NGRX IS:

An Angular implementation of Redux

Why do we care about Redux?

If you experience the following symtoms you might need Redux:

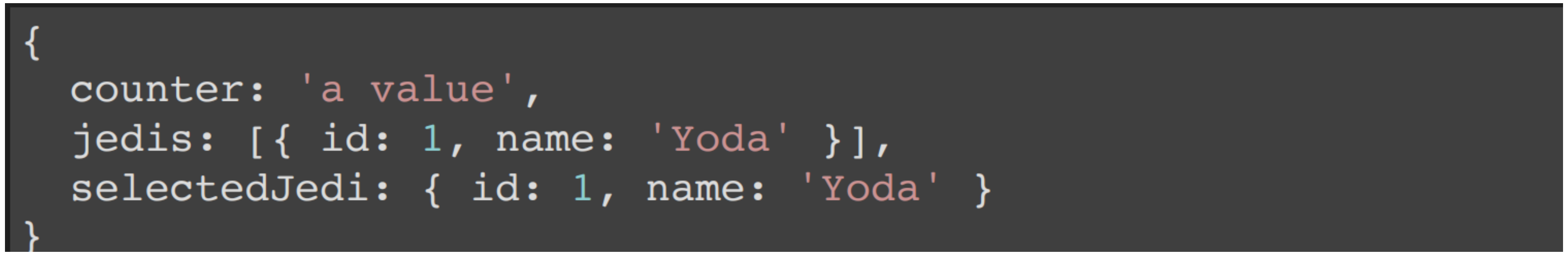
- A feeling that the state is spread out
- There are issues with updating state
- Some things keep changing the state but you don't know who or what

- Solution: a single source of truth with reducers guarding state change. Also enhanced predictability with immutable data structures

CORE CONCEPTS

- Store, our data store
- Reducer, a function that takes state + action and produces a new state
- Selector, selects a slice of state
- Action, an intention of state change
- Action creator, produces an intention that may include data

Typical Store content, just an object

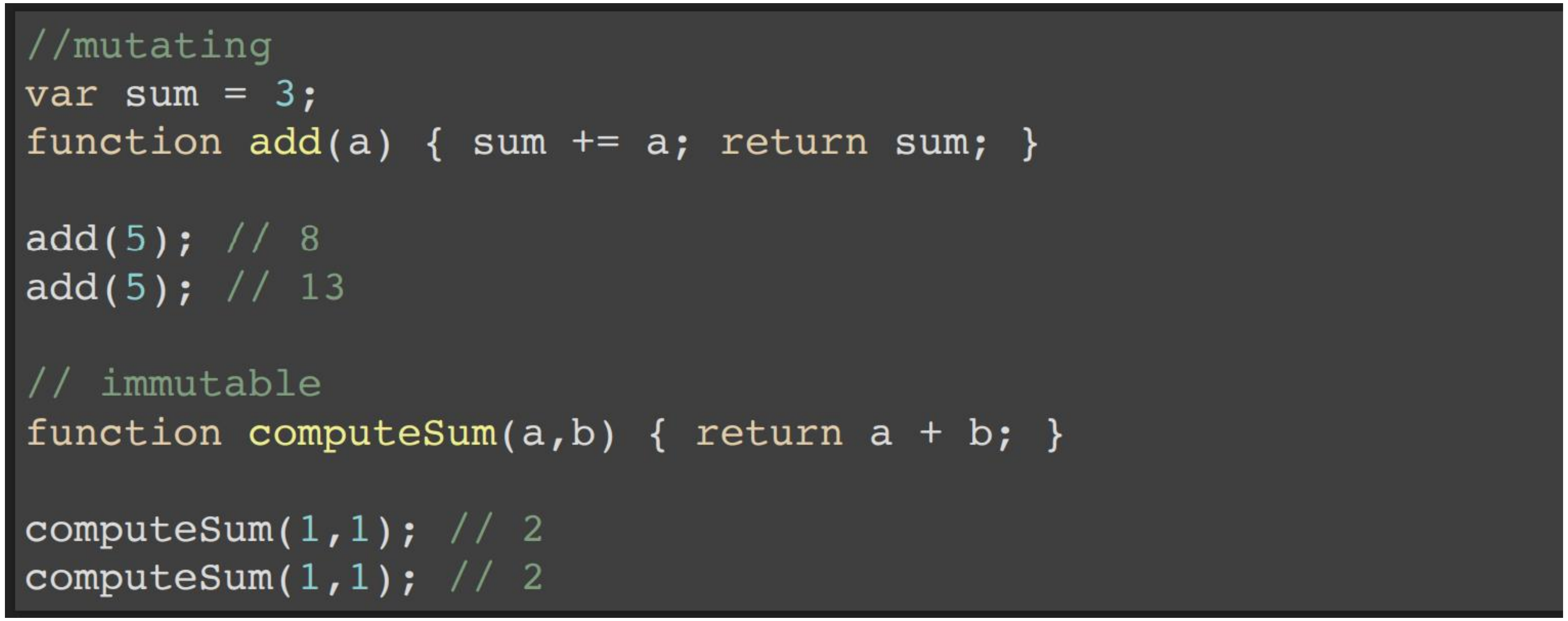


```
{
  counter: 'a value',
  jedis: [{ id: 1, name: 'Yoda' }],
  selectedJedi: { id: 1, name: 'Yoda' }
}
```

REDUCER

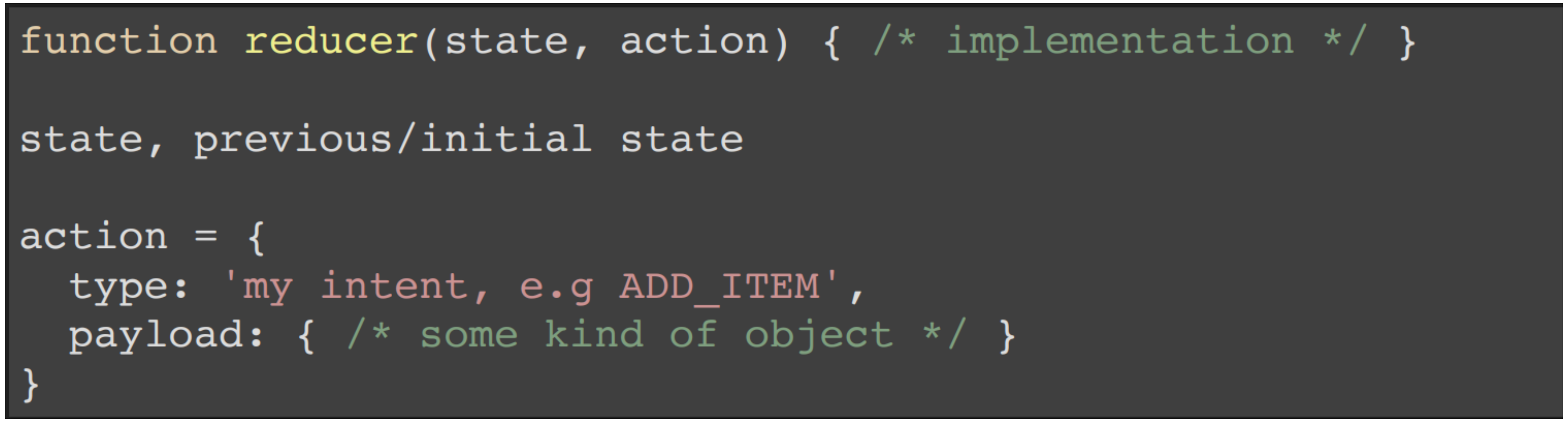
NEW STATE = STATE + ACTION

Mathematical function, immutable, just a calculation



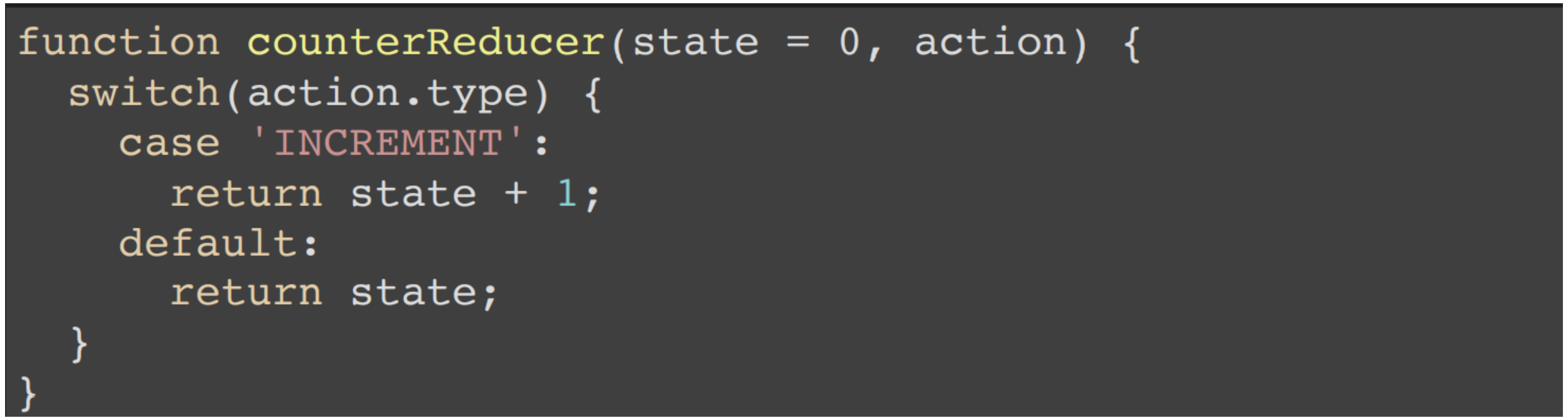
```
//mutating
var sum = 3;
function add(a) { sum += a; return sum; }

add(5); // 8
add(5); // 13

// immutable
function computeSum(a,b) { return a + b; }

computeSum(1,1); // 2
computeSum(1,1); // 2
```

A reducer looks like the following:



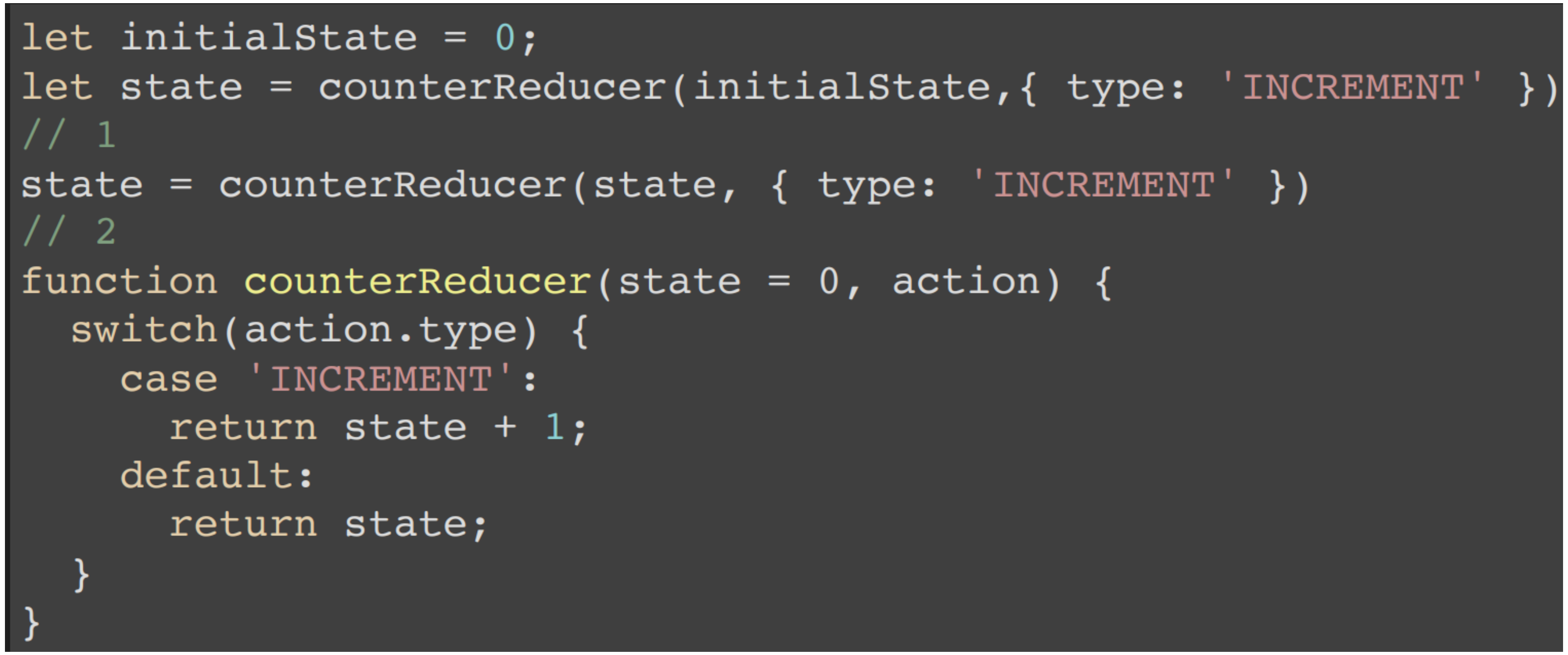
```
function reducer(state, action) { /* implementation */ }

state, previous/initial state

action = {
  type: 'my intent, e.g ADD_ITEM',
  payload: { /* some kind of object */ }
}
```

state + 1, instead of state +=1, immutable



```
function counterReducer(state = 0, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1;
    default:
      return state;
  }
}
```

Usage



```javascript
let initialState = 0;
let state = counterReducer(initialState,{ type: 'INCREMENT' })
// 1
state = counterReducer(state, { type: 'INCREMENT' })
// 2
function counterReducer(state = 0, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1;
    default:
      return state;
  }
}
```

A simple store

```
class Store {
    constructor(initialState) { this.state = initialState; }
    dispatch(action) {
        this.state = calc(this.state, action);
    }
    calc(state, action) {
        return {
            counter: counterReducer(state.counter, action)
        }
    }
}
```

Usage, store

- `let store = new Store({ counter: 0 });`
- `store.dispatch({ type: 'INCREMENT' });`

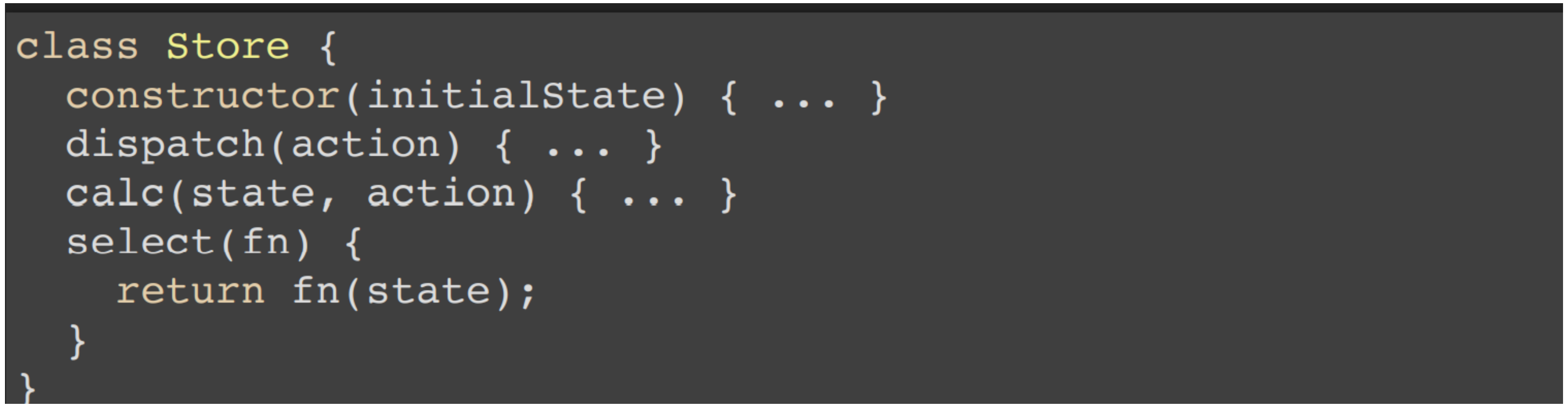
Action, an object with a property 'type' and 'payload’

    'type' = intent

    'payload' = the change

```
{ type: ‘INCREMENT', payload: {} }
{ type: 'ADD_USER', payload: { id: 1, name: ‘user_name' } }
```

Selector, slice of state



```
class Store {
  constructor(initialState) { ... }
  dispatch(action) { ... }
  calc(state, action) { ... }
  select(fn) {
    return fn(state);
  }
}
```

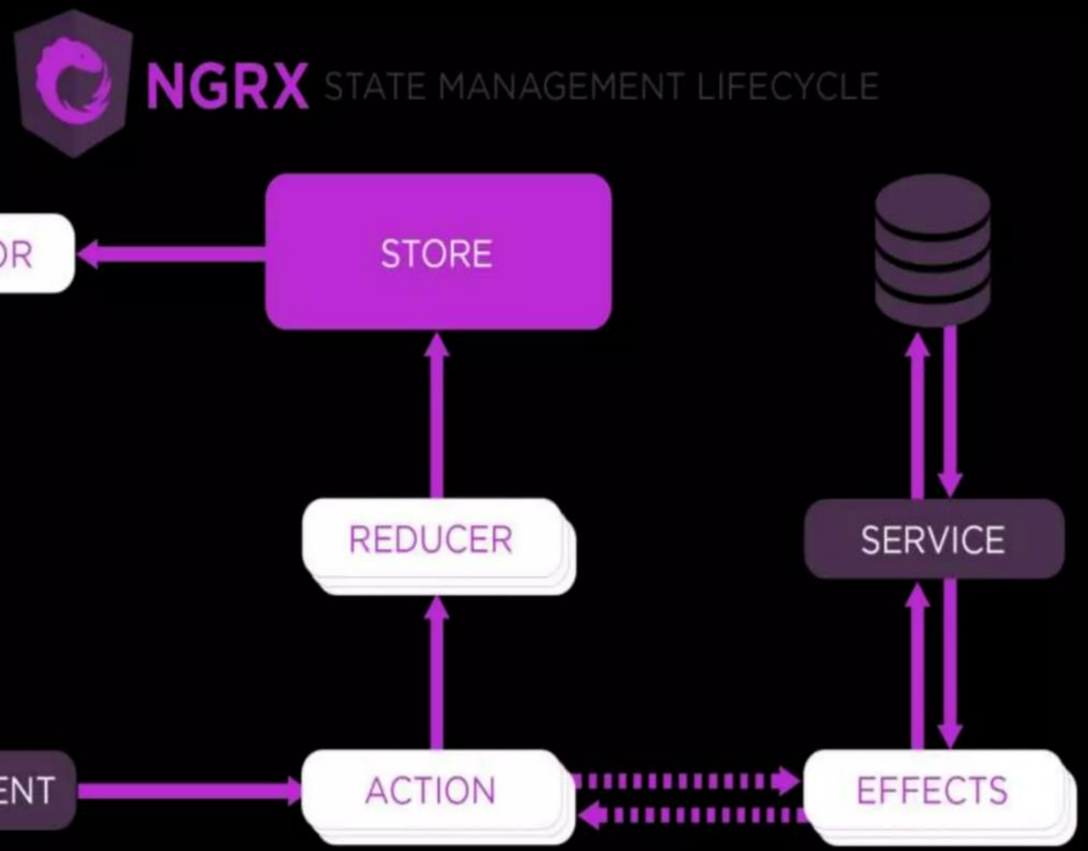
Selector, definitions

```
const getCounter = (state) => state.counter;
```

OVERVIEW OF LIBRARIES

- @ngrx/store, the store
- @ngrx/store-devtools, a debug tool that helps you track dispatched actions
- @ngrx/effects, handles side effects
- @ngrx/router-store, lets you put the routing state in the store
- @ngrx/entites, handles records
- @ngrx/schematics

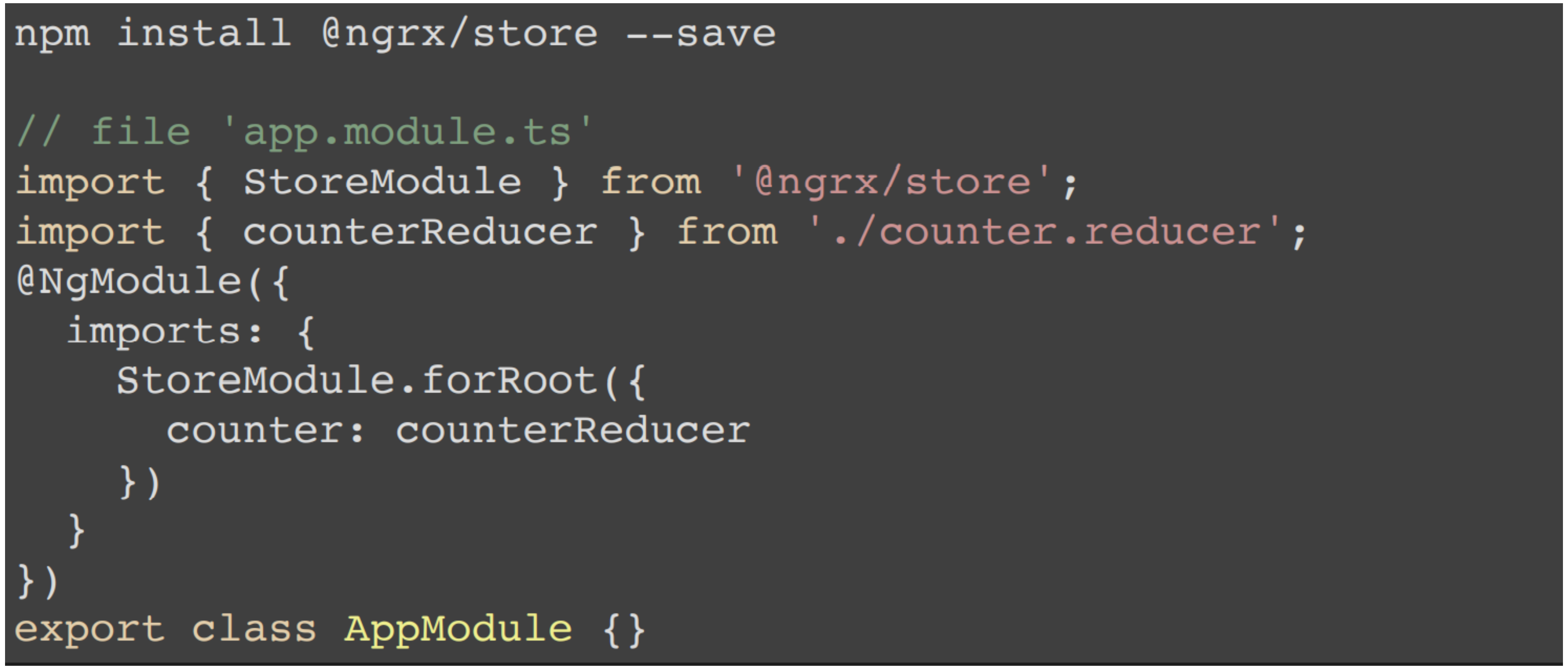
- Actions
- Reducers



STORE

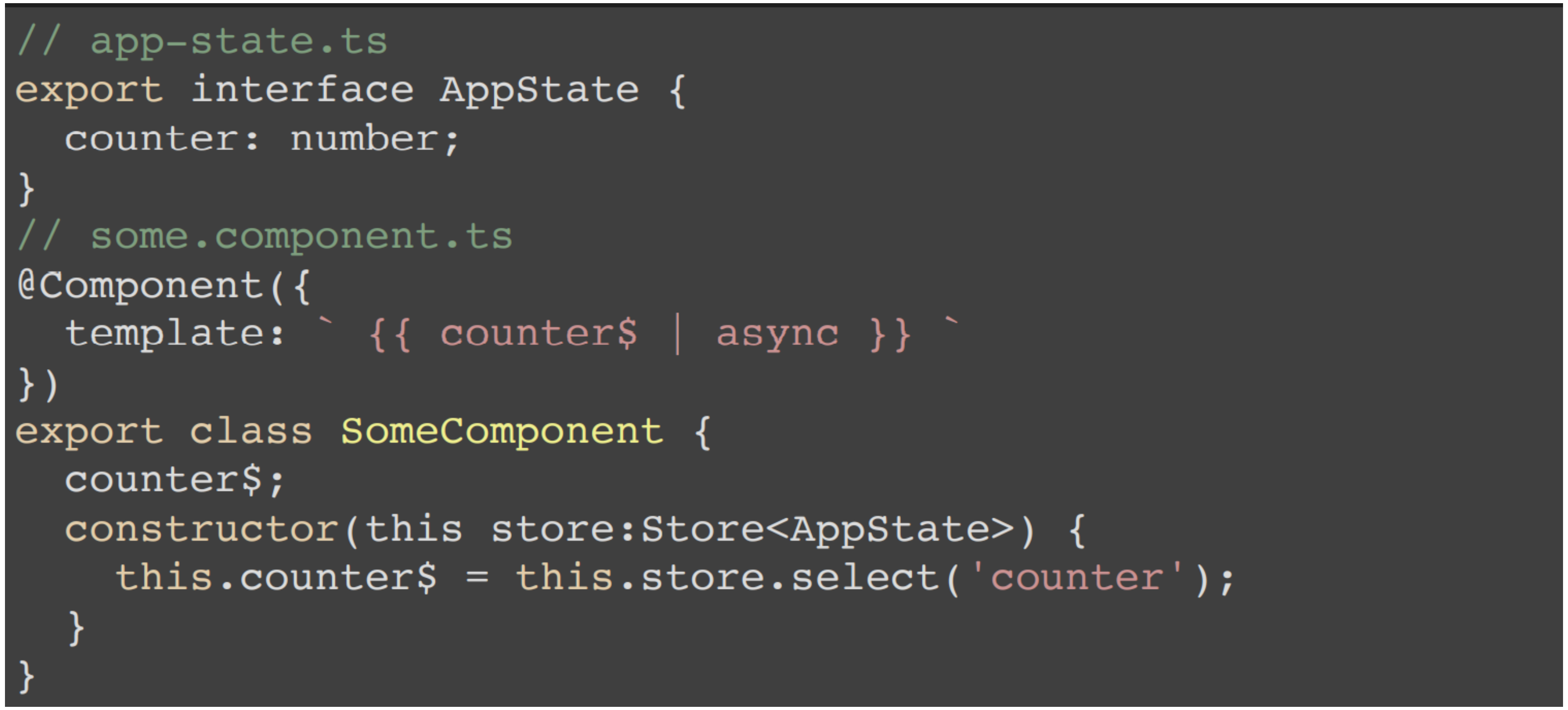
WHERE THE STATE LIVES

INSTALLATION AND SET UP



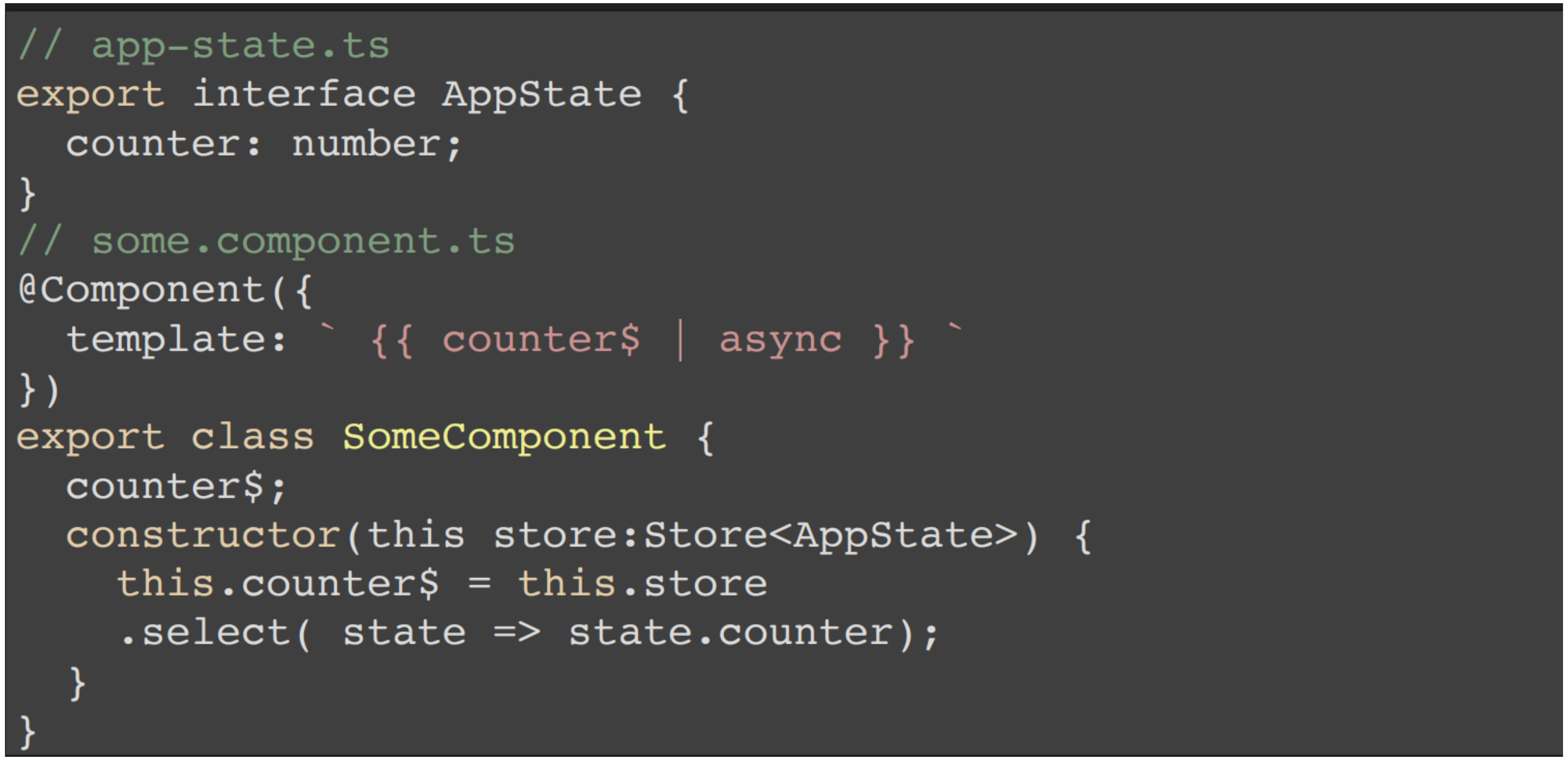
```
npm install @ngrx/store --save

// file 'app.module.ts'
import { StoreModule } from '@ngrx/store';
import { counterReducer } from './counter.reducer';
@NgModule({
  imports: {
    StoreModule.forRoot({
      counter: counterReducer
    })
  }
})
export class AppModule {}
```

SHOW DATA FROM STORE



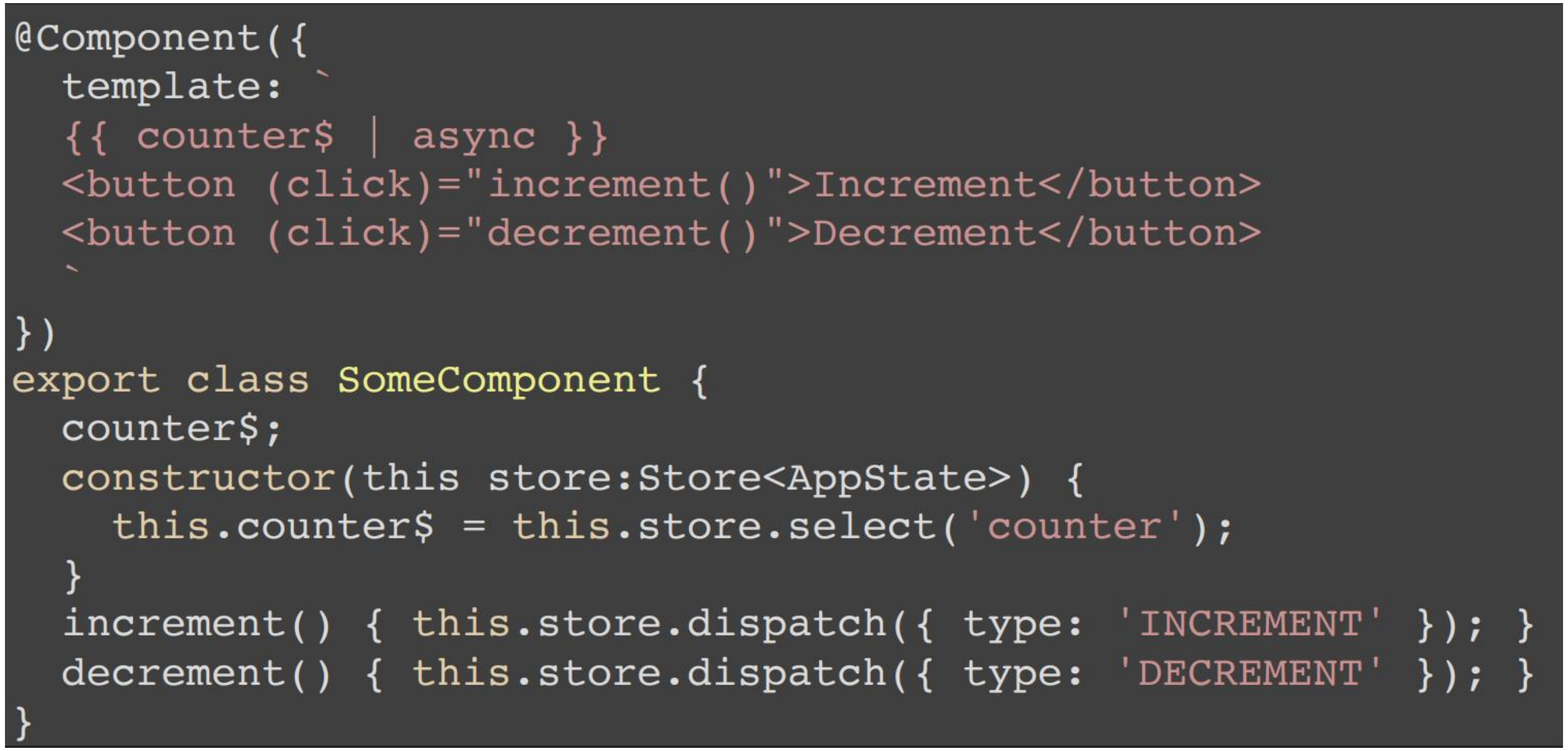
```
// app-state.ts
export interface AppState {
  counter: number;
}
// some.component.ts
@Component({
  template: ` {{ counter$ | async }} `
})
export class SomeComponent {
  counter$;
  constructor(this store:Store<AppState>) {
    this.counter$ = this.store.select('counter');
  }
}
```

SHOW DATA FROM STORE, SELECTOR FUNCTION



```
// app-state.ts
export interface AppState {
  counter: number;
}
// some.component.ts
@Component({
  template: ` {{ counter$ | async }} `
})
export class SomeComponent {
  counter$;
  constructor(this store:Store<AppState>) {
    this.counter$ = this.store
    .select( state => state.counter);
  }
}
```

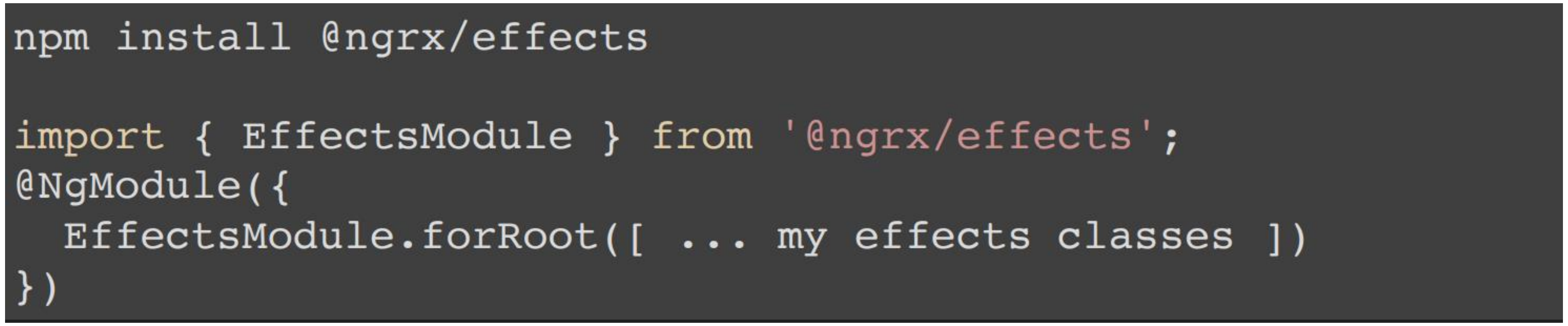
DISPATCH DATA



```
@Component({
  template: `
  {{ counter$ | async }}
  <button (click)="increment()">Increment</button>
  <button (click)="decrement()">Decrement</button>
  `
})
export class SomeComponent {
  counter$;
  constructor(this store:Store<AppState>) {
    this.counter$ = this.store.select('counter');
  }
  increment() { this.store.dispatch({ type: 'INCREMENT' }); }
  decrement() { this.store.dispatch({ type: 'DECREMENT' }); }
}
```

NGRX EFFECTS

- INSTALLATION AND SETUP

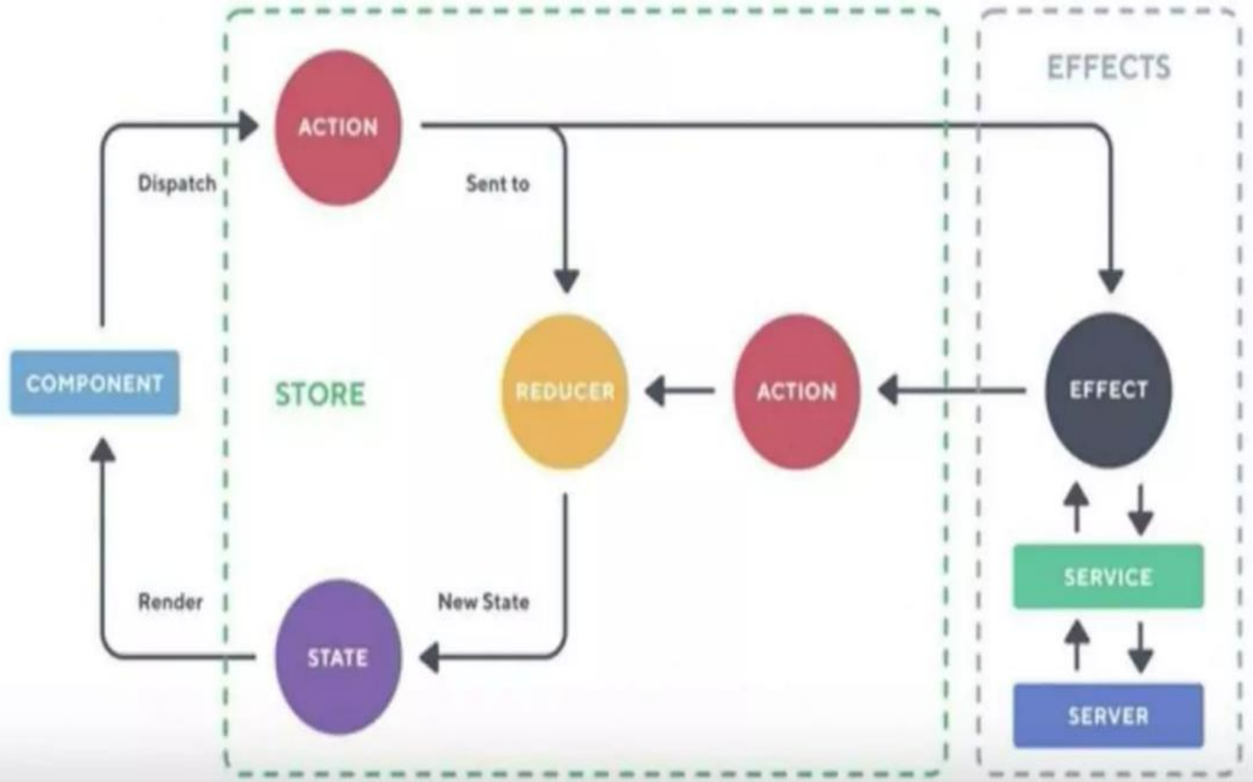


```
npm install @ngrx/effects

import { EffectsModule } from '@ngrx/effects';
@NgModule({
  EffectsModule.forRoot([ ... my effects classes ])
})
```

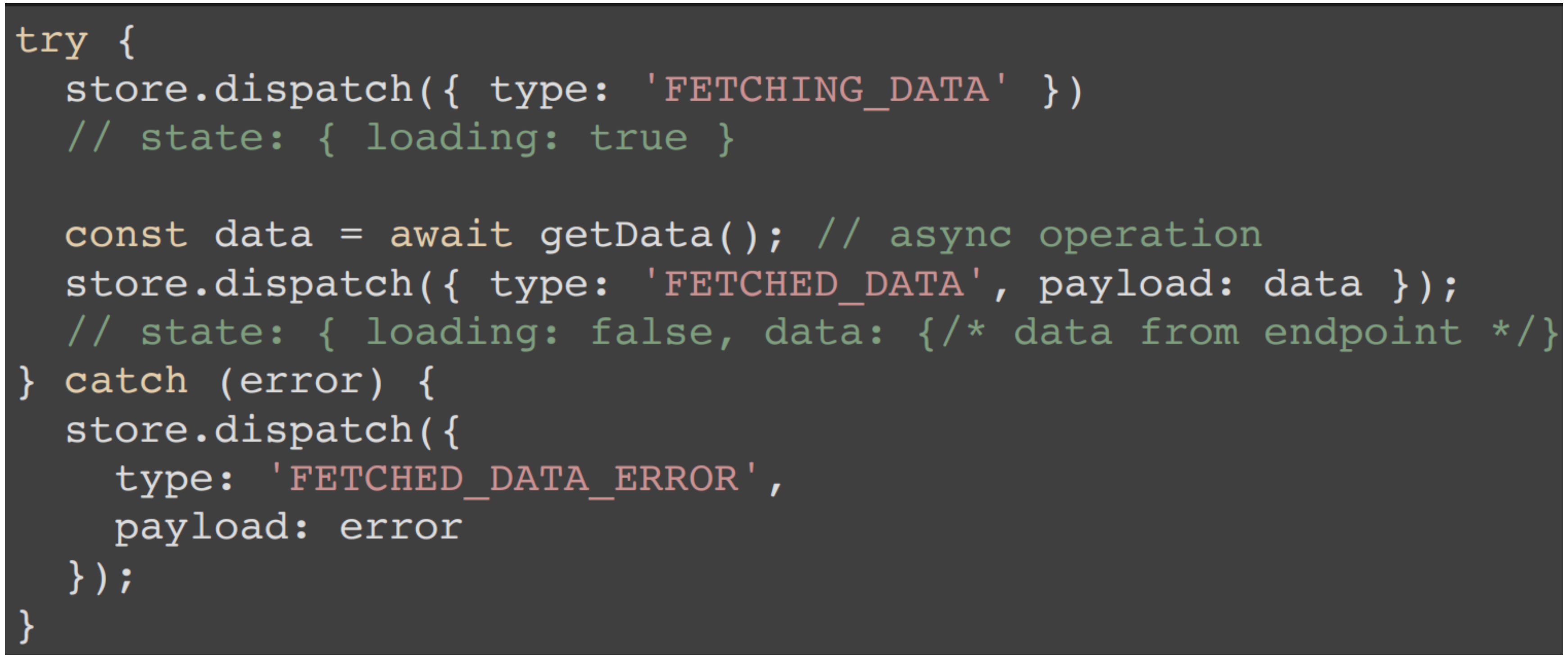
What kind of behaviour do we want?

- Set a loading flag, show spinner
- Do AJAX call
- Show fetched data or error
- Set loading flag to false, hide spinner

Effects flow

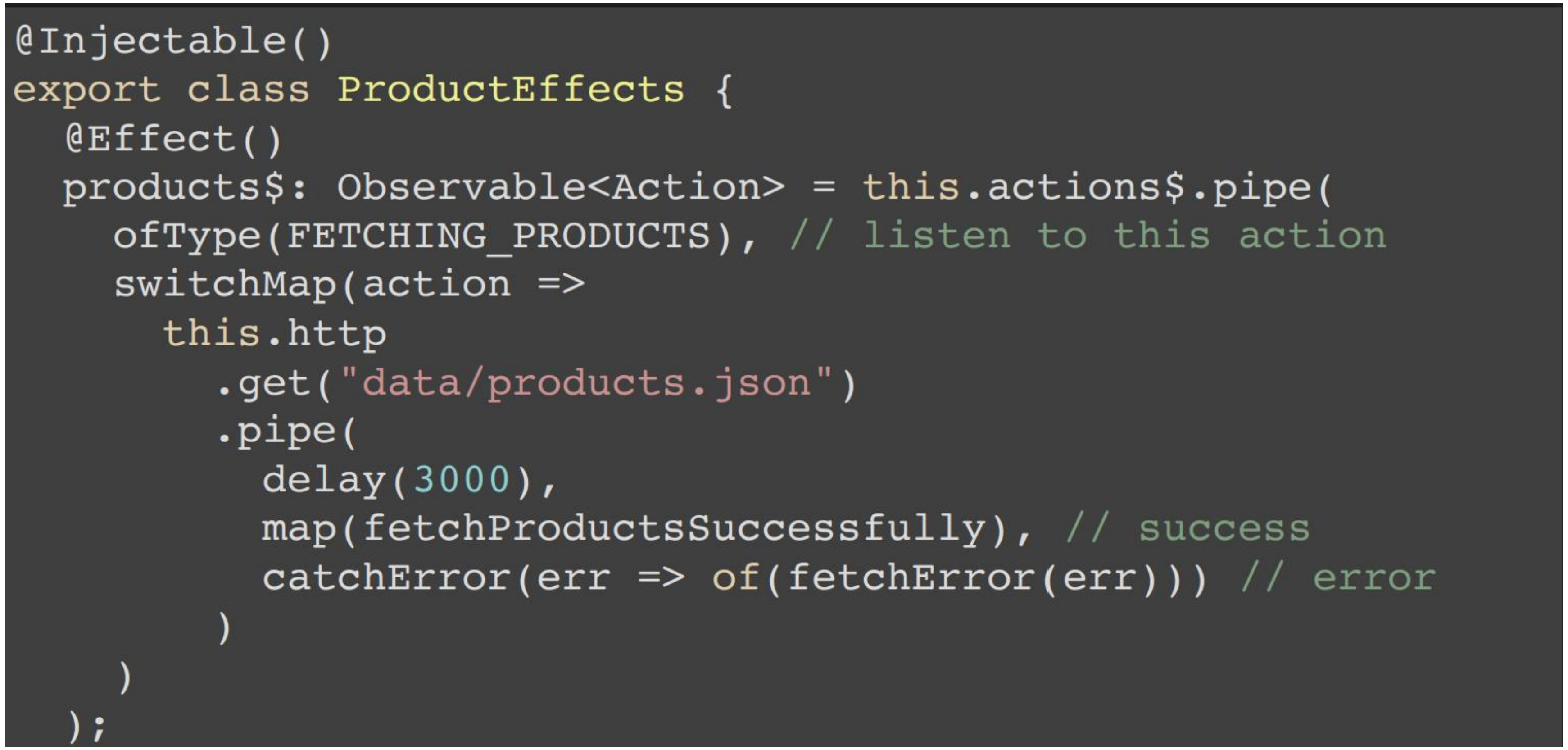


NGRX approach

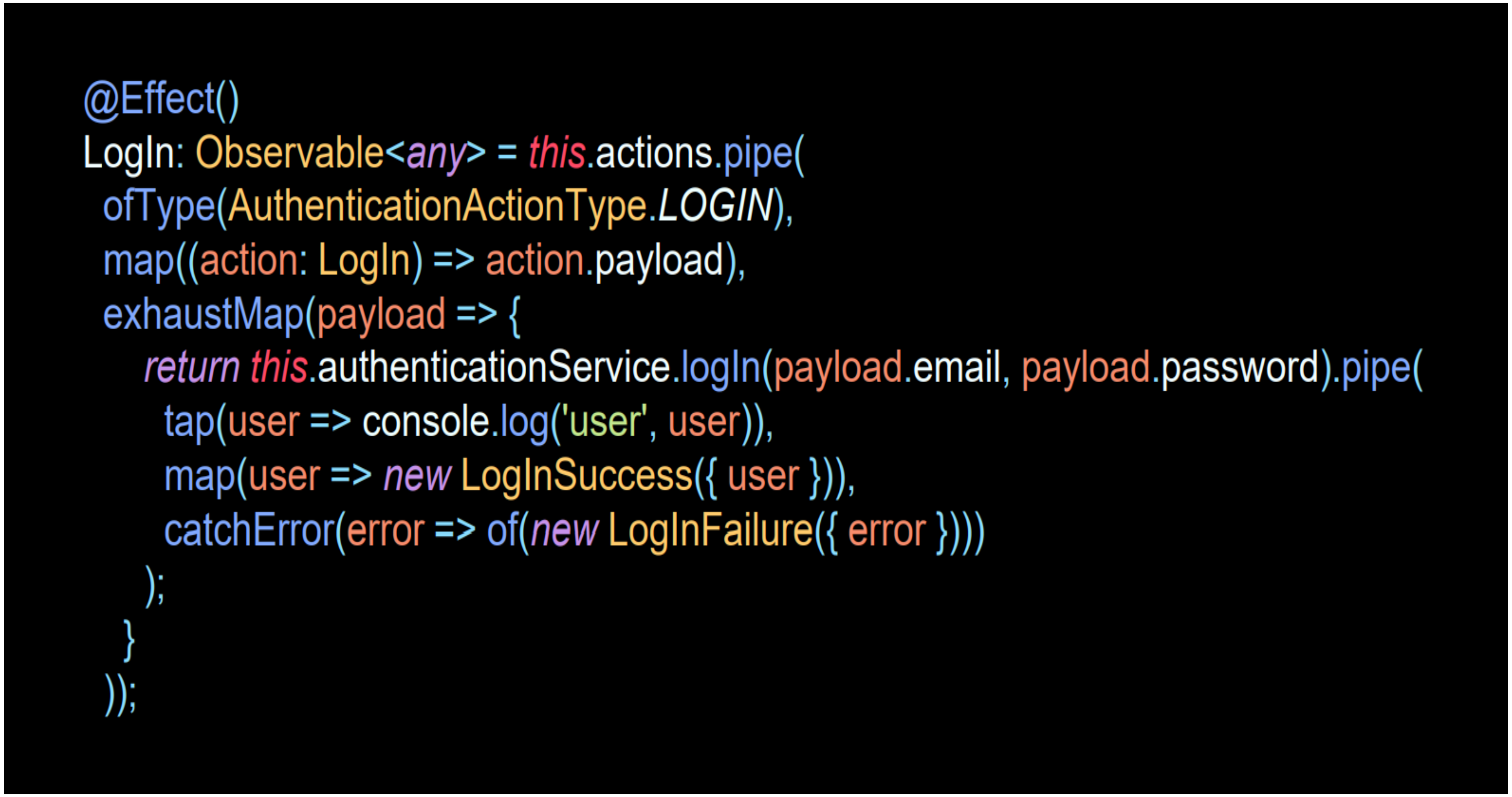


```
try {
  store.dispatch({ type: 'FETCHING_DATA' })
  // state: { loading: true }

  const data = await getData(); // async operation
  store.dispatch({ type: 'FETCHED_DATA', payload: data });
  // state: { loading: false, data: {/* data from endpoint */}
} catch (error) {
  store.dispatch({
    type: 'FETCHED_DATA_ERROR',
    payload: error
  });
}
```

Effect - calling HTTP



```
@Injectable()
export class ProductEffects {
  @Effect()
  products$: Observable<Action> = this.actions$.pipe(
    ofType(FETCHING_PRODUCTS), // listen to this action
    switchMap(action =>
      this.http
        .get("data/products.json")
        .pipe(
          delay(3000),
          map(fetchProductsSuccessfully), // success
          catchError(err => of(fetchError(err))) // error
        )
    )
  );
```



```
@Effect()
LogIn: Observable<any> = this.actions.pipe(
  ofType(AuthenticationActionType.LOGIN),
  map((action: LogIn) => action.payload),
  exhaustMap(payload => {
      return this.authenticationService.logIn(payload.email, payload.password).pipe(
        tap(user => console.log('user', user)),
        map(user => new LogInSuccess({ user })),
        catchError(error => of(new LogInFailure({ error })))
      );
    }
  ));
```