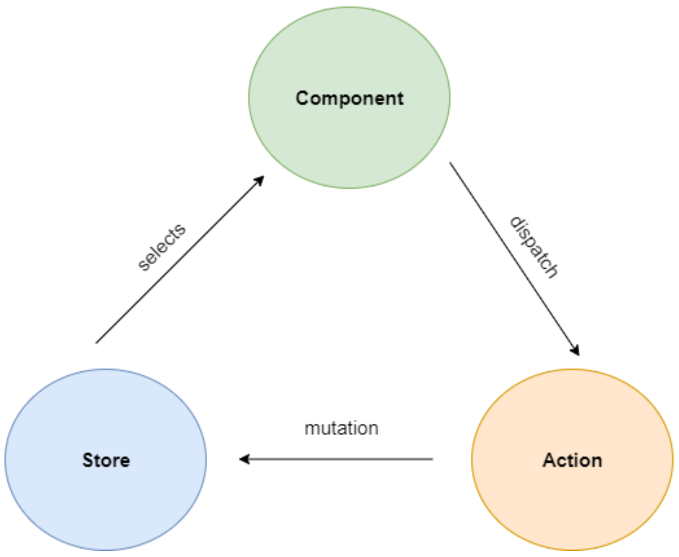

Image 2. NGXS flow

Below is an example representing state management in NGXS with countries.

countries.action.js

```
import { CountryModel } from 'src/app/models/all';
import {
  CountriesGetRequest,
  CountriesInsertRequest
} from 'src/app/models/requests/all';

export class CountriesStateModel {
  countries?: CountryModel[];
  filter?: CountriesGetRequest;
}

export class CountriesGet {
  static readonly type = '[Countries] Get';
  constructor(public request?: CountriesGetRequest) {}
}

export class CountriesInsert {
  static readonly type = '[Countries] Insert';
  constructor(public request: CountriesInsertRequest) {}
}
```

Here we define the action names and their constructors, what the actions will receive when being dispatched through components.

countries.state.js

```
import { Injectable } from '@angular/core';
import { tap } from 'rxjs';
import { Action, Selector, State, StateContext } from '@ngxs/store';

import {
  CountriesStateModel,
  CountriesGet,
  CountriesInsert,
} from './countries.actions';
import { CountryModel } from 'src/app/models/all';
import { CountriesService } from 'src/app/services/all';

@State<CountriesStateModel>({
  name: 'countries',
  defaults: {},
})
@Injectable()
export class CountriesState {
  constructor(private countriesService: CountriesService) {}

  @Selector()
  static countries(
    state: CountriesStateModel
  ): CountryModel[] | undefined {
    return state.countries;
  }

  @Action(CountriesGet)
  get(context: StateContext<CountriesStateModel>, params: CountriesGet)
{
    return this.countriesService.get(params.request).pipe(
      tap((result: CountryModel[]) => {
        context.patchState({
          countries: result,
          filter: params.request,
        });
      })
    );
  }

  @Action(CountriesInsert)
  insert(context: StateContext<CountriesStateModel>, params:
CountriesInsert) {
    return this.countriesService.insert(params.request).pipe(
      tap(() => {
        const previousState = context.getState();

        context.dispatch(new CountriesGet(previousState.filter));
      })
    );
  }
}
```

This is the whole logic happening in the store and it represents state mutation with action names before every implementation of the mutation.

Once the store is set up we use our selects and dispatches in the components we want. The only thing we need to do is inject the store in our wanted component constructor, as so:

```
export class CountriesComponent implements OnInit {
  title = 'countries';
  constructor(private store: Store) {}

  @Select(CountriesState.countries) countries$:
Observable<CountryModel[]>;

  ngOnInit() {
    this.store.dispatch(new CountriesGet()); //params go in parentheses
  }
}
```

In conclusion, state management is as crucial in frontend frameworks, as the database is to the backend. Every state management library has its downsides as well, but in our recent practice, NGXS showed as a very good solution. Both NGXS and NGRX are a good choice and the decision depends on the wanted structure of the code. Both are based on Redux, and will require proper handling of the actions.