netmind
a bts company

Angular 14 - 15

Data Store Pattern

SPAs at scale

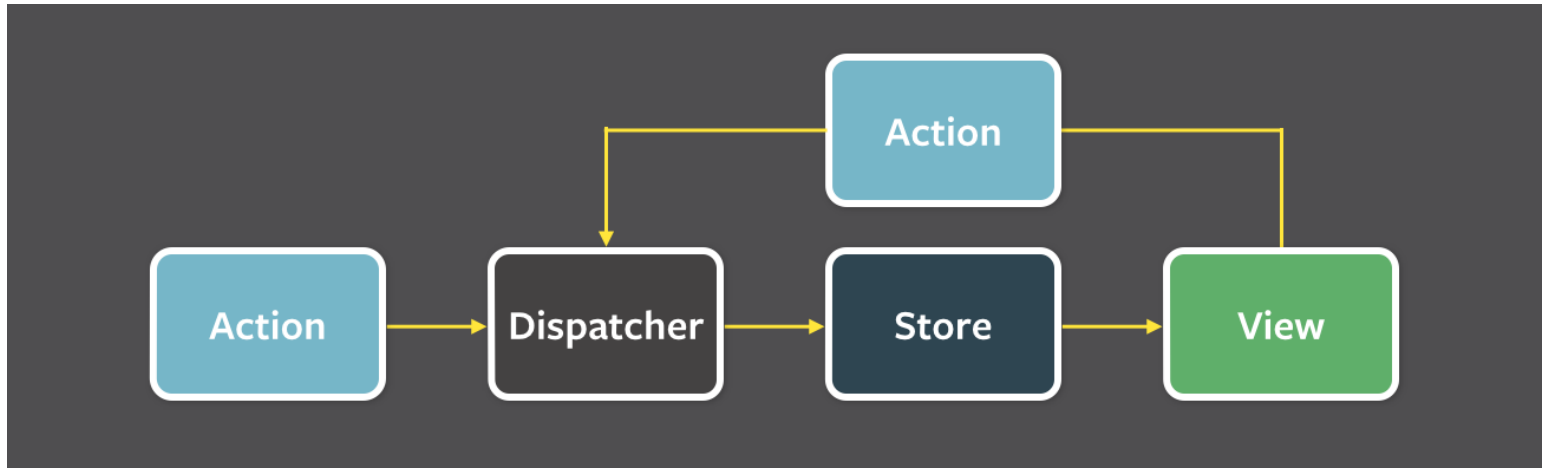
- Scalable apps are a challenging. In front-end, we can think of increasing complexity, more and more business rules, a growing amount of data loaded into the application and large teams often distributed around the world.
- In order to deal with it and gain high quality of delivery and prevent technical debt, robust and well-grounded architecture is necessary.
- Angular itself is a opinionated framework, forcing developers to do things *the proper way*, yet there are a lot of places where things can go wrong.
- Our ultimate goal is design Angular application in order to maintain **sustainable development speed** and **ease of adding new features** in the long run.
- To achieve these goals, we need apply:
  - proper abstractions between application layers,
  - unidirectional data flow,
  - reactive state management,
  - modular design,
  - smart and dumb components pattern.

SPA State

- As we have seen, observables are a flexible and efficient way to communicate services with components.
- In a real application, many components will need access to common application data.
- Take, for example, a simple project and task application. We could have views of projects (list, detail, create/edit) and tasks (list, detail, create/edit).
- Since there is a dependency between the two models (tasks belong to a project, projects have tasks), we will need the components to have access to this common data to display it in the interface.
- This set of data (model values) that we manage in an application is called the **application state**.

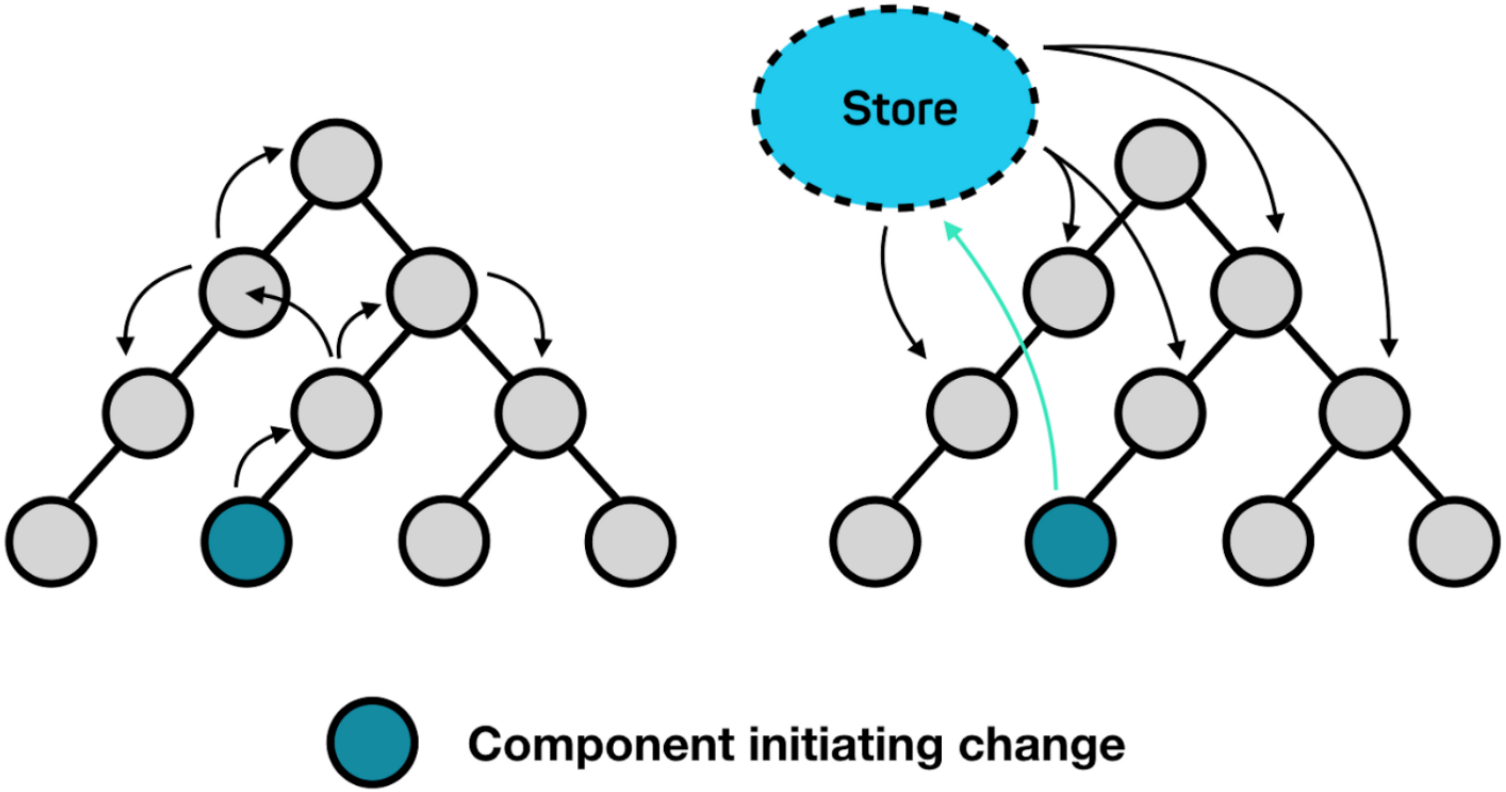
Flux pattern

- One of the more interesting problems that is addressed during application development is managing state.
- New paradigms have changed the way web applications treat and manipulate their state, moving away from two-way data binding mechanisms (as done in the old days of AngularJS) to a more functional **one-way flow**, as posed by the Flux pattern with React and Redux.
- In this architecture, the data flow is unidirectional and updates the state of the application stored in a centralized element, the **store**, based on actions objects.



Flux pattern

- Components are mere consumers of this store and are updated as the store changes state
- This approach can be seen in Redux, which is an example of a simplified Flux pattern implementation.



Data store in Angular

- A variant of this pattern can be implemented in Angular without third-party dependencies using the framework's own tools that address most state management concerns:
- **@Input** and **@Output** – Pass state through a chain of nested components using attributes and event emitters.

```
<td><delete-button [ task ]="aTask" ( deleteClicked )="delete($event)"></delete-button></td>
```

- **Providers / Services** : manage data through dependency injection

```
@Injectable({
  providedIn: 'root',
})
export class ProductsService {
  private _products: IProduct[] = [
  ];
  ...
}
```

Data store service

- When it comes to a fairly large application, we want to avoid services that contain too much business logic and cover a lot of mutable data.
- Therefore we can use the services as a "data store" in a simple way using their state and observables.
- The idea behind this architecture is threefold:
  - **Isolate** a piece of data in a service so that it can be shared throughout an application
  - **Moderate** mutations to this data so changes can be easily tracked
  - **Limit** the degree to which the state can be changed to avoid unwanted side effects.

Data store service

- A store service would have the following form (using the API client service):

```
private _products: IProduct[] = [];
  private _productsObs: Observable<IProduct[]> = {} as any;

  getProducts(): Observable<IProduct[]> {
    if (this._products) {
      return of(this._products);
    } else if (this._productsObs) {
      return this._productsObs;
    } else {
      this._productsObs = this._productAPI.getProductsFromAPI().pipe(
        tap((data) => (this._products = data)),
        catchError(this.errorHandl)
      );
      return this._productsObs;
    }
  }
```

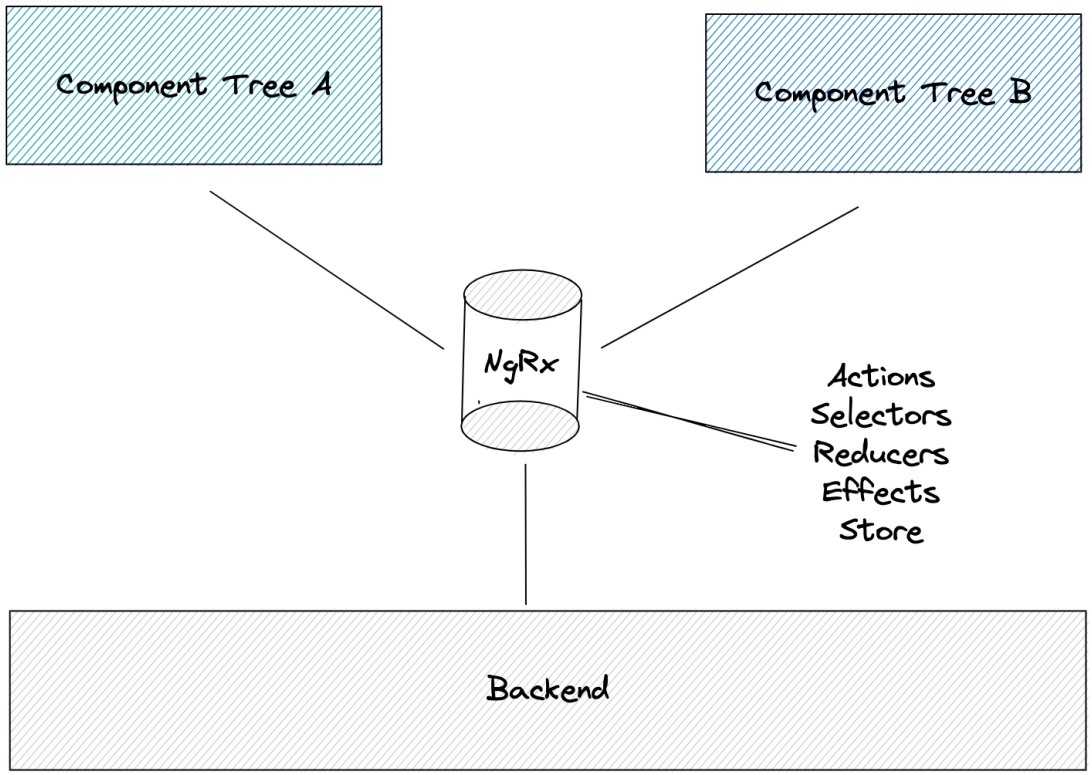
- In the component

```
  constructor( private _productStore: ProductStoreService ) {}

  $products: IProduct[] | null = null;

  ngOnInit() {
    this._productStore.getProducts().subscribe(data=>this.$products=data);
  }
```

Data store service

- Although this component is quite simple, the basic principles of a data store still applies.
- Inside the component, we pass our data store through Angular's DI (Dependency Injection) `constructor( private _productStore: ProductStoreService ) {}` so that we can access the app state in the component.
- Within the ngOnInit() lifecycle function, we subscribe to the application's state (in this case, a list of tasks) which will provide us with a new state each time the store is changed via an observable from a BehaviorSubject.
- To modify the state we interact with the interface (for example deleting a task) and call the service to mutate the data.
- Thanks to the subscription, the store contains the updated data of the application.
  - Every time the data is changed, all subscribers to the store (for example, the projects component) will be updated automatically.
  - With the help of the private BehaviorSubject, any component that wants to change state in the store has to do so through the store, ensuring that no one can manipulate the data in a way that breaks the application.

NgRx



- NgRx is a framework for building reactive web applications in Angular. It implements **the Flux-Pattern**.
  - https://ngrx.io/
- NgRx Store provides reactive state management for Angular apps inspired by Redux. Unify the events in your application and derive state using RxJS.
- NgRx provides a global store and various building blocks around the store, such as **Actions**, **Reducers**, **Selectors**, and **Effects**, to manage this store.
- NgRx is a good choice, if:
  - the state should be accessed by many components and services
  - the state needs to be available when re-entering routes
  - the state is impacted by actions from different sources.

NgRx actions

- Actions express **unique events** that occur while using your web application.
- These can be user interaction with a particular page or external interaction through network requests or direct interaction with, for example, the device API.
- Actions are dispatched via Store in NgRx and observed by NgRx's Reducers and Effects.

```
import { createAction, props } from '@ngrx/store';

export const getProducts = createAction('[Product List] Get products');

export const addProduct = createAction(
  '[Product List] Add product',
  props<{ product: IProduct }>()
);
```

- You can create a group of related actions using **createActionGroup**.

```
export const productListActions = createActionGroup({
  source: 'Product List',
  events: {
    'Get products': emptyProps(),
    'Add product': props<{product: Iproduct}>(),
  }
});
```

NgRx store

- The Store is the most important component of NgRx based on a **single, immutable data structure**.
- It **provides a single store** to express a global, application-wide state.
- To access the Store, you can simply inject it. In the store, you can select data using selectors or dispatch actions.

```
import {Store} from "@ngrx/store";
import {getProducts} from "../../actions/products-page.actions";
...

constructor(private _store: Store) {}
...
this.store.dispatch(getProducts());
```

NgRx reducers

- Reducers are responsible for **state transitions** from the store.
- The great thing is that reducers are **pure functions**, meaning they produce the same output for a given input.

```
export const productsReducer = createReducer(
  initialState,
  on(productsLoadedSuccessfully, (store, result) => ({
    ...store,
    products: result.data
  }))
);
```

NgRx selectors

- Selectors give you slices of a store state.
- They also help to compose different selectors.

```
export const selectProductsState = (state: ShopState) => state.productsFeature;

export const selectproducts = createSelector(
  selectProductsState,
  (productsFeature: productsFeatureState | undefined) => {
    return productsFeature?.products;
  }
);
```

Binding data from selectors

- With the **@ngrx/component** package you get a directive called ***ngrxLet**.
- With this directive you can easily bind data, e.g. from a selector in the markup

```
<ng-container *ngrxLet="number$ as n">
    <app-number [number]="n"></app-number>
</ng-container>

<ng-container *ngrxLet="number$; let n">
    <app-number [number]="n"></app-number>
</ng-container>
```

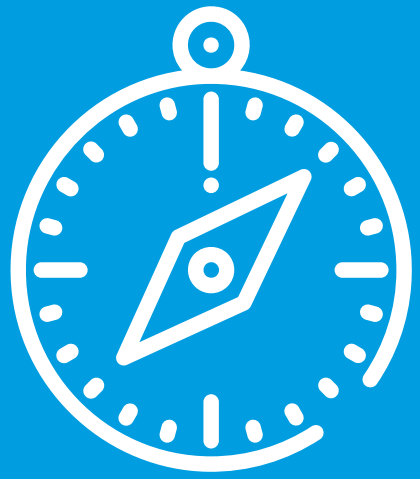
NgRx effects

- Effects are an **RxJS based side effect** model for the store.
- Effects use streams to provide new sources for actions and isolate side effects from components.
- This gives us "**purer**" components that select state and perform actions.

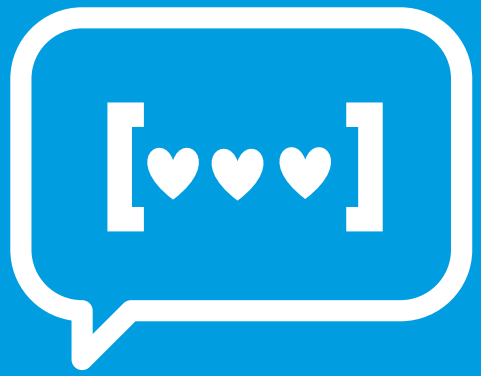
```
loadProducts$ = createEffect(() =>
  this.actions$.pipe(
    ofType(getProducts),
    mergeMap(() =>
      this.productService
        .getProducts()
        .pipe(map((products) => productsLoadedSuccessfully({ data: products })))
    )
  )
);
```

Let's put it into practice: Tasks/Projects App

- Create a simple Data Store for tasks and projects and make them use the API services to update their status.
- Use NgRx for implemented the store.



Next steps



We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time and effort you have taking in answering all of that is important in order to improve our training plans so that you will always be satisfied with having chosen us
quality@netmind.es

netmind
a bts company

Thanks!

Follow us: