**Why ngRx Effects ?**

In any service based application which interacts with a database server, component is the one which interacts with external resources directly through services. So, the component is responsible to manage the view dynamically as well as interacting with external sources.

***Effects*** help us handle this more efficiently. Effects provide us a way to interact with the services and isolate them from the components.

**What are ngRx Effects ?**

**Effects**are an**RxjS powered** side effect model for Store. Effects uses streams and provide us with new sources of action to handle the state based on external interactions such as long running tasks which produce multiple events, web-socket messages and network requests to fetch, add or modify data.

The use of **Effects**becomes much more important in larger applications as they may have multiple sources of data with multiple services required to fetch those data

**Features of Effects:-**

1. It helps to have more pure components that can select the state from the store and dispatch actions, by isolating the side effects.
2. Effects are long running services which listens to every action dispatched.
3. Effects use RxJS operators to filter the actions based on the type of action.
4. Effects can handle synchronous as well as asynchronous tasks and return a new action.

**ngRx Effects Installation:-**

To use ngRx Effects in our angular applications to perform the side effects, we need to install it using the below command:

1. npm install @ngrx/effects --save

We can use Effects to isolate the **side effects/external interactions** from the components by creating a ***Effect***class which listens to events and performs tasks.

**Effect**class is an ***injectable***service class and has the following distinct parts:

**1.**An injectable ***Actions***service which provides an observable of all the dispatched actions after the latest state changes.

1. */\* Example \*/*
2. import { Injectable } from '@angular/core';
3. import { Actions } from '@ngrx/effects';
4. @Injectable()
5. export class ProductEffects {
6. constructor(private actions$: Actions) { }
7. }

**2.** Metadata is attached to the observable of dispatched actions using ***createEffect*** function of **@ngrx/effects.**The **createEffect**function registers the streams that are subscribed to the store. Any action returned from the effect stream is then dispatched back to the Store.

1. */\* Example \*/*
2. import { Injectable } from '@angular/core';
3. import { Actions, createEffect } from '@ngrx/effects';
4. @Injectable()
5. export class ProductEffects {
6. constructor(private actions$: Actions) { }
7. loadProducts$ = createEffect(() => this.actions$.pipe());
8. }

**3.**Actions are filtered using the ngRx effects pipeable **ofType** operator. It takes one or more action types as argument to filter those on which it needs to act upon.

1. */\* Example \*/*
2. import { Injectable } from '@angular/core';
3. import { Actions, createEffect, ofType } from '@ngrx/effects';
4. @Injectable()
5. export class ProductEffects {
6. constructor(private actions$: Actions, private productService: ProductService) { }
7. loadProducts$ = createEffect(() => this.actions$.pipe(
8. ofType('[ProductList Component] GET\_ALL\_PRODUCTS')
9. ));
10. }

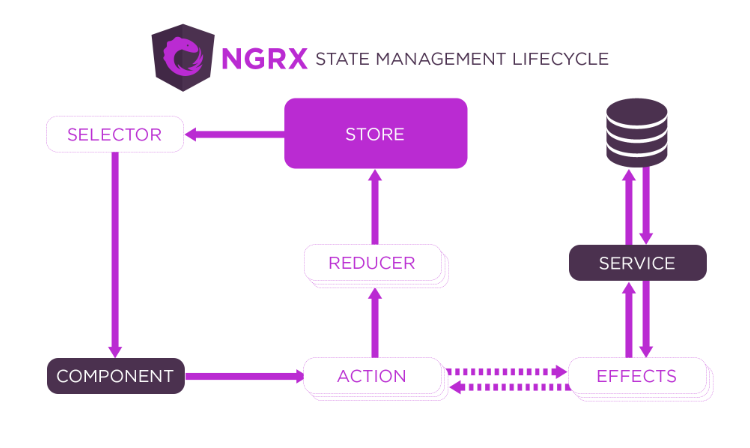
**4.**Effects are subscribed to the Store observable and Services are injected into the effects to interact with the external API's.

1. import { Injectable } from '@angular/core';
2. import { Actions, createEffect, ofType } from '@ngrx/effects';
3. import { of } from 'rxjs';
4. import { ProductService } from '../../product-list/product.service';
5. import { map, mergeMap, catchError } from 'rxjs/operators';
6. @Injectable()
7. export class ProductEffects {
8. constructor(private actions$: Actions, private productService: ProductService) { }
9. loadProducts$ = createEffect(() => this.actions$.pipe(
10. ofType('[ProductList Component] GET\_ALL\_PRODUCTS'),
11. mergeMap(() => this.productService.getAllProducts()
12. .pipe(
13. map(products => ({ type: '[ProductList Component] GET\_ALL\_PRODUCTS SUCCESS', allProducts: products })),
14. catchError(() => of({ type: '[ProductList Component] GET\_ALL\_PRODUCTS ERROR', errorMessage: 'No Products Found' }))
15. ),
16. ),
17. )
18. );
19. }

**The flow Explanations:**

* The **loadProducts$** effect is listening for all dispatched actions through the Actions stream, but is only interested in the **'[ProductList Component] GET\_ALL\_PRODUCTS'**event using the ofType operator.
* The stream of actions is then flattened and mapped into a new observable using the mergeMap operator.
* The **productService#getAllProducts()** method returns an observable that maps the **products**to a new action on success, and currently returns an empty observable if an error occurs.
* The action is dispatched to the Store where it can be handled by reducers when a state change is needed.
* It's also important to handle errors when dealing with observable streams so that the effects continue running.

**Angular App flow with ngRx Effects:-**



1. Component **C** Dispatches an action **A**.
2. The action**A**invokes an Effect **E.**
3. The Effect **E**then invokes a service **Srv**to interact with the database. After service call the **Effect E**again returns an action, either **A1 or A2**depending upon service call success or error**.**
4. **A1/A2**should not be the same as **A (otherwise, this will lead to infinite loop).**
5. These actions **A1/A2**then go to the reducer **R** and update the store **S**.
6. Once the store **S** has been updated, the component **C**can fetch the value using a **Selector SS.**

With this basic understanding of Effects, lets see a demo on how to use effects in angular apps.