El patrón Redux con NgRx en Angular

Le pasa a todas las aplicaciones, crecen y crecen en funcionalidad y complejidad. En Angular estamos preparados para modularizar, componentizar e inyectar servicios. Pero con grandes aplicaciones, o con grandes equipos, parece que nada es suficiente. Se necesita una **gestión del estado centralizada** como la del patrón Redux.

Si decíamos que **Redux no hace rápido lo simple, sino mantenible lo complejo**, ahora te digo que **NgRx no hace rápido a Redux, sino mantenible su boilerplate.** Así que si tienes delante un desarrollo funcionalmente complejo, te recomiendo que uses *NgRX*; la solución estándar para implementar **Redux con Angular**.

Partiendo del código tal cómo quedó en Redux, flujo reactivo unidireccional con Angular y RxJs. Al finalizar tendrás una aplicación que gestiona centralizadamente los cambios, que permite conocer qué ocurrió y predecir lo que ocurrirá usando NgRx.

> Código asociado a este tutorial en *GitHub*: AcademiaBinaria/angular-boss

1 Instalación y configuración

NgRx es el estándar de facto para implementar *Redux* en Angular. Está basada en *RxJS* y es una librería modular con todo lo necesario para crear grandes aplicaciones. Esto son los módulos que la componen:

- **store**: Es el módulo principal con el administrador del estado centralizado y reactivo.
- **store-devtools**: Instrumentación para depurar desde el navegador. Vale su peso en oro.
- **router-Store** : Almacena el estado del *router* de Angular en el *store*, tratando cada evento como una acción Redux.
- **effects**: Los reductores son funciones puras sin efectos colaterales. Este módulo es la solución para comandos asíncronos.
- **schematics, entity, ngrx-data**: Son otros módulos opcionales con ayudas y plantillas de NgRX.

1.1 Instalación de NgRx

Para agregar *NgRx* a un app te propongo que uses los *schematics* de NxDev pues disponen de más opciones de configuración inmediata. Aunque la alternativa con el CLI también es suficiente para empezar.

```
# with Nx.dev tools
ng g @nrwl/angular:ngrx app --module=apps/shop/src/app/app.module.ts --root --
minimal
# with only CLI
ng add @ngrx/store
```

Con esto habrás instalado y configurado NgRx y tu `AppModule` tendrá algo así:

```
@NgModule({
  imports: [
    CommonModule,
    RouterModule,
    StoreModule.forRoot(
      {},
      {
        metaReducers: !environment.production ? [] : [],
        runtimeChecks: {
          strictActionImmutability: true,
          strictStateImmutability: true
        }
      }
    ),
    EffectsModule.forRoot([])
  ]
})
export class AppModule {}
```

> El código generado por cualquiera de los dos schematics no es para todos los gustos. Tómalo como un punto de partida y crea la estructura que mejor te encaje.

1.2 DevTools

La razón principal de usar Redux es la capacidad que da al programador para comprender cómo el estado de sus variables ha variado en el tiempo. Es decir ¿Cómo hemos llegado hasta aquí?. Esta es claramente una cuestión de análisis y depuración de código. Y para ello necesitamos herramientas de desarrollador. Presentamos las **Redux DevTools for Chrome**.

Este es un plugin que se engancha a cualquier solución Redux (no sólo NgRx, ni siquiera sólo Angular) y que permite explorar el valor del estado y lo más interesante: la secuencia de acciones que se ha despachado y el efecto que cada una tuvo sobre el valor del estado.

Para realizar el enganche sirve la línea que verás también en el `AppModule`. La cual activa la monitorización cuando no estamos en producción. Normal, esto es un herramienta para el desarrollador, nunca para el usuario.

```
    !environment.production ? StoreDevtoolsModule.instrument() : [],
```

Para verlo en funcionamiento tienes que ejecutar y abrir tu app en Chrome. Después en la pantalla de inspección del código aparecerá una pestaña al final de todo (más allá de Network y Application...) llamada Redux. La interfaz es muy intuitiva; no tardarás en hacerte con ella. aunque ahora mismo un poco pobre...

1.3 Router

Para darle algo de contenido y de paso acercarnos a las capacidades de Redux y NgRx vamos a conectarle el `RouterModule` de la aplicación. Para ello sirve la siguiente línea que tienes que poner también en la importaciones del `AppModule`.

```
  StoreRouterConnectingModule.forRoot({ routerState: RouterState.Minimal })
```

A partir de este momento cada evento de navegación del *router* va a generar una acción que *NgRx* recogerá y registrará. Esas acciones y su impacto en el estado se pueden visualizar con el plugin de Redux. Podrás ver los cambios y reproducirlos hacia adelante y atrás. Es como viajar en el tiempo dentro de la ejecución de app.

2 Actions

Hasta el momento ha sido todo muy de infraestructura, pero es hora de que hagamos algo funcional. Un ejemplo sería agregar a nuestra app un módulo para la gestión de medios de pago de un usuario. Algo así:

```
As a: customer,
  I want: to add payment methods
  so that: I can pay with them

As a: customer,
  I want: to select one as preferred
  so that: I can make fewer clicks

As a: customer,
  I want: to change de expiration date
  so that: I get my cards up to date
```

Para ello, y valiendo una vez más de los schematics vamos a agregar una *feature*

```
ng g m payments --project=shop --module=app.module.ts --routing --route=payments
ng g @ngrx/schematics:feature payments/store/paymentMethod --project=shop --
module=payments/payments.module.ts --no-flat --no-spec --creators
```

Esto habrá generado un buen montón de código. Pero todo demasiado genérico. Vamos a empezar a adaptarlo a nuestro problema aportando un modelo de datos.

apps\shop\src\app\payments\store\payment-method\payment-method.model.ts

```
export interface PaymentMethod {
  id: string;
  expiration: Date;
}

export interface PaymentMethods {
  list: PaymentMethod[];
  preferred: string;
}
```

Y ahora sí, vamos a por las acciones.

2.1 Create

Uno de los parámetros usados durante la generación de la *feature* fue `--creators`. Esto le indicó a NgRx que preferimos usar funciones en lugar de clases para definir nuestras acciones. Así que en el fichero de definición de acciones te encontrarás con llamadas a una función como esta `createAction('Descripción de la acción', props<{ parametro: Tipo }>() )`. Con esto tenemos que cubrir los dos requerimientos de Redux: las acciones han de tener un tipo y una *payload*.

`apps\shop\src\app\payments\store\payment-method\payment-method.actions.ts`

```
export const loadPaymentMethods = createAction(
  '[PaymentMethod] Load PaymentMethods'
);

export const addPaymentMethod = createAction(
  '[PaymentMethod] Add PaymentMethod',
  props<{ newPaymentMethod: PaymentMethod }>()
);

export const selectPreferredPaymentMethod = createAction(
  '[PaymentMethod] Select preferred PaymentMethod',
  props<{ preferredId: string }>()
);

export const setExpirationPaymentMethod = createAction(
  '[PaymentMethod] Set Expiration Date on PaymentMethod',
  props<{ updatedPaymentMethod: PaymentMethod }>()
);
```

El código puede resultar extraño a primera vista; pero es siempre igual y acabas familiarizando muy rápido. Esencialmente se trata de dar un nombre a la acción y luego configurarla. Lo imprescindible va a ser un string que la describa. Es recomendable seguir el convenio con el formato `[Origen] Nombre acción descriptivo`.

El segundo argumento de la función `creteAction` es a su vez otra función. La parte interesante de la función `propos<>()` está en su tipo genérico. Es ahí dónde definimos el tipo de datos de la *payload* que puede transportar la acción.

2.2 Dispatch

Definir el catálogo de acciones es la primera parte del problema. Ahora nos toca invocarlas, o como se dice en el argot Redux: despacharlas. Para ello voy a crear un servicio auxiliar que encapsule toda la interacción con NgRx. Es una implementación del patrón fachada para unificar y facilitar el trabajo con Redux desde fuera.

`apps\shop\src\app\payments\store\payment-method\payment-method.service.ts`

```
import { Injectable } from '@angular/core';
import { Store } from '@ngrx/store';
import * as PaymentMethodActions from './payment-method.actions';
import { PaymentMethod, PaymentMethods } from './payment-method.model';
@Injectable({
  providedIn: 'root'
})
export class PaymentMethodService {
  constructor(private store: Store<PaymentMethods>) {}
}
```

Y aquí puedo crear los métodos funcionales que despachen las acciones sobre el *store* de NgRx.

```
public loadPaymentMethods() {
  this.store.dispatch(PaymentMethodActions.loadPaymentMethods());
}
public addPaymentMethod(newPaymentMethod: PaymentMethod) {
    this.store.dispatch(
      PaymentMethodActions.addPaymentMethod({
        newPaymentMethod: { ...newPaymentMethod }
      })
    );
  }
public selectPreferredPaymentMethod(preferredId: string) {
  this.store.dispatch(
    PaymentMethodActions.selectPreferredPaymentMethod({ preferredId })
  );
}
public setExpirationPaymentMethod(updatedPaymentMethod: PaymentMethod) {
  this.store.dispatch(
    PaymentMethodActions.setExpirationPaymentMethod({
      updatedPaymentMethod: { ...updatedPaymentMethod }
    })
  );
}
```

Es necesario enviar siempre un clon de los argumentos para desconectarlos del origen. En Redux queremos tener la fuente única de la verdad, y eso sólo es posible si los argumentos que recibimos están sellados y aislados del resto del mundo.

Para una primera aproximación, reconozco que es mucho código para tan poca funcionalidad. Claro que para quien use esta fachada el mundo es mucho más sencillo. Y por dentro es muy potente y controlable.

`apps\shop\src\app\payments\payments.component.ts`

```
export class PaymentsComponent implements OnInit {
  constructor(private paymentMethodService: PaymentMethodService) {}
```

```
  ngOnInit() {
    this.paymentMethodService.loadPaymentMethods();
    const visa: PaymentMethod = {
      id: '1234 7896 3214 6549',
      expiration: new Date(2020, 6-1, 30)
    };
    this.paymentMethodService.addPaymentMethod(visa);
    this.paymentMethodService.selectPreferredPaymentMethod(visa.id);
    visa.expiration = new Date(2021, 12-1, 31);
    this.paymentMethodService.setExpirationPaymentMethod(visa);
  }
}
```

3 State reducer

3.1 State

El estado en *redux* es un objeto tipado a partir de una interfaz, inicialmente llamada `State` a secas, aunque yo prefiero identificarla como `PaymentMethodsState`. Tendrá propiedades para almacenar objetos más o menos complejos. Cada propiedad tendrá su propio tipo complejo y necesita un estado inicial. Eso es lo que hace este código:

`apps\shop\src\app\payments\store\payment-method\payment-method.reducer.ts`

```
export const paymentMethodFeatureKey = 'paymentMethod';

export interface PaymentMethodsState {
  paymentMethods: PaymentMethods;
}

export const initialState: PaymentMethodsState = {
  paymentMethods: { list: [], preferred: null }
};
```

3.2 Create function

Pero todo estado necesita su reductor, y una vez más lo crearemos mediante una función. En este caso es la `createReducer(state, on(action1, function1) )`. Su encarnación más básica tiene la siguiente pinta:

```
const paymentMethodReducer = createReducer(
  initialState,
  on(PaymentMethodActions.loadPaymentMethods, state => state)
);
```

Lo que hace es definir una serie de `hooks` para enganchar funciones de mutación en respuesta a las acciones despachadas. Empezamos con la acción `loadPaymentMethods` y en este caso no por ahora hacemos ninguna

mutación al estado. Es un reductor transparente e innecesario. Lo pongo para familiarizarnos con la sintaxis.

Vamos a algo más complejo y útil como la acción `addPaymentMethod`. Funcionalmente queremos agregar el nuevo método de pago que viene en su carga al array actual de métodos de pago. Nada complejo, ya sí deben ser todas las funciones reductoras.

Pero, siempre hay un pero, resulta que **una función reductora debe ser también una función pura**. Y esto nos obliga a mutar el estado sin mutar los argumentos; es decir, nos obliga a clonar el estado y sus propiedades antes de cambiarlas.

```
on(PaymentMethodActions.addPaymentMethod, (state, { newPaymentMethod }) => {
  return {
    ...state,
    paymentMethods: {
      ...state.paymentMethods,
      list: [...state.paymentMethods.list, newPaymentMethod]
    }
  };
})
```

El proceso de clonado puedes hacerlo con mayor o menor profundidad, pero al menos un clonado superficial del estado es obligatorio. Yo aquí he optado por clonar incluso el array interno, aunque no sea estrictamente obligatorio.

3.3 Register in Store

El resultado de esta invocación a `createReducer()` se será a su vez otra función que yo almacené en la contante `paymentMethodReducer`. Por requerimiento del proceso de empaquetado con `webpack` que nada tiene que ver con NgRx, estamos obligados a exportar la función mediante un *wrapper* como este:

`apps\shop\src\app\payments\store\payment-method\payment-method.reducer.ts`

```
export function reducer(state: PaymentMethodsState | undefined, action: Action) {
  return paymentMethodReducer(state, action);
}
```

Hecho esto tenemos ya una función reductora que debemos registrar en el *store*. Lo que hacemos es decirle a NgRx que cuando llegue una acción de mutado invoque a esta acción pasándole la propiedad del estado encargada gestionar.

`apps\shop\src\app\payments\payments.module.ts`

```
import * as fromPaymentMethod from './store/payment-method/payment-
method.reducer';
@NgModule({
  imports: [
    StoreModule.forFeature(
```

```
      fromPaymentMethod.paymentMethodFeatureKey,
      fromPaymentMethod.reducer
    )]
```

Normalmente el nombre de dicha propiedad se estable en una constante exportada para evitar fallos de tecleo con consecuencias imprevisibles.

4 Selectors

Hasta ahora hemos definido el estado, y creado las acciones que despachamos conta él. Y hemos creado las funciones reductoras que lo modifican al recibir las acciones. No es poco, pero es la mitad del camino. Nos queda enterarnos de los cambios que se ha realizado. Y para eso nos ofrecen un concepto muy de base de datos `select`.

4.1 Create selector

Los selectores, mejor dicho las funciones selectoras, se crean mediante una función factoría igual que hemos visto para las reductoras y las acciones. La diferencia es que aquí las tenemos que llamar a dos niveles: el de la funcionalidad y el de la propiedad específica que nos interesa.

`apps\shop\src\app\payments\store\payment-method\payment-method.selectors.ts`

```
import { createFeatureSelector, createSelector } from '@ngrx/store';
import { paymentMethodFeatureKey, State } from './payment-method.reducer';

export const getPaymentMethodState = createFeatureSelector<State>(
  paymentMethodFeatureKey
);

export const getPaymentMethodsList = createSelector(
  getPaymentMethodState,
  (state: State) => state.paymentMethods.list
);

export const getPreferredPaymentMethod = createSelector(
  getPaymentMethodState,
  (state: State) => state.paymentMethods.preferred
);
```

Podemos tener tantos selectores como queramos. Se aconseja crearlos para cada tema de interés que pueda tener el resto de la aplicación. Pueden ser simples vista parciales del estado o complejas transformaciones a gusto del consumidor. NgRx lo hace de forma que optimiza cálculos y llamadas así que nos sugieren que abusemos de ellos y encapsulemos aquí todo el acceso al valor y a los cambios del estado.

4.2 Selecting data

Una vez más, si queremos que nuestra aplicación se desacople lo más posible de ngRx, debemos llamar a los selectores desde un servicio fachada. Así que añadimos al `apps\shop\src\app\payments\store\payment-method\payment-method.service.ts` un par de nuevos métodos públicos.

```
  public getPaymentMethodsList$(): Observable<PaymentMethod[]> {
    return this.store.select(PaymentMethodSelectors.getPaymentMethodsList);
  }

  public getPreferredPaymentMethod$(): Observable<string> {
    return this.store.select(PaymentMethodSelectors.getPreferredPaymentMethod);
  }
```

4.3 Showing data

Estos métodos devuelven observables a los que suscribirse para mostrar los valores cambiantes del estado. Gracias a la fachada anterior el componente consumidor es completamente inconsciente de la existencia de NgRx.

```
export class PaymentsComponent implements OnInit {
  public paymentMethodsList$: Observable<PaymentMethod[]>;
  public preferredPaymentMethod$: Observable<string>;
  constructor(private paymentMethodService: PaymentMethodService) {}

  ngOnInit() {
    this.paymentMethodsList$ = this.paymentMethodService.getPaymentMethodsList$();
    this.preferredPaymentMethod$ =
this.paymentMethodService.getPreferredPaymentMethod$();
  }
}
```

Por supuesto, desde la *template* lo tratamos como a cualquier otro dato asíncrono.

```
<p>Payment Methods List:</p>
<pre>{{ paymentMethodsList$ | async | json }}</pre>
<p>Preferred Payment Method:</p>
<pre>{{ preferredPaymentMethod$ | async | json }}</pre>
```

5 Effects

Las funciones reductoras, como ya se ha dicho, deben ser puras. La idea es que puedan ser auditadas, re-ejecutadas y testeadas sin que necesiten servicios externos ni causen efectos colaterales. Y esto es un problema con la cantidad de **ejecuciones asíncronas en las aplicaciones web**. Cualquier tentación de lanzar una llamada *AJAX* dentro de un reductor debe ser eliminada de inmediato.

> Dos razones: por un lado en Angular se necesita invocar al *httpClient* de alguna manera para realizar la llamada *AJAX*. Y ya que la función reductora no pertenece a ninguna clase Angular, no puede haber constructor que reclame la inyección de la dependencia a dicho servicio. Tampoco las funciones puras tienen permitido usar nada que no venga entre sus argumentos. Por otra parte las funciones puras han de ser predecibles, y una llamada a un servidor remoto no es en absoluto predecible. Puede pasarle de todo, así que **los reductores no son país para procesos asíncronos**.

La solución que proponen *NgRX* es usar un artificio llamado efecto, porque será encargado de **los efectos secundarios que provocan las las instrucciones asíncronas**. De una forma simplista, diremos que las acciones asíncronas se multiplicarán por tres. El comando que genera la llamada, y los dos potenciales eventos con la respuesta correcta o el error.

Para manejarlo todo incluyen en la librería el módulo `EffectsModule` que ha de registrarse junto al `StoreModule`. Desde ese momento *NgRX* activa un sistema de seguimiento que trata las acciones como un stream de *RxJS* y permite subscribirse a la invocación de dichas acciones y tratarlas adecuadamente.

5.1 Install

Según el proceso que hayas escogido para agregar Ngrx puede que ya tengas los efectos agregados o que tengas que hacerlo manualmente mediante un instrucción como esta:

```
ng add @ngrx/effects
```

En cualquier caso aparecerá una instrucción de registro en el módulo raíz o funcional que lo necesite. Por ejemplo el de `PaymentsModule` es:

```
EffectsModule.forFeature([])
```

5.2 Efecto básico

En un ejemplo inicial vamos a intentar almacenar y recuperar los métodos de pago desde el *local storage* del navegador. Esta es claramente una actividad de riesgo (y no lo dogo sólo por el evidente peligro de seguridad). Pero el acceso puede fallar o no estar disponible. Y en el mejor de los casos es un recurso externo que nunca podría ir dentro de una función reductora.

Así que vamos a ver el trabajo necesario para montar esto *como Redux manda*.

Acciones

Para empezar agregamos dos nuevas acciones en respuesta a los potenciales eventos que puedan surgir del acceso al *local storage*

`apps\shop\src\app\payments\store\payment-method\payment-method.actions.ts`

```
export const loadPaymentMethods = createAction(
  '[PaymentMethod] Load Payment Methods'
```

```
);

export const loadPaymentMethodsSucess = createAction(
  '[PaymentMethod] Load Payment Methods Success',
  props<{ paymentMethodList: PaymentMethod[] }>()
);

export const loadPaymentMethodsError = createAction(
  '[PaymentMethod] Load Payment Methods Error'
);
```

Para simplificarlo no estoy reportando nada del error específico. La `loadPaymentMethodsError` servirá para indicar que el proceso ha fallado. sin más.

Definición

Ahora viene lo bueno. La creación de un efecto. Empezamos como siempre usando una función factoría : `createEffect()`. Sólo que esta vez va asociada a una propiedad pública de un servicio de Angular. Y aquí ya tenemos todos nuestros juguetes disponibles.

Por ejemplo la inyección de dependencias; en el constructor `constructor(private actions$: Actions) {}` solicitamos como argumento un observable de todas las acciones que se hayan despachado en nuestro `store`. Ese *stream* servirá para escuchar y filtrar las acciones que nos interesen. Después haremos nuestra operativa con todas las llamadas que hagan falta.

Sólo un requisito. Al terminar hemos de retornar un observable con una nueva acción que el sistema encauzará para ser despachada. Esto obliga a conocer bien los operadores *RxJs* que están en la base do todo este montaje. De ahí el uso `concatMap()` y por supuesto del creador observable más básico: `of()`.

`apps\shop\src\app\payments\store\payment-method\payment-method.effects.ts`

```
public loadPaymentMethods$ = createEffect(() =>
  this.actions$.pipe(
    ofType(PaymentMethodActions.loadPaymentMethods),
    concatMap(() => {
      try {
        let storedList = JSON.parse(
          window.localStorage.getItem(this.storeKey)
        );
        if (!storedList) {
          storedList = initialState.paymentMethods.list;
          window.localStorage.setItem(
            this.storeKey,
            JSON.stringify(storedList)
          );
        }
        return of(
          PaymentMethodActions.loadPaymentMethodsSucess({
            paymentMethodList: storedList
          })
        );
```

```
        } catch (e) {
          return of(PaymentMethodActions.loadPaymentMethodsError);
        }
      })
    )
  );
```

Tómate tu tiempo para entender este efecto, y recuerda que por raro que te parezca al principio, te acabarás acostumbrando muy pronto porque siempre sigue el mismo esquema.

Reducer y Register

Por si te olvidabas, todo este trabajo no altera para nada al estado. Recuerda que sólo las reductoras asociadas a las acciones lo hacen. Así que incrementamos nuestro código con nuevos casos para el reductor.

apps\shop\src\app\payments\store\payment-method\payment-method.reducer.ts

```
on(
    PaymentMethodActions.loadPaymentMethodsSucess,
    (state, { paymentMethodList }) => {
      return {
        ...state,
        paymentMethods: { ...state.paymentMethods, list: paymentMethodList }
      };
    }
  ),
  on(PaymentMethodActions.loadPaymentMethodsError, state => state),
```

Y para finalizar registramos la clase que contiene al efecto (como un provider especial) en el módulo de efectos.

apps\shop\src\app\payments\payments.module.ts

```
EffectsModule.forFeature([PaymentMethodEffects])
```

5.3 Api async

Una vez familiarizado con la sintaxis de los efectos básicos, es hora de pasar a los que sin duda se utilizan más: los efectos sobre servicios asíncronos. Es decir casi todas las llamadas a un API. Algo muy sencillo podría ser agregar a nuestra tienda un converso de divisas para saber los precios en distintas monedas.

```
As a: customer,
  I want: to see the current exchange rate in several currencies
  so that: I can decide
```

Actions

A estas alturas te pido un último esfuerzo para que interpretes el siguiente código:

`apps\shop\src\app\rates\store\exchange-rate\exchange-rate.actions.ts`

```
export const loadExchangeRates = createAction(
  '[ExchangeRate] Load ExchangeRates'
);

export const loadExchangeRatesSuccess = createAction(
  '[ExchangeRate] Load ExchangeRates Success',
  props<{ rates: any }>()
);

export const loadExchangeRatesError = createAction(
  '[ExchangeRate] Load ExchangeRates Error',
  props<{ rates: any }>()
);
```

Efectivamente, para cada llamada usaremos tres acciones. El comando que la inicia y los dos eventos que pueden ocurrir *success and error*.

Effect

`apps\shop\src\app\rates\store\exchange-rate\exchange-rate.effects.ts`

```
export class ExchangeRateEffects {
  public loadExchangeRates$ = createEffect(() =>
    this.actions$.pipe(
      ofType(ExchangeRateActions.loadExchangeRates),
      concatMap(() =>
        this.http.get<any>('https://api.exchangeratesapi.io/latest').pipe(
          map(res =>
            ExchangeRateActions.loadExchangeRatesSuccess({ rates: res.rates })
          ),
          catchError(err =>
            of(ExchangeRateActions.loadExchangeRatesError({ rates: err }))
          )
        )))
  );
  constructor(private actions$: Actions, private http: HttpClient) {}
}
```

Paradójicamente un efecto que incluya una llamada *http* es más sencillo, pues la naturaleza asíncrona de *Ajax* junto con el uso de *Observables* en `httpClient` nos dejan medio trabajo hecho.

Lo que realmente hacemos es encauzar el flujo de acciones `pipe(concatMap())` hacia un flujo de respuestas *http* `this.http.get()`. Pero claro esto es trampa, así que de nuevo encauzamos la respuesta hacia una

acción `pipe(map(res=> actionSuccess(res)))`. Cuando la respuesta no llega volvemos a lo que ya sabemos, crear un nuevo observable con `of(actionError(err))`.

Aquí lo dejamos. Soy consciente de que Redux es complejo. Soy consciente de que NgRx añade mucho código de infraestructura. Pero también te animo a revisar este artículo y practicar con un caso sencillo antes de tomar cualquier decisión. Incluir NgRx o no hacerlo tiene un alto impacto en el mantenimiento de tu proyecto. No te tires a de cabeza ni lo rechaces de plano.

Descarga y revisa la aplicación de ejemplo. Si la dominas, entonces es que ya tienes los conocimientos para gestionar de manera centralizada, auditable y predecible el estado de tus programas. El patrón Redux lucirá más cuanto más grande y compleja sea tu aplicación.

Continúa tu formación avanzada para crear aplicaciones PWA, entre la web y las apps con Angular y verás como aprendes a programar con Angular.

> Aprender, programar, disfrutar, repetir. -- *Saludos, Alberto Basalo*s