State management in Angular

NgRx

Benjamin Canape - 31/12/24

What is NgRx ?

- Manage the state of the application
- Based on Redux
- Optimisation of the data sharing between components

Why use NgRx?

- A Single Source of Truth
- State predictability
- Easy Debugging tools

When ?

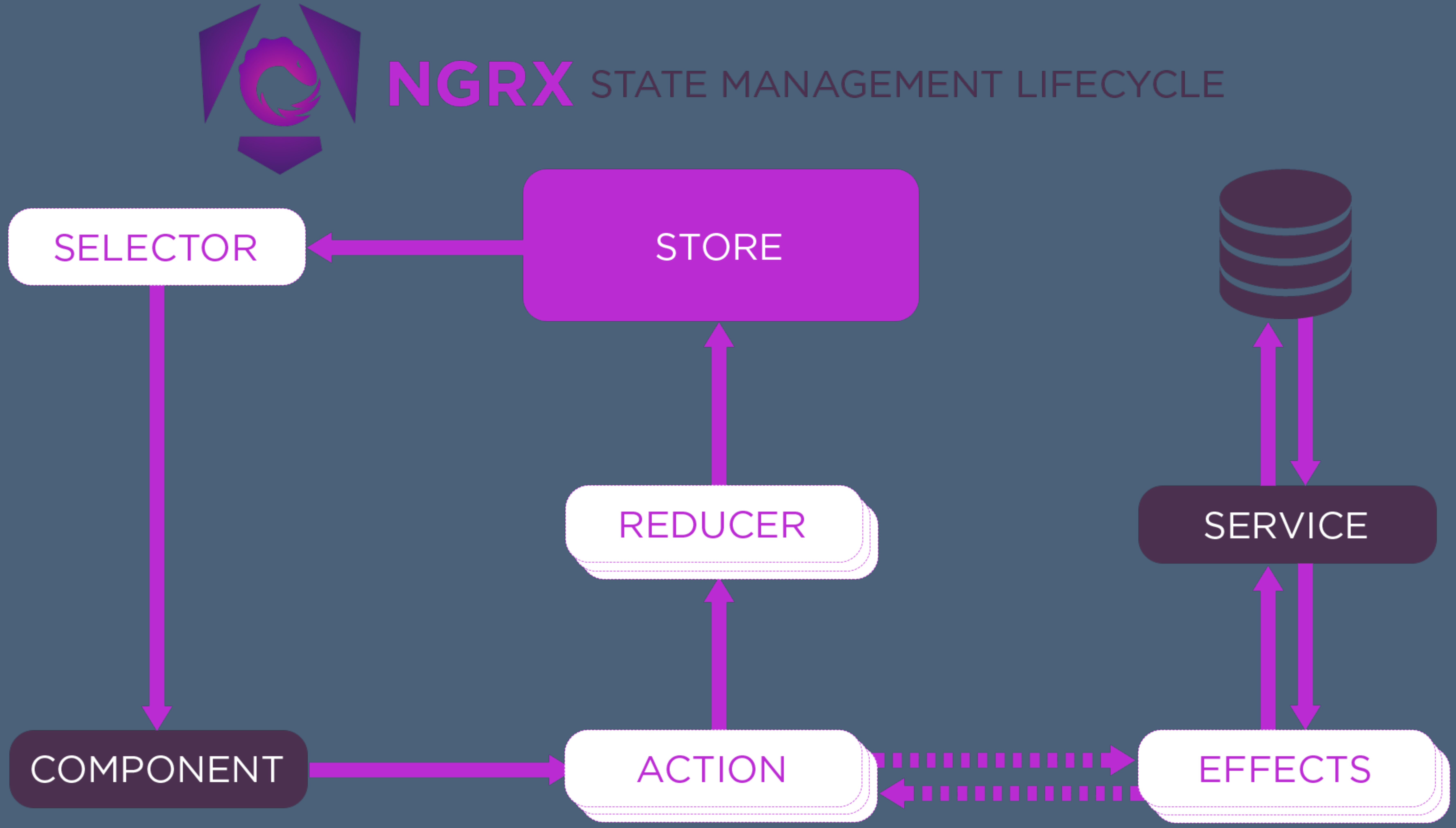
SHARI principle

- 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

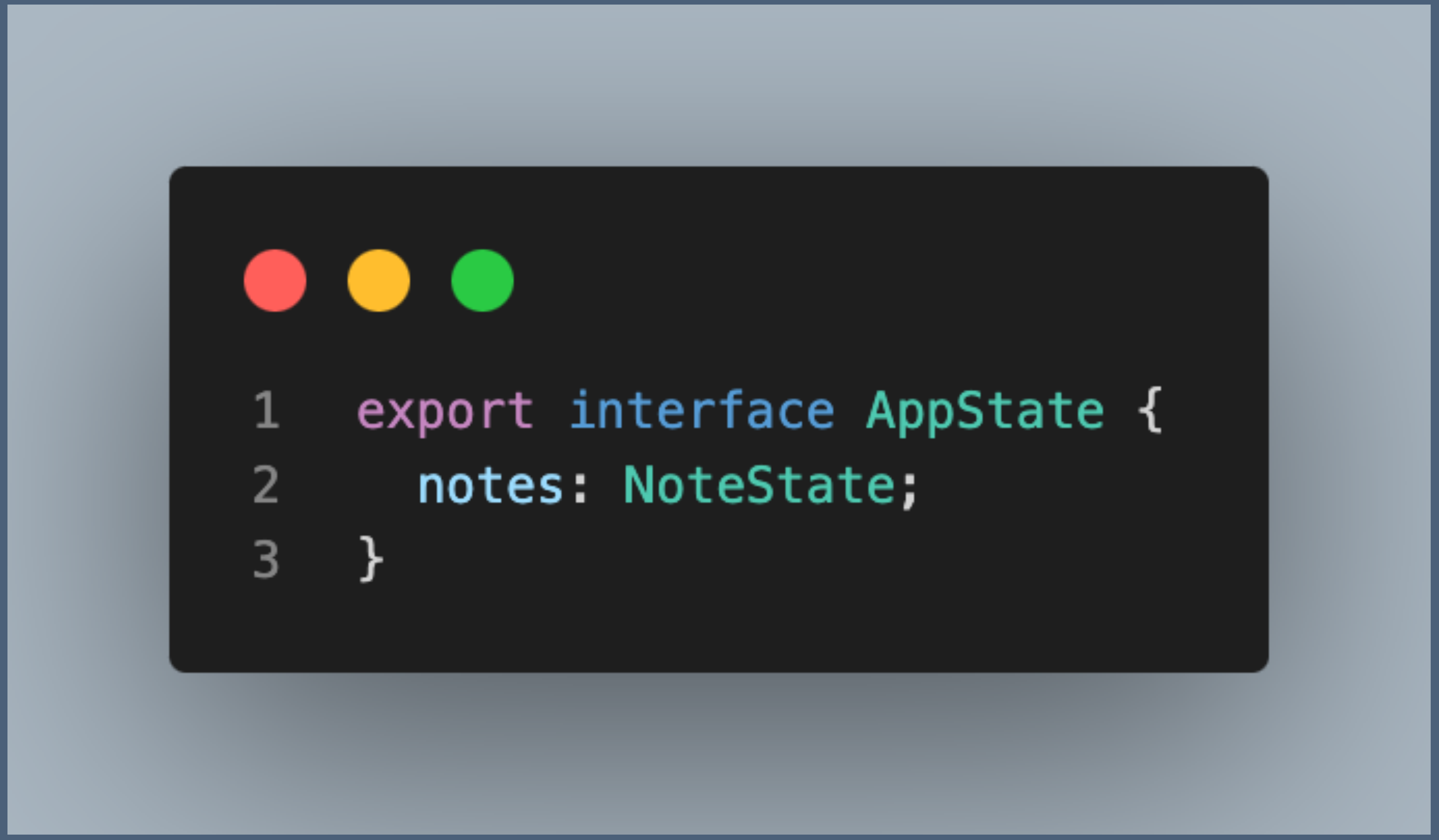
- Store: Global state of the application
- Actions: Actions that happens in the application to modify the state
- Reducers: Produce a new state with the last one and an action
- Selectors: Extract the data in the store
- Effects: Calls services (like Api calls) when an action is launched

How does it work ?

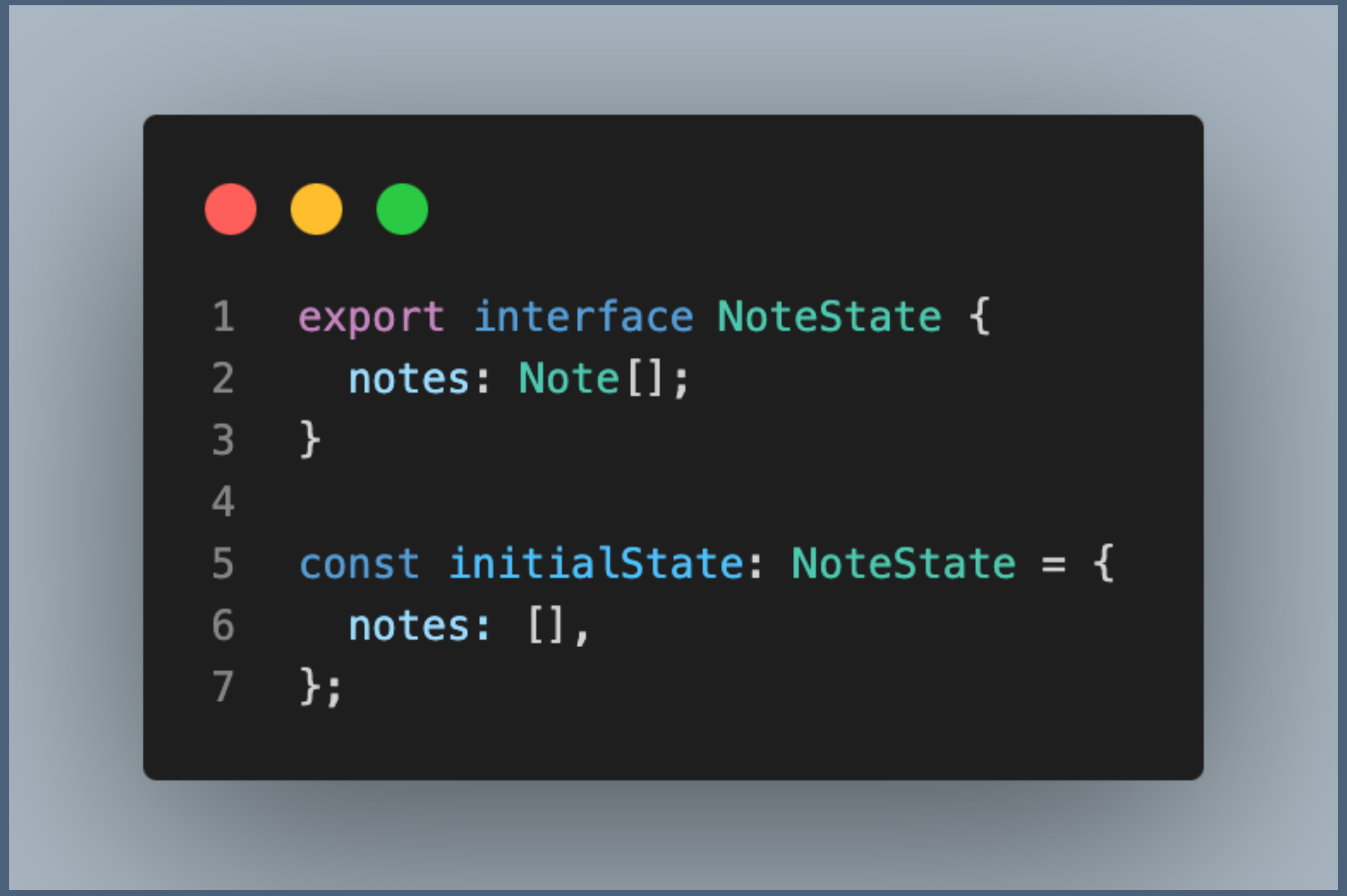


State

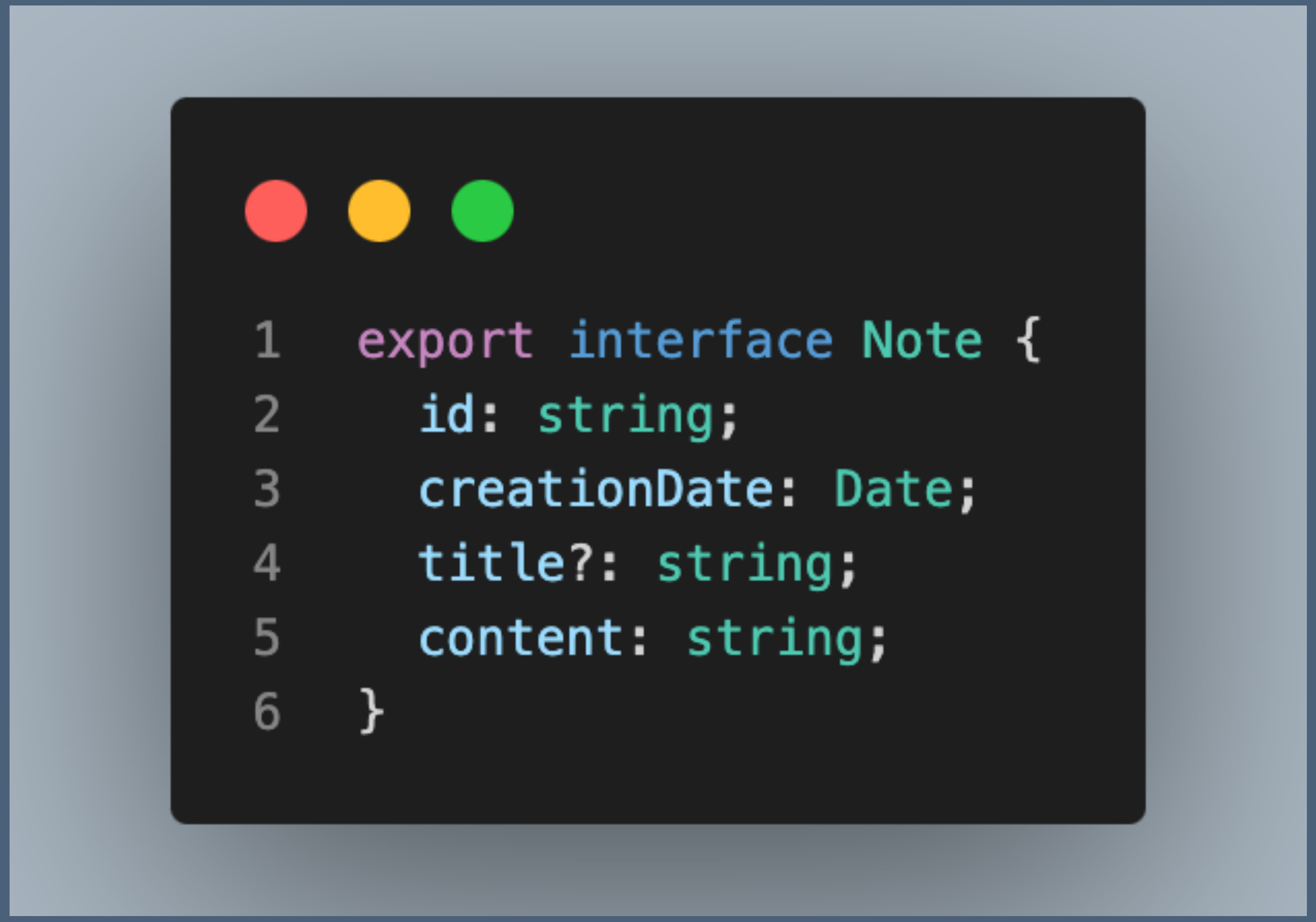
the global state of the application



```typescript
export interface AppState {
  notes: NoteState;
}
```



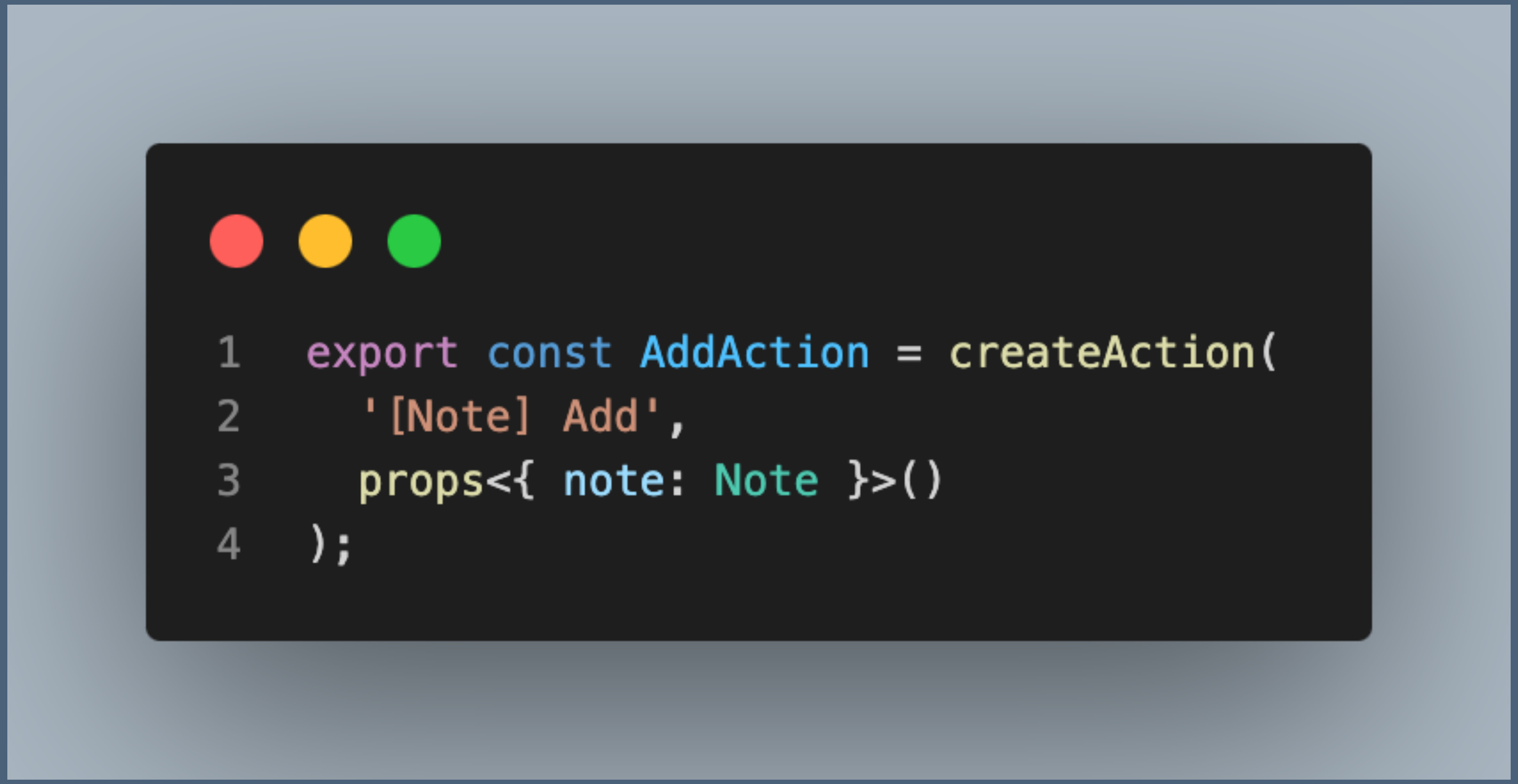
```typescript
export interface NoteState {
  notes: Note[];
}

const initialState: NoteState = {
  notes: [],
};
```



```typescript
export interface Note {
  id: string;
  creationDate: Date;
  title?: string;
  content: string;
}
```

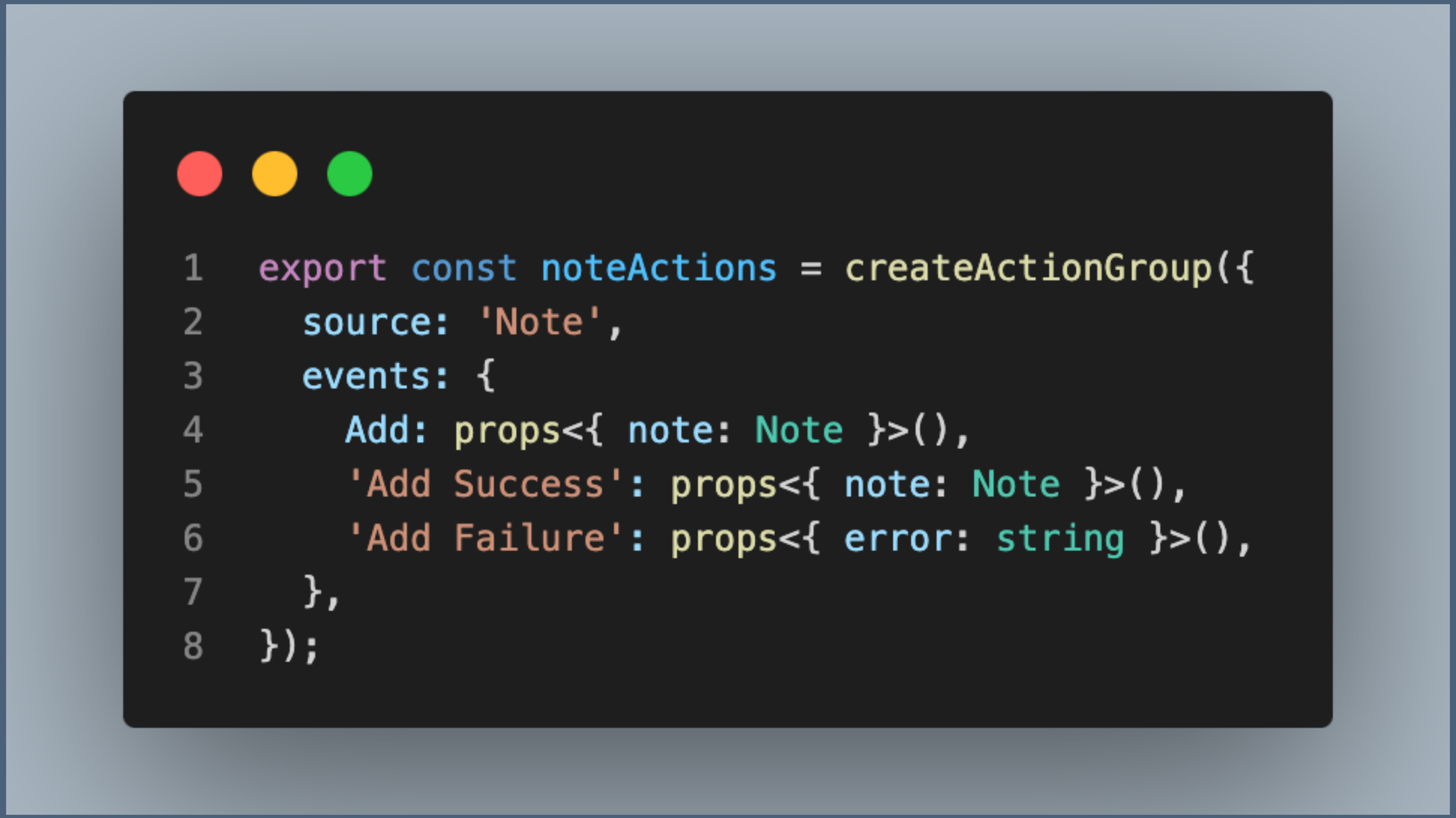
Actions

Actions that happens in the application



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

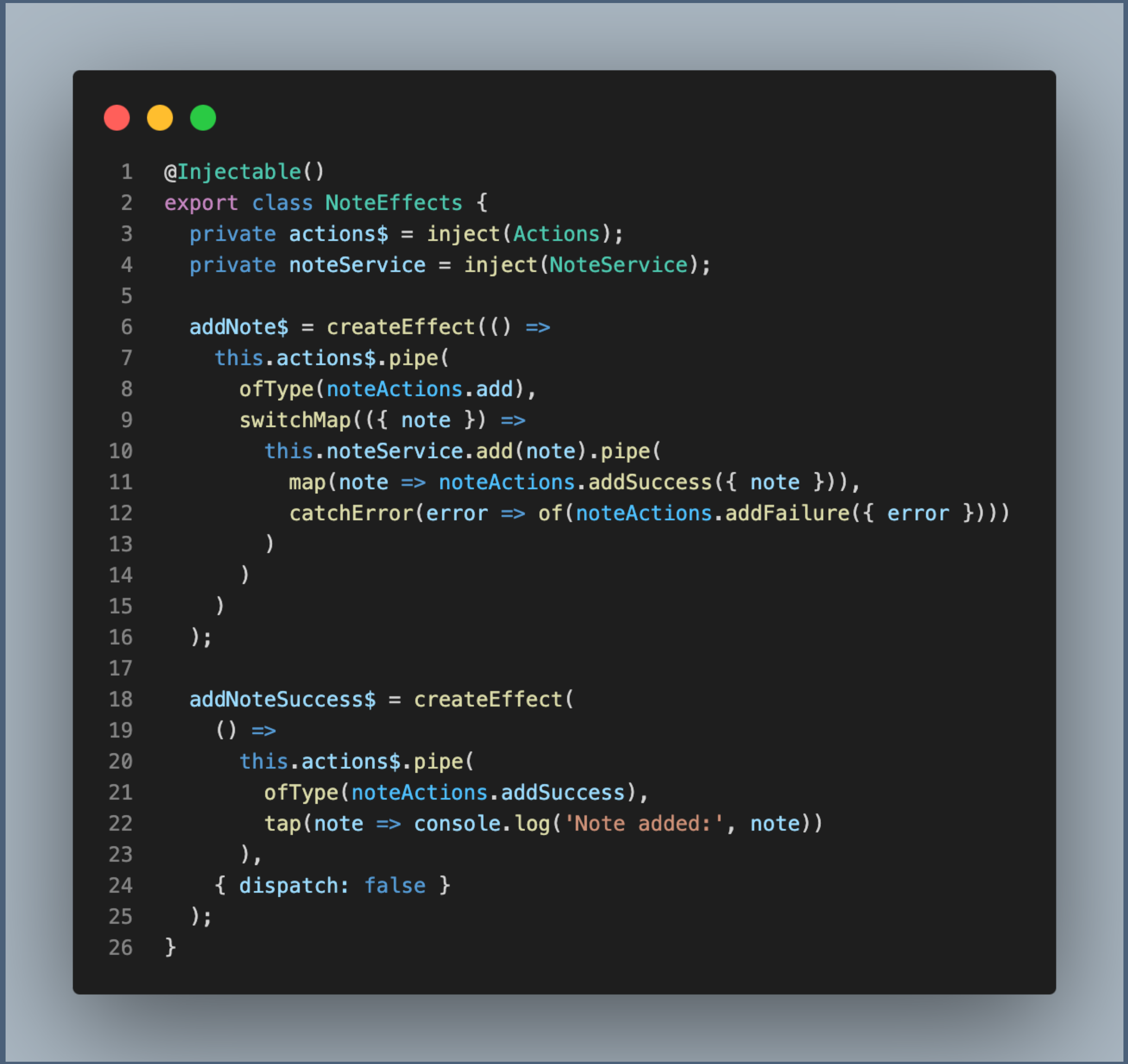
Action Groups



```typescript
export const noteActions = createActionGroup({
  source: 'Note',
  events: {
    Add: props<{ note: Note }>(),
    'Add Success': props<{ note: Note }>(),
    'Add Failure': props<{ error: string }>(),
  },
});
```

Effects

Do something, not state related, 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.

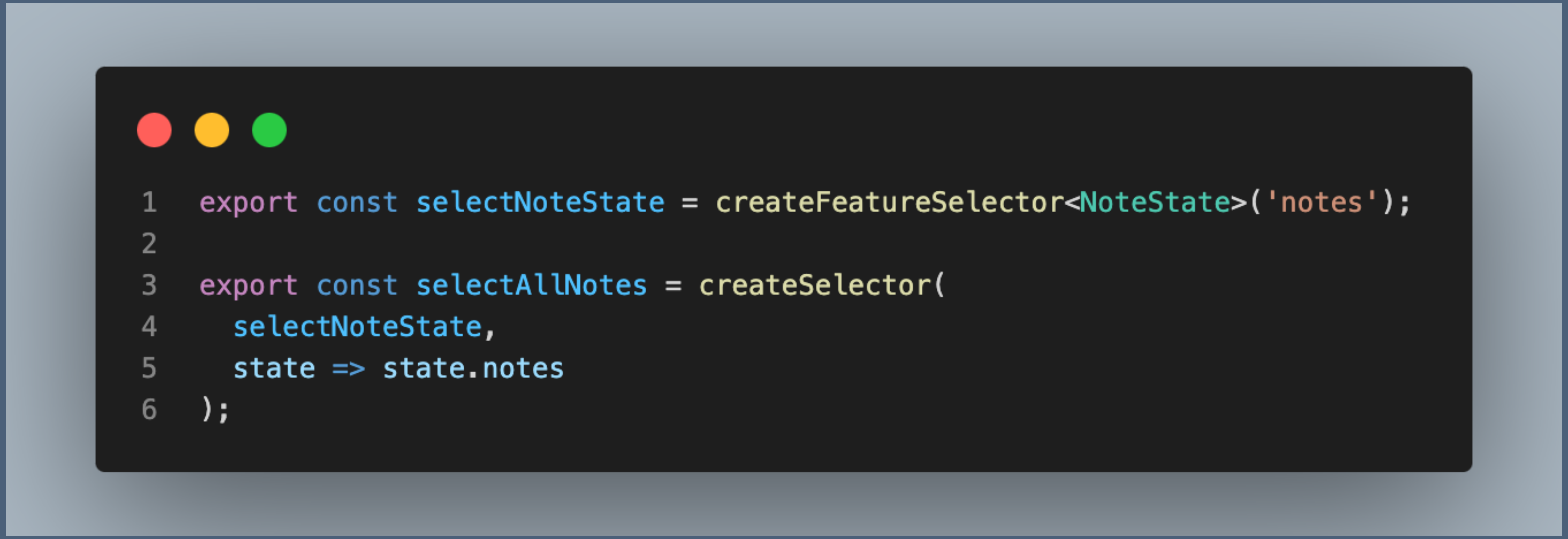
If not, don't forget to add

{ dispatch: false } as the

second parameter of createAction method

Selectors

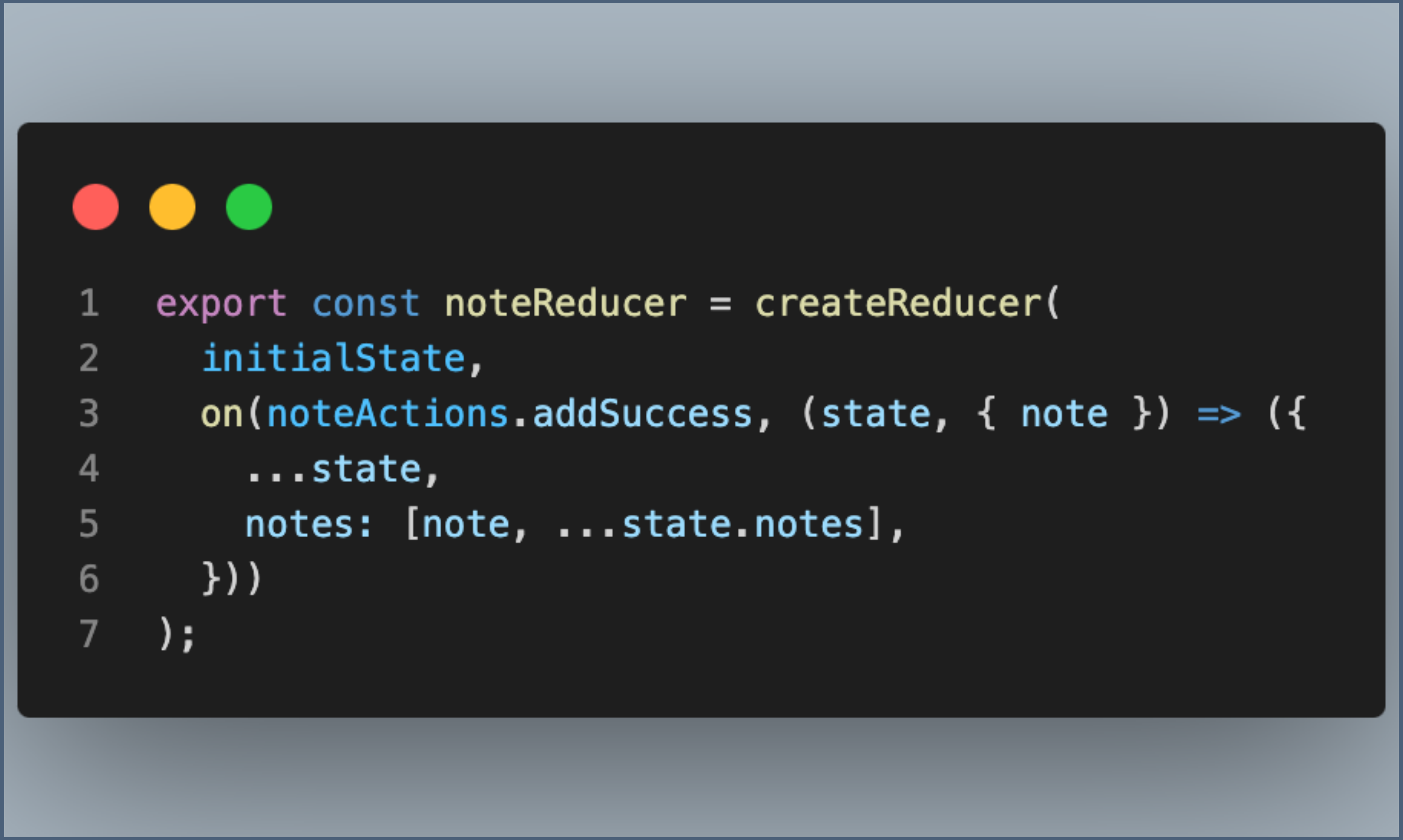
Extract the data in the store



```
export const selectNoteState = createFeatureSelector<NoteState>('notes');

export const selectAllNotes = createSelector(
  selectNoteState,
  state => state.notes
);
```

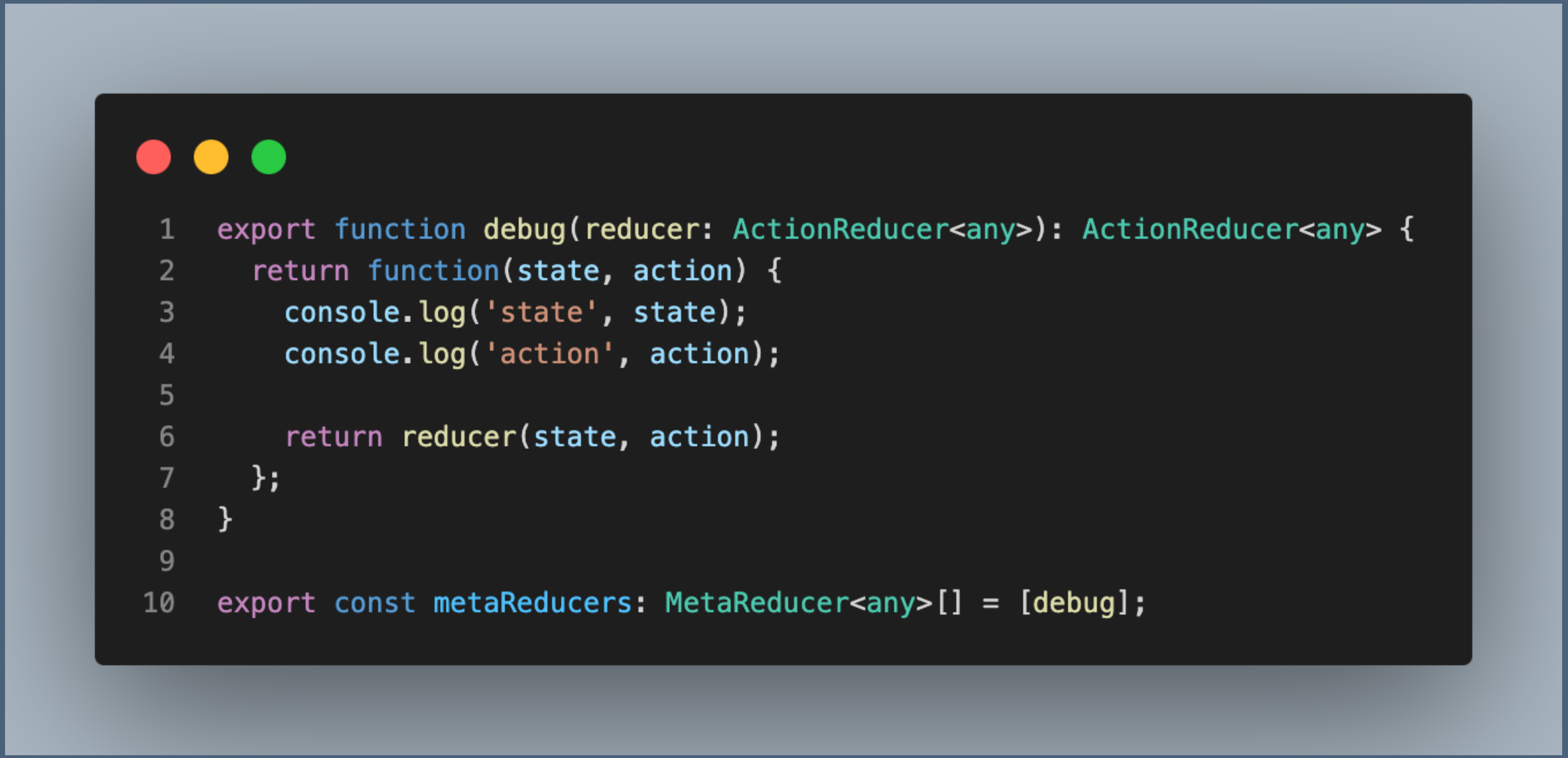
Reducers

Produce a new state with the last one and an action



```
export const noteReducer = createReducer(
  initialState,
  on(noteActions.addSuccess, (state, { note }) => ({
    ...state,
    notes: [note, ...state.notes],
  }))
);
```

Meta-reducers



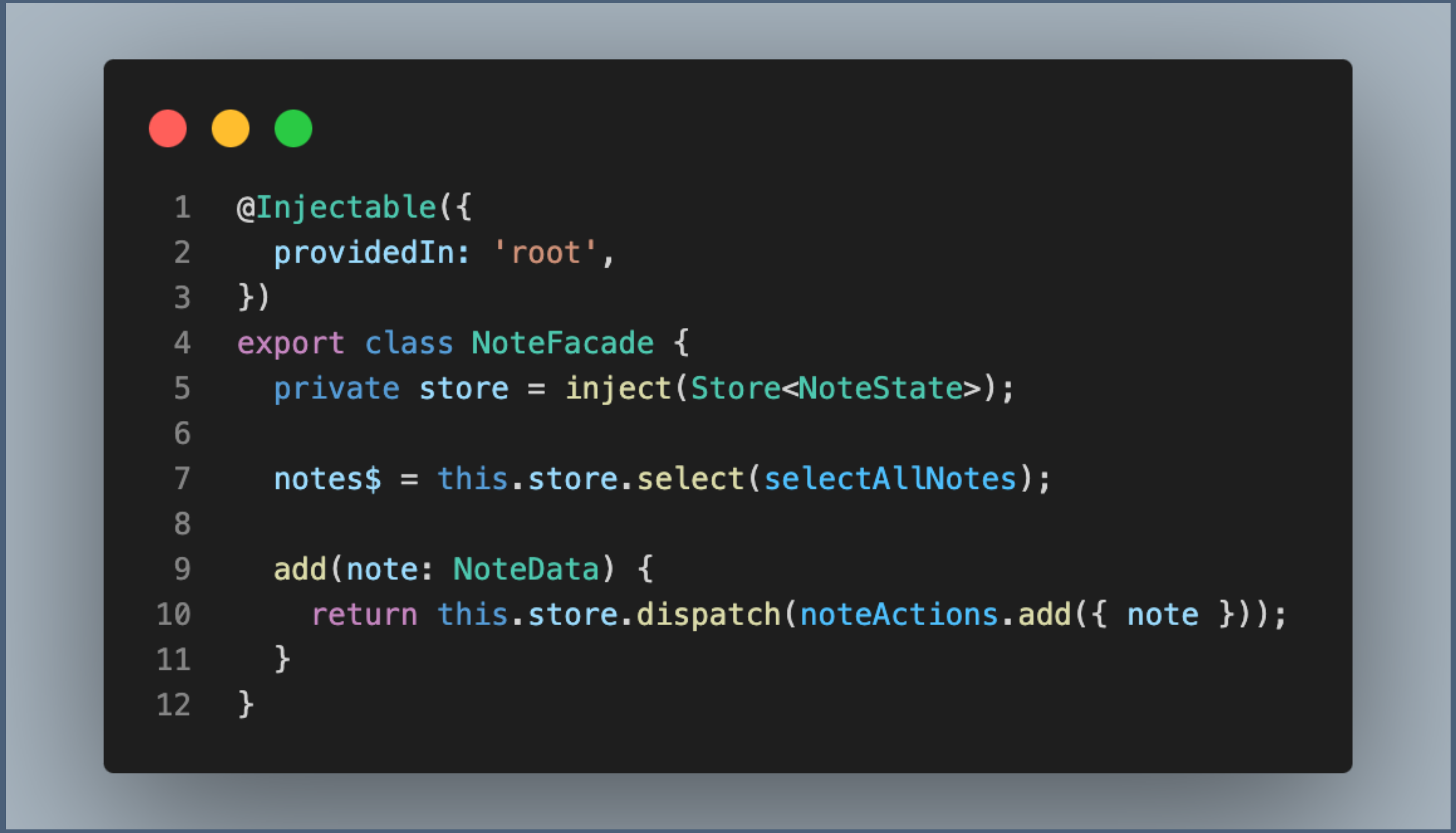
```typescript
export function debug(reducer: ActionReducer<any>): ActionReducer<any> {
  return function(state, action) {
    console.log('state', state);
    console.log('action', action);

    return reducer(state, action);
  };
}

export const metaReducers: MetaReducer<any>[] = [debug];
```

Called before the « normal » reducers

Allows to pre-process actions

How to use in your app ?

The best is to create a facade



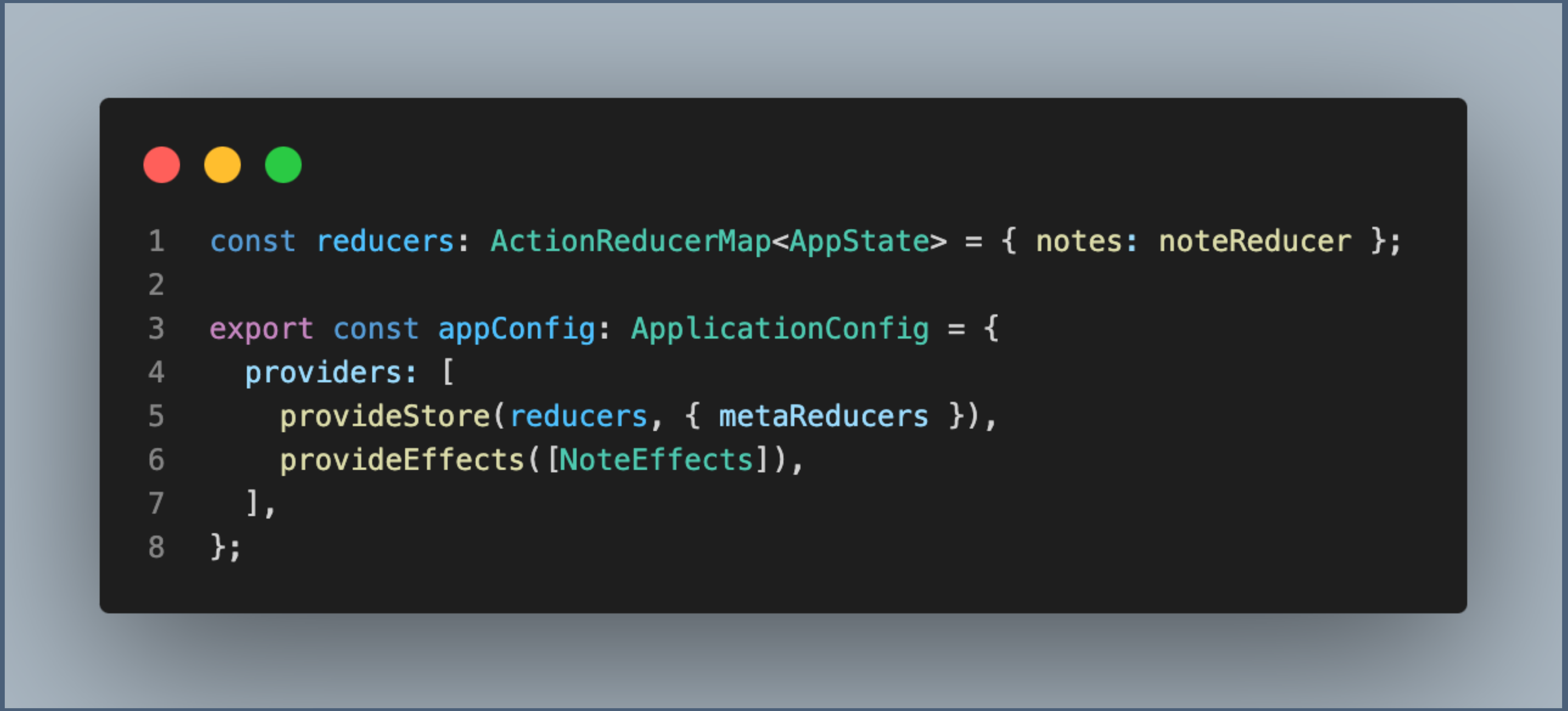
```
@Injectable({
  providedIn: 'root',
})
export class NoteFacade {
  private store = inject(Store<NoteState>);

  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



```
const reducers: ActionReducerMap<AppState> = { notes: noteReducer };

export const appConfig: ApplicationConfig = {
  providers: [
    provideStore(reducers, { metaReducers }),
    provideEffects([NoteEffects]),
  ],
};
```

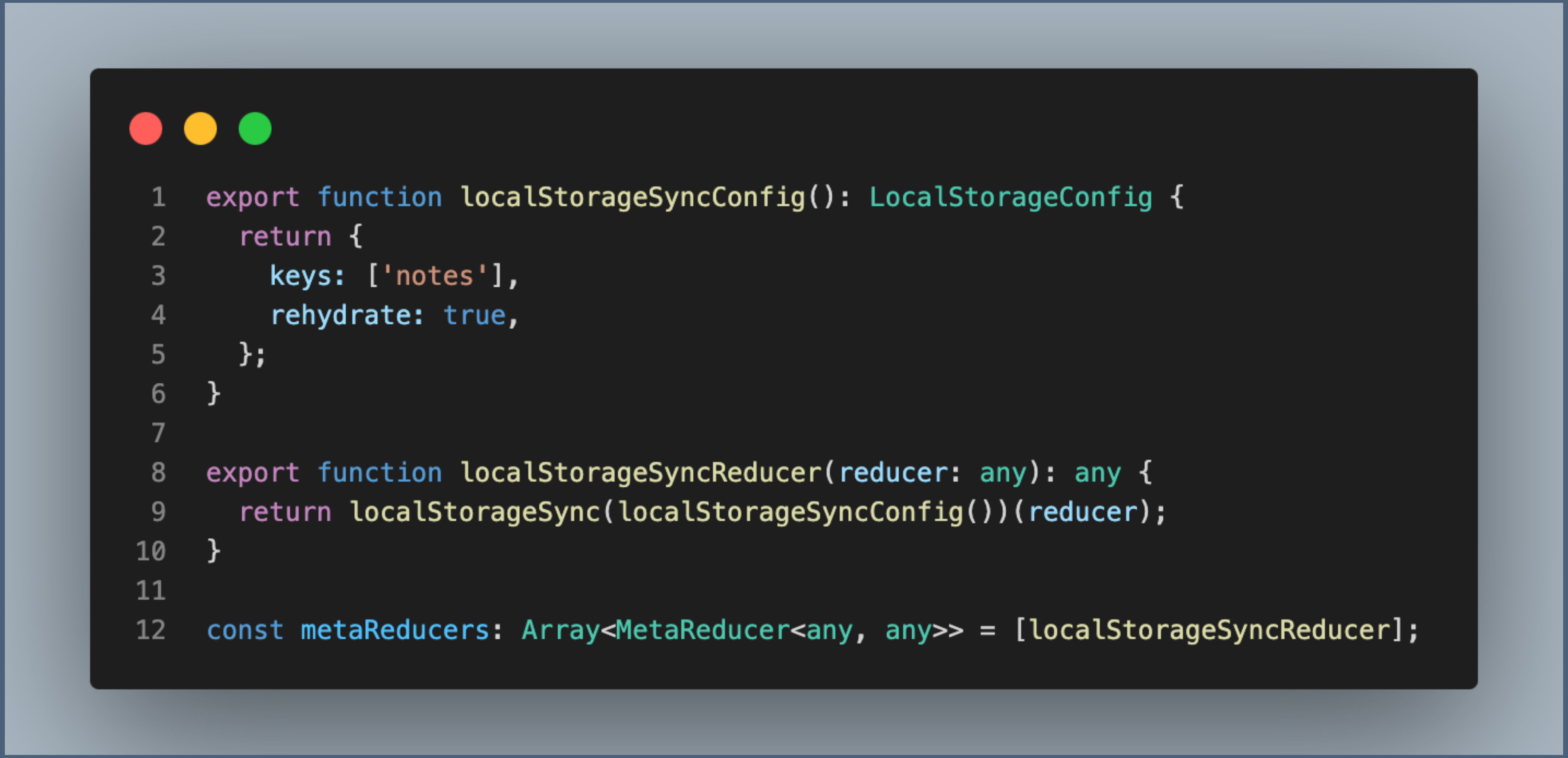
You can also use ProvideState especially

if you organize your different sub-state

into features

Saved state on page refresh

ngrx-store-localstorage



```
export function localStorageSyncConfig(): LocalStorageConfig {
  return {
    keys: ['notes'],
    rehydrate: true,
  };
}

export function localStorageSyncReducer(reducer: any): any {
  return localStorageSync(localStorageSyncConfig())(reducer);
}

const metaReducers: Array<MetaReducer<any, any>> = [localStorageSyncReducer];
```

Use cases

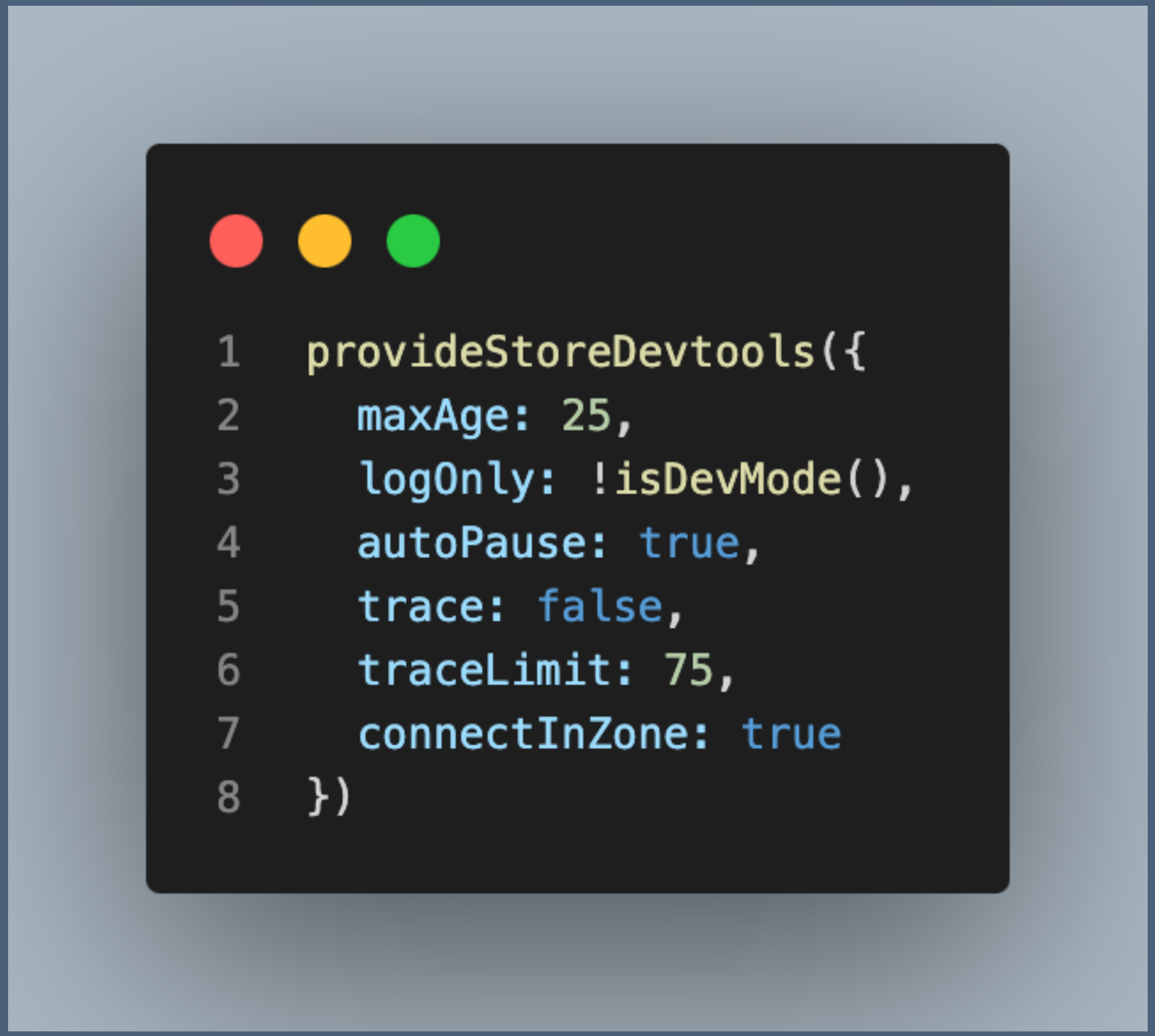
- Synchronizing the data with an api
- Sharing state data between many components
- Complex state

Debugging

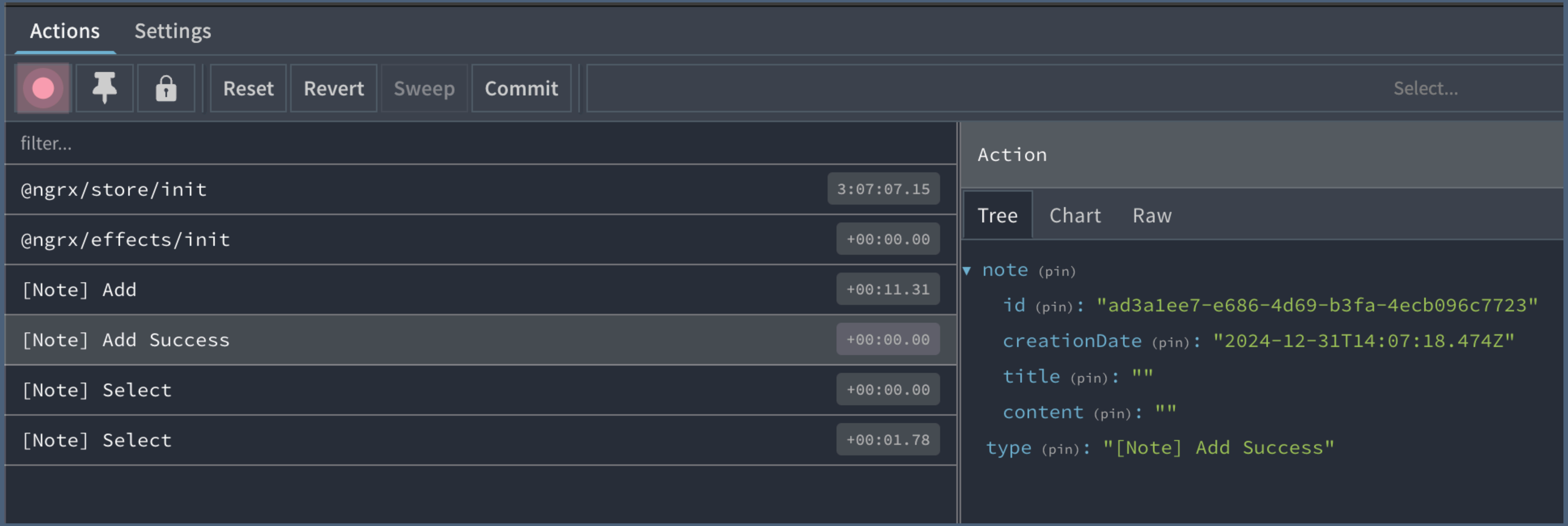
Redux Devtools

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

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



```
provideStoreDevtools({
  maxAge: 25,
  logOnly: !isDevMode(),
  autoPause: true,
  trace: false,
  traceLimit: 75,
  connectInZone: true
})
```



```
@ngrx/store/init          3:07:07.15
@ngrx/effects/init        +00:00.00
[Note] Add                +00:11.31
[Note] Add Success        +00:00.00
[Note] Select             +00:00.00
[Note] Select             +00:01.78
```

```
▾ note (pin)
    id (pin): "ad3a1ee7-e686-4d69-b3fa-4ecb096c7723"
    creationDate (pin): "2024-12-31T14:07:18.474Z"
    title (pin): ""
    content (pin): ""
  type (pin): "[Note] Add Success"
```

Best practices

- Don’t overuse the store (just what is necessary).
- Create Features to have different modules

Thanks a lot

Questions ?

NgRx documentation: https://ngrx.io/

Contact info: benjamin.canape@gmail.com