TEMA 2

REDUX (Parte 1)

Desarrollo front-end avanzado

Máster Universitario en Desarrollo de sitios y aplicaciones web

Universitat Oberta de Catalunya

Contenido

Parte 1

- Introducción Redux
- Counter App sin Redux
- Counter App con Redux

Parte 2

- TODO App con Redux

Introducción

En este documento vamos a estudiar el patrón **Redux** pero desde un punto de vista más práctico para facilitar así su comprensión.

Vamos a organizar este documento, que entregaremos en dos partes, de la siguiente manera:

- **Parte 1**:
    - Primero repasaremos los conceptos más relevantes del patrón **Redux** con un ejemplo teórico para ilustrar los conceptos más relevantes como son **action**, **state**, **reducer** y **store**.
    - Posteriormente, implementaremos una aplicación sencilla en Angular que establece la comunicación entre tres componentes, y con esta simple aplicación veremos que para establecer dicha comunicación entre componentes el código necesario empieza a tener cierta complejidad.
    - A partir de la aplicación anterior, en el siguiente bloque lo que haremos es estudiar cómo le podemos aplicar el patrón **Redux** y veremos así las ventajas de su aplicación, cómo se reduce el código y cómo queda la gestión de la información centralizada en el **store**.
- **Parte 2**:
    - Finalmente, implementaremos una nueva aplicación de tipo TODO list directamente con el patrón **Redux** e introduciremos el concepto de los **effects**.

Redux

Sabemos que **Redux** es un patrón para gestionar la información de una aplicación. Esta información gracias a este patrón se encuentra centralizada en un solo lugar, el **STORE**.

Aplicando este patrón conseguiremos dar respuestas a preguntas como:

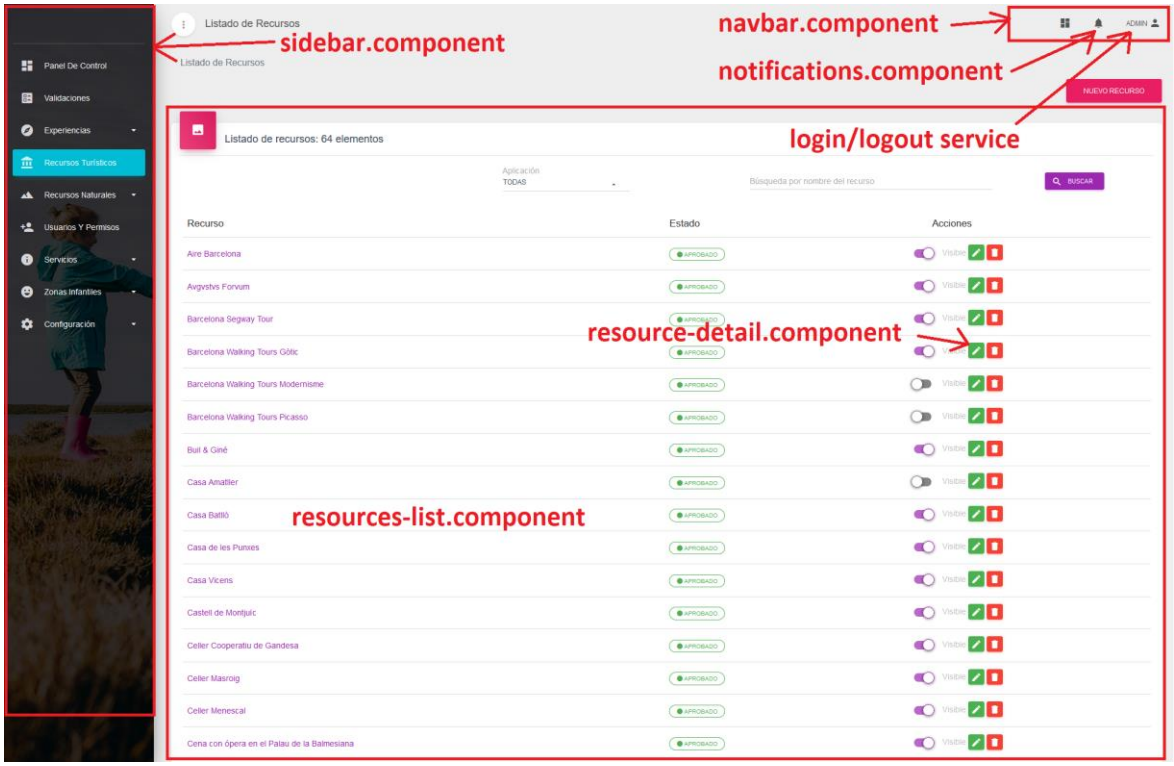
- Cuál es el estado de mi aplicación.
- Cuál es el valor de una determinada variable.
- Quién cambió cierta variable.
- Cómo cambió determinada información.
- …

Vale, estas preguntas anteriores que resuelve al patrón **Redux**, ¿por qué nos las planteamos? ¿Cómo trabajamos normalmente cuando desarrollamos una aplicación en Angular?

Suponemos que debemos desarrollar una aplicación para administrar actividades turísticas (lo que llamaríamos un *backoffice*).

Con este *backoffice* se guardaría toda la información en la base de datos que después consumiría un *front-end* determinado.

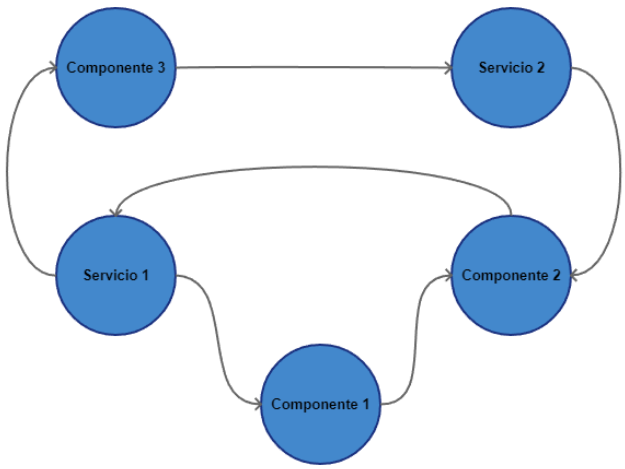
La aplicación podría tener un aspecto como el siguiente:



Sea esta aplicación o sea otra diferente, aquí lo importante es darnos cuenta de la cantidad de componentes y servicios que podríamos llegar a tener. Podemos ver que tendríamos diferentes componentes que se mostrarían en la vista al mismo tiempo y componentes y servicios que interactúan entre ellos.

Cada uno de estos componentes o servicios en un momento dado emitirán información que podrán recibir otros componentes o servicios. De esta manera, tendríamos un flujo de información que podría llegar a ser muy complejo.

Observemos el siguiente diagrama:



Imaginemos que **partimos** del servicio 1, podría ser por ejemplo el servicio que carga toda la información de los recursos turísticos. Este servicio **le pasa** la información al componente 1 (**resource-list.component**) para mostrar el listado de recursos. Este componente **reacciona** a una interacción de un usuario (por ejemplo se pulsa el botón de editar recurso) y **le pasa** información al componente 2 (**resource-detail.component**) donde podríamos editar las diferentes propiedades del recurso que estaríamos editando en su vista detalle. Este componente a su vez puede **pedir** información al servicio 1, por ejemplo, y éste **emitir** información hacía el componente 3 que podría ser otro cualquiera. A su vez, éste último componente podría pedir información a otro servicio y así podríamos ir describiendo todo un flujo de información o intercambio de datos que podría llegar a ser muy complejo.

¿Dónde está aquí el problema si una aplicación es muy grande? En que cualquier elemento puede modificar la información y en ciertos escenarios podemos decir que perdemos el control del estado de dicha información.

Con **Redux** solucionamos esta problemática ya que:

- Toda la información de la aplicación estará en una estructura previamente definida (**STATE**)
- Toda esta información estará guardada en un único lugar llamado **STORE**.
- El **STORE** nunca se manipulará de forma directa, lo haremos a través de las acciones (**ACTIONS**).
- Cualquier interacción del usuario o de código dispara acciones que describen qué sucedió.
- El valor actual de la información de la aplicación es el estado (**STATE**).
- Un nuevo estado es creado mediante la combinación del estado anterior y una acción, y esto se consigue con las funciones llamadas **REDUCER**.

Vale, con todo esto, aclaremos conceptos, ¿qué es un estado de la aplicación?

A nivel general podríamos destacar dos conceptos del estado (**STATE**):

- Es de sólo lectura.
- Nunca se mutará el **STATE** de forma directa.

Pongamos un ejemplo:

```
let appState = {
  TODOS: [
    {
      name: 'Task 1',
      done: true
    },
    {
      name: 'Task 2',
      done: true
    },
    {
      name: 'Task 3',
      done: false
    }
  ],
  USER: {
    name: 'Emma',
    age: 34
  }
};
```

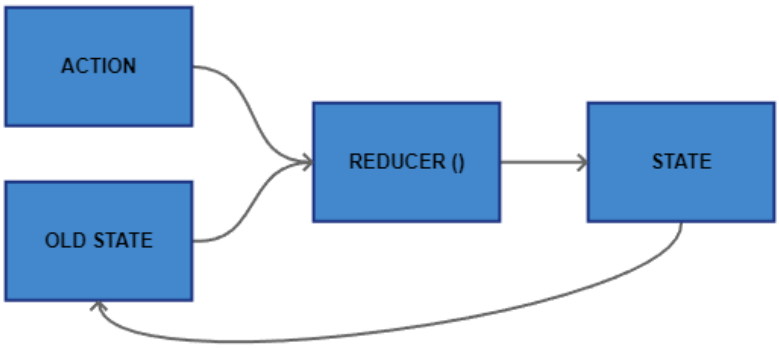
Suponemos que tenemos entre manos una aplicación para gestionar el estado de las tareas, y éstas pueden estar realizadas o no, la típica aplicación TODO. Además, tenemos un usuario en la aplicación. (En aplicaciones 'reales' al ser más complejas el estado también será más complejo, poco a poco lo iremos viendo).

Vale, pues la captura que tenemos a la izquierda sería un ejemplo de estado de esta aplicación, que no es más que un JSON con la 'foto' de los datos de la aplicación en un momento determinado, normalmente consideraremos el estado actual y el estado anterior.

Normalmente a nivel de código declararemos el **STATE** como una interfaz, lo veremos más adelante.

Vamos a introducir todos los conceptos implicados, y cómo se relacionan entre ellos.

Fijémonos en el siguiente diagrama:



Ahora suponemos que queremos modificar este estado de la aplicación que acabamos de comentar, suponemos que queremos establecer la tarea 3 como hecha (nótese que ahora la tarea 3 está con la propiedad *done* a *false*), por lo tanto, queremos asignar *true* al campo *done* de la *'Task 3'*.

Como hemos dicho, no podemos modificar directamente el estado, tenemos que hacerlo mediante acciones. Por lo tanto, dispararemos una acción, esta acción describe lo que sucederá, en este caso 'actualizar el estado de la tarea 3'. La acción no dispara el evento, describe lo que se va a pedir que haga.

Con esta acción junto con el estado anterior, tendremos los dos argumentos que pasaremos al *reducer*. Es decir, tenemos que ver al *reducer* como una función con dos argumentos, ACTION y OLD_STATE. De esta manera, al ejecutarse el **REDUCER** se generará el nuevo **STATE**.

Profundicemos con los conceptos ACTION, REDUCER, STATE y STORE

Pensemos que estamos en el contexto del ejemplo anterior:

- **Definir como hecha la tarea 3, es decir, asignar *true* al campo *done* de la *'Task 3'***

ACTION

Una acción es la herramienta que tenemos para modificar el **STORE**. Es la única fuente de información que se envía debido a interacciones de usuario o de código.

Las acciones tienen dos propiedades:

- **Type**: describe que es lo que quiere hacer la acción.
    - En nuestro ejemplo anterior, el *type* podría ser 'COMPLETE_TASK'
- **Pyload**: es opcional, lo utilizaremos si necesitamos mandar información para realizar una cierta acción.
    - En nuestro ejemplo anterior podríamos mandar un índice para saber qué tarea del array de tareas queremos completar (asignar campo a *true*)

REDUCER

Función que recibe dos argumentos, una acción y el estado anterior, y siempre devuelve un estado (normalmente un 'nuevo' estado, pero podría ser que el *reducer* no haga ninguna acción y se devolviera el mismo estado sin alteración).

Vamos a ponerlo en el contexto del ejemplo que estamos comentando de la aplicación TODO.

En nuestro ejemplo anterior, el OLD_STATE sería el siguiente:

```
let appState = {
  TODOS: [
    {
      name: 'Task 1',
      done: true
    },
    {
      name: 'Task 2',
      done: true
    },
    {
      name: 'Task 3',
      done: false
    }
  ],
  USER: {
    name: 'Emma',
    age: 34
  }
};
```

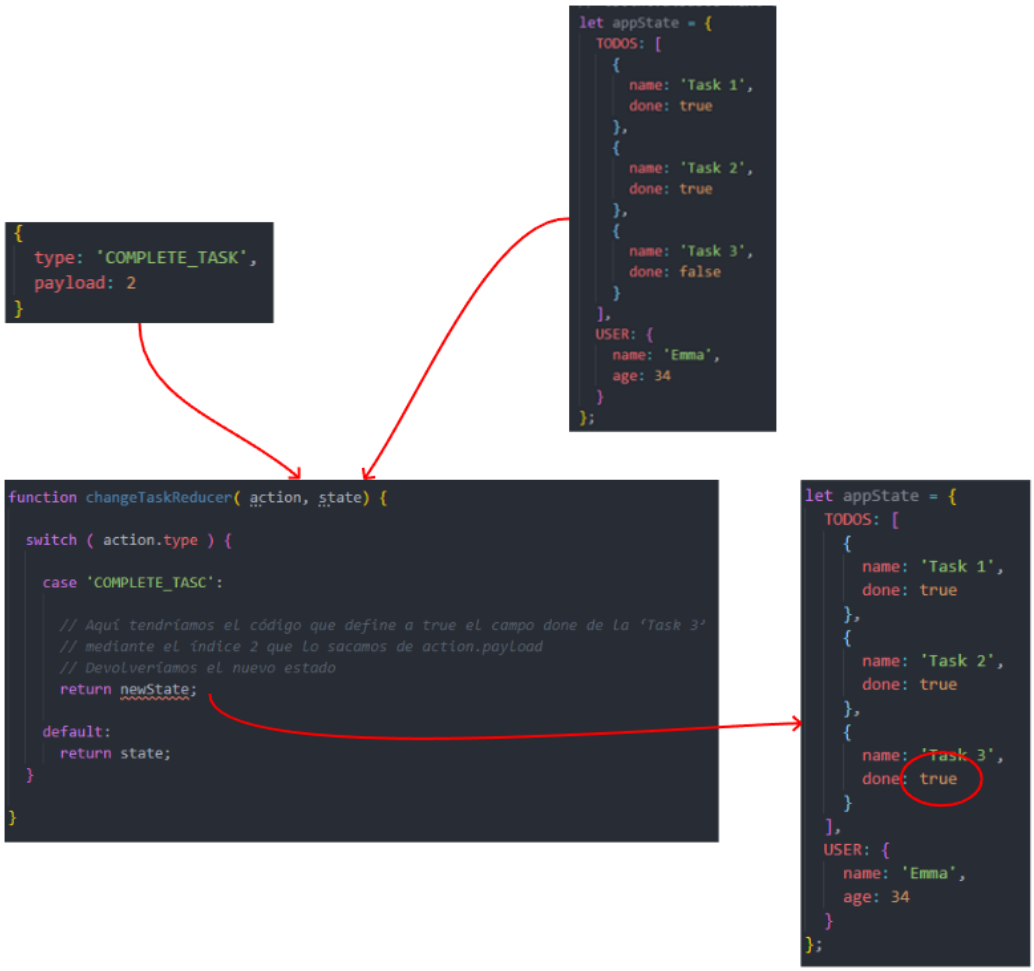
Este OLD_STATE no es más que el estado actual de la aplicación. Por lo tanto, este sería uno de los argumentos que le pasaríamos al *reducer*.

La *action* que le pasaríamos al *reducer* sería:

```
{
  type: 'COMPLETE_TASK',
  payload: 2
}
```

Es decir, describimos que va a realizar dicha acción, que, por el momento, lo podemos ver como una simple cadena de texto. En este caso de ejemplo necesitamos indicar que índice de la tabla de tareas queremos actualizar, por lo tanto, le podríamos pasar este índice en la propiedad opcional **payload**.

Ya tenemos los dos argumentos que le tenemos que pasar al *reducer*, la acción y el estado actual. Vale, pero ¿qué es eso del *reducer*?



Dentro del *reducer* imaginemos un switch/case en el que para la opción 'COMPLETE_TASK' coge el *payload* = 2 y coge el OLD_STATE y va la posición 2 del array ('Task 3'), define el campo *done* a *true* y devuelve el nuevo estado.

El nuevo estado generado por la función *reducer* sería el OLD_STATE para el siguiente cambio. Este nuevo estado quedaría guardado en el **STORE**.

Y aquí empezamos a hablar del **STORE**.

¿Verdad que si implementamos una clase cualquiera, por ejemplo, la clase **User** tendríamos por ejemplo las propiedades **name**, **email**,… con sus funciones de lectura (**getters**) y sus funciones de modificación (**setters**)?

El **STORE** sería muy parecido, lo podemos ver simplemente como una clase con su propiedad **STATE**, que hemos podido ver anteriormente que simplemente es un **JSON** o interfaz con la 'foto' de los datos actuales de la aplicación, con su **getState** y su **dispatch**, para consultar o modificar respectivamente el estado de la aplicación.

Vamos a repasar los conceptos fundamentales del **STORE**:

Tiene las siguientes responsabilidades:

- Contiene el estado actual de la aplicación.
- Permite la lectura del estado mediante la función **getState().**
- Permite crear un nuevo estado mediante la función **dispatch(ACTION).**
- Permite notificar los cambios de estado mediante **subscribe().**

Ahora tenemos los conceptos generales del patrón **Redux**, vamos a implementar una aplicación en Angular, y después le aplicaremos este patrón para contrastar estos conceptos **action**, **state**, **reducer** y **store**.

Counter App sin Redux

Una vez estudiados los conceptos más importantes del patrón **Redux**, vamos a implementar una aplicación en Angular donde se produzca la comunicación entre tres componentes. Tendremos un componente **madre**, un componente **hija** y un componente **nieta**. Cada uno de estos componentes lo que hará es modificar el valor de un contador de diferentes maneras. La **madre** podrá incrementar o decrementar el contador, la **hija** podrá duplicar el contador y la **nieta** podrá ponerlo a 0. La gracia estará en que sea quien sea quien modifique el contador, los tres componentes tienen que reflejar el cambio, es decir, tiene que mostrar el valor correcto del contador.

Veremos como poco a poco va aumentando la complejidad del código para una aplicación a priori sencilla y en un siguiente apartado aplicaremos el patrón **Redux** para poder ver las ventajas de éste.

Pero de momento, vamos a crear nuestra aplicación en **Angular**.

Crearemos un nuevo proyecto en angular, lo podemos llamar **basic-redux-app**:

```
H:\Projectes>ng new basic-redux-app
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? CSS
```

Una vez terminado el proceso de instalación y tengamos el proyecto creado lo podemos abrir con el **Visual Code**.

También podemos ejecutar la instrucción:

```
H:\Projectes\basic-redux-app>ng serve --open
```

Para validar que no tenemos ningún error y que podemos ejecutar la aplicación.

Para tener un aspecto visual un poco más agradable, nos podemos instalar **Bootstrap** por ejemplo, ejecutando la siguiente instrucción (este paso sería opcional):

```
H:\Projectes\basic-redux-app>npm install bootstrap
```

Para que nos funcione, una vez instalado, tendremos que indicarle a **Angular** que utilice esta librería, por lo tanto, iremos al fichero *angular.json* y añadiremos:

```
],
"styles": [
  "src/styles.css",
  "node_modules/bootstrap/dist/css/bootstrap.min.css"
],
```

Ahora lo que haremos es crearnos los componentes necesarios para el desarrollo de nuestra aplicación. Recordemos que queremos establecer la comunicación entre tres componentes, estos tres componentes lo que harán es ir modificando el valor de un contador. Nos crearemos un componente que llamaremos **hija** y otro componente que llamaremos **nieta**. La **madre** será directamente el componente **app.component**.

Por lo tanto, ahora haríamos:

```
H:\Projectes\basic-redux-app>ng g c counter/daughter
CREATE src/app/counter/daughter/daughter.component.html (23 bytes)
CREATE src/app/counter/daughter/daughter.component.spec.ts (640 bytes)
CREATE src/app/counter/daughter/daughter.component.ts (283 bytes)
CREATE src/app/counter/daughter/daughter.component.css (0 bytes)
UPDATE src/app/app.module.ts (491 bytes)

H:\Projectes\basic-redux-app>ng g c counter/granddaughter
CREATE src/app/counter/granddaughter/granddaughter.component.html (28 bytes)
CREATE src/app/counter/granddaughter/granddaughter.component.spec.ts (675 bytes)
CREATE src/app/counter/granddaughter/granddaughter.component.ts (303 bytes)
CREATE src/app/counter/granddaughter/granddaughter.component.css (0 bytes)
UPDATE src/app/app.module.ts (609 bytes)
```

Nótese que creamos estos dos componentes dentro de una carpeta **counter**, simplemente para tenerlos un poco agrupados.

Dejaríamos el proyecto ejecutándose y podemos eliminar todo el contenido del fichero **app.component.html**.

Este componente **app.component** actuará como **madre**, e inicialmente haremos lo siguiente, en su controlador:

app.component.ts

src > app > app.component.ts > ...

```typescript
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  title = 'Basic Redux App';

  counter: number;

  constructor() {
    this.counter = 20;
  }

  increase(): void {
    this.counter = this.counter + 1;
  }

  decrease(): void {
    this.counter = this.counter - 1;
  }
}
```

Podemos ver que tenemos declarada una variable pública **counter** que inicializamos a 20 y dos métodos para augmentar o disminuir el contador en una unidad respectivamente.

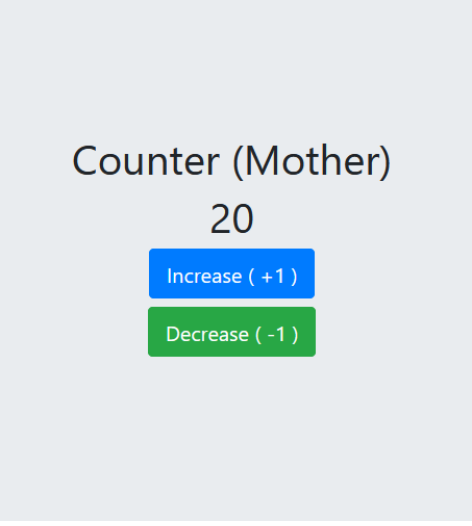
La vista, implementada en el fichero **app.component.html**, podría hacerse de la siguiente manera:

app.component.html

```
src > app > app.component.html > ...
Unsaved changes (cannot determine recent change or authors)
<div class="jumbotron d-flex align-items-center min-vh-100">
  <div class="container text-center">
    <div class="row" style="text-align: center">
      <div class="col">
        <h1>Counter (Mother)</h1>
        <h1>{{ counter }}</h1>
      </div>
    </div>

    <div class="row" style="text-align: center">
      <div class="col">
        <button
          (click)="increase()"
          type="button"
          class="btn btn-primary btn-lg"
        >
          Increase ( +1 )
        </button>
      </div>
    </div>

    <div class="row mt-2" style="text-align: center">
      <div class="col">
        <button
          (click)="decrease()"
          type="button"
          class="btn btn-success btn-lg"
        >
          Decrease ( -1 )
        </button>
      </div>
    </div>
  </div>
</div>
```

Simplemente la idea de este código es mostrar en el centro de la pantalla el contador junto con los dos botones de incrementar y decrementar dicho contador.

En el navegador podremos ver el siguiente resultado:



Pulsando los botones de incrementar o decrementar podremos ver cómo va variando el valor del contador.

Hasta aquí, fácil. ¡Seguimos!

Ahora vamos a 'añadir' a la ecuación el componente **hija**.

En el componente **madre** le añadiremos el siguiente código:

src > app > app.component.html > ...

```
<div class="jumbotron d-flex align-items-center min-vh-100">
  <div class="container text-center">
    <div class="row" style="text-align: center">
      <div class="col">
        <h1>Counter (Mother)</h1>
        <h1>{{ counter }}</h1>
      </div>
    </div>

    <div class="row" style="text-align: center">
      <div class="col">
        <button
          (click)="increase()"
          type="button"
          class="btn btn-primary btn-lg"
        >
          Increase ( +1 )
        </button>
      </div>
    </div>

    <div class="row mt-2" style="text-align: center">
      <div class="col">
        <button
          (click)="decrease()"
          type="button"
          class="btn btn-success btn-lg"
        >
          Decrease ( -1 )
        </button>
      </div>
    </div>

    <hr />
    <hr />
    <app-daughter [counter]="counter"></app-daughter>
  </div>
</div>
```

Aquí podemos ver que mostraríamos el componente **hija** y a éste le mandamos el valor del contador. Por lo tanto, en el componente **hija** necesitaremos implementar un **@input**.

Después la vista del componente **hija** podría ser así:

src > app > counter > daughter > daughter.component.html > ...

```
<div class="row" style="text-align: center">
  <div class="col">
    <h1>Counter (Daughter)</h1>
    <h1>{{ counter }}</h1>
  </div>
</div>
```

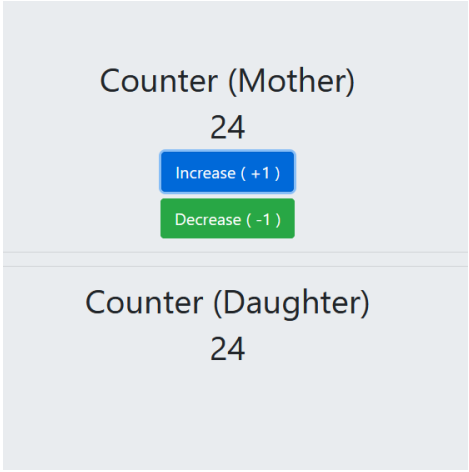
En principio lo único que queremos es mostrar el valor del contador que nos facilitará la **madre**.

Y su controlador se podría implementar así:

```
src > app > counter > daughter > daughter.component.ts > ...
1   import { Component, Input, OnInit } from '@angular/core';
2
3   @Component({
4     selector: 'app-daughter',
5     templateUrl: './daughter.component.html',
6     styleUrls: ['./daughter.component.css']
7   })
8   export class DaughterComponent implements OnInit {
9
10    @Input() counter: number;
11
12    constructor() { }
13
14    ngOnInit(): void {
15    }
16
17  }
```

Necesitamos recibir de la **madre** el valor del contador que nos envía a través de la vista que hemos visto anteriormente, por lo tanto, definimos el parámetro **input**.

Con este pequeño cambio, podemos ver que si pulsamos los botones del componente **madre** los valores del contador tanto de la **madre** como de la **hija** se actualizan correctamente:



Ahora vamos a añadir que la **hija** también pueda modificar el valor del contador. Vamos a suponer que la **hija** tendrá un botón que lo que hará al pulsarse es multiplicar por dos el valor del contador. Vamos a ello:

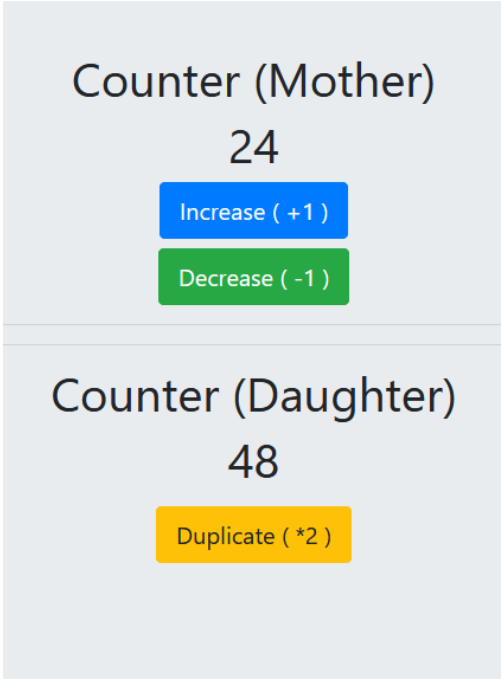
```
src > app > counter > daughter > daughter.component.html > ...
1   <div class="row" style="text-align: center">
2     <div class="col">
3       <h1>Counter (Daughter)</h1>
4       <h1>{{ counter }}</h1>
5     </div>
6   </div>
7
8   <div class="row mt-2" style="text-align: center">
9     <div class="col">
10      <button (click)="duplicate()" type="button" class="btn btn-warning btn-lg">
11        Duplicate ( *2 )
12      </button>
13    </div>
14  </div>
```

Añadimos el código remarcado en rojo, que al final es un botón para lanzar la acción de que multiplique el valor por dos. Su implementación la haríamos en el fichero **daughter.component.ts** y se vería de la siguiente manera:

src > app > counter > daughter > daughter.component.ts > ...

```
import { Component, Input, OnInit } from '@angular/core';

@Component({
  selector: 'app-daughter',
  templateUrl: './daughter.component.html',
  styleUrls: ['./daughter.component.css'],
})
export class DaughterComponent implements OnInit {
  @Input() counter: number;

  constructor() {}

  ngOnInit(): void {}

  duplicate(): void {
    this.counter = this.counter * 2;
  }
}
```

Si ejecutamos la aplicación tal y como la tenemos ahora:



Podremos ver que, si la **madre** cambia el valor del contador, tanto en la **madre** como en la **hija** se refleja el cambio, pero si es la **hija** quien duplica el valor, la **madre** no es consciente del cambio. Vamos a solucionar esto:

En el **daughter.component.ts** vamos a añadir el código para que cuando se modifique el valor del contador éste se emita:

src > app > counter > daughter > daughter.component.ts > ...

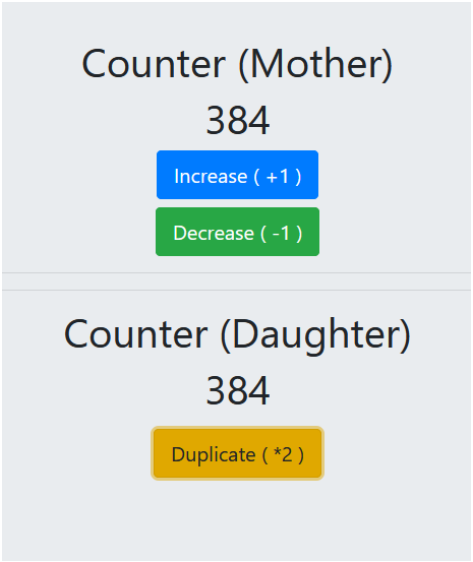
```ts
import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';

@Component({
  selector: 'app-daughter',
  templateUrl: './daughter.component.html',
  styleUrls: ['./daughter.component.css'],
})
export class DaughterComponent implements OnInit {
  @Input() counter: number;
  @Output() changeCounter = new EventEmitter<number>();

  constructor() {}

  ngOnInit(): void {}

  duplicate(): void {
    this.counter = this.counter * 2;
    this.changeCounter.emit(this.counter);
  }
}
```

Y, por otra parte, nos quedaría adaptar el código de la **madre** para que cuando la **hija** realice algún cambio ésta lo recoja, por lo tanto, en el **app.component.html** añadiremos:

src > app > app.component.html > ...

```html
<div class="jumbotron d-flex align-items-center min-vh-100">
  <div class="container text-center">
    <div class="row" style="text-align: center">
      <div class="col">
        <h1>Counter (Mother)</h1>
        <h1>{{ counter }}</h1>
      </div>
    </div>

    <div class="row" style="text-align: center">
      <div class="col">
        <button
          (click)="increase()"
          type="button"
          class="btn btn-primary btn-lg"
        >
          Increase ( +1 )
        </button>
      </div>
    </div>

    <div class="row mt-2" style="text-align: center">
      <div class="col">
        <button
          (click)="decrease()"
          type="button"
          class="btn btn-success btn-lg"
        >
          Decrease ( -1 )
        </button>
      </div>
    </div>

    <hr />
    <hr />
    <app-daughter [counter]="counter" (changeCounter)="counter = $event"></app-daughter>
  </div>
</div>
```

Si ejecutamos la aplicación, podremos ver que ahora funcionaría correctamente:



Para terminar con nuestra aplicación, vamos a sumar a la ecuación la interacción con el componente **nieta**. Vamos a suponer que la **nieta** lo que hará será siempre es poner el contador a 0. ¡Vamos a ello!

Primero lo que haremos es mostrar el contador en la vista **nieta**, es decir, en el fichero **granddaughter.component.html**:

src > app > counter > granddaughter > granddaughter.component.html > ...

```
<div class="row" style="text-align: center">
  <div class="col">
    <h1>Counter (Grand Daughter)</h1>
    <h1>{{ counter }}</h1>
  </div>
</div>
```

Después, en su controlador le tendremos que añadir la sentencia **@Input** para recibir el valor del contador desde el componente **hija**:

src > app > counter > granddaughter > granddaughter.component.ts > ...

```
import { Component, Input, OnInit } from '@angular/core';

@Component({
  selector: 'app-granddaughter',
  templateUrl: './granddaughter.component.html',
  styleUrls: ['./granddaughter.component.css'],
})
export class GranddaughterComponent implements OnInit {
  @Input() counter: number;
  constructor() {}

  ngOnInit(): void {}
}
```

Finalmente, iremos a la vista de la **hija** para mandarle a la **nieta** dicho contador:

src > app > counter > daughter > daughter.component.html > ...

```
<div class="row" style="text-align: center">
  <div class="col">
    <h1>Counter (Daughter)</h1>
    <h1>{{ counter }}</h1>
  </div>
</div>

<div class="row mt-2" style="text-align: center">
  <div class="col">
    <button (click)="duplicate()" type="button" class="btn btn-warning btn-lg">
      Duplicate ( *2 )
    </button>
  </div>
</div>

<hr />
<hr />
<app-granddaughter [counter]="counter"></app-granddaughter>
```

Si ejecutamos nuestra aplicación, podremos ver que:

- Si incrementamos/decrementamos en la **madre** todos los contadores se actualizan (la **madre** se lo manda a la **hija** y la **hija** a la **nieta**).

- Si la **hija** duplica el contador, ésta se lo manda a la **madre**, y como la **madre** tiene una relación con el **@input** la **nieta** también recibe el valor y, por lo tanto, los tres contadores se muestran correctamente.

Para finalizar, añadimos que la **nieta** pueda poner a 0 el contador de todos los componentes.

En la vista de la **nieta** añadimos:

src > app > counter > granddaughter > granddaughter.component.html > ...

```
<div class="row" style="text-align: center">
  <div class="col">
    <h1>Counter (Grand Daughter)</h1>
    <h1>{{ counter }}</h1>
  </div>
</div>

<div class="row mt-2" style="text-align: center">
  <div class="col">
    <button (click)="reset()" type="button" class="btn btn-info btn-lg">
      Reset ( 0 )
    </button>
  </div>
</div>
```

En su controlador implementamos el método de **reset**:

```
src > app > counter > granddaughter > granddaughter.component.ts > ...
1   import { Component, Input, OnInit } from '@angular/core';
2
3   @Component({
4     selector: 'app-granddaughter',
5     templateUrl: './granddaughter.component.html',
6     styleUrls: ['./granddaughter.component.css'],
7   })
8   export class GranddaughterComponent implements OnInit {
9     @Input() counter: number;
10    constructor() {}
11
12    ngOnInit(): void {}
13
14    reset(): void {
15      this.counter = 0;
16    }
17  }
```

Si ejecutamos la aplicación, podremos ver que al pulsar el botón de **reset** sólo se pone a 0 el contador de este componente, y es normal, nos falta una última implementación para que al resetear se notifique a la **madre** y la **hija**. Hacemos lo siguiente:

Añadimos lo siguiente al controlador de la **nieta**:

```
src > app > counter > granddaughter > granddaughter.component.ts > ...
1   import { EventEmitter, Output } from '@angular/core';
2   import { Component, Input, OnInit } from '@angular/core';
3
4   @Component({
5     selector: 'app-granddaughter',
6     templateUrl: './granddaughter.component.html',
7     styleUrls: ['./granddaughter.component.css'],
8   })
9   export class GranddaughterComponent implements OnInit {
10    @Input() counter: number;
11    @Output() changeCounter = new EventEmitter<number>();
12    constructor() {}
13
14    ngOnInit(): void {}
15
16    reset(): void {
17      this.counter = 0;
18      this.changeCounter.emit(this.counter);
19    }
20  }
```

Porque necesitamos que cuando ésta cambie el valor del contador se emita el cambio para que se 'dé cuenta' el componente **hija**.

Entonces iríamos a la vista del componente **hija** y haríamos:

src > app > counter > daughter > daughter.component.html > ...

```
<div class="row" style="text-align: center">
  <div class="col">
    <h1>Counter (Daughter)</h1>
    <h1>{{ counter }}</h1>
  </div>
</div>

<div class="row mt-2" style="text-align: center">
  <div class="col">
    <button (click)="duplicate()" type="button" class="btn btn-warning btn-lg">
      Duplicate ( *2 )
    </button>
  </div>
</div>

<hr />
<hr />
<app-granddaughter [counter]="counter" (changeCounter)="counter = $event"></app-granddaughter>
```

Si guardamos y ejecutamos, podremos ver que cuando hacemos el **reset** des del componente **nieta**, se actualiza el contador de la **nieta** y de la **hija,** pero no de la **madre**. Nos quedaría un último paso para solucionar la falta de esta comunicación.

Tenemos que modificar esta última implementación en el componente **hija** de la siguiente manera:

src > app > counter > daughter > daughter.component.html > ...

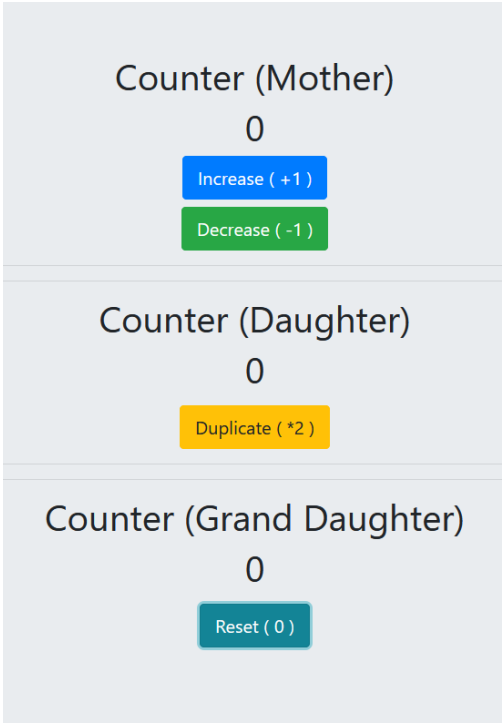
```
<div class="row" style="text-align: center">
  <div class="col">
    <h1>Counter (Daughter)</h1>
    <h1>{{ counter }}</h1>
  </div>
</div>

<div class="row mt-2" style="text-align: center">
  <div class="col">
    <button (click)="duplicate()" type="button" class="btn btn-warning btn-lg">
      Duplicate ( *2 )
    </button>
  </div>
</div>

<hr />
<hr />
<app-granddaughter [counter]="counter" (changeCounter)="resetGrandDaughter($event)"></app-granddaughter>
```

Y en su controlador:

src > app > counter > daughter > daughter.component.ts > ...

```
import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';

@Component({
  selector: 'app-daughter',
  templateUrl: './daughter.component.html',
  styleUrls: ['./daughter.component.css'],
})
export class DaughterComponent implements OnInit {
  @Input() counter: number;
  @Output() changeCounter = new EventEmitter<number>();

  constructor() {}

  ngOnInit(): void {}

  duplicate(): void {
    this.counter = this.counter * 2;
    this.changeCounter.emit(this.counter);
  }

  resetGrandDaughter( newCounter: number ): void {
    this.counter = newCounter;
    this.changeCounter.emit(this.counter);
  }
}
```

Y podremos ver que la aplicación nos funcionará correctamente:



- Si incrementamos/decrementamos en la **madre** todos los contadores se actualizan (la **madre** se lo manda a la **hija** y la **hija** a la **nieta**)

- Si la **hija** duplica el contador, ésta se lo manda a la **madre**, y como la madre tiene una relación con el **@input** la **nieta** también recibe el valor y, por lo tanto, los tres contadores se muestran correctamente

- Si la **nieta** resetea el contador, los tres contadores ven reflejado el valor 0, ya que la **nieta** emite y la **hija** recoge el cambio, y al cambiar el valor, la **hija** emite el cambio y la **madre** recoge el cambio.

Podemos pensar que tenemos muchos **@inputs**/**@outputs** para una tarea relativamente sencilla, y que podríamos mejorar o limpiar un poco el código si utilizáramos un servicio de **Angular** por ejemplo, pero continuaríamos teniendo una complejidad que en principio no nos compensaría por la sencillez de la aplicación.

Ahora vamos a aplicar el patrón **Redux** a la aplicación que acabamos de implementar para ver cómo nos queda el código de limpio y entendible al tener la gestión de la información centralizada.

¡Vamos a ello!

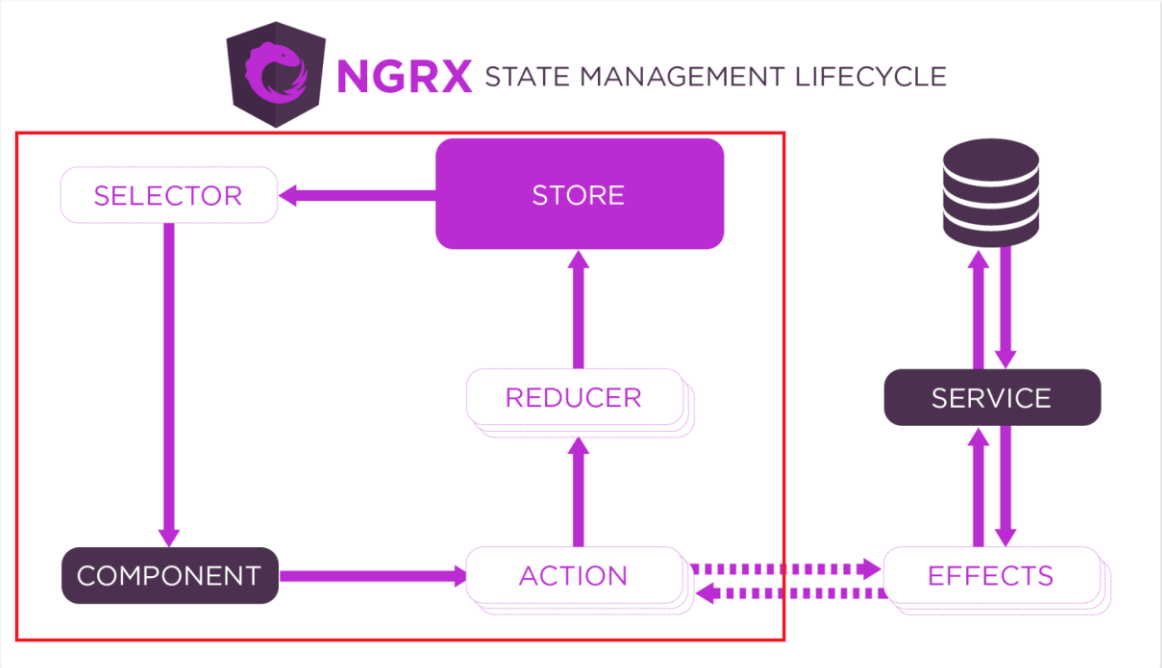
Counter App con Redux

Para poder trabajar con **Redux** necesitamos instalar un paquete a nuestro proyecto.

Podemos encontrar información al respecto en la página oficial del proyecto:

https://ngrx.io/

Si vamos al apartado *getting started* veremos el siguiente gráfico:



La parte izquierda resaltada en rojo es la que implementaremos a continuación en nuestra aplicación. Desde los componentes lanzaremos las acciones, que mediante el *reducer* modificarán el *store*. Y aquí vemos un concepto nuevo que sería el *selector*, que lo veremos en breve, pero lo podemos ver como la manera de notificar a un componente de un cambio y además teniendo la posibilidad de seleccionar la propiedad que queremos notificar.

Apuntar que la parte de los *effects* la estudiaremos en la parte 2 del documento cuando implementemos una aplicación de tipo TODO app.

Si vamos al apartado *installation* podremos ver que para instalar el paquete tendremos que ejecutar el siguiente comando:

```
H:\Projectes\basic-redux-app>npm install @ngrx/store --save
```

Una vez instalado, vamos a empezar a aplicar el patrón **Redux**.

Lo primero que haremos es crearnos las acciones. Pensemos en el contexto que estamos. Tenemos:

- Un componente **madre** que puede incrementar o decrementar el contador.
- Un componente **hija** que puede duplicar el contador.
- Un componente **nieta** que puede resetear el contador.

Cada una de estas funciones que modifican el valor del contador serán una acción. Por lo tanto, vamos a empezar creándonos un fichero donde implementaremos dichas acciones, podemos crearnos un fichero **counter.actions.ts** dentro de la carpeta **counter**.

De momento escribiremos el siguiente código:

src > app > counter > counter.actions.ts > ...

```
1  import { createAction } from '@ngrx/store';
2
3  export const increment = createAction('[Counter Component] Increment');
4  export const decrement = createAction('[Counter Component] Decrement');
5
```

Recordemos que las acciones describen que es lo que hace dicha acción, por lo tanto, de momento nos vale con indicar una cadena de texto para indicar si incrementamos o decrementamos.

Ahora crearemos el fichero para manejar el *reducer*. Para que se entienda mejor cómo funciona el *reducer*, vamos a implementarlo nosotros mismos, de igual manera que lo comentamos en la introducción de este documento. Posteriormente lo implementaremos de la manera que nos proponen en la guía oficial.

Recordemos que el *reducer* es una función que recibe dos argumentos, una acción y un estado, y la lógica que implementa esta función no es más que un *switch/case* donde en función del *action.type* hará la modificación determinada en el *state*.

Creamos dentro de la carpeta **counter** el fichero **counter.reducer.ts**, e implementamos el siguiente código:

src > app > counter > counter.reducer.ts > ...

```
1  import { Action } from '@ngrx/store';
2  import { decrement, increment } from './counter.actions';
3
4  export function counterReducer(state: number = 20, action: Action) {
5
6    switch (action.type) {
7
8      case increment.type:
9        return state + 1;
10
11     case decrement.type:
12       return state - 1;
13
14     default:
15       return state;
16
17   }
18
19 }
```

Si volvemos al primer apartado de la introducción de este documento, veríamos que tenemos algo muy parecido. En el *reducer* que nos estamos creando tenemos una función que llamamos **counterReducer** que tiene dos argumentos, un estado y una acción.

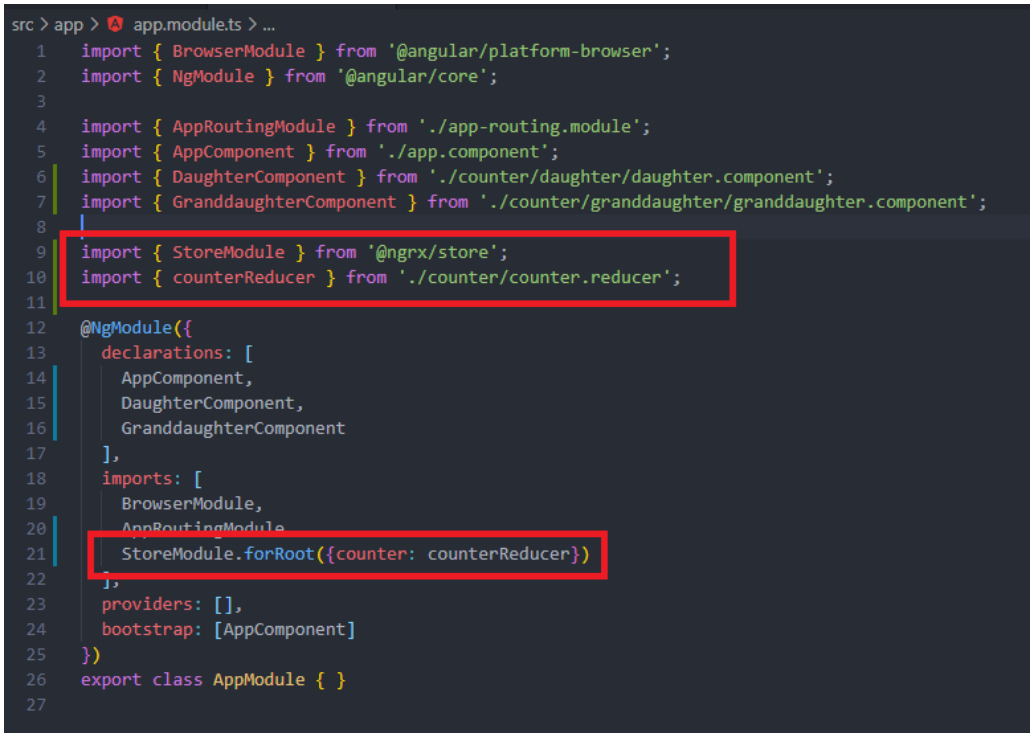
Respecto al estado, tenemos que pensar que en esta aplicación tan sencilla es simplemente un contador, y respecto a la acción, pues de momento tenemos dos posibles, la acción incrementar y decrementar implementadas justo en el paso anterior a este.

Vale, pues como decíamos, tenemos que ver al *reducer* como un *switch/case*, de manera que, si la acción es incrementar, le sumamos 1 al estado y si es decrementar, le restamos 1. Recordemos que una acción tenía dos propiedades, el **type** y el **payload**. De momento utilizamos el **type** para saber si queremos incrementar o decrementar.

Es importante destacar que dentro del *reducer* no tenemos que hacer llamadas a terceros, no puede haber llamadas a apis, servicios, … lo que se haga en el *reducer* se tiene que hacer con la acción y el estado que se le pasa por argumento, nada más.

Digamos que ya tenemos dos acciones implementadas y el *reducer* que utilizaríamos para modificar el estado de nuestra aplicación, pero nos faltaría hacer una configuración para poder guardar este nuevo estado en el **store**, por lo tanto, vamos a configurarlo.

Tenemos que ir al fichero **app.module.ts** y añadir lo siguiente:



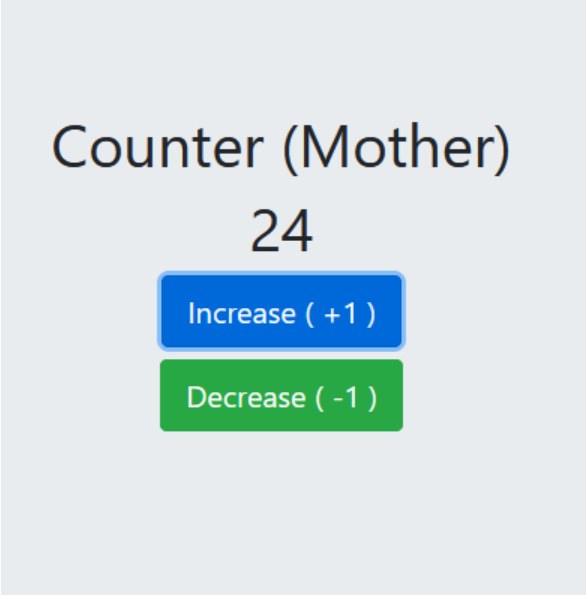
En el apartado **imports** vemos que hemos añadido un código que nos sirve para describir como es el **estado** de mi aplicación.

Llegados a este punto tendríamos las acciones incrementar y decrementar y el **reducer** implementados y el **store** configurado, vamos a utilizarlo.

Nos vamos al componente **app.component.html** que sería la **madre** y comentamos el código para que no nos aparezca el componente **hija**:

```
src > app > app.component.html > ...
<div class="jumbotron d-flex align-items-center min-vh-100">
  <div class="container text-center">
    <div class="row" style="text-align: center">
      <div class="col">
        <h1>Counter (Mother)</h1>
        <h1>{{ counter }}</h1>
      </div>
    </div>

    <div class="row" style="text-align: center">
      <div class="col">
        <button
          (click)="increase()"
          type="button"
          class="btn btn-primary btn-lg"
        >
          Increase ( +1 )
        </button>
      </div>
    </div>

    <div class="row mt-2" style="text-align: center">
      <div class="col">
        <button
          (click)="decrease()"
          type="button"
          class="btn btn-success btn-lg"
        >
          Decrease ( -1 )
        </button>
      </div>
    </div>

    <!--
    <hr />
    <hr />
    <app-daughter [counter]="counter" (changeCounter)="counter = $event"></app-daughter>
    -->
  </div>
</div>
```

De esta manera la aplicación se vería así:



Y podemos ver que funciona correctamente.

Ahora nos iríamos al fichero **app.component.ts** y haremos los siguientes cambios:

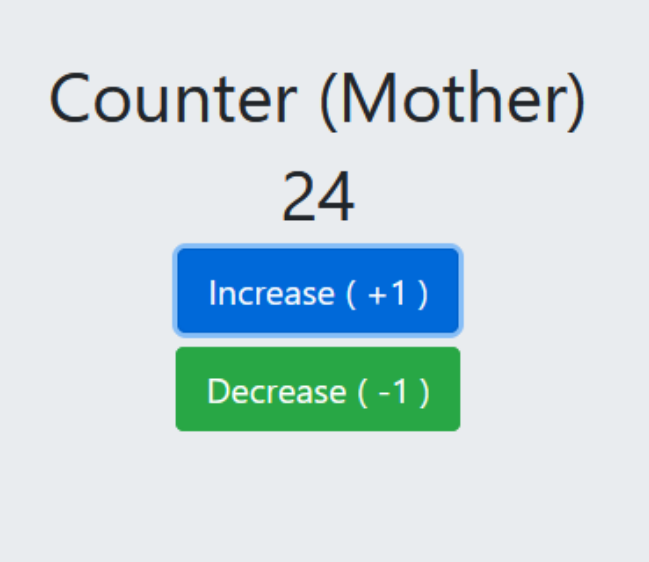
A) Tenemos que inyectar el servicio para gestionar el **store**. Fijémonos que le pasamos **{counter: number}**. Esto sería lo que tiene que gestionar el **store**, el estado, que en nuestro caso es muy sencillo ya que simplemente es un contador. Recordemos que el estado normalmente será algo más complejo y lo parametrizaremos mediante una interfaz, pero esto lo veremos más adelante.

B) Nos suscribimos a cualquier cambio que suceda en el **store**, y cuando se produzca, asignamos el resultado a la variable pública **counter** que mostramos en la vista de la **madre**.

C) Lanzamos la acción de incrementar para que nos modifique el **store**.

D) Lanzamos la acción de decrementar para que nos modifique el **store**.

Si vamos al navegador podemos ver que la aplicación funciona correctamente, a través del patrón **Redux**:

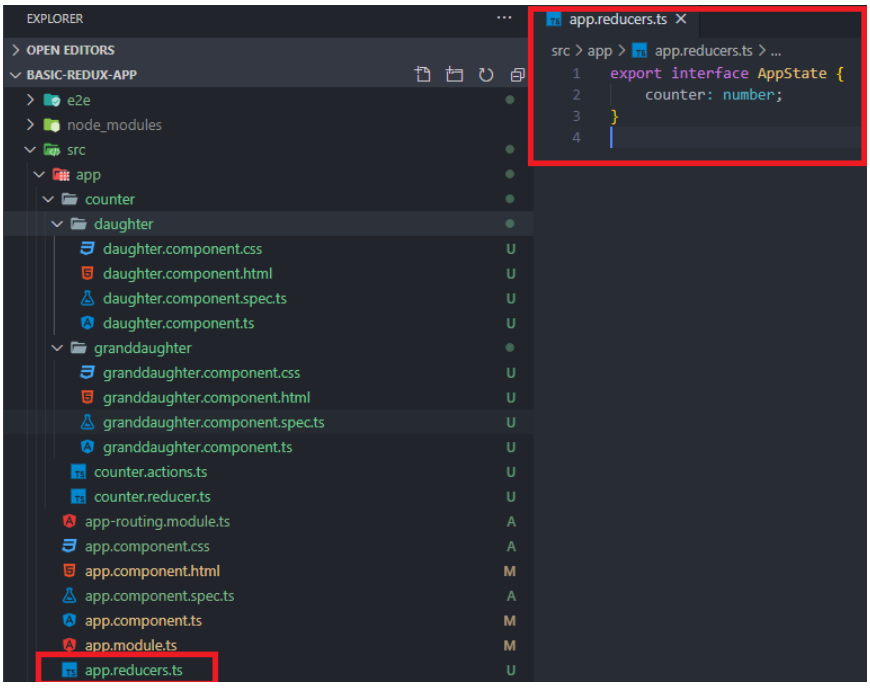


Ahora que tenemos aplicado el patrón **Redux** en el componente **madre**, podemos ver que realmente no nos implica demasiados cambios, pero antes de implementarlo en el componente **hija** y **nieta** y ver como reducimos el código de manera destacable, vamos a hacerle unas mejoras a nuestro código.

1.- Mejoramos cómo gestionamos el estado en el store:

Hemos visto en el **app.component.ts** que inyectamos el servicio para gestionar el **store** y a éste le pasamos el estado de la forma **{ counter: number }**. Claro, esto porque en nuestra aplicación sólo tenemos que gestionar un contador, pero como hemos dicho alguna vez, el estado de una aplicación normalmente será un objeto más complejo. Por lo tanto, vamos a implementarlo de una manera que se acercaría más a la realidad.

Nos crearemos un fichero **app.reducers.ts** en la raíz de la carpeta **app** y dentro copiaremos el siguiente código:



Como podemos ver, y como explicamos con un ejemplo al principio de todo de este documento, podemos definir el estado de esta aplicación como una interfaz, que en este caso, sólo tiene una propiedad para guardar el valor de un contador.

Ahora lo que haremos es mejorar como le pasamos el estado al argumento cuando inyectamos el servicio del store en el **app.component.ts**:

Al store le pasamos directamente la interfaz, que define cómo es nuestro estado.

Podremos ver que nuestra aplicación continúa funcionando correctamente.

2.- Mejoramos cómo importamos las acciones:

Si empezamos a tener muchas acciones los **imports** de arriba del todo del fichero quedará muy cargado, por lo que podemos hacer este tipo de **import** que remarcamos en rojo, importar todo el contenido con un alias, en nuestro caso **actions**, y luego utilizar el alias **actions.** <<la acción que sea >>. De esta manera nos quedará la sección de los **imports** un poco más limpia.

3.- Mejoramos a que queremos suscribirnos cuando un elemento del store se modifique.

Antes hemos definido nuestro estado en una interfaz accesible desde cualquier parte del proyecto. Esta interfaz es la que define la estructura de nuestro estado y es a la que le pasamos al **store** en el constructor del **app.component.ts** para inicializarlo.

Vale, en nuestro caso, solo tenemos el contador, pero en una aplicación real tendríamos más propiedades, que formarían parte de un objeto más complejo. Tal como tenemos el **subscribe** ahora mismo dentro del fichero **app.component.ts**, está bien, en el sentido de que estaremos al caso cuando se produzca un cambio en el estado y por tanto, podremos mostrar correctamente el valor del contador. Pero con esta implementación, estamos al caso de cualquier cambio que puede tener el estado, de todo el estado, y esto no siempre es lo que queremos. Porque ahora sólo tenemos la propiedad **counter**, pero

si también tuviéramos por ejemplo una propiedad **user**, una propiedad **tasks**, que fuera un array de tareas, … Nos puede interesar saber cuándo se produce un cambio de un elemento especifico de nuestro estado. Para hacer esto, solo tenemos que utilizar la sentencia **select**, lo vemos a continuación:

app.component.ts

src > app > app.component.ts > ...

```
import { Component } from '@angular/core';
import { Store } from '@ngrx/store';
import { AppState } from './app.reducers';
// import { decrement, increment } from './counter/counter.actions
import * as actions from './counter/counter.actions';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  title = 'Basic Redux App';

  counter: number;

  // constructor( private store: Store<{ counter: number }>) {
  constructor( private store: Store<AppState>) {
    // this.counter = 20;

    /*
    this.store.subscribe( state => {
      this.counter = state.counter;
    });
    */

    this.store.select('counter').subscribe( counter => {
      this.counter = counter;
    });
  }

  increase(): void {
    // this.counter = this.counter + 1;
    this.store.dispatch( actions.increment() );
  }

  decrease(): void {
    // this.counter = this.counter - 1;
    this.store.dispatch( actions.decrement() );
  }
}
```

Podemos probar la aplicación y veremos que funciona correctamente.

Ahora lo que vamos a hacer es mejorar el *reducer* que hemos implementado. Si bien es verdad que nuestro *reducer* sería válido, con el *switch/case* y en función de la acción que haga lo que tenga que hacer al estado, desde **NgRx** se propone una manera de implementar los *reducer* de manera más optima. Vamos a coger nuestra implementación y vamos a cambiarla de la manera que nos proponen:

Recordemos que teníamos esto:

counter.reducer.ts

```
src > app > counter > counter.reducer.ts > ...
import { Action } from '@ngrx/store';
import { decrement, increment } from './counter.actions';

export function counterReducer(state: number = 20, action: Action) {

  switch (action.type) {

    case increment.type:
      return state + 1;

    case decrement.type:
      return state - 1;

    default:
      return state;

  }

}
```

Y ahora tendremos igualmente un **counterReducer** pero de la manera que nos dicen los creadores del patrón, sería la siguiente:

(lo hemos hecho primero de esta manera para que conceptualmente se entienda mejor, ya que el código que nos proponen es un poco más complejo de entender)

```
src > app > counter > counter.reducer.ts > ...
import { Action, createReducer, on } from '@ngrx/store';
import { decrement, increment } from './counter.actions';

/*
export function counterReducer(state: number = 20, action: Action) {

  switch (action.type) {

    case increment.type:
      return state + 1;

    case decrement.type:
      return state - 1;

    default:
      return state;

  }

}
*/

export const initialState = 20;

const _counterReducer = createReducer(
  initialState,
  on(increment, (state) => state + 1),
  on(decrement, (state) => state - 1)
);

export function counterReducer(state, action) {
  return _counterReducer(state, action);
}
```

Dejamos comentado el código que teníamos para que podamos comparar. Fijémonos que la función del *reducer* **counterReducer** mantiene el nombre, que es el que se utiliza en el **app.module.ts**.

Esta función que es la que queda expuesta para ser utilizada en otras partes del proyecto internamente llama a la función **_counterReducer** que recibe los dos parámetros, acción y estado.

Esta función podemos ver que llama a una función propia del patrón **Redux createReducer** y está más optimizada que nuestro *switch/case*. Vemos que se le pasa un estado inicial (asignación de 20 al contador) tal como teníamos anteriormente, y luego con cada instrucción **on** definimos las acciones, y cada una hará la modificación determinada al estado.

Además, posteriormente veremos cómo esta estructura nos ayudará, cuando trabajemos con estados que serán objectos, a no mutar dichos estados.

Como vemos, hacen lo mismo, pero esta última es la manera en la que nos proponen desde la documentación oficial.

Continuemos con nuestro desarrollo, vamos a gestionar el **store** del componente **hija**.

Primero tenemos que descomentar el código de la vista del fichero **app.component.html** para poder mostrar el componente **hija**.

src > app > app.component.html > ...

```
<div class="jumbotron d-flex align-items-center min-vh-100">
  <div class="container text-center">
    <div class="row" style="text-align: center">
      <div class="col">
        <h1>Counter (Mother)</h1>
        <h1>{{ counter }}</h1>
      </div>
    </div>

    <div class="row" style="text-align: center">
      <div class="col">
        <button
          (click)="increase()"
          type="button"
          class="btn btn-primary btn-lg"
        >
          Increase ( +1 )
        </button>
      </div>
    </div>

    <div class="row mt-2" style="text-align: center">
      <div class="col">
        <button
          (click)="decrease()"
          type="button"
          class="btn btn-success btn-lg"
        >
          Decrease ( -1 )
        </button>
      </div>
    </div>


    <hr />
    <hr />
    <app-daughter [counter]="counter" (changeCounter)="counter = $event"></app-daughter>

  </div>
</div>
```

Al descomentar este código en nuestra aplicación deberíamos volver a ver todos los componentes. Aunque a priori parece que todo funciona, nos quedan algunas cosas por hacer.

Pensemos qué situación tenemos ahora. Se está gestionando con el **store** la modificación de nuestro estado de nuestra aplicación (un simple contador) cuando lo modifica el componente **madre** cuando lanza las acciones de incrementar o decrementar.

En cambio, tanto el componente **hija** cuando duplica como el componente **nieta** cuando resetea, modifican directamente el estado de nuestra aplicación y esto es incorrecto. Por lo tanto, vamos a hacer estas últimas adaptaciones.

En el **app.component.html** en el código donde se instancia al componente **<app-daughter>** se le pasa la variable **counter** para que lo reciba dicho componente y también tenemos un **changeCounter** para que emita información. Todo esto ahora no lo necesitaremos ya que lo gestionaremos a través del **store**, por tanto, el código simplemente nos quedaría así:

```
<hr />
<hr />
<!--<app-daughter [counter]="counter" (changeCounter)="counter = $event"></app-daughter>-->
<app-daughter></app-daughter>
```

Ahora nos centraremos en el código de la **hija**, por tanto, nos vamos al fichero **daughter.component.ts** :

src > app > counter > daughter > daughter.component.ts > DaughterComponent

```
1  import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
2  import { Store } from '@ngrx/store';
3  import { AppState } from 'src/app/app.reducers';
4
5  @Component({
6    selector: 'app-daughter',
7    templateUrl: './daughter.component.html',
8    styleUrls: ['./daughter.component.css'],
9  })
10 export class DaughterComponent implements OnInit {
11   // @Input() counter: number;
12   // @Output() changeCounter = new EventEmitter<number>();          (A)
13
14   counter: number;                                                 (B)
15
16   constructor( private store: Store<AppState>) {}                  (C)
17
18   ngOnInit(): void {
19     // para obtener el valor del contador
20     this.store.select('counter').subscribe( counter => this.counter = counter );   (D)
21   }
22
23   duplicate(): void {
24     // this.counter = this.counter * 2;                            (E)
25     // this.changeCounter.emit(this.counter);
26   }
27
28   /*
29   resetGrandDaughter( newCounter: number ): void {                 (F)
30     this.counter = newCounter;
31     this.changeCounter.emit(this.counter);
32   }
33   */
34 }
```

A) Quitaremos los **@input/outputs** ya que no necesitaremos recoger o emitir información.

B) Necesitaremos una variable local que será la que pintaremos en la vista.

C) Inyectamos el servicio para poder gestionar el **store**, le pasamos nuestro **state**.

D) Necesitamos obtener el estado actual de la aplicación, en este caso, necesitamos obtener el valor actual del contador y asignárselo a la variable local que pintaremos en la vista.

E) Comentamos el código del método **duplicate** ya que lo haremos posteriormente mediante una acción.

F) Comentamos el método **resetGrandDaughter** ya que directamente esto no será necesario, lo terminaremos posteriormente, al descomentar este método, nos dará error en la vista **daughter.component.html** ya que se utiliza, podemos eliminar esta llamada de esta vista también.

Para poder implementar el método **duplicate** primero necesitamos añadir la acción, recordemos que tenemos implementadas sólo las de incrementar y decrementar. Vamos a ello:

Aunque sabemos que la acción de duplicar simplemente sería multiplicar por 2, lo que vamos a hacer es que a esta acción le podamos mandar un parámetro, que en nuestro caso sería un 2, pero así vemos como funciona esto de los parámetros en las acciones.

En este caso, lo que podemos hacer es añadir la función **props**, y le indicamos que le mandaremos una variable que se llama **number** que será de tipo numérico.

Ahora nos iremos a nuestro *reducer* para añadir la gestión de esta nueva acción, por tanto, iríamos al fichero **counter.reducers.ts** :

src > app > counter > counter.reducer.ts > ...

```ts
import { Action, createReducer, on } from '@ngrx/store';
import { decrement, duplicate, increment } from './counter.actions';

/*
export function counterReducer(state: number = 20, action: Action) {

  switch (action.type) {

    case increment.type:
      return state + 1;

    case decrement.type:
      return state - 1;

    default:
      return state;

  }

}
*/

export const initialState = 20;

const _counterReducer = createReducer(
  initialState,
  on(increment, (state) => state + 1),
  on(decrement, (state) => state - 1),
  on(duplicate, ( state, { number } ) => state * number)
);

export function counterReducer(state, action) {
  return _counterReducer(state, action);
}
```

Esta sería la manera de recibir el parámetro **number** y lo utilizaríamos para hacer el producto con nuestro **counter**.

Ahora nos iríamos al componente **hija** y implementaríamos el método **duplicate** mediante el **store**:

src > app > counter > daughter > daughter.component.ts > ...

```ts
import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
import { Store } from '@ngrx/store';
import { AppState } from 'src/app/app.reducers';
import { duplicate } from '../counter.actions';

@Component({
  selector: 'app-daughter',
  templateUrl: './daughter.component.html',
  styleUrls: ['./daughter.component.css'],
})
export class DaughterComponent implements OnInit {
  // @Input() counter: number;
  // @Output() changeCounter = new EventEmitter<number>();

  counter: number;

  constructor( private store: Store<AppState>) {}

  ngOnInit(): void {
    // para obtener el valor del contador
    this.store.select('counter').subscribe( counter => this.counter = counter );
  }

  duplicate(): void {
    // this.counter = this.counter * 2;
    // this.changeCounter.emit(this.counter);
    this.store.dispatch( duplicate({number: 2}) );
  }

  /*
  resetGrandDaughter( newCounter: number ): void {
    this.counter = newCounter;
    this.changeCounter.emit(this.counter);
  }
  */
}
```

Fijémonos qué limpio nos quedaría el código, simplemente llamaríamos al **dispatch** para que se lance la acción de duplicar y enviaríamos el número 2.

Vamos a gestionar el **store** del componente **nieta**.

Adaptamos la vista **daughter.component.html**:

```
src > app > counter > daughter > daughter.component.html > ...
<div class="row" style="text-align: center">
  <div class="col">
    <h1>Counter (Daughter)</h1>
    <h1>{{ counter }}</h1>
  </div>
</div>

<div class="row mt-2" style="text-align: center">
  <div class="col">
    <button (click)="duplicate()" type="button" class="btn btn-warning btn-lg">
      Duplicate ( *2 )
    </button>
  </div>
</div>

<hr />
<hr />
<!-- <app-granddaughter [counter]="counter" (changeCounter)="resetGrandDaughter($event)"></app-granddaughter>-->
<app-granddaughter></app-granddaughter>
```

Creamos la acción reset:

```
src > app > counter > counter.actions.ts > ...
import { createAction, props } from '@ngrx/store';

export const increment = createAction('[Counter Component] Increment');
export const decrement = createAction('[Counter Component] Decrement');

export const duplicate = createAction('[Counter Component] Duplicate', props<{number: number}>() );

export const reset = createAction('[Counter Component] Reset');
```

Añadimos la acción al *reducer*:

```
src > app > counter > counter.reducer.ts > ...
import { Action, createReducer, on } from '@ngrx/store';
import { decrement, duplicate, increment, reset } from './counter.actions';

/*
export function counterReducer(state: number = 20, action: Action) {

  switch (action.type) {

    case increment.type:
      return state + 1;

    case decrement.type:
      return state - 1;

    default:
      return state;

  }

}
*/

export const initialState = 20;

const _counterReducer = createReducer(
  initialState,
  on(increment, (state) => state + 1),
  on(decrement, (state) => state - 1),
  on(duplicate, (state, { number }) => state * number),
  on(reset, (state) => initialState),
);

export function counterReducer(state, action) {
  return _counterReducer(state, action);
}
```

Vamos a hacer que la función *reset* en este caso se asigne el estado inicial que en nuestro ejemplo era 20. Simplemente es un ejemplo.

Finalmente adaptaríamos el fichero **granddaughter.component.ts**:

A) Quitaremos los **@input/outputs** ya que no necesitaremos recoger o emitir información.

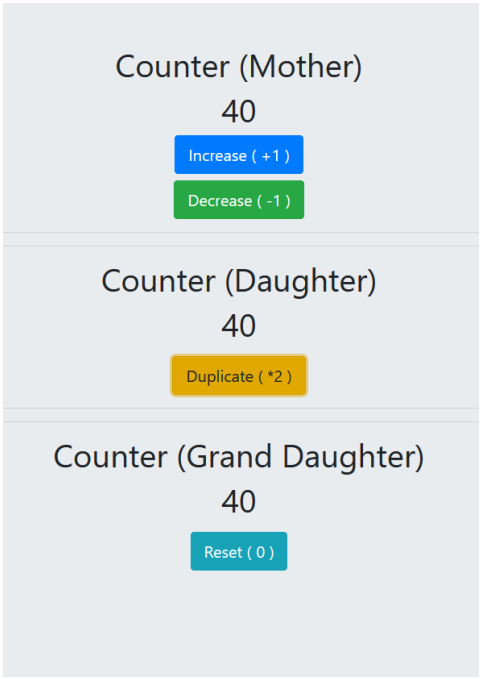
B) Necesitaremos una variable local que será la que pintaremos en la vista.

C) Inyectamos el servicio para poder gestionar el **store**, le pasamos nuestro **state**.

D) Necesitamos obtener el estado actual de la aplicación, en este caso, necesitamos obtener el valor actual del contador y asignárselo a la variable local que pintaremos en la vista.

E) Comentamos el código del método **reset** ya que lo haremos mediante una acción que gestiona el **store**.

La aplicación:

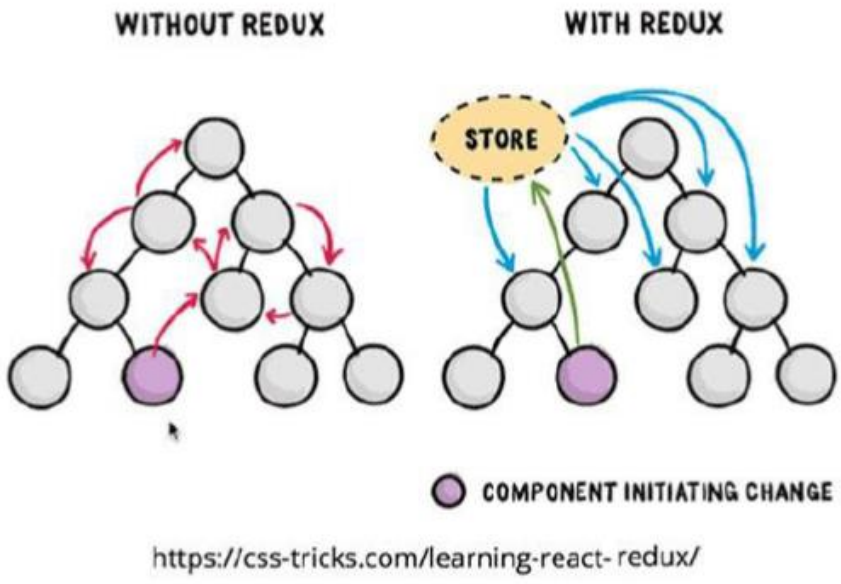


Funcionando 100% con **Redux.**

Ya hemos terminado.

Aquí sería muy interesante para terminar de ver cómo se comporta el patrón internamente configurar el **Store DevTools** para ver cómo se interactúa con el **store**, cómo va cambiando el estado, …

Después de esta implementación, el siguiente grafico deberíamos tenerlo claro:



En la parte izquierda, sin **Redux**, cada flecha roja sería un @input/@output. Cada componente podría emitir y modificar la información.

En la parte derecha, un componente realiza una acción, esta hace su tarea y se guarda el nuevo estado en el **store**, y todos los componentes, servicios, … que estén suscritos al elemento concreto del estado serán notificados.