Effects

What are NgRx Effects?

As the reducer and actions are considered a pure function, they lack the ability to call out to an external source for data. Effects are the solution to this problem. When an action is called that requires access to an outside data source it will trigger an effect. The effect will make the subsequent call to the server, retrieve or update data and return and attach any necessary data to an action. This action will then follow the standard protocol where it is picked up by the reducer and the data if any is processed into the immutable state.

Actions
Payloads
Effects

Redux Pattern

The redux pattern is very handy for updating and tracking changes to a data store that multiple resources rely on for up to date information simultaneously. How does this pattern keep all components dependent on a single source synchronized to one data source?

1. It uses selectors to get the latest information that is stored in the store/state.
2. This information is gathered in the component and displayed in the view.
3. When the user interacts with view and changes the state of the application this information and new state will be packaged into actions in the Component and sent to the reducers.
4. The reducers are listening for specific actions that carry payloads of new information about the changes to the state
5. The reducers copy the existing data and augment it with the payload from the actions carrying the new changes to state using a spread operator, creating the new state.
6. The new state is saved to the store
7. The selectors that have been listening for changes to the state, notify all the components subscribed simultaneously of the new state.

The process happens over and over again. The store cannot be changed directly and must follow this strict process.

Getting Started w/ Effects

To use NgRx Effects your application will require you to install the NgRx Effects package via npm or yarn. In this example I prefer to use yarn.

```
> npm install @ngrx/effects --save
```

To complete the initialization of the NgRx Effects module you need to import it into the appModule

```
@NgModule({
imports: [
        ...
                EffectsModule.forRoot([]),

})
```

Implementation

When you create an NgRx effects they are located in an effects file in their respective feature state folder.
The file follows the standard naming convention with the feature in the title... 'feature'.effects.ts. In this example we are using the product feature module and adding an effect that will get an array of products from the server. It is important to note that at the top of the file we import Actions, Effect, and ofType from @ngrx/effects. An effect relies on rxjs operators like mergeMap, map, and catchError to manage the return data from the observables. An Effect also operates similar to a service as it uses the @Injectable() decorator.
After attempting to fetch the data the effect will map and pipe the results and either dispatch a LoadSuccess() Action or a LoadFail() Action. The LoadSuccess() Action if successful will carry the resultant array of products as the payload. The LoadFail() Action will carry a string detailing the error.

```
import { Injectable } from '@angular/core';
import { Actions, Effect, ofType } from '@ngrx/effects';
import { ProductService } from '../product.service';
import * as productActions from '../state/product.actions';
import {
        mergeMap, map, catchError
        } from 'rxjs/operators';
import { Product } from '../product';
import { of } from 'rxjs';

@Injectable()
export  class ProductEffects  {
        constructor (
                private action$: Actions,
                private productService: ProductService)  { }

@Effect()
LoadProducts$ = this.acitons$.pipe(
        ofType(productActions.ProductActionTypes.Load),
        mergeMap((action: productActions.Load) =>
                this.productService.getProducts()
                .pipe(
                        map((products: Product [ ]) =>
                        (new productActions.LoadSuccess(products))),
                        catchError(err => of (new productActions.LoadFail(err)))
                )
        )
);
}
```

app > products > state > product.effects.ts

Regardless of whether the Effect is successful The Reducer is waiting patiently to process the resultant actions of the effect.