State management in Angular

NgRx

Benjamin Canape - 31/12/24

What is NgRx ?

- Centralize the state of the application
- Clear flow to modify the state
- Based on Redux
- Optimisation of the data sharing between components

Why use NgRx?

- A Single Source of Truth
- State predictability
- Scalability for large applications
- Easy Debugging tools

When ?

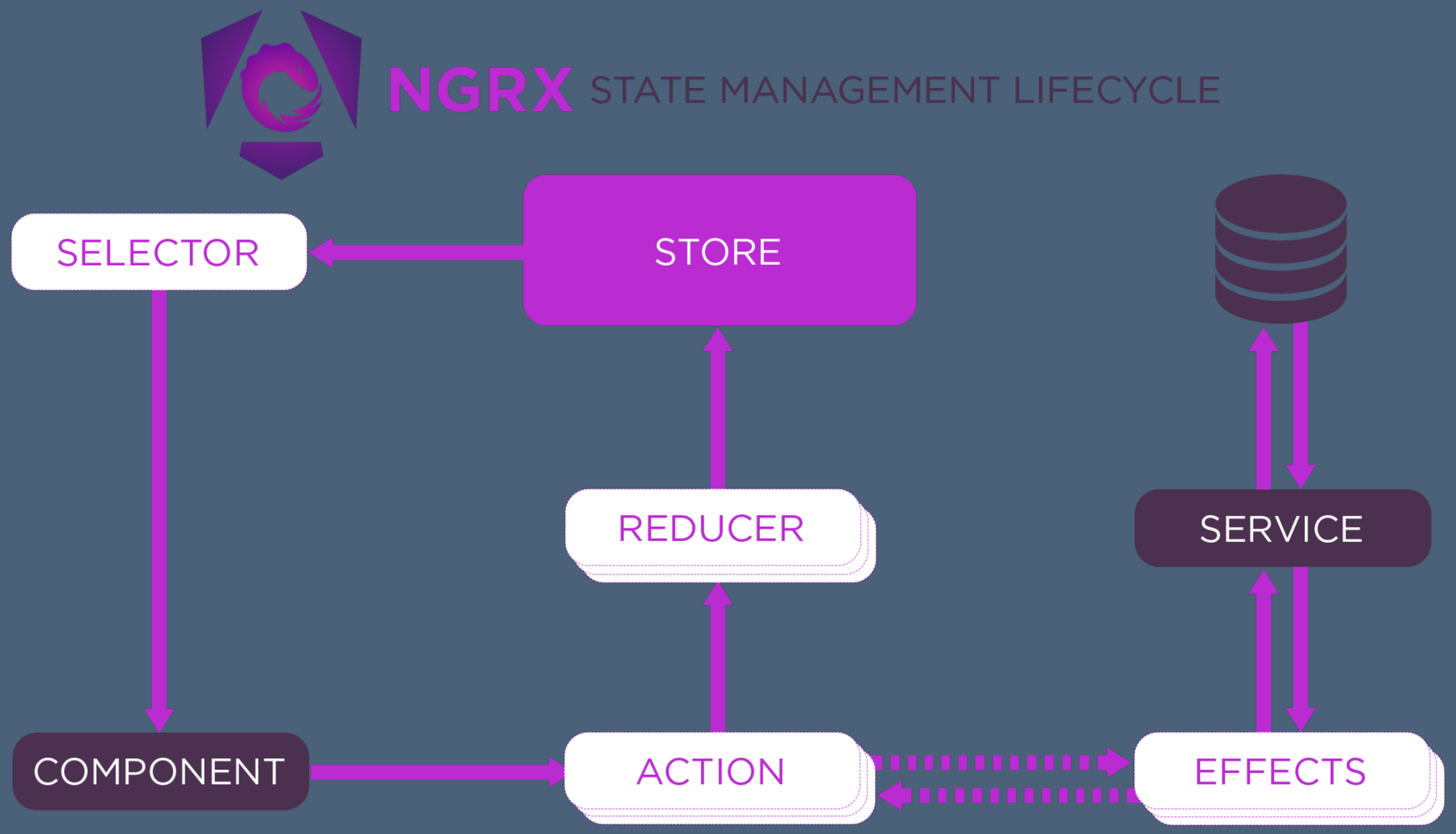
SHARI principle

- Shared: state that is accessed by many components and services
- Hydrated: state that is persisted and rehydrated from external storage
- Available: state that needs to be available when re-entering routes
- Retrieved: state that must be retrieved with a side-effect
- Impacted: state that is impacted by actions from other sources

Key concepts

- Store: Global state of the application
- Actions: Plain objects that describe an intention to change the state
- Reducers: Produce a new state with the current state and an action
- Selectors: Extract some data from the store
- Effects: Handle side effects like calls to services when an action is launched

How does it work ?



State

The global state of the application

```
1  export interface AppState {
2    notes: NoteState;
3  }
```

```
1  export interface NoteState {
2    notes: Note[];
3  }
4
5  const initialState: NoteState = {
6    notes: [],
7  };
```

```
1  export interface Note {
2    id: string;
3    creationDate: Date;
4    title?: string;
5    content: string;
6  }
```

Actions

Plain objects that describe an intention to change the state

```
1  export const AddAction = createAction(
2    '[Note] Add',
3    props<{ note: Note }>()
4  );
```

Action Groups

```
1  export const noteActions = createActionGroup({
2    source: 'Note',
3    events: {
4      Add: props<{ note: Note }>(),
5      'Add Success': props<{ note: Note }>(),
6      'Add Failure': props<{ error: string }>(),
7    },
8  });
```

Reducers

How the state should change in response to an action

```
1  export const noteReducer = createReducer(
2    initialState,
3    on(noteActions.addSuccess, (state, { note }) => ({
4      ...state,
5      notes: [note, ...state.notes],
6    }))
7  );
```

Effects

Handle side effects on action launched

```
@Injectable()
export class NoteEffects {
  private actions$ = inject(Actions);
  private noteService = inject(NoteService);

  addNote$ = createEffect(() =>
    this.actions$.pipe(
      ofType(noteActions.add),
      switchMap(({ note }) =>
        this.noteService.add(note).pipe(
          map(note => noteActions.addSuccess({ note })),
          catchError(error => of(noteActions.addFailure({ error })))
        )
      )
    )
  );

  addNoteSuccess$ = createEffect(
    () =>
      this.actions$.pipe(
        ofType(noteActions.addSuccess),
        tap(note => console.log('Note added:', note))
      ),
    { dispatch: false }
  );
}
```

For instance: call a service

Can dispatch an action in return or not.

If not, don’t forget to add

{ dispatch: false } as the

second parameter of createAction method

Selectors

Extract the data from the store

```
1  export const selectNoteState = createFeatureSelector<NoteState>('notes');
2
3  export const selectAllNotes = createSelector(
4    selectNoteState,
5    state => state.notes
6  );
```

How to use in your app ?

The best is to create a facade

```
@Injectable({
  providedIn: 'root',
})
export class NoteFacade {
  private store = inject(Store<NoteState>);

  notes$ = this.store.select(selectAllNotes);

  add(note: NoteData) {
    return this.store.dispatch(noteActions.add({ note }));
  }
}
```

In your components or services,

you just have to call the facade

Import the state in your app

Don't forget this step, you will not have any error if you miss it

```
1  const reducers: ActionReducerMap<AppState> = { notes: noteReducer };
2
3  export const appConfig: ApplicationConfig = {
4    providers: [
5      provideStore(reducers, { metaReducers }),
6      provideEffects([NoteEffects]),
7    ],
8  };
```

You can also use ProvideState especially

if you organize your different sub-state

into features

Meta-reducers

```
1  export function debug(reducer: ActionReducer<any>): ActionReducer<any> {
2    return function(state, action) {
3      console.log('state', state);
4      console.log('action', action);
5
6      return reducer(state, action);
7    };
8  }
9
10 export const metaReducers: MetaReducer<any>[] = [debug];
```

Called before the « normal » reducers

Allows to pre-process actions

Rehydrate state on page refresh

ngrx-store-localstorage

```
1  export function localStorageSyncConfig(): LocalStorageConfig {
2    return {
3      keys: ['notes'],
4      rehydrate: true,
5    };
6  }
7
8  export function localStorageSyncReducer(reducer: any): any {
9    return localStorageSync(localStorageSyncConfig())(reducer);
10 }
11
12 const metaReducers: Array<MetaReducer<any, any>> = [localStorageSyncReducer];
```

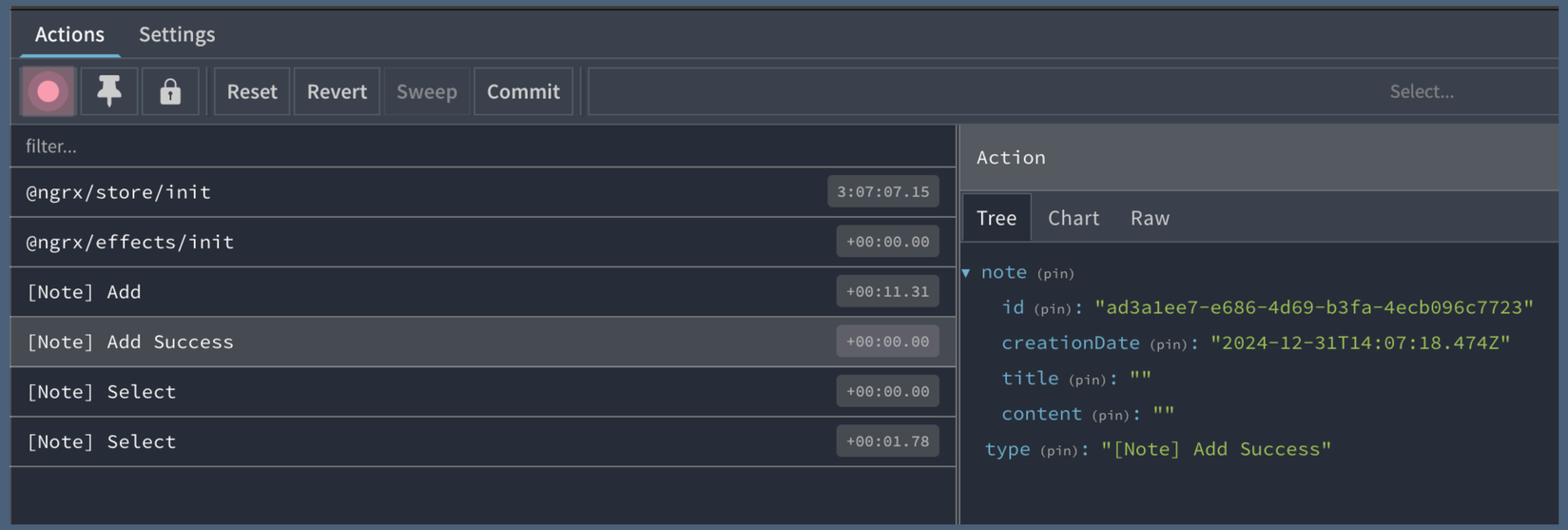
Debugging

Redux Devtools

Chrome extension: https://chromewebstore.google.com/detail/redux-devtools

Firefox extension: https://addons.mozilla.org/fr/firefox/addon/reduxdevtools/

```
1  provideStoreDevtools({
2    maxAge: 25,
3    logOnly: !isDevMode(),
4    autoPause: true,
5    trace: false,
6    traceLimit: 75,
7    connectInZone: true
8  })
```



Best practices

- Don’t overuse the store (just when it is necessary). Sometimes, just Input/Output data sharing between components is sufficient
- Create Features to have different modules

Thanks a lot

Questions ?

NgRx documentation: https://ngrx.io/

Contact info: benjamin.canape@gmail.com