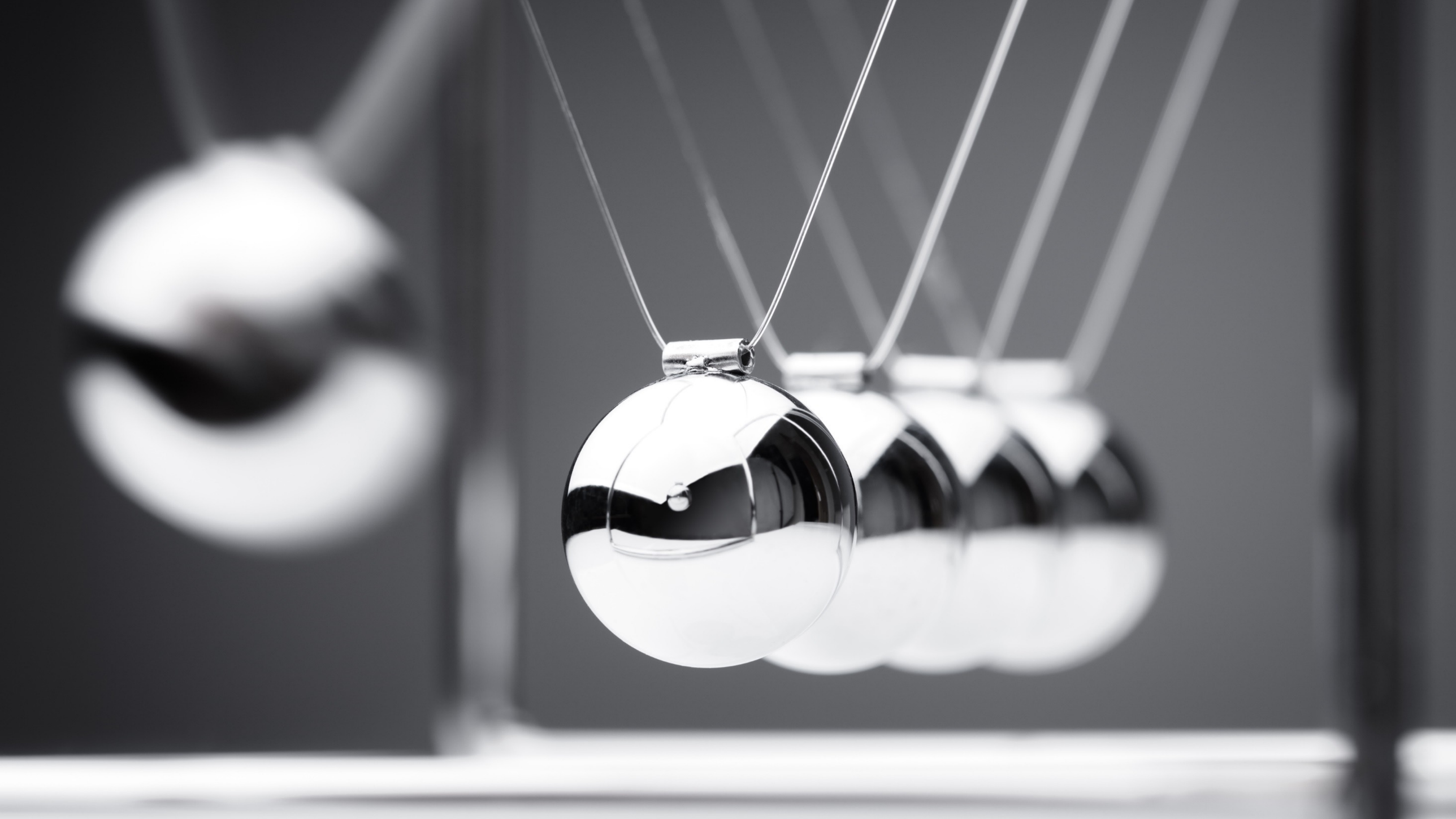
Working with Effects

Duncan Hunter
Lead Frontend Developer

@dunchunter | duncanhunter.com.au



Module Introduction

Why use effects?

Add @ngrx/effects

Define an effect

Register an effect

Use an effect

Exception handling in effects

Effects are an RxJS powered side effect model for NgRx Store.

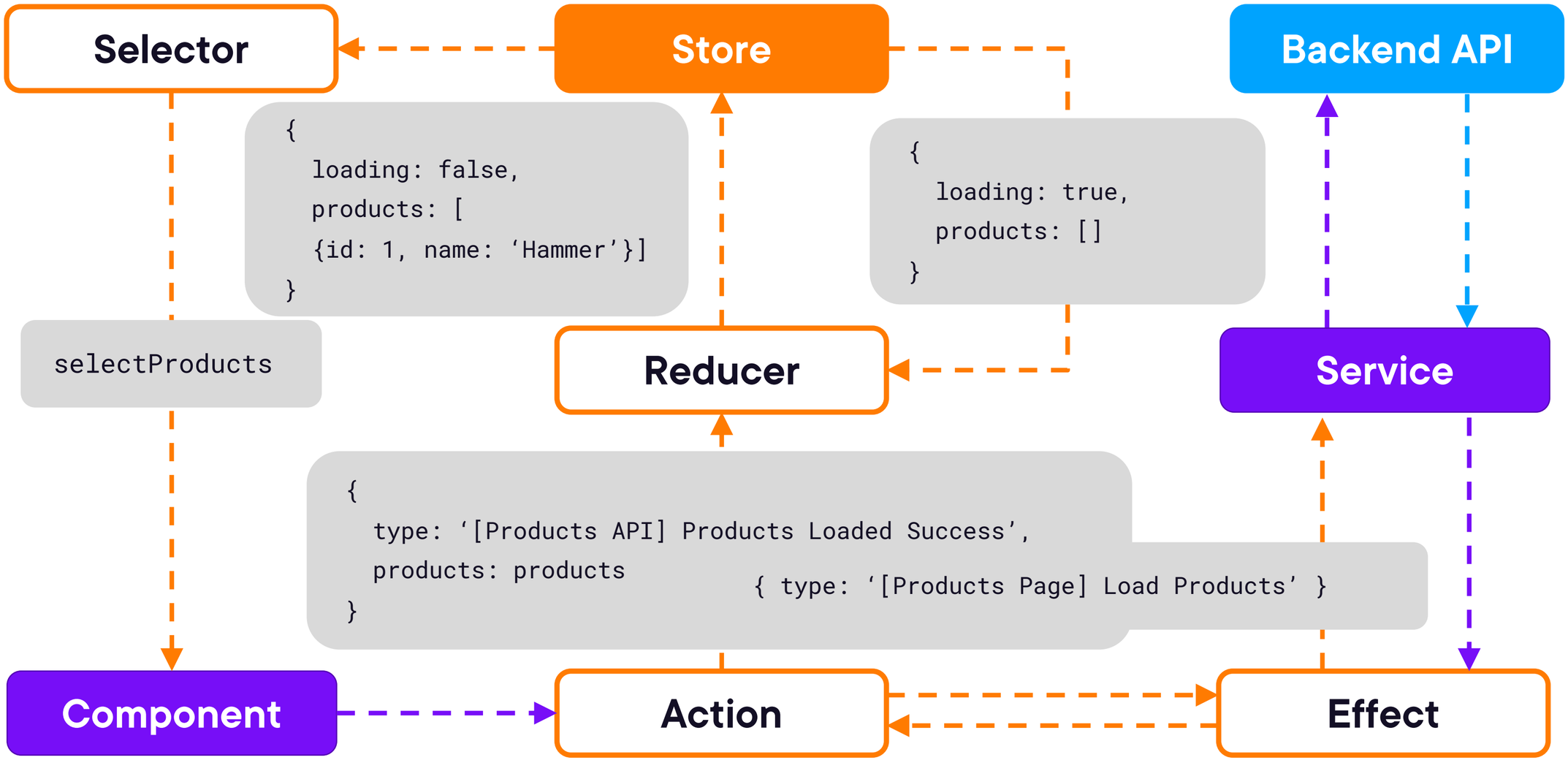
Effects Keep Components Pure

```
constructor(private productsService: ProductsService, private store: Store) {

getProducts() {
  this.store.dispatch(ProductsPageActions.loadProducts());
  this.productsService.getAll().subscribe({
    next: (products) => {
      this.store.dispatch(ProductsAPIActions.productsLoadedSuccess({ products }));
    }
  });
}
```

Products Reducer

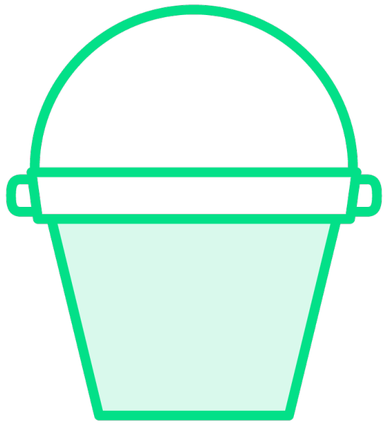
```
import { createReducer, on } from '@ngrx/store';
import { ProductsPageActions } from './products.actions';

this.productsService.getAll().subscribe({
    next: (products) => {
      this.store.dispatch(ProductsAPIActions.productsLoadedSuccess({ products
}))}

export const productsReducer = createReducer(
 initialState,
 on(createAction(ProductsPageActions.loadProducts), (state) => ({
  ...state,
  loading: true,
 }))
);
```

State Lifecycle – Load Products

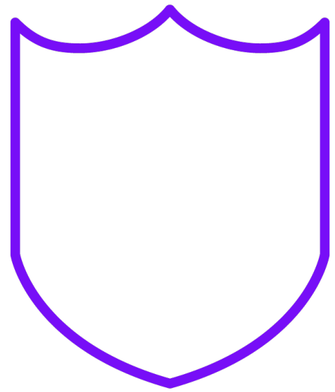


Benefits of Effects

Keep components pure



Isolate side effects



Easier to test

Defining an Effect

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions) {}




};
```

Defining an Effect

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}




};
```

Defining an Effect

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>



 );

};
```

Defining an Effect

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>
  this.actions$.pipe(



 ));

};
```

Defining an Effect

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>
  this.actions$.pipe(
   ofType(ProductsPageActions.loadProducts),


 ));

};
```

Defining an Effect

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>
  this.actions$.pipe(
   ofType(ProductsPageActions.loadProducts),
   concatMap(

 ));

};
```

Defining an Effect

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>
  this.actions$.pipe(
   ofType(ProductsPageActions.loadProducts),
   concatMap(() => this.productsService.getAll()

 ));

};
```

Defining an Effect

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>
  this.actions$.pipe(
   ofType(ProductsPageActions.loadProducts),
   concatMap(() => this.productsService.getAll().pipe(
     map((products) => ProductsAPIActions.productsLoadedSuccess({ products })),
 )));

};
```

Defining an Effect

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>
  this.actions$.pipe(
   ofType(ProductsPageActions.loadProducts),
   concatMap(() => this.productsService.getAll().pipe(
     map((products) => ProductsAPIActions.productsLoadedSuccess({ products })),
 )));

};
```

Defining an Effect

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>
  this.actions$.pipe(
   ofType(ProductsPageActions.loadProducts),
   concatMap(() => this.productsService.getAll().pipe(
     map((products) => ProductsAPIActions.productsLoadedSuccess({ products })),
 )));

};
```

Defining an Effect

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>
  this.actions$.pipe(
   ofType(ProductsPageActions.loadProducts),
   concatMap(() => this.productsService.getAll().pipe(
     map((products) => ProductsAPIActions.productsLoadedSuccess({ products })),
 )));

};
```

Demo

Install and define an effect

Defining an Effect

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>
  this.actions$.pipe(
   ofType(ProductsPageActions.loadProducts),
   concatMap(() => this.productsService.getAll().pipe(
     map((products) => ProductsAPIActions.productsLoadedSuccess({ products })),
 )));

};
```

Defining an Effect

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>
  this.actions$.pipe(
   ofType(ProductsPageActions.loadProducts),
   concatMap(() => this.productsService.getAll().pipe(
     map((products) => ProductsAPIActions.productsLoadedSuccess({ products })),
 )));

};
```

Exception Handling in Effects

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>
  this.actions$.pipe(
   ofType(ProductsPageActions.loadProducts),
   concatMap(() => this.productsService.getAll().pipe(
     map((products) => ProductsAPIActions.productsLoadedSuccess({ products })),
     catchError(
       (error) => of(ProductsAPIActions.productsLoadedFail({ message: error})
     )
 )));

};
```

Exception Handling in Reducer

```
export interface ProductsState {
 showProductCode: boolean;
 loading: boolean;
 products: Product[];
 errorMessage: string;
}
```

Exception Handling in Reducer

```
export interface ProductsState {
 showProductCode: boolean;
 loading: boolean;
 products: Product[];
 errorMessage: string;
}

const initalState: ProductsState = {
 showProductCode: true,
 loading: false,
 products: [],
 errorMessage: ‘’
};
```

Exception Handling in Reducer

```
export interface ProductsState {
 showProductCode: boolean;
 loading: boolean;
 products: Product[];
 errorMessage: string;
}

const initalState: ProductsState = {
 showProductCode: true,
 loading: false,
 products: [],
 errorMessage: ‘’
};

export const productsReducer = createReducer(
 initalState,
 on(ProductsAPIActions.productsLoadedFail, (state, { message }) => ({
  errorMessage: message,
})));
```

Demo

Add exception handling to effect

RxJS Mapping Operators

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>
  this.actions$.pipe(
   ofType(ProductsPageActions.loadProducts),
   concatMap(() => this.productsService.getAll().pipe(
     map((products) => ProductsAPIActions.productsLoadedSuccess({ products })),
     catchError(
       (error) => of(ProductsAPIActions.productsLoadedFail({ message: error})
     )
 )));

};
```

RxJS Mapping Operators

```
@Injectable()
export class ProductEffects {

 constructor(private actions$: Actions, private productsService: ProductsService) {}

 loadProducts$ = createEffect(() =>
  this.actions$.pipe(
   ofType(ProductsPageActions.loadProducts),
   concatMap(() => this.productsService.getAll().pipe(
     map((products) => ProductsAPIActions.productsLoadedSuccess({ products })),
     catchError(
       (error) => of(ProductsAPIActions.productsLoadedFail({ message: error})
     )
 )));

};
```

Warning – Possible Race Conditions

Operator	When	Possible Race Condition
concatMap	**Runs subscriptions/requests in order and is less performant** Use for get, post and put requests when order is important	**No**

Warning – Possible Race Conditions

Operator	When	Possible Race Condition
concatMap	**Runs subscriptions/requests in order and is less performant** Use for get, post and put requests when order is important	**No**
mergeMap	**Runs subscriptions/requests in parallel** Use for get, put, post and delete methods when order is not important	**Yes**

Warning – Possible Race Conditions

Operator	When	Possible Race Condition
concatMap	**Runs subscriptions/requests in order and is less performant** Use for get, post and put requests when order is important	**No**
mergeMap	**Runs subscriptions/requests in parallel** Use for get, put, post and delete methods when order is not important	**Yes**
switchMap	**Cancels the current subscription/request** Use for get requests or cancelable requests like searches	**Yes**

Warning – Possible Race Conditions

Operator	When	Possible Race Condition
concatMap	**Runs subscriptions/requests in order and is less performant** Use for get, post and put requests when order is important	**No**
mergeMap	**Runs subscriptions/requests in parallel** Use for get, put, post and delete methods when order is not important	**Yes**
switchMap	**Cancels the current subscription/request** Use for get requests or cancelable requests like searches	**Yes**
exhaustMap	**Ignores all subsequent subscriptions/requests until complete** Use when you do not want more requests until the initial one completes	**Yes**

Demo

Add more effects:

- Add
- Update
- Delete