

# TEMA 2

## REDUX ( Parte 2 )

**Desarrollo front-end avanzado**

**Máster Universitario en Desarrollo de  
sitios y aplicaciones web**

UOC

Universitat Oberta  
de Catalunya

### Contenido

#### Parte 1

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

#### Parte 2

- TODO App con Redux



# TODO App con Redux

En este documento vamos a desarrollar de manera guiada una aplicación de tipo **TODO list** con la implementación del patrón **Redux** para asimilar todos los conceptos estudiados en la parte 1 de esta documentación. Además, cuando finalicemos este desarrollo, también haremos una introducción a cómo se utilizan los **effects** mediante algún ejemplo y dejaremos el proyecto estructurado como si fuera un proyecto real, preparado para añadir en un momento dado más acciones y **reducers**.

A modo de estructurar el documento, podemos decir que tendremos una aplicación donde trabajaremos con la entidad **todo** (*“por hacer”* en inglés), que tendrá las propiedades **id**, **title** y **done**, las cuales serán un identificador, el título de la tarea y si está terminada o no, respectivamente. Con esta entidad podremos hacer las siguientes acciones:

- Crear una nueva tarea.
- Editar el título de una tarea determinada.
- Dar por terminada una tarea determinada.
- Eliminar una tarea determinada, que puede estar terminado o no.
- Si una tarea está terminada, entonces solo podremos eliminarla, ninguna acción más.

Este es un poco el escenario que queremos estudiar e implementar con **Redux**, ¡así que vamos a ello!

Primero de todo nos creamos un nuevo proyecto en **Angular**, lo podemos llamar **todo-app**:

```
H:\Projectes>ng new todo-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\todo-app>ng serve --open
```

Para validar que no tenemos ningún error y ver 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\todo-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"
  ],
```

Una vez creado el proyecto y echas las configuraciones básicas, empecemos el desarrollo. Nos creamos una carpeta **todos** en la raíz del proyecto, y luego ejecutamos el siguiente comando:

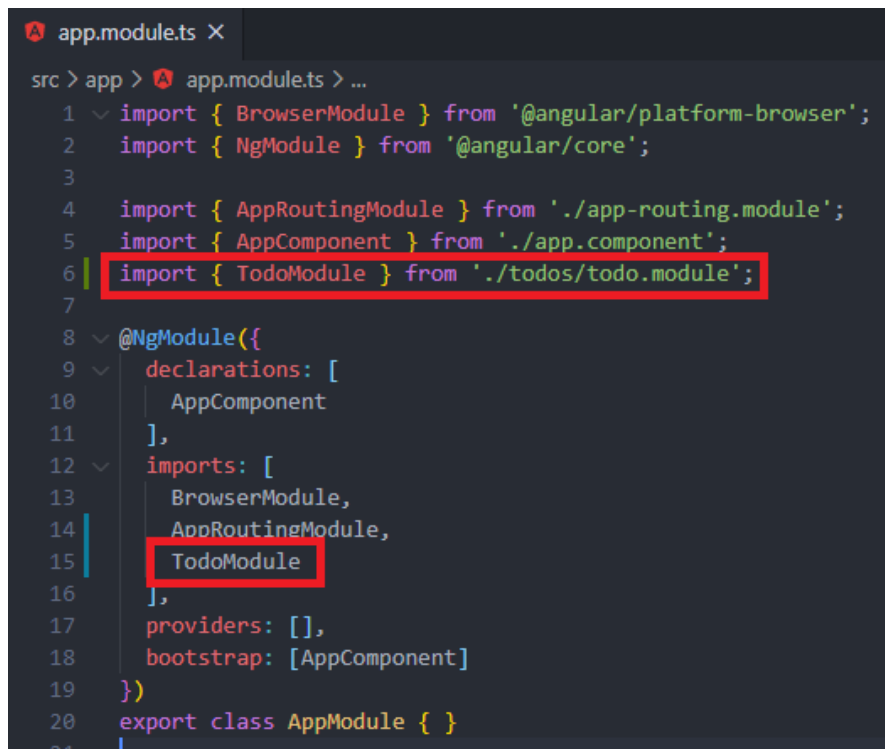
```
H:\Projectes\todo-app>ng g m todos/todo --flat
CREATE src/app/todos/todo.module.ts (190 bytes)
```

Esto sería opcional, pero vamos a crearnos un módulo para centralizar todo el trabajo en dicho módulo. Realmente para este ejercicio que estamos haciendo no nos haría falta, pero es una buena práctica utilizar módulos para agrupar todo lo que implica a una entidad o funcionalidad, ya que posteriormente, si en el proyecto utilizáramos el **lazy loading**, ya lo tendríamos preparado para poder utilizarlo. A grandes rasgos, aplicar **lazy loading** lo que nos aporta es que la aplicación tiene la capacidad de cargar solo aquellos módulos que necesita. De lo contrario, cuando arrancamos una aplicación en **Angular**, se cargan todos los componentes, módulos, ... y puede ser que nada más arrancar la aplicación, al cargarlo todo por 'debajo', vaya lenta.

En todo caso, seguimos.

La instrucción anterior nos creará el módulo dentro de la carpeta **todos**.

Ahora tendremos que ir al **app.module.ts** para importar el módulo que acabamos de crear para que podamos trabajar con él a nivel de aplicación, por tanto, haríamos:



```
app.module.ts X
src > app > app.module.ts > ...
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppRoutingModule } from './app-routing.module';
5 import { AppComponent } from './app.component';
6 import { TodoModule } from './todos/todo.module';
7
8 @NgModule({
9   declarations: [
10     AppComponent
11   ],
12   imports: [
13     BrowserModule,
14     AppRoutingModule,
15     TodoModule
16   ],
17   providers: [],
18   bootstrap: [AppComponent]
19 })
20 export class AppModule { }
21
```

Después nos crearemos los componentes necesarios para resolver las necesidades que hemos planteado al inicio del documento:

```
H:\Projectes\todo-app>ng g c todos/todo-list
CREATE src/app/todos/todo-list/todo-list.component.html (24 bytes)
CREATE src/app/todos/todo-list/todo-list.component.spec.ts (641 bytes)
CREATE src/app/todos/todo-list/todo-list.component.ts (286 bytes)
CREATE src/app/todos/todo-list/todo-list.component.css (0 bytes)
UPDATE src/app/todos/todo.module.ts (276 bytes)

H:\Projectes\todo-app>ng g c todos/todo-list-item
CREATE src/app/todos/todo-list-item/todo-list-item.component.html (29 bytes)
CREATE src/app/todos/todo-list-item/todo-list-item.component.spec.ts (670 bytes)
CREATE src/app/todos/todo-list-item/todo-list-item.component.ts (305 bytes)
CREATE src/app/todos/todo-list-item/todo-list-item.component.css (0 bytes)
UPDATE src/app/todos/todo.module.ts (382 bytes)

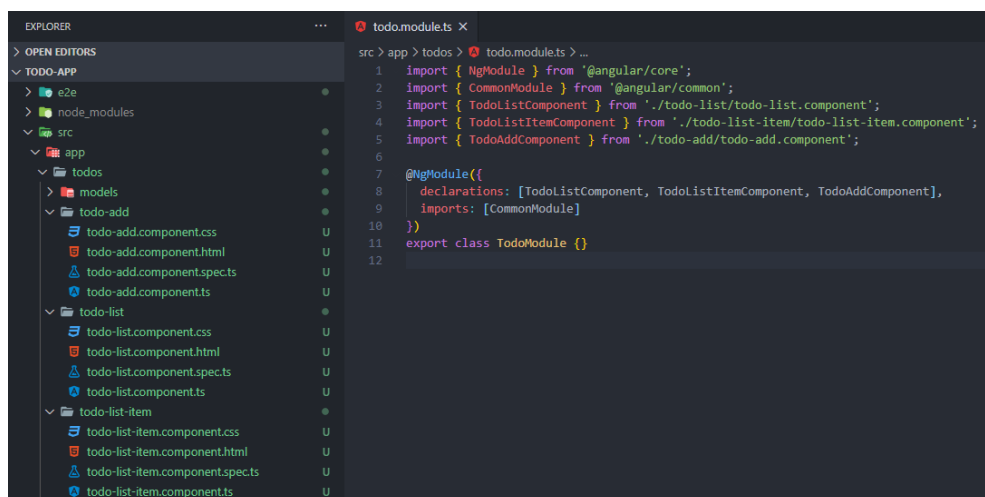
H:\Projectes\todo-app>ng g c todos/todo-add
CREATE src/app/todos/todo-add/todo-add.component.html (23 bytes)
CREATE src/app/todos/todo-add/todo-add.component.spec.ts (634 bytes)
CREATE src/app/todos/todo-add/todo-add.component.ts (282 bytes)
CREATE src/app/todos/todo-add/todo-add.component.css (0 bytes)
UPDATE src/app/todos/todo.module.ts (585 bytes)
```

De momento, podemos pensar que necesitaremos:

- Un componente para listar las tareas.
- Un componente para mostrar el detalle de las tareas, que, en principio, utilizaremos solo para mostrar el nombre de la tarea, y las acciones que le podamos hacer, ya sea editar, completar, eliminar, ...
- Un componente a parte que podemos utilizar para añadir una nueva tarea. Podríamos utilizar directamente el componente anterior, el del detalle de la tarea, pero separándolo nos facilitará el trabajo y la complejidad del código, una vez lo vayamos haciendo, nos daremos cuenta de que nos irá mejor hacerlo de esta manera.

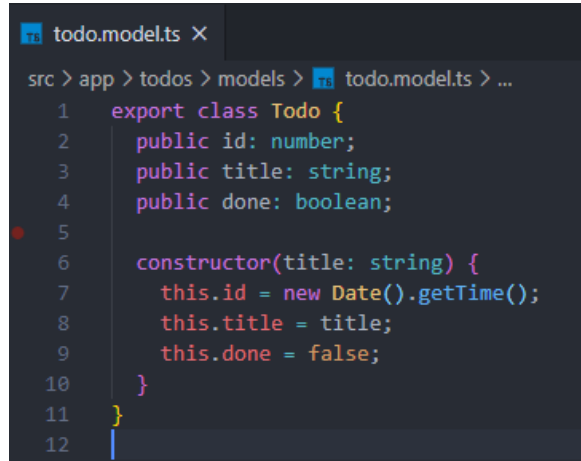
Asegurémonos que añade en los **imports** del fichero **todo.module.ts** los tres componentes que acabamos de crear.

De momento deberíamos tener una estructura así:



Vamos a crearnos el modelo ahora.

Creamos una carpeta **models** dentro de la carpeta **todos**, y dentro de la carpeta **models** nos creamos un fichero **todo.model.ts**:



```

1  export class Todo {
2      public id: number;
3      public title: string;
4      public done: boolean;
5
6      constructor(title: string) {
7          this.id = new Date().getTime();
8          this.title = title;
9          this.done = false;
10     }
11 }
12

```

Podemos escribir el código que vemos en la captura anterior. Para nosotros/as nuestra entidad **todo** podría tener:

- **Id**: para identificar la tarea, lo que sería un **id** típico de la base de datos que se utilizaría como clave primaria.
- **Title**: campo de texto para guardar el nombre de la tarea.
- **Done**: propiedad booleana para indicar si la tarea esta realizada o no. Por defecto indicaríamos que vale **false**.

Al **id** de momento le asignamos la función **Date().getTime()** simplemente para que nos devuelva un número. Como si fuera un identificador generado por una base de datos. Podríamos utilizar la función **random** de **Javascript** o cualquier otro método que queramos, simplemente es para generar un numero aleatorio y asignárselo cada vez que creamos una tarea.

Con todo esto podemos empezar a deducir como manejaríamos un **crud**:

- Cuando crearemos una tarea por defecto le asignaremos **done** a **false** y le pasaremos como argumento un texto insertado por teclado que le asignaremos a la propiedad **title**.
- Cuando editemos una tarea, le pasaremos como argumento su identificador, buscaremos en nuestro sistema la tarea con este identificador y modificaremos su título con el texto pasado por argumento el cual será insertado por teclado.
- Para eliminar una tarea, necesitaremos pasar como argumento el identificador y utilizarlo para eliminar dicha tarea del sistema.
- De manera semejante, para indicar que una tarea estará completada, necesitaremos pasar como argumento el identificador de esta, buscaremos en nuestro sistema la tarea con este identificador y modificaremos el valor de la propiedad **done**.

Una vez el modelo creado, vamos a hacer las configuraciones necesarias para poder utilizar el patrón **Redux**.

Instalamos el paquete **ngrx** para poder trabajar con el **store** con el siguiente comando:

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

Una vez el paquete instalado, vamos a crearnos nuestra primera acción.

Nos creamos el fichero **todo.actions.ts** dentro de la carpeta **todos** y escribiremos lo siguiente:

```
src > app > todos > todo.actions.ts > ...
1  import { createAction, props } from '@ngrx/store';
2
3  export const createTodo = createAction(
4    '[TODO] Create todo',
5    props<{ title: string }>()
6  );
7
```

A la acción **createTodo** le pasaríamos un parámetro como argumento que sería el texto que asignaríamos al título de la tarea. Posteriormente veremos cómo obtenemos este texto, pero ya podemos pensar que simplemente tendremos un **input** en la vista y capturaremos su contenido, por ejemplo, utilizando formularios reactivos.

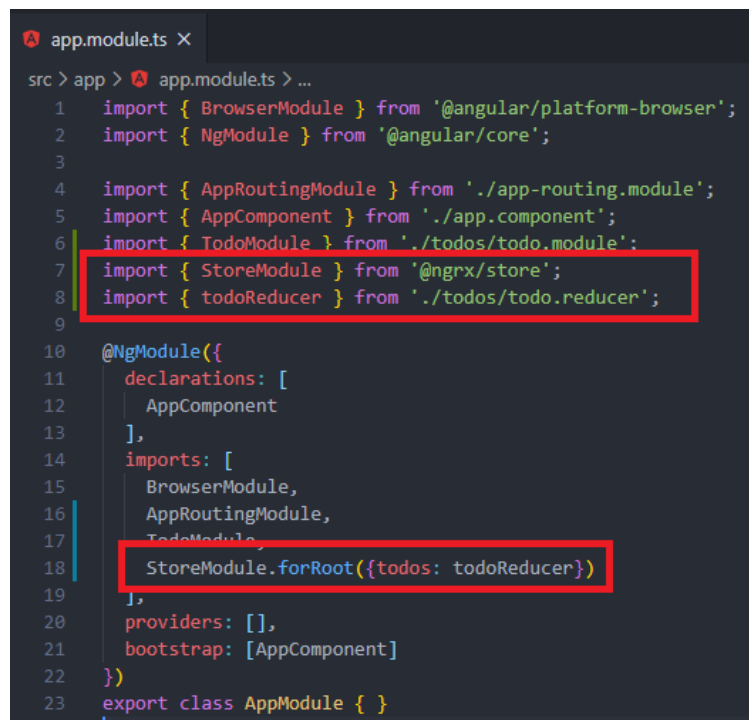
Una vez creada nuestra acción de crear tarea, vamos a implementarla, pero primero, nos vamos a crear nuestro **reducer**. Creamos el fichero **todo.reducer.ts** dentro de la carpeta **todos** y escribimos lo siguiente:

```
src > app > todos > todo.reducer.ts > ...
1  import { createReducer, on } from '@ngrx/store';
2  import { Todo } from '../models/todo.model';
3  import { createTodo } from './todo.actions';
4
5  export const initialState: Todo[] = []; (A)
6
7  const _todoReducer = createReducer(
8    initialState,
9    on(createTodo, (state, { title }) => [...state, new Todo(title)])
10 );
11                                     (B)          (C)
12 export function todoReducer(state, action) {
13   return _todoReducer(state, action);
14 }
15
```

- A) Para nosotros/as, el estado inicial de la aplicación sería un array vacío de tareas, y lo que podemos hacer es indicar el tipo del array, que en nuestro caso es definido por el modelo **Todo**. Tengamos en cuenta eso, que nuestro estado inicial por ahora sería un array vacío de **todos**.

- B) Declaramos nuestra acción **createTodo**, ésta recibirá el estado y el texto que pasamos por parámetro, de esta manera recuperaríamos dicho argumento. Posteriormente cuando utilicemos esta acción en el controlador veremos cómo recuperamos el valor de este argumento.
- C) Aquí viene una **parte importante**. En el ejemplo de la aplicación anterior del contador trabajábamos con un simple contador, ahora tenemos un objeto definido por el modelo **Todo** y además tenemos un array de objetos. Podríamos pensar que lo que tenemos que hacerle a nuestro estado cuando creamos una tarea sería lo siguiente:
- I. Algo así → `state.push( new Todo(title) )`
  - II. Pero en **JavaScript** los objetos se pasan por referencia y trabajando de esta manera podemos mutar el estado y entonces, nuestro patrón de diseño dejaría de funcionar como es debido. Por lo tanto, en la captura de pantalla se muestra la forma correcta de hacerlo. En este punto C le estamos diciendo, añade al array de tareas la nueva tarea con título **title**.
  - III. **Con los 3 puntos lo que hace es extraer todos los elementos del array de manera independiente, y junto con el new Todo(title) devuelve un nuevo array, de esta manera nos aseguramos de no mutar el estado. Esta sería la manera de hacerlo.**

Terminemos de configura **ngrx**:



```

app.module.ts X
src > app > app.module.ts > ...
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3
4  import { AppRoutingModule } from './app-routing.module';
5  import { AppComponent } from './app.component';
6  import { TodoModule } from './todos/todo.module';
7  import { StoreModule } from '@ngrx/store';
8  import { todoReducer } from './todos/todo.reducer';
9
10 @NgModule({
11   declarations: [
12     AppComponent
13   ],
14   imports: [
15     BrowserModule,
16     AppRoutingModule,
17     TodoModule,
18     StoreModule.forRoot({todos: todoReducer})
19   ],
20   providers: [],
21   bootstrap: [AppComponent]
22 })
23 export class AppModule { }

```

En el fichero **app.module.ts** deberíamos añadir el código que remarcamos en rojo. Básicamente le estamos indicando como se gestionará nuestro estado indicando el estado y el gestor de las acciones (**reducer**).

Al final del documento, en el apartado de los **effects**, veremos como dejar esto preparado para el caso de que tengamos más de un **reducer**.

Ahora vamos a instalar una herramienta que nos irá muy bien para poder ver los cambios de estado de nuestra aplicación de una forma muy visual. Es una herramienta adrede para inspeccionar el flujo de datos del patrón **Redux**, tenemos que instalar un paquete a nuestro proyecto y una extensión a nuestro navegador (disponible para **Google Chrome** i **Firefox**).

Por lo tanto, instalamos las **Redux dev-tools** para poder depurar y revisar que no mutemos el estado de nuestra aplicación:

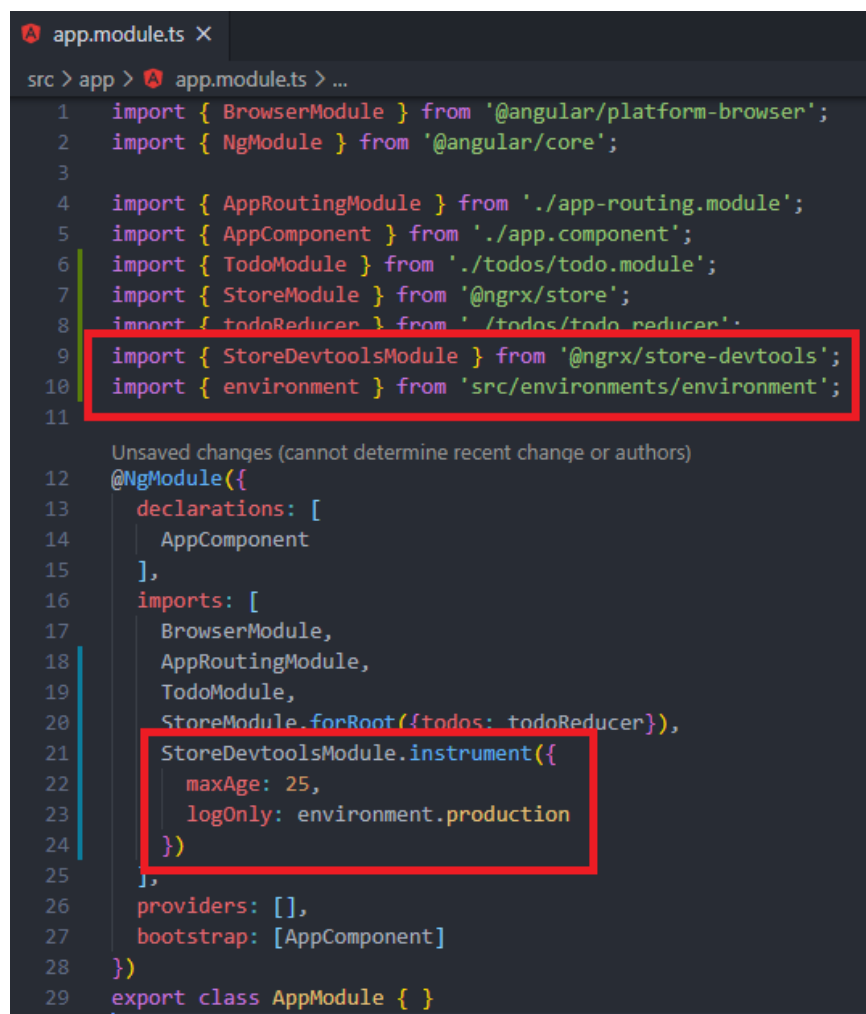
- Instalamos extensión (en nuestro caso para **Chrome**):

<https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbebkcpmknkioeibfkpmmfbljd>

- Instalamos el paquete ejecutando la siguiente instrucción:

```
H:\Projectes\todo-app>npm install @ngrx/store-devtools --save
```

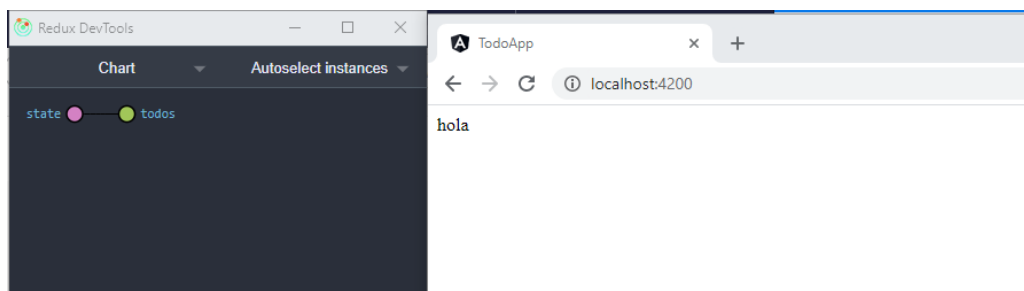
Finalmente hacemos la siguiente configuración:



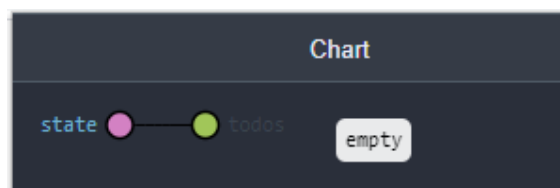
```
app.module.ts X
src > app > app.module.ts > ...
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3
4  import { AppRoutingModule } from './app-routing.module';
5  import { AppComponent } from './app.component';
6  import { TodoModule } from './todos/todo.module';
7  import { StoreModule } from '@ngrx/store';
8  import { todoReducer } from './todos/todo.reducer';
9  import { StoreDevtoolsModule } from '@ngrx/store-devtools';
10 import { environment } from 'src/environments/environment';
11
12  @NgModule({
13    declarations: [
14      AppComponent
15    ],
16    imports: [
17      BrowserModule,
18      AppRoutingModule,
19      TodoModule,
20      StoreModule.forRoot({todos: todoReducer}),
21      StoreDevtoolsModule.instrument({
22        maxAge: 25,
23        logOnly: environment.production
24      })
25    ],
26    providers: [],
27    bootstrap: [AppComponent]
28  })
29  export class AppModule { }
```



Podremos ver el panel en nuestro navegador **Chrome**:



Si situamos el cursor encima de **todos** podremos ver que el estado inicial es un array vacío:

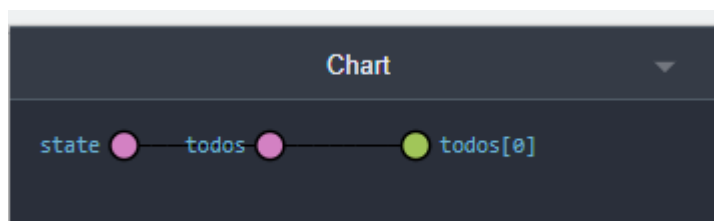


¡Y esto sería correcto!

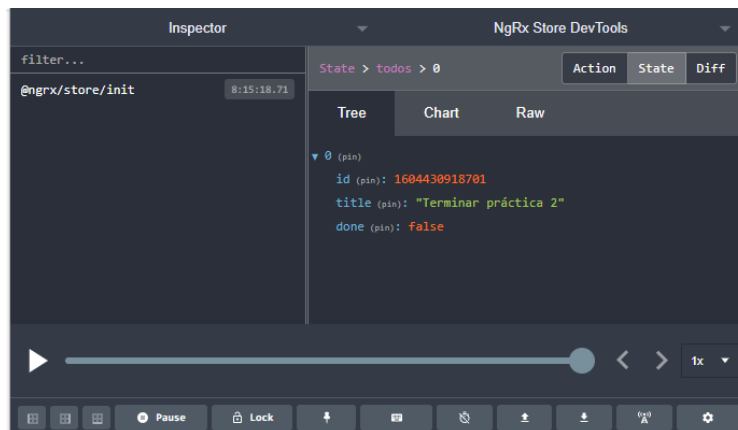
Hagamos una prueba:



Si en nuestro **reducer**, donde tenemos definido nuestro estado inicial, le decimos que inicialmente tenga una tarea que sea 'Terminar práctica 2', vamos a ver cómo arranca la aplicación:



Y si hacemos clic en la tarea:



Podremos ver sus datos. Esto sería correcto. Vamos a dejar el estado inicial tal y como está ahora.

Ahora vamos a definir el **AppState**. Nos creamos el fichero **app.reducer.ts** en la raíz de la carpeta **app**. En este fichero definiremos el estado global de nuestra aplicación. Escribiremos lo siguiente:

```

app.reducer.ts X
src > app > app.reducer.ts > ...
1  import { Todo } from './todos/models/todo.model';
2
3  export interface AppState {
4    todos: Todo[];
5  }
6

```

Tendremos un array de tareas, en principio sencillo. Podríamos tener más elementos si fuera una aplicación más compleja, un array de usuarios, algunas propiedades más, ... Pero para nuestro caso de ejemplo, suficiente.

Vamos ahora a utilizar la creación de una tarea. Nos hará falta importar los formularios reactivos en el **app.module.ts**. También nos hará falta añadir en los **exports** los componentes **TodoAddComponent**, **TodoListComponent** y **TodoListItemComponent**.

```

todo.module.ts X
src > app > todos > todo.module.ts > ...
1  import { NgModule } from '@angular/core';
2  import { CommonModule } from '@angular/common';
3  import { TodoListComponent } from './todo-list/todo-list.component';
4  import { TodoListItemComponent } from './todo-list-item/todo-list-item.component';
5  import { ReactiveFormsModule } from '@angular/forms';
6  import { TodoAddComponent } from './todo-add/todo-add.component';
7
8  @NgModule({
9    declarations: [TodoListComponent, TodoListItemComponent, TodoAddComponent],
10   imports: [CommonModule, ReactiveFormsModule],
11   exports: [TodoListItemComponent, TodoListComponent, TodoAddComponent]
12 })
13 export class TodoModule {}
14

```

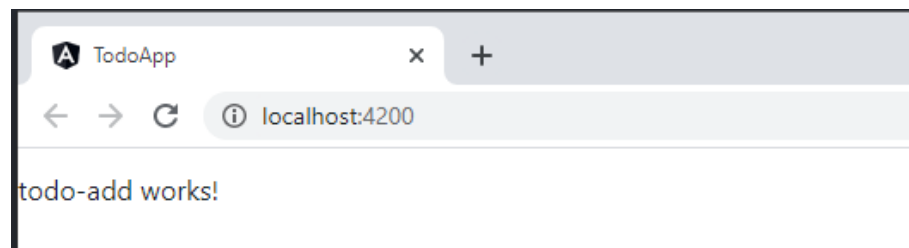
Esto nos permitirá en el **app.component.html** llamar a los componentes **todo-add**, **todo-list** o **todo-list-item** fuera del propio módulo **todo**. Añadimos la siguiente línea al fichero **app.component.html**:

```

app.component.html X
src > app > app.component.html > app-todo-add
1
2
3 | <app-todo-add></app-todo-add>

```

Y en la aplicación deberíamos ver algo así:



Ahora vamos a implementar un **input** para poder insertar el nombre de la tarea y ver como se guarda en el **store** con la acción de crear tarea que hemos implementado en los pasos anteriores. La vista podría tener el siguiente código:

```

todo-add.component.html X
src > app > todos > todo-add > todo-add.component.html > ...
1
2 <div class="container mt-5">
3   <div class="input-group mb-3">
4     <input type="text" class="form-control" placeholder="Insert title for task ..." [formControl]="titleInput" />
5     <div class="input-group-append">
6       <button (click)="addTodoTask()" class="btn btn-success" type="button">ADD TODO TASK</button>
7     </div>
8   </div>
9 </div>
10

```

Aquí podemos ver que definimos dos cosas:

- Un input con la propiedad **titleInput** que gestionaremos en el controlador.
- Un botón que al pulsarse llamará a la función **addTodoTask** y se encargará de llamar a la acción que hemos implementado anteriormente.

Y el controlador:

```

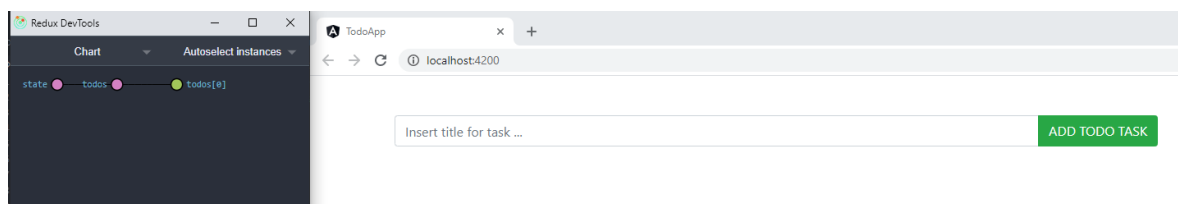
src > app > todos > todo-add > todo-add.component.ts > TodoAddComponent
1 import { Component, OnInit } from '@angular/core';
2 import { FormControl, Validators } from '@angular/forms';
3 import { Store } from '@ngrx/store';
4 import { AppState } from 'src/app/app.reducer';
5 import { createTodo } from '../todo.actions';
6
7 @Component({
8   selector: 'app-todo-add',
9   templateUrl: './todo-add.component.html',
10  styleUrls: ['./todo-add.component.css']
11 })
12 export class TodoAddComponent implements OnInit {
13
14   public titleInput: FormControl;
15
16   constructor(private store: Store<AppState>) {
17     this.titleInput = new FormControl('', Validators.required);
18   }
19
20   ngOnInit(): void {
21   }
22
23   addTodoTask() {
24     if (this.titleInput.valid) {
25       this.store.dispatch(createTodo({ title: this.titleInput.value }));
26       this.titleInput.reset();
27     }
28   }
29 }
30

```

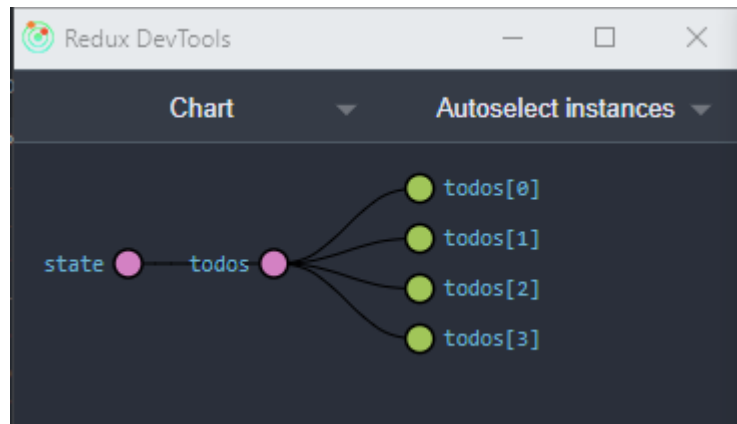
Podemos ver las siguientes cosas:

- Inyectamos el **store** en el constructor para poder utilizarlo.
- En el propio constructor, definimos que el input donde escribiremos el título de la tarea sea requerido y el contenido lo guardaremos en la variable **titleInput**. Nótese que esta variable es la que se vincula con la vista con la cláusula **[formControl]="titleInput"**.
- Finalmente tenemos implementado el método **addTodoTask** que se ejecuta cuando se pulsa el botón de la vista anterior. Lo que tenemos que hacer básicamente es llamar al **dispatch** del **store** para que lance la acción de **createTodo**, y le pasamos como argumento a dicha acción el argumento **title** al que le asignamos el contenido del input (**this.titleInput.value**). Podemos ver que le añadimos una pequeña validación, de manera que se lance la acción sólo si el input es válido, que, en nuestro caso, es cuando éste esté informado.

Si ejecutamos la aplicación podremos ver el estado inicial:



Y podremos ver que si vamos añadiendo tareas se irán añadiendo en el **store** los diferentes **todos**:



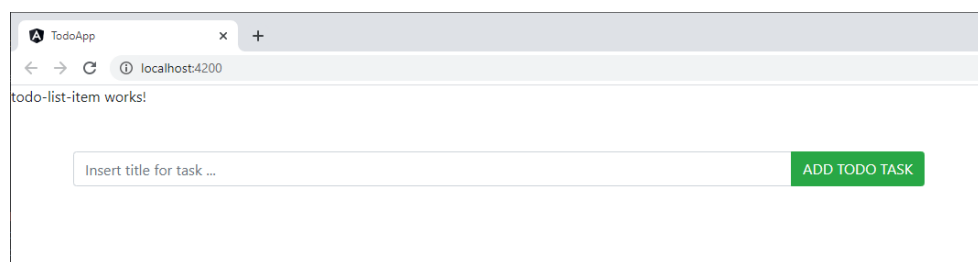
Podemos ir variando las vistas que nos ofrece la extensión para ver la evolución de los datos en el **store**:



Vamos a mostrar ahora el listado de las tareas que vamos añadiendo a nuestra aplicación. En la vista **app.component.html** podemos añadir el componente **app-todo-list**:

```
app.component.html X
src > app > app.component.html > ...
1
2 <app-todo-list></app-todo-list>
3
4 <app-todo-add></app-todo-add>
5
```

Si observamos el navegador deberíamos visualizar algo así:



En el **todo-list-component.html** haremos:

```
todo-list.component.html X
src > app > todos > todo-list > todo-list.component.html > ...
1 <div class="row">
2   <div class="col">
3     <app-todo-list-item [todo]="todo" *ngFor="let todo of todos"></app-todo-list-item>
4   </div>
5 </div>
6
```

Básicamente haremos un bucle del componente **app-todo-list-item** por cada tarea que tengamos en el array de tareas, y este componente **app-todo-list-item** será el que mostrará el detalle de la tarea (aunque solo sea mostrar el título) y los botones de acciones que le podamos hacer a dicha tarea.

Podemos ver también que tenemos la cláusula **[todo]="todo"**, esto nos servirá para mandar la tarea en cuestión al componente **app-todo-list-item** para poder acceder a sus propiedades. Después en su controlador veremos cómo lo recogemos con un **@input**.

Vemos por lo tanto que tenemos un **ngFor** de **todos**, vamos a ver como obtenemos estos **todos**.

En su controlador haremos:

```

src > app > todos > todo-list > todo-list.component.ts > ...
1  import { Component, OnInit } from '@angular/core';
2  import { Store } from '@ngrx/store';
3  import { AppState } from 'src/app/app.reducer';
4  import { Todo } from '../models/todo.model';
5
6  @Component({
7    selector: 'app-todo-list',
8    templateUrl: './todo-list.component.html',
9    styleUrls: ['./todo-list.component.css']
10 })
11 export class TodoListComponent implements OnInit {
12
13   todos: Todo[] = []; (A)
14
15   constructor(private store: Store<AppState>) { } (B)
16
17   ngOnInit(): void {
18     this.store.select('todos').subscribe( todos => this.todos = todos );
19   } (C)
20
21 }

```

- A) Declaramos un array de **todos** local que será el que se recorrerá en la vista que acabamos de comentar.
- B) Inyectamos el **store** en el constructor para poder utilizarlo.
- C) En el **ngOnInit** nos suscribimos al estado de nuestra aplicación para leer el estado actual, concretamente, queremos saber el valor de **todos**, por eso utilizamos la sentencia **select**. Recordemos que en nuestro estado ahora solo tenemos el array de **todos**, pero en una aplicación real tendríamos más propiedades, y normalmente nos interesaría suscribirnos a propiedades concretas en lugar de todo el estado. Por lo tanto, de esta línea de código se asigna el array de **todos** de nuestro estado a la variable **todos** que utilizamos para hacer el **ngFor** en la vista del componente.

Vale, ahora nos vamos a la vista detalle, concretamente a **todo-list-item-component.html** y de momento, haremos:

```

src > app > todos > todo-list-item > todo-list-item.component.html > ...
1  <div class="container mt-2">
2    {{ todo.title }}
3  </div>

```

De momento pongámonos como objetivo mostrar el nombre de la tarea. Esto como se llamará en el **ngFor** del componente **todo-list.component** nos servirá para que aparezca el listado de nombres de las tareas que tengamos en nuestra aplicación.

En su controlador haremos:

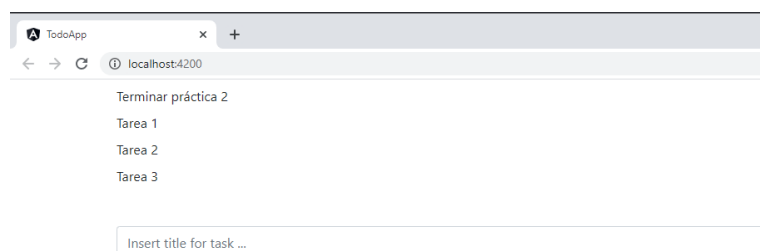
```

1 import { Component, Input, OnInit } from '@angular/core';
2 import { Todo } from '../models/todo.model';
3
4 @Component({
5   selector: 'app-todo-list-item',
6   templateUrl: './todo-list-item.component.html',
7   styleUrls: ['./todo-list-item.component.css'],
8 })
9 export class TodoListItemComponent implements OnInit {
10
11   @Input() todo: Todo;
12
13   constructor() {}
14
15   ngOnInit(): void {}
16
17 }

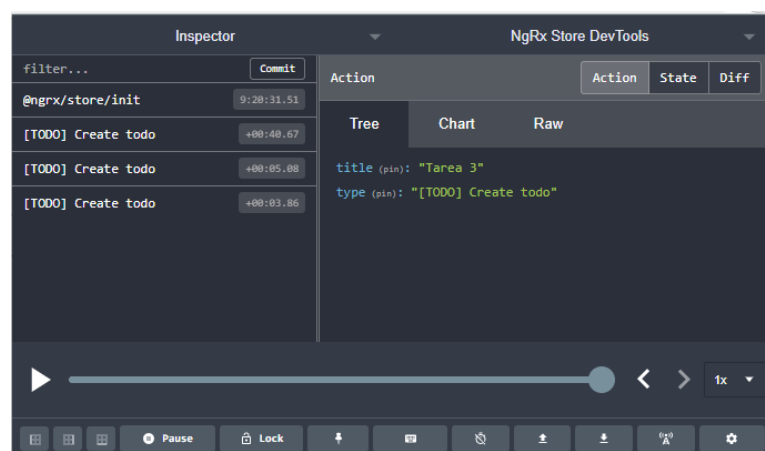
```

Podemos ver que tenemos un **@input** para recoger los datos de la entidad **todo** que corresponda a la iteración determinada del bucle **ngFor** del componente **todo-list.component**.

Si nos vamos al navegador, y añadimos varias tareas, podemos empezar a ver cómo funciona nuestra aplicación, empezaremos a ver como en la parte superior se van mostrando todas las tareas que vamos dando de alta:



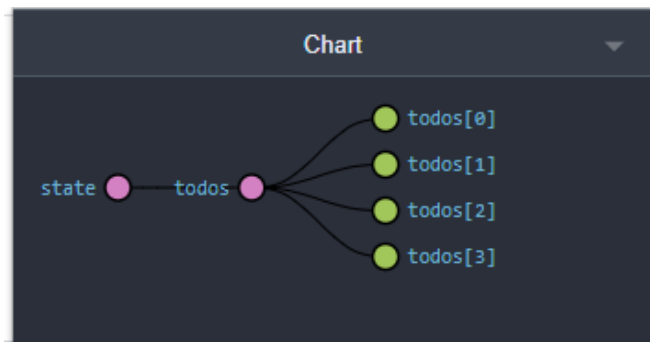
Si observamos la herramienta **Redux dev-tools** podremos ver:



Como aparecen las tres acciones de creación de tarea.



Si ponemos la vista **chart**, podremos ver de forma muy visual cómo va evolucionando nuestro estado de la aplicación:



Vamos a modificar el componente **todo-list-item** para que podamos manejar las acciones de editar, eliminar y completar tareas. Estas acciones las iremos implementando poco a poco, de momento, implementamos la vista.

Pero primero, pensemos en lo que necesitamos, las casuísticas que tendremos serán:

- Si tenemos una tarea creada, se mostraría de la siguiente manera:

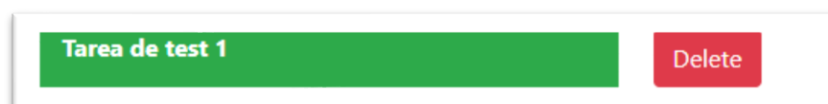


- Cuando hagamos clic en **delete** se eliminará y no se mostrará.
- Cuando hagamos clic en **edit**, veremos lo siguiente:



Podremos modificar el texto y al pulsar **submit** guardaremos el cambio y volveremos a la vista anterior.

- Y si pulsamos el botón **done** veríamos:



Nos quedará la tarea como realizada y la única acción posterior que podremos hacer es eliminar dicha tarea.

Por lo tanto, estamos buscando este comportamiento que iremos desarrollando poco a poco.

Seguidamente, en el fichero **todo-list-item.component.html** haríamos:

```

1  <!-- EDIT mode -->
2  <div class="container mt-2" *ngIf="isEditing">
3
4  <div class="row">
5    <div class="col-4">
6      <input type="text" class="form-control hidden" [formControl]="titleInput" />
7    </div>
8    <div class="col-8">
9      <button (click)="submitTask()" class="btn btn-primary ml-2" type="button">Submit</button>
10    </div>
11  </div>
12 </div>
13
14 <!-- READ mode -->
15 <div class="container mt-2" *ngIf="!isEditing">
16
17 <div class="row">
18   <div class="col-4">
19     {{ todo.title }}
20   </div>
21   <div class="col-8">
22     <button (click)="completeTask()" class="btn btn-success" type="button">Done</button>
23     <button (click)="editTask()" class="btn btn-primary ml-2" type="button">Edit</button>
24     <button (click)="deleteTask()" class="btn btn-danger ml-2" type="button">Delete</button>
25   </div>
26 </div>
27 </div>
28
29

```

Podemos ver que mediante **ngIf** y en función de si la variable **isEditing** es cierta o no, mostraremos un bloque u otro. El bloque de **EDIT mode** tendrá el input para poder modificar el texto de la tarea y el botón de **submitTask** para guardar dicho cambio. El bloque de **READ mode** básicamente mostrará el título de la tarea y los tres botones de acción **completeTask**, **editTask** y **deleteTask**. De momento lo implementamos de esta manera y poco a poco iremos consiguiendo el comportamiento definido anteriormente.

Vamos a ver como implementamos en el controlador, para gestionar el input y la variable lógica **isEditing**. En el fichero **todo-list-item.component.ts** haríamos:

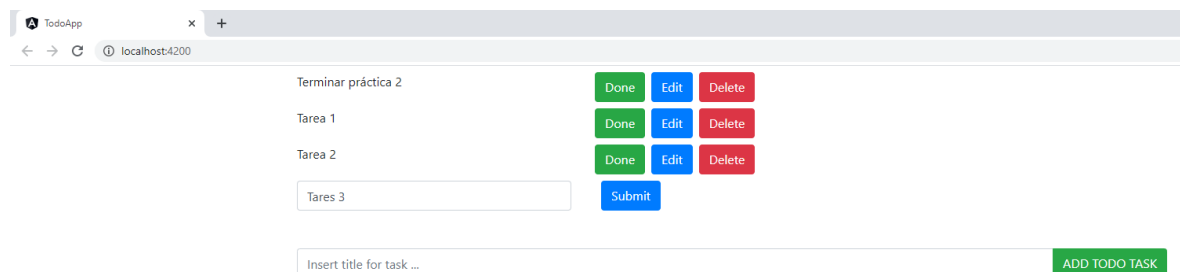
```

1  import { Component, Input, OnInit } from '@angular/core';
2  import { FormControl, Validators } from '@angular/forms';
3  import { Todo } from '../models/todo.model';
4
5  @Component({
6    selector: 'app-todo-list-item',
7    templateUrl: './todo-list-item.component.html',
8    styleUrls: ['./todo-list-item.component.css'],
9  })
10 export class TodolistItemComponent implements OnInit {
11
12   @Input() todo: Todo;
13
14   public titleInput: FormControl; (A)
15   public isEditing: boolean; (B)
16
17   constructor() {}
18
19   ngOnInit(): void {
20
21     this.isEditing = false; (C)
22     this.titleInput = new FormControl(this.todo.title, Validators.required); (D)
23   }
24
25   completeTask(): void{
26
27   }
28
29   editTask(): void{
30     this.isEditing = true; (E)
31   }
32
33   deleteTask(): void{
34   }
35
36   submitTask(): void{
37     this.isEditing = false; (F)
38   }
39 }

```

- A) Definimos la variable del input **titleInput** donde el usuario escribirá el título que quiera modificar cuando estemos en modo edición.
- B) Definimos la variable booleana **isEditing** que nos servirá para mostrar un bloque u otro en la vista y poder discriminar así si estamos editando o no.
- C) Inicialmente definimos el modo edición a falso.
- D) Posteriormente, asignamos al input el valor del título de la tarea que estamos mostrando y le decimos que cuando estemos en modo de edición, este campo es requerido.
- E) Cuando hacemos clic a **editTask**, de momento definimos el modo edición a cierto para que nos modifique la vista, para que nos ponga el input en modo edición y podamos escribir el texto necesario y tengamos disponible el botón de **submitTask** para poder guardar los cambios (Esta funcionalidad de guardar los cambios la implementaremos más adelante).
- F) También nos hará falta cuando se pulse el botón de **submitTask** guardar los cambios que se hayan podido hacer en el título de la tarea (lo haremos después) y asignar la variable **isEditing** a falso para que nos muestre la vista correspondiente, es decir, el bloque de lectura, con los botones de editar, completar y eliminar.

Con esta implementación tendremos lo siguiente:



Podremos ir añadiendo tareas, y éstas se irán mostrando en el listado de la parte superior. Cada tarea tendrá las tres acciones y de momento, si pulsamos editar, se habilitará el input y podríamos modificar el texto, aunque todavía no se guardaría el cambio ya que no tenemos esta funcionalidad implementada. También podemos ver que si estamos en modo edición se habilita el botón de **submit** que guardaría los cambios, y si lo pulsáramos volveríamos al modo lectura inicial.

Por lo tanto, la lógica inicial para manejar este comportamiento a priori lo tendríamos.

Vamos a por las acciones que nos hacen falta.

**Acción COMPLETE:** Implementemos la finalización de una tarea (dar por echa una tarea).

**Objetivo:** definir a **true** el campo **done** de la tarea que queremos finalizar o dar por echa.

Primero nos crearemos la acción, vamos a **todo.actions.ts** :

```

1  import { createAction, props } from '@ngrx/store';
2
3  export const createTodo = createAction(
4    '[TODO] Create todo',
5    props<{ title: string }>()
6  );
7
8  export const completeTodo = createAction(
9    '[TODO] Complete todo',
10   props<{ id: number }>()
11 );
12

```

Crearemos la acción y lo que necesitaremos para poder modificar el campo **done** a **true** de la tarea en cuestión será el **id** para poder identificar que tarea es. Después tendremos que implementar el **reducer**, por lo tanto iremos a **todo.reducer.ts** y haremos:

```

1  import { createReducer, on } from '@ngrx/store';
2  import { Todo } from '../models/todo.model';
3  import { completeTodo, createTodo } from '../todo.actions';
4
5  export const initialState: Todo[] = [new Todo('Terminar práctica 2')];
6
7  const _todoReducer = createReducer(
8    initialState,
9    on(createTodo, (state, { title }) => [...state, new Todo(title)]),
10   on(completeTodo, (state, { id }) => {
11     return state.map((todo) => {
12       if (todo.id === id) {
13         return {
14           ...todo,
15           done: true,
16         };
17       } else {
18         return todo;
19       }
20     });
21   });
22 );
23
24 export function todoReducer(state, action) {
25   return _todoReducer(state, action);
26 }
27

```

Aquí conceptualmente sabemos que lo que tendríamos que hacer es recorrer el array de **todos**, buscar cual tiene el **id** que pasamos por parámetro y asignarle la propiedad **done** a **true**. Pero de la misma manera que para la acción de crear una tarea no podemos hacer un **push** de **JavaScript** para evitar mutar el estado, y lo hicimos como es debido, aquí tampoco podemos manipular directamente el array, por lo tanto, lo haríamos de la siguiente manera para asegurarnos que devolvemos un nuevo array (estado).

Al hacer un **state.map**, la propiedad **map** (si leemos su descripción) nos dice que devuelve un **nuevo** array que contiene los resultados. Por lo tanto, con esto nos aseguramos de devolver un nuevo estado sin mutarlo.

¿Qué hace el código en rojo?

Es decir, el **map** devuelve todos los elementos del array en un nuevo array, por cada elemento del array, es decir, por cada **todo**, mira si coincide el **id**. Si no coincide, a ese **todo** no tenemos que hacerle nada, ya que no es el que queremos dar por **done**, por lo tanto, lo devolvemos tal cual al nuevo array. En cambio, si coincide el **id**, se trata de la tarea la cual, si queremos completar, por lo tanto, tenemos que hacer dos cosas, devolver todas sus propiedades que no son la **done** (instrucción **...todo**) que en este caso solo son **id** y **title**, y poner a **true** la propiedad **done**.

Con esto tendríamos el **reducer** implementado. Lo importante es destacar lo que hacemos para asegurarnos que devolvemos un estado nuevo, con **map** y devolviendo las propiedades del elemento **todo** con los tres puntos.

Vamos a utilizar esta acción. Tendremos que implementarla en el botón **Done**:

```

src > app > todos > todo-list-item > todo-list-item.component.ts > ...
1  import { Component, Input, OnInit } from '@angular/core';
2  import { FormControl, Validators } from '@angular/forms';
3  import { Store } from '@ngrx/store';
4  import { AppState } from 'src/app/app.reducer';
5  import { Todo } from '../models/todo.model';
6  import { completeTodo } from '../todo.actions';
7
8  @Component({
9    selector: 'app-todo-list-item',
10   templateUrl: './todo-list-item.component.html',
11   styleUrls: ['./todo-list-item.component.css'],
12 })
13 export class TodoListItemComponent implements OnInit {
14   @Input() todo: Todo;
15
16   public titleInput: FormControl;
17   public isEditing: boolean;
18
19   constructor(private store: Store<AppState>) {}
20
21   ngOnInit(): void {
22     this.isEditing = false;
23     this.titleInput = new FormControl(this.todo.title, Validators.required);
24   }
25
26   completeTask(): void {
27     this.store.dispatch(completeTodo({ id: this.todo.id }));
28   }
29
30   editTask(): void {
31     this.isEditing = true;
32   }
33
34   deleteTask(): void {}
35
36   submitTask(): void {
37     this.isEditing = false;
38   }
39 }

```

Llamamos a la acción y le pasamos el **id** del **todo** que estamos editando. Validemos que esto funciona. Ejecutamos la aplicación tal cual, recordemos que inicializamos el estado con una tarea que se llama 'Terminar práctica 2'. El estado inicial es:

Terminar práctica 2

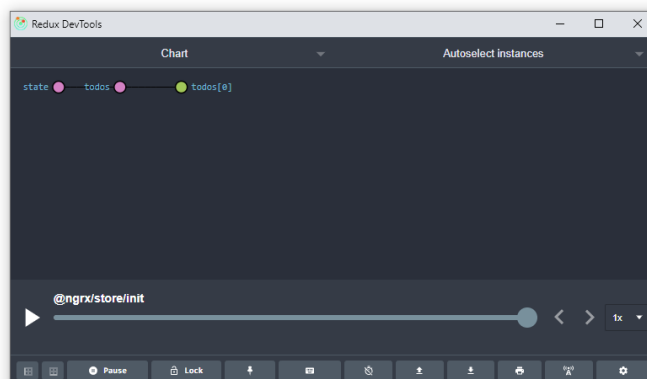
Done

Edit

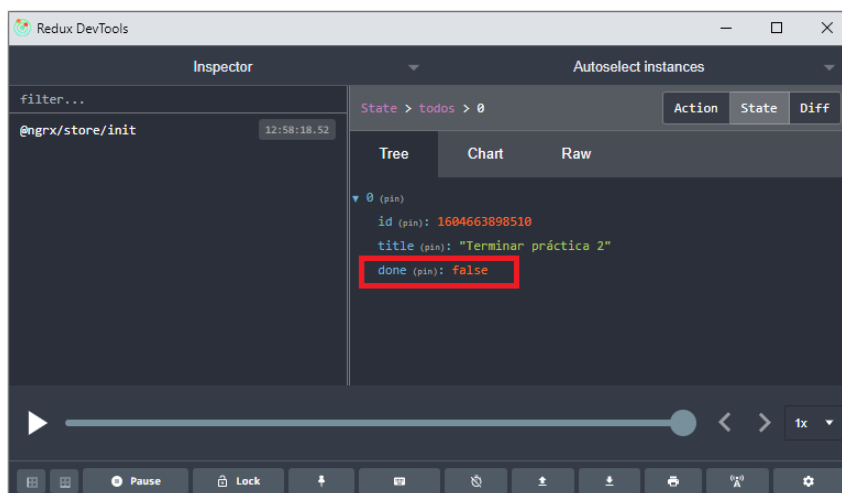
Delete

Insert title for task ...

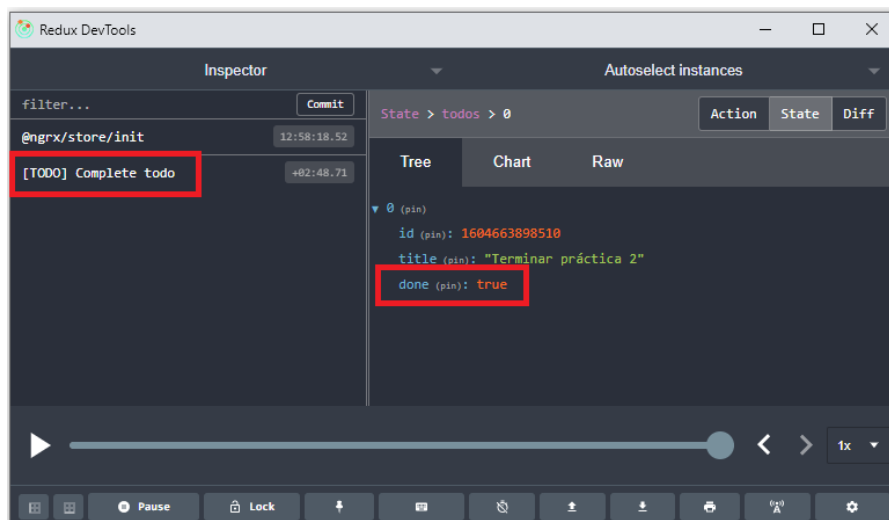
ADD TODO TASK



Si vemos las propiedades del estado inicial:

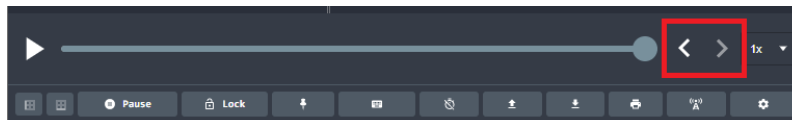


Vemos que la tarea no está completada, lo cual es correcto, es como está definida inicialmente en el **inicialState** del **todo.reducer.ts**. Y si le damos clic a **Done** y volvemos a inspeccionar podemos ver que:



Se ha ejecutado la acción de completar y se ha actualizado el campo **done** a **true**.

Una funcionalidad muy útil de esta herramienta es que podemos volver a estados anteriores, si hacemos clic en



Podemos ir hacia atrás y hacia adelante para ver cómo van evolucionando los cambios en nuestro estado de la aplicación.

Para terminar de validar esta implementación, añadamos varias tareas y hagamos clic en **done** en alguna de ellas e inspeccionemos la herramienta **Redux dev-tools** para asegurarnos. En principio tendríamos una implementación correcta.

Ahora podríamos hacer, simplemente por mejorar un poco la funcionalidad de la aplicación, que si la tarea esta completada, que lo sepamos visualmente, por ejemplo, cambiando el color de fondo de la propia tarea, y también dejaremos solo la acción de eliminar activa. ¡Vamos a ello!

En el fichero de estilos del proyecto nos definimos la siguiente clase:

```
styles.css
src > styles.css > ...
1  /* You can add global styles to this f
2
3  .completed {
4    background-color: #28a745;
5    font-weight: bold;
6    color: white;
7  }
```

Simplemente nos creamos una clase que asignaremos cuando una tarea esté completa, de manera que nos ponga el fondo verde, el texto en negrita y de color blanco. En la vista del componente **todo-list-item.component.html** añadiremos lo siguiente:

```
todo-list-item.component.html
src > app > todos > todo-list-item > todo-list-item.component.html > ...
4  <div class="container mt-2" *ngIf="isEditing">
5    <div class="row">
6      <div class="col-4">
7        <input type="text" class="form-control hidden" [formControl]="titleInput" />
8      </div>
9      <div class="col-8">
10       <button (click)="submitTask()" class="btn btn-primary ml-2" type="button">Submit</button>
11     </div>
12   </div>
13 </div>
14
15 <!-- READ mode -->
16
17 <div class="container mt-2" *ngIf="!isEditing">
18   <div class="row">
19     <div class="col-4" [class.completed]="todo.done">
20       {{ todo.title }}
21     </div>
22     <div class="col-8">
23       <button (click)="completeTask()" class="btn btn-success" type="button" *ngIf="!todo.done">Done</button>
24       <button (click)="editTask()" class="btn btn-primary ml-2" type="button" *ngIf="!todo.done">Edit</button>
25       <button (click)="deleteTask()" class="btn btn-danger ml-2" type="button">Delete</button>
26     </div>
27   </div>
28 </div>
29
30 <hr>
31 <hr>
32
```

Realmente esto se puede hacer de muchas maneras, pero vamos a hacerlo lo más sencillo posible, y en nuestro caso, como la misma propiedad **done** es una variable booleana, la podemos aprovechar para mostrar u ocultar los elementos en la vista.

Aquí lo que estamos diciendo es, si el estado es completado aplicas la clase **completed**, que simplemente pone un fondo verde, y el título de la tarea en negrita y de color blanco. De la misma manera, si la tarea esta completa, ocultamos los botones de editar y completar. La única acción que podremos hacer una vez una tarea esté completada será eliminarla. ¡Sería momento de validar lo que acabamos de implementar!

### Acción EDIT: Implementemos la edición del título de una tarea.

Nueva acción:

```

1  import { createAction, props } from '@ngrx/store';
2
3  export const createTodo = createAction(
4    '[TODO] Create todo',
5    props<{ title: string }>()
6  );
7
8  export const completeTodo = createAction(
9    '[TODO] Complete todo',
10   props<{ id: number }>()
11 );
12
13 export const editTodo = createAction(
14   '[TODO] Edit todo',
15   props<{ id: number, title: string }>()
16 );
17

```

La acción es muy parecida a la acción de completar, pero en este caso, necesitaremos pasarle por argumento el **id** para identificar qué tarea es y el texto del título que queramos modificar.

En el **reducer**:

```

1  import { createReducer, on } from '@ngrx/store';
2  import { Todo } from '../models/todo.model';
3  import { completeTodo, createTodo, editTodo } from '../todo.actions';
4
5  export const initialState: Todo[] = [new Todo('Terminar práctica 2')];
6
7  const _todoReducer = createReducer(
8    initialState,
9    on(createTodo, (state, { title }) => [...state, new Todo(title)]),
10   on(completeTodo, (state, { id }) => {
11     return state.map((todo) => {
12       if (todo.id === id) {
13         return {
14           ...todo,
15           done: true,
16         };
17       } else {
18         return todo;
19       }
20     });
21   }),
22   on(editTodo, (state, { id, title }) => {
23     return state.map((todo) => {
24       if (todo.id === id) {
25         return {
26           ...todo,
27           title: title,
28         };
29       } else {
30         return todo;
31       }
32     });
33   })
34 );
35
36 export function todoReducer(state, action) {
37   return _todoReducer(state, action);
38 }

```



Podemos observar que la implementación es prácticamente la misma que en la acción de completar, pero en este caso modificamos la propiedad **title** que recibiremos como argumento.

Nos vamos al componente **todo-list-item.component.ts** y haremos:

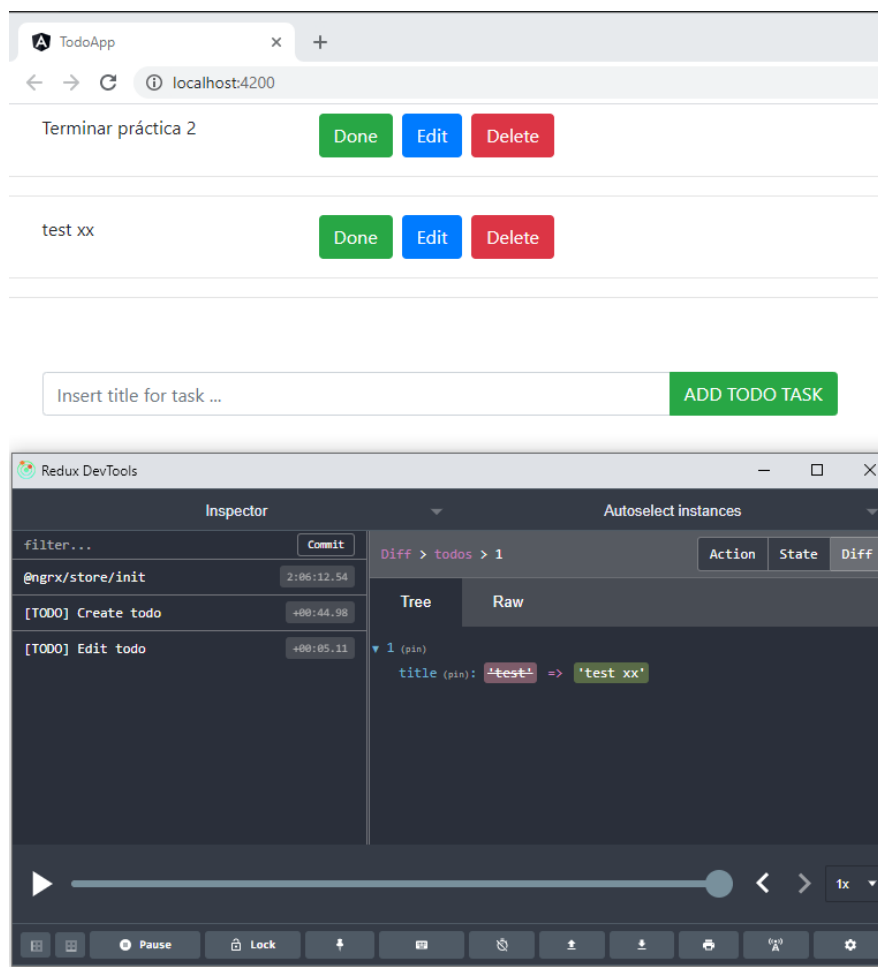
```

1  todo-list-item.component.ts X
src > app > todos > todo-list-item > todo-list-item.component.ts > ...
2  import { FormControl, Validators } from '@angular/forms';
3  import { Store } from '@ngrx/store';
4  import { AppState } from 'src/app/app.reducer';
5  import { Todo } from '../models/todo.model';
6  import { completeTodo, editTodo } from '../todo.actions';
7
8  @Component({
9    selector: 'app-todo-list-item',
10   templateUrl: './todo-list-item.component.html',
11   styleUrls: ['./todo-list-item.component.css'],
12 })
13 export class TodoListItemComponent implements OnInit {
14   @Input() todo: Todo;
15
16   public titleInput: FormControl;
17   public isEditing: boolean;
18
19   constructor(private store: Store<AppState>) {}
20
21   ngOnInit(): void {
22     this.isEditing = false;
23     this.titleInput = new FormControl(this.todo.title, Validators.required);
24   }
25
26   completeTask(): void {
27     this.store.dispatch(completeTodo({ id: this.todo.id }));
28   }
29
30   editTask(): void {
31     this.isEditing = true;
32   }
33
34   deleteTask(): void {}
35
36   submitTask(): void {
37     this.isEditing = false;
38
39     if (!this.titleInput.invalid && this.titleInput.value !== this.todo.title) {
40       this.store.dispatch(
41         editTodo({ id: this.todo.id, title: this.titleInput.value })
42       );
43     }
44   }
45 }

```

Cuando editamos una tarea y pulsamos **submit**, ejecutaremos el nuevo código resaltado en rojo. La acción es muy parecida a la acción de completar, en este caso de editar lo que hacemos es modificar el campo **title** de la tarea. Lo que podemos hacer es que actualice dicho caso si el input pasa la validación que le hemos puesto, en este caso, simplemente es que esté el campo informado, y también podemos añadir que lance la acción si se modifica el título, ya que, si entramos en modo edición, pero por lo que sea no se modifica el texto, no tiene sentido llamar al **store**.

Si añadimos una tarea **test** y luego la editamos y le añadimos **xx** podemos ver el cambio de estado correcto.



Podemos validar que, si pulsamos editar y no cambiamos el texto, al pulsar **submit** no se lanza ninguna acción del **store**. También podemos validar que, si pulsamos editar y eliminamos el contenido del input, y pulsamos **submit** no se lanza ninguna acción. Lo que si podemos ver es que, en este caso, si volvemos a editar la misma tarea, ésta a 'perdido' el título cuando estamos editando, esto lo solucionamos simplemente añadiendo esta línea en el método **editTask** del fichero **todo-list-item.component.ts**:

```
editTask(): void {
  this.isEditing = true;
  this.titleInput.setValue( this.todo.title );
}
```

## Acción DELETE: Implementamos la eliminación de una tarea.

Nueva acción:

```

src > app > todos > todo.actions.ts > deleteTodo
1  import { createAction, props } from '@ngrx/store';
2
3  export const createTodo = createAction(
4    '[TODO] Create todo',
5    props<{ title: string }>()
6  );
7
8  export const completeTodo = createAction(
9    '[TODO] Complete todo',
10   props<{ id: number }>()
11 );
12
13 export const editTodo = createAction(
14   '[TODO] Edit todo',
15   props<{ id: number, title: string }>()
16 );
17
18 export const deleteTodo = createAction(
19   '[TODO] Delete todo',
20   props<{ id: number }>()
21 );
22

```

En el reducer:

```

src > app > todos > todo.reducer.ts > ...
1  import { createReducer, on } from '@ngrx/store';
2  import { Todo } from '../models/todo.model';
3  import { completeTodo, createTodo, deleteTodo, editTodo } from './todo.actions';
4
5  export const initialState: Todo[] = [new Todo('Terminar práctica 2')];
6
7  const _todoReducer = createReducer(
8    initialState,
9    on(createTodo, (state, { title }) => [...state, new Todo(title)]),
10   on(completeTodo, (state, { id }) => {
11     return state.map((todo) => {
12       if (todo.id === id) {
13         return {
14           ...todo,
15           done: true,
16         };
17       } else {
18         return todo;
19       }
20     });
21   }),
22   on(editTodo, (state, { id, title }) => {
23     return state.map((todo) => {
24       if (todo.id === id) {
25         return {
26           ...todo,
27           title: title,
28         };
29       } else {
30         return todo;
31       }
32     });
33   }),
34   on(deleteTodo, (state, { id }) => state.filter( todo => todo.id !== id ))
35 );
36
37 export function todoReducer(state, action) {
38   return _todoReducer(state, action);
39 }

```

De manera semejante a los casos anteriores, tenemos que asegurarnos de no mutar el estado. En este caso podemos utilizar la función **filter** la cual devuelve un **nuevo array** de elementos, por lo tanto, podemos utilizarla para devolver todos los elementos menos el que tenga el **id** que le pasamos por parámetro, de esta manera sería como si lo elimináramos.

Utilizamos la acción de eliminar en el controlador:

```

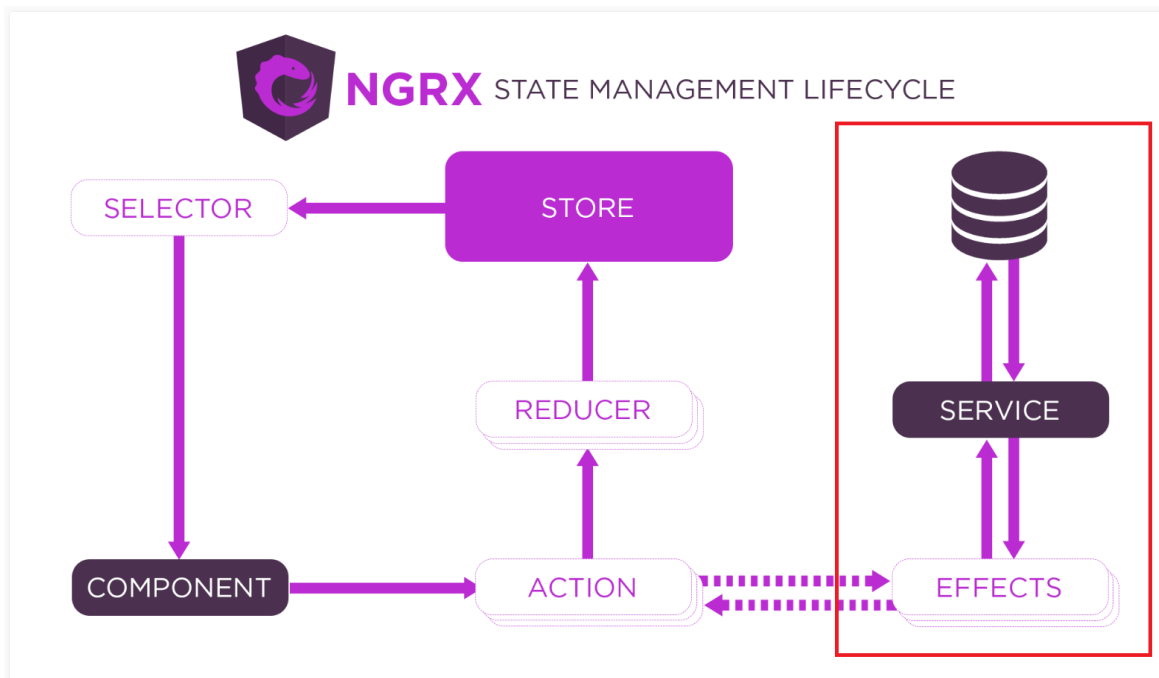
15
16 public titleInput: FormControl;
17 public isEditing: boolean;
18
19 constructor(private store: Store<AppState>) {}
20
21 ngOnInit(): void {
22   this.isEditing = false;
23   this.titleInput = new FormControl(this.todo.title, Validators.required);
24 }
25
26 completeTask(): void {
27   this.store.dispatch(completeTodo({ id: this.todo.id }));
28 }
29
30 editTask(): void {
31   this.isEditing = true;
32   this.titleInput.setValue( this.todo.title );
33 }
34
35 deleteTask(): void {
36   this.store.dispatch( deleteTodo({id: this.todo.id}) );
37 }
38
39 submitTask(): void {
40   this.isEditing = false;
41
42   if (!this.titleInput.invalid && this.titleInput.value !== this.todo.title) {
43     this.store.dispatch(
44       editTodo({ id: this.todo.id, title: this.titleInput.value })
45     );
46   }
47 }
48

```

Podemos validar que funciona correctamente con el inspector **Redux dev-tools**.

Llegados a este punto, ¡ya tendríamos la aplicación resuelta!

## EFFECTS



Pensemos que tenemos acciones, estas acciones se gestionan a través del **reducer** y se hace la lógica que sea. Vale, pongamos ahora que determinadas acciones queremos que llamen a un **backend** para recuperar datos o para hacer actualizaciones o la tarea que sea. Pues las acciones que a nosotros/as nos interese pueden ser escuchadas por un efecto, y éste reaccionará de la manera que nosotros/as le indiquemos. Por ejemplo, llamando a otra acción, o normalmente, llamando a un servicio de **Angular** que lo podremos utilizar, por ejemplo, para hacer las peticiones **http** a un servidor.

Cuando el efecto termina su tarea, notifica a la acción, y dicha acción hace las tareas determinadas.

Por lo tanto, podemos decir que:

- Los efectos escuchan acciones que son disparadas por el **store**.
- Simplifican la lógica de los componentes (controladores).
- Son los encargados normalmente de comunicarse con el **backend** mediante peticiones **http** a través de servicios de **Angular**.

Ya dijimos que las acciones tienen que resolverse con los argumentos que reciben y nunca deben utilizar datos de terceros. De esta manera, se encapsula la gestión de la información y son los efectos que al escuchar una determinada acción se conectan por ejemplo a un servicio **Rest** para recuperar ciertos datos o para modificarlos.

Por lo tanto, estamos acostumbrados a que los componentes interactúen con terceros a través de los servicios y con los efectos tendremos una herramienta que nos ayudará a que éstos interactúen con terceros mediante los servicios y esto nos permitirá aislarlos de los componentes. Esto nos dejara unos componentes (controladores) con un código más limpio y mantenible.

Ahora haremos un ejemplo de como estamos acostumbrados a trabajar y cómo lo deberíamos hacer si trabajamos con el **store** y los efectos siguiendo la aplicación que estábamos desarrollando.

Pensemos en la manera como trabajamos normalmente. Pongamos un ejemplo.

Suponiendo que continuamos con la aplicación anterior, y queremos que, al iniciar la aplicación, se haga una petición al **backend** para cargar la lista de tareas. ¿Como lo hacemos esto?

Por ejemplo, en el **ngOnInit** del **todo-list.component.ts** nos suscribiríamos a un servicio y éste haría la petición **http.get** para recuperar el array de **todos**.

¡Vamos a implementarlo!

Creámonos primero un **MockObject** que nos servirá como base de datos. Dentro de la carpeta **assets** nos podemos crear una carpeta **mocks** y dentro de ésta podemos crearnos un fichero **todos.json**. En este fichero podemos crearnos por ejemplo unas diez tareas tal que así:

```

{
  "todos": [
    {
      "id": 1,
      "title": "Task 1",
      "done": true
    },
    {
      "id": 2,
      "title": "Task 2",
      "done": true
    },
    {
      "id": 3,
      "title": "Task 3",
      "done": true
    },
    {
      "id": 4,
      "title": "Task 4",
      "done": true
    },
    {
      "id": 5,
      "title": "Task 5",
      "done": true
    },
    {
      "id": 6,
      "title": "Task 6",
      "done": true
    },
    {
      "id": 7,
      "title": "Task 7",
      "done": false
    },
    {
      "id": 8,
      "title": "Task 8",
      "done": false
    },
    {
      "id": 9,
      "title": "Task 9",
      "done": false
    },
    {
      "id": 10,
      "title": "Task 10",
      "done": false
    }
  ]
}

```

Ahora vamos a crearnos el servicio para simular las peticiones **http** al **backend** para recuperar los datos de los **todos**, que directamente los cogeremos del **mockobject** creado en el paso anterior. Nos podemos crear una carpeta **services** dentro de la carpeta **todos** y posteriormente ejecutamos la siguiente instrucción:

```
H:\Projectes\todo-app>ng g s todos/services/todo --flat
CREATE src/app/todos/services/todo.service.spec.ts (347 bytes)
CREATE src/app/todos/services/todo.service.ts (133 bytes)
```

Antes de implementar el servicio, como haremos peticiones **http**, necesitamos importar el módulo:

```
todo.module.ts X
src > app > todos > todo.module.ts > ...
1 import { NgModule } from '@angular/core';
2 import { CommonModule } from '@angular/common';
3 import { TodoListComponent } from '../todo-list/todo-list.component';
4 import { TodoListItemComponent } from '../todo-list-item/todo-list-item.component';
5 import { ReactiveFormsModule } from '@angular/forms';
6 import { TodoAddComponent } from '../todo-add/todo-add.component';
7 import { HttpClientModule } from '@angular/common/http';
8
9 @NgModule({
10   declarations: [TodoListComponent, TodoListItemComponent, TodoAddComponent],
11   imports: [CommonModule, ReactiveFormsModule, HttpClientModule],
12   exports: [TodoListItemComponent, TodoListComponent, TodoAddComponent]
13 })
14 export class TodoModule {}
15
```

En el servicio vamos a implementar la petición para recuperar todas las tareas:

```
todo.service.ts X
src > app > todos > services > todo.service.ts > ...
1 import { Injectable } from '@angular/core';
2 import { HttpClient } from '@angular/common/http';
3 import { Todo } from '../models/todo.model';
4 import { Observable } from 'rxjs';
5
6 @Injectable({
7   providedIn: 'root',
8 })
9 export class TodoService {
10   constructor(private http: HttpClient) {}
11
12   getAllTodos(): Observable<Todo[]> {
13     return Object.assign(this.http.get('../assets/mocks/todos.json'));
14   }
15 }
16
```

Esta sería la típica llamada **get**, pero en lugar de pasarle una **url** de una **api** le pasamos directamente la ruta de nuestro **MockObject** local.

En el controlador:

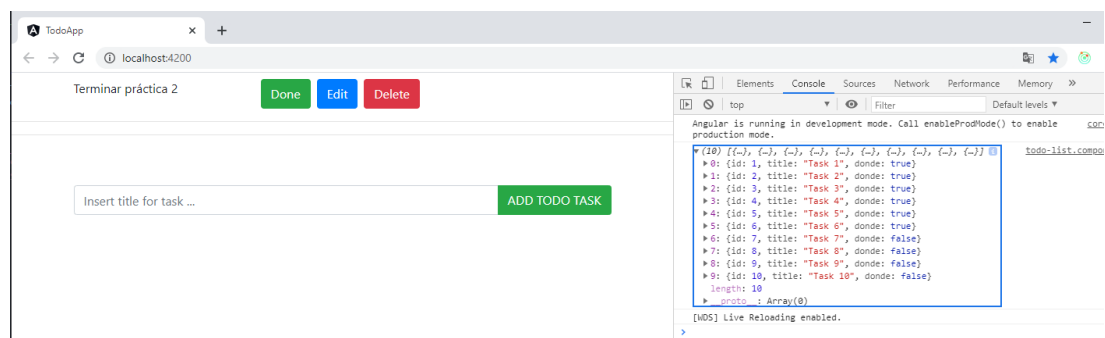
```

1  import { Component, OnInit } from '@angular/core';
2  import { Store } from '@ngrx/store';
3  import { AppState } from 'src/app/app.reducer';
4  import { Todo } from '../models/todo.model';
5  import { TodoService } from '../services/todo.service';
6
7  @Component({
8    selector: 'app-todo-list',
9    templateUrl: './todo-list.component.html',
10   styleUrls: ['./todo-list.component.css'],
11 })
12 export class TodoListComponent implements OnInit {
13   todos: Todo[] = [];
14
15   constructor(
16     private store: Store<AppState>,
17     private todoService: TodoService
18   ) {}
19
20   ngOnInit(): void {
21     this.store.select('todos').subscribe((todos) => (this.todos = todos));
22     this.todoService.getAllTodos().subscribe((todos) => console.log(todos));
23   }
24 }

```

Inyectaríamos el servicio en el constructor y en el **ngOnInit** llamamos al servicio y nos suscribimos, y simplemente mostramos la respuesta (listado de tareas) por consola.

Y vemos en la consola del navegador como recibimos los datos:



Si le asignáramos directamente la respuesta de este servicio a la variable **todos** que se utiliza para mostrar la lista de **todos** en la vista, y quitamos provisionalmente la recuperación de este dato del **store**, tendríamos:

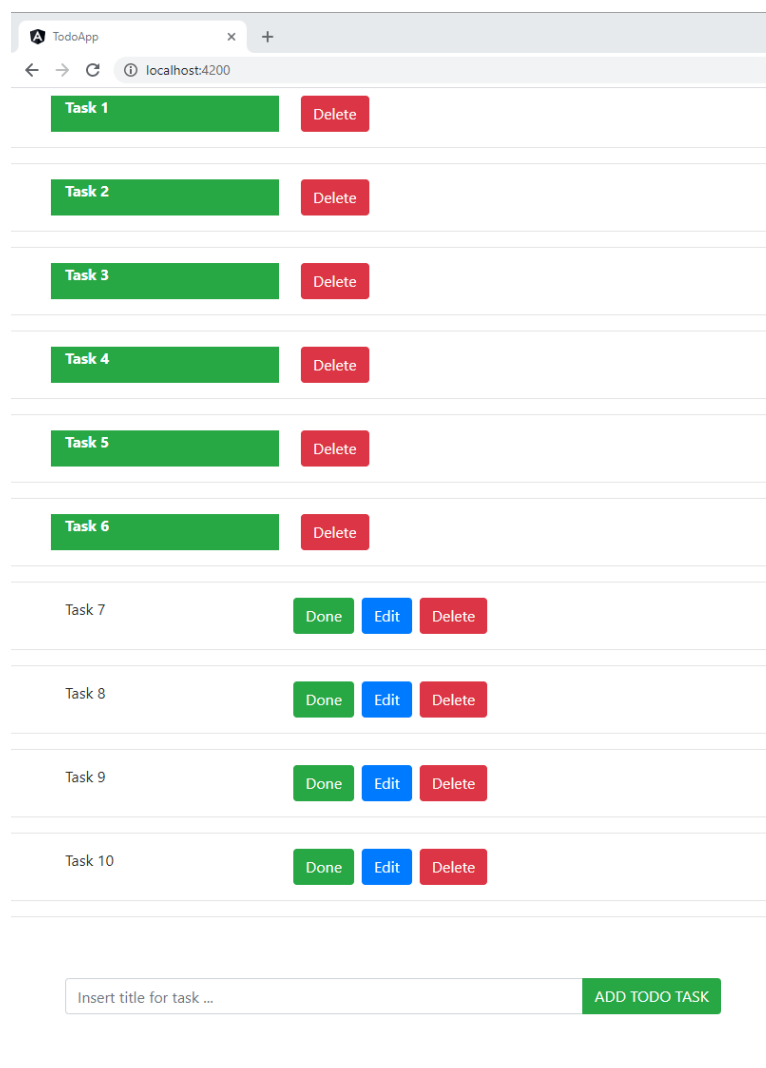


```

src > app > todos > todo-list > todo-list.component.ts > ...
2 import { Store } from '@ngrx/store';
3 import { AppState } from 'src/app/app.reducer';
4 import { Todo } from '../models/todo.model';
5 import { TodoService } from '../services/todo.service';
6
7 @Component({
8   selector: 'app-todo-list',
9   templateUrl: './todo-list.component.html',
10  styleUrls: ['./todo-list.component.css'],
11 })
12 export class TodoListComponent implements OnInit {
13   todos: Todo[] = [];
14
15   constructor(
16     // private store: Store<AppState>
17     private todoService: TodoService
18   ) {}
19
20   ngOnInit(): void {
21     // this.store.select('todos').subscribe((todos) => (this.todos = todos));
22     this.todoService.getAllTodos().subscribe((todos) => this.todos = todos);
23   }
24 }
25

```

Y en la vista:



Podremos ver cómo se cargan todas las tareas. Evidentemente, no nos funcionarían las acciones ya que no se carga el estado inicial que recordemos que teníamos una sola tarea 'Terminar práctica 2' y aunque hiciéramos acciones, no las veríamos en el listado de tareas ya que hemos comentado el código donde nos suscribimos al **store**.

Esto simplemente ha sido un ejemplo de un servicio que hace una petición **http** y cómo nos suscribimos al controlador y mostramos los resultados a la vista.

### Vamos ahora a implementarlo con los *effects*.

Lo que vamos a hacer ahora es cambiar esta manera de recuperar los datos a través de efectos, para inicializar el estado inicial de nuestra aplicación con estas diez tareas.

El objetivo será llamar a **getAllTodos** con los **effects** y establecer el estado inicial de la aplicación y poder trabajar con normalidad tal y como lo teníamos ahora.

¡Vamos a ello!

Añadimos las acciones que vamos a necesitar para llamar a **getAllTodos**.

```

src > app > todos > todo.actions.ts > ...
1  import { createAction, props } from '@ngrx/store';
2  import { Todo } from '../models/todo.model';
3
4  export const createTodo = createAction(
5    '[TODO] Create todo',
6    props<{ title: string }>()
7  );
8
9  export const completeTodo = createAction(
10   '[TODO] Complete todo',
11   props<{ id: number }>()
12 );
13
14 export const editTodo = createAction(
15   '[TODO] Edit todo',
16   props<{ id: number, title: string }>()
17 );
18
19 export const deleteTodo = createAction(
20   '[TODO] Delete todo',
21   props<{ id: number }>()
22 );
23
24 export const getAllTodos = createAction('[TODO] Get all'); (A)
25
26 export const getAllTodosSuccess = createAction( (B)
27   '[TODO] Get all success',
28   props<{ todos: Todo[] }>()
29 );
30
31 export const getAllTodosError = createAction( (C)
32   '[TODO] Get all error',
33   props<{ payload: any }>()
34 );
35

```

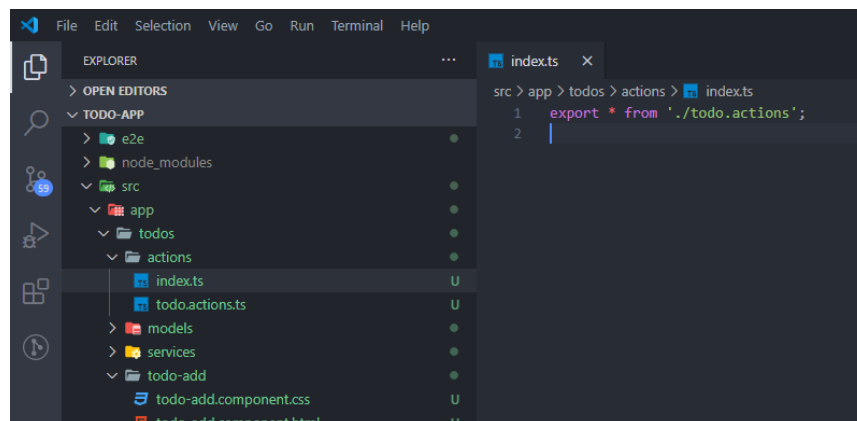
- A) Acción para indicar al sistema que necesitamos empezar a cargar el listado de **todos**.
- B) Acción para indicar que se cargaron los **todos** de forma correcta. Esta acción recibirá el array de **todos** cuando la petición al **backend** se realice correctamente.
- C) Acción para indicar que se ha producido algún error al hacer la petición del listado de **todos**. De momento le pasamos un **payload** como argumento y posteriormente veremos cómo lo manejamos.

A continuación, veremos que iremos reestructurando algunos directorios y ficheros para que nuestro proyecto se asemeje más a un proyecto real.

Nos crearemos la carpeta **actions** dentro de la carpeta **todos** y moveremos el fichero **todo.actions.ts** que acabamos de editar dentro.

En principio nos actualizará los **imports** donde se estuvieran utilizando las acciones, pero no estaría de más que nos aseguraremos de que todo sigue funcionando.

Ahora nos creamos un fichero **index.ts** dentro de la carpeta **actions** y pondremos lo siguiente:



Este fichero será el encargado de exponer todas las acciones y de esta manera sólo tendremos una sola exportación.

Ahora haremos algunos cambios más.

Recordemos que teníamos en el fichero **app.reducer.ts** la definición del **AppState**. Lo que haremos es definir un **TodoState** dentro del **todo.reducer.ts** y en el fichero **app.reducer.ts** en el **AppState** utilizaremos este nuevo **TodoState**, de manera que el **AppState** del fichero **app.reducer.ts** será donde centralizaremos los diferentes estados que podríamos tener.

En el **reducer** tendremos que añadir las nuevas acciones que hemos creado anteriormente que nos servirán para cargar inicialmente todas las tareas. Antes de empezar a desarrollar esta parte, nos crearemos la carpeta **reducers** dentro de la carpeta **todos** y moveremos el fichero **todo.reducer.ts** dentro.

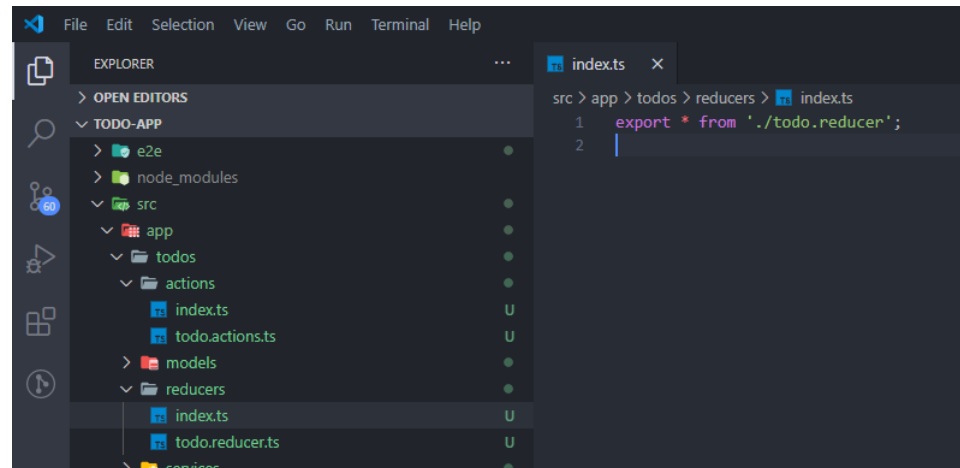
```

1  import { createReducer, on } from '@ngrx/store';
2  import { AppState } from '../../app.reducer';
3  import { Todo } from '../models/todo.model';
4  import { completeTodo, createTodo, deleteTodo, editTodo } from '../actions'; (A)
5
6  // export const initialState: Todo[] = [new Todo('Terminar práctica 2')]; (B)
7
8  export interface TodoState { (C)
9    todos: Todo[];
10   loading: boolean;
11   loaded: boolean;
12   error: any;
13 }
14
15 export const initialState: TodoState = { (D)
16   todos: [new Todo('Terminar práctica 2')],
17   loading: false,
18   loaded: false,
19   error: null
20 };
21
22 const _todoReducer = createReducer(
23   initialState,
24   // on(createTodo, (state, { title }) => [...state, new Todo(title)]), (E)
25   on(createTodo, (state, { title }) => ({ (F)
26     ...state,
27     loading: false,
28     loaded: false,
29     todos: [...state.todos, new Todo(title)]
30   })),
31   /*
32   on(completeTodo, (state, { id }) => {
33     return state.map((todo) => {
34       if (todo.id === id) {
35         return {
36           ...todo,
37           done: true,
38         }; (G)
39       } else {
40         return todo;
41       }
42     });
43   }),
44   on(editTodo, (state, { id, title }) => {
45     return state.map((todo) => {
46       if (todo.id === id) {
47         return {
48           ...todo,
49           title: title,
50         };
51       } else {
52         return todo;
53       }
54     });
55   }),
56   on(deleteTodo, (state, { id }) => state.filter( todo => todo.id !== id ))
57   */
58 );
59
60
61
62 export function todoReducer(state, action) {
63   return _todoReducer(state, action);
64 }

```

- A) Démonos cuenta en los **imports** de las acciones que al definir un **index.ts** y exportar todas las acciones dentro de la carpeta **actions** ahora el **import** nos quedaría como mostramos en el punto A.
- B) Comentamos el **initialState** ya que con el nuevo **TodoState** tendremos que modificarlo, lo vemos en el siguiente punto.
- C) En lugar de tener definido el estado en el fichero **app.reducer.ts** como un **AppState**, lo que haremos es crearnos este estado como **TodoState**, de manera que organizamos por entidad o funcionalidad un estado que nos interesa gestionar. Del array de **todos** que teníamos antes le añadiremos las propiedades:
  - **Loading**: nos servirá para indicar que es cierta cuando lancemos la acción de **getAllTodos**.
  - **Loaded**: nos servirá para indicar que es cierta una vez tengamos respuesta de la petición anterior y dispongamos de los datos, en nuestro caso, del array de **todos**.
  - **Error**: nos servirá para gestionar el error en caso de que éste sea la respuesta de la petición al servidor del array de **todos**.
- D) Definimos el cómo será el estado inicial de nuestra aplicación, será un array de tareas (**Todo**) con la tarea 'Terminar práctica 2' y además tendremos las propiedades **loading**, **loaded** y **error** que las utilizaremos para la lógica cuando llamemos a las acciones **getAll...**
- E) Comentamos la acción **createTodo**.
- F) Implementamos la acción **createTodo** con el nuevo estado. Vemos que devolvemos cada elemento del estado, para las propiedades **loading** y **loaded** las devolvemos a falso ya que para esta acción realmente no nos interesan, y lo que hacíamos en la versión anterior lo haremos ahora para asignar el nuevo array de tareas a la propiedad **todos**, devolviendo todos los elementos del array con los tres puntos y añadiendo la nueva tarea cuyo título lo asignaremos con el parámetro que recibimos.
- G) El resto de las acciones la comentamos y las implementaremos después, primero tenemos que validar que la que acabamos de modificar funciona correctamente.

Ahora tendremos que hacer una pequeña configuración semejante a lo que hemos hecho con las acciones. Dentro de la carpeta **reducers** nos creamos el fichero **index.ts** y escribiremos lo siguiente:



Así podremos exponer todo lo que haya dentro del **todo.reducer.ts** para tener unas importaciones más sencillas, podríamos decir que esto es como un convenio de como deberíamos hacerlo.

Ahora nos iríamos a nuestro **app.reducer.ts** y haríamos lo siguiente:



- A) Con el fichero **index** que hemos creado dentro de la carpeta **reducer** con este **import** podemos importar de forma fácil todos los **reducers** y estados que tengamos definidos, esto nos irá bien para la configuración que haremos a continuación.
- B) Comentamos el **AppState** que teníamos, ya que hemos implementado el **TodoState** dentro de su **reducer** y ahora el **AppState** se encargara de centralizar los diferentes estados que podamos tener en nuestra ampliación.

C) Ahora nosotros/as le diremos que el estado **todosApp** estará implementado por la interfaz **TodoState** y lo utilizaremos como **AppState**. Después en el fichero **todo-list.component.ts** cuando nos tengamos que suscribir a dicho estado veremos su aplicación, veremos que en la cláusula **select** utilizaremos el nombre **totosApp**.

D) Esta será la nueva definición de los **reducers** para tenerlos de forma centralizada.

Con los puntos C y D anteriores, nos quedaría una ultima configuración para que nos funcionara.

Vamos al fichero **app.module.ts** y hacemos:

```

app.module.ts X
src > app > app.module.ts > AppModule
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3
4  import { AppRoutingModule } from './app-routing.module';
5  import { AppComponent } from './app.component';
6  import { TodoModule } from './todos/todo.module';
7  import { StoreModule } from '@ngrx/store';
8  import { todoReducer } from './todos/reducers/todo.reducer';
9  import { StoreDevtoolsModule } from '@ngrx/store-devtools';
10 import { environment } from 'src/environments/environment';
11 import { appReducers } from './app.reducer';
12
13 @NgModule({
14   declarations: [
15     AppComponent
16   ],
17   imports: [
18     BrowserModule,
19     AppRoutingModule,
20     TodoModule,
21     // StoreModule.forRoot({todos: todoReducer}),
22     StoreModule.forRoot([appReducers]),
23     StoreDevtoolsModule.instrument({
24       maxAge: 25,
25       logOnly: environment.production
26     })
27   ],
28   providers: [],
29   bootstrap: [AppComponent]
30 })
31 export class AppModule { }
32

```

De esta manera le estamos diciendo con que estructuras trabaja nuestro **store**, que pueden ser más de una.

Finalmente, nos queda validar que la acción de crear una tarea funciona correctamente. Para ello, nos quedará adaptar el código para que cuando haya cambios en el **store**, en este caso, cuando se vayan añadiendo tareas, éstas se muestren en la vista. Por lo tanto, tenemos que modificar la manera como nos suscribimos al **store** a la nueva estructura, nos vamos al fichero **todo-list.component.ts** y hacemos:

```

src > app > todos > todo-list > todo-list.component.ts > ...
1  import { Component, OnInit } from '@angular/core';
2  import { Store } from '@ngrx/store';
3  import { AppState } from 'src/app/app.reducer';
4  import { Todo } from '../models/todo.model';
5  import { TodoService } from '../services/todo.service';
6
7  @Component({
8    selector: 'app-todo-list',
9    templateUrl: './todo-list.component.html',
10   styleUrls: ['./todo-list.component.css'],
11 })
12 export class TodoListComponent implements OnInit {
13
14   todos: Todo[] = [];
15
16   constructor(
17     private store: Store<AppState>,
18     // private todoService: TodoService
19   ) {}
20
21   ngOnInit(): void {
22
23     // this.store.select('todos').subscribe((todos) => (this.todos = todos));
24
25     this.store.select('todosApp').subscribe( todosResponse => {
26       this.todos = todosResponse.todos;
27     });
28
29     // this.todoService.getAllTodos().subscribe((todos) => this.todos = todos);
30   }
31 }
32

```

Comentamos el servicio que habíamos implementado al principio de la sección de los **effects** y vemos que también tenemos comentada la línea anterior en la que nos suscribíamos al **store** ya que la tenemos que modificar.

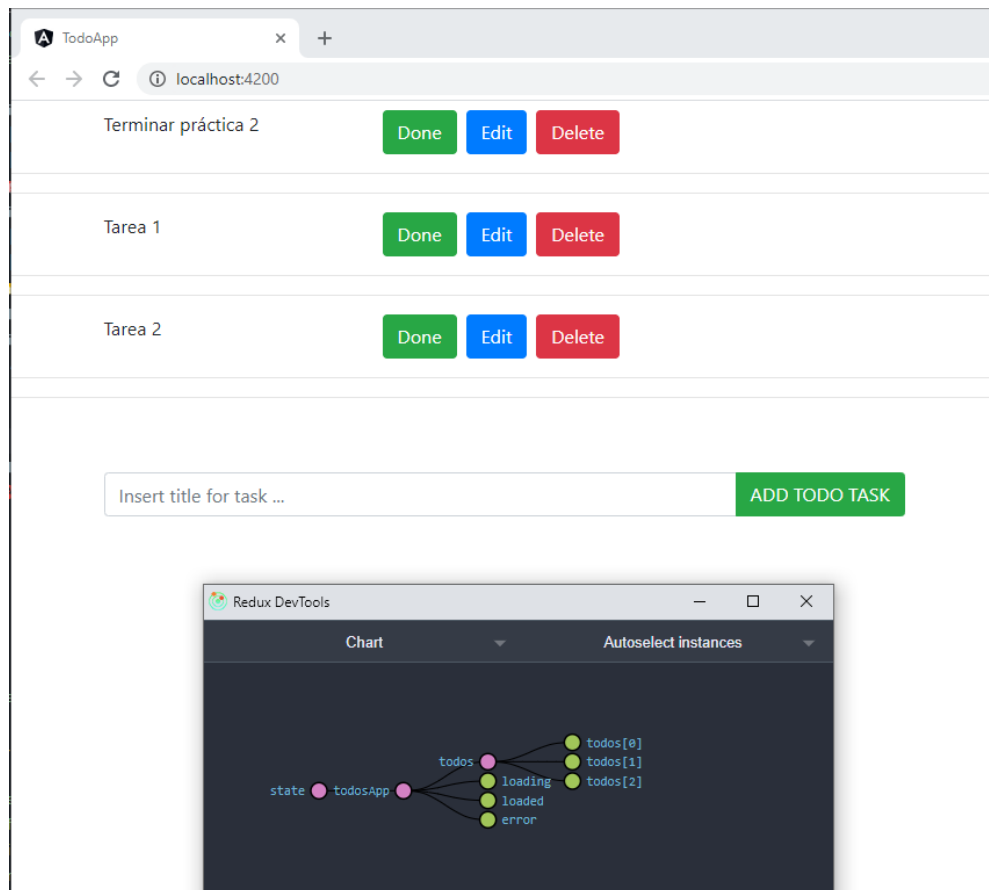
La corrección es lo que destacamos en rojo. Nótese que nos suscribimos a **todosApp** que como hemos visto anteriormente es una propiedad del **AppState** del fichero **app.reducer.ts**.

Entonces de la respuesta obtenida, nos quedamos con el array de **todos** y lo asignamos a la variable que recorre la vista para pintar la información.

En este momento tenemos la aplicación funcionando, pero solo la acción de crear tarea, pero adaptada a la nueva estructura que nos ira bien para gestionar los **effects**.

Si vemos la ejecución y la salida de la herramienta **Redux dev-tools** vemos que tenemos la tarea inicial que está definida al estado inicial más las dos tareas que hemos añadido mediante la acción de añadir tarea.





Perfecto, ahora nos quedan los siguientes pasos:

- Adaptar **completeTodo**, **editTodo** y **deleteTodo** a la nueva estructura.
- Implementar las acciones de **getAllTodos** utilizando **effects** para que cargue el **mockobject** de las diez tareas que hay definidas.

En el fichero **todo.reducer.ts** modificamos la acción **completeTodo**:

```
/*
on(completeTodo, (state, { id }) => {
  return state.map((todo) => {
    if (todo.id === id) {
      return {
        ...todo,
        done: true,
      };
    } else {
      return todo;
    }
  });
}),
*/
on(completeTodo, (state, { id }) => ({
  ...state,
  loading: false,
  loaded: false,
  todos: [...state.todos.map((todo) => {
    if (todo.id === id) {
      return {
        ...todo,
        done: true,
      };
    } else {
      return todo;
    }
  })]
})),
/*
```

Modificamos la acción **editTodo**:

```
/*
on(editTodo, (state, { id, title }) => {
  return state.map((todo) => {
    if (todo.id === id) {
      return {
        ...todo,
        title: title,
      };
    } else {
      return todo;
    }
  });
}),
*/
on(editTodo, (state, { id, title }) => ({
  ...state,
  loading: false,
  loaded: false,
  todos: [...state.todos.map((todo) => {
    if (todo.id === id) {
      return {
        ...todo,
        title
      };
    } else {
      return todo;
    }
  })]
})),
/*
```

Finalmente modificamos la acción **deleteTodo**:

```
// on(deleteTodo, (state, { id }) => state.filter( todo => todo.id !== id ))
on(deleteTodo, (state, { id }) => ({
  ...state,
  loading: false,
  loaded: false,
  todos: [...state.todos.filter( todo => todo.id !== id )]
}))
```

Una vez tenemos todo nuestro proyecto adaptado, vamos a finalizarlo implementando en nuestro **reducer** las acciones que habíamos definido inicialmente **getAllTodos**, **getAllTodosSuccess** y **getAllTodosError**.

```
on( getAllTodos, state => ({...state, loading: true})),
on ( getAllTodosSuccess, ( state, {todos}) => ({
  ...state,
  loading: false,
  loaded: true,
  todos: [ ...todos]
})),
on ( getAllTodosError, ( state, {payload}) => ({
  ...state,
  loading: false,
  loaded: false,
  error: payload
}))
```

- **getAllTodos**: lanza la acción de cargar tareas y asigna la variable **loading** a **true**.
- **getAllTodosSuccess**: cargar las tareas que recibimos cuando la acción funciona correctamente.
- **getAllTodosError**: si la llamada funciona mal, trataremos el error de manera controlada.

Nos queda el ultimo paso, implementar los **effects** para que escuche las acciones **getAllTodos** ... y llame al servicio **getAllTodos** del fichero **todo.service.ts** y devuelva el listado de tareas del **mockobject**, simulando que es una llamada a una api.

**Configuramos el effect.**

Instalamos el paquete siguiente:

```
H:\Projectes\todo-app>npm install @ngrx/effects --save
```

Creamos una carpeta **effects** dentro de la carpeta **todos** y nos creamos el fichero **todos.effects.ts** y dentro escribimos:

```

src > app > todos > effects > todos.effects.ts > ...
1  import { Injectable } from '@angular/core';
2  import { Actions, createEffect, ofType } from '@ngrx/effects';
3  import { getAllTodos, getAllTodosError, getAllTodosSuccess } from '../actions';
4  import { TodoService } from '../services/todo.service';
5  import { mergeMap, map, catchError } from 'rxjs/operators';
6  import { of } from 'rxjs';
7
8  @Injectable()
9  export class TodosEffects {
10
11    constructor(
12      private actions$: Actions,
13      private todosService: TodoService
14    ) {}
15
16    getTodos$ = createEffect(() =>
17      this.actions$.pipe(
18        ofType(getAllTodos),
19        mergeMap(() =>
20          this.todosService.getAllTodos().pipe(
21            map((todos) => getAllTodosSuccess({ todos: todos })),
22            catchError((err) => of(getAllTodosError({ payload: err })))
23          )
24        )
25      );
26  };
27
28  }

```

Con el signo de **\$** indicamos que es un observable. Es un estándar. Definimos un observable que está escuchando todas las acciones de nuestra aplicación. En nuestro caso, mediante **pipe** y **ofType** acotaremos esto y le diremos que escuche solo la acción **getAllTodos**.

Después queremos disparar otro observable que se encargue de pedir la información y se una al anterior, esto lo hacemos con **mergeMap**. Dentro del **mergeMap** básicamente llamaremos al servicio **Angular** que hará la comunicación con terceros, normalmente peticiones **http** a un **backend**. Por lo tanto, llamaremos al método **getAllTodos** del servicio **TodoService**. Para poder utilizarlo tendremos que inyectarlo en el constructor.

Posteriormente, sabemos que la respuesta de este servicio podría ser el array de **todos** o un elemento de **error** en caso de que algo no saliera bien.

Por lo tanto, si la respuesta es correcta, se la pasamos a la acción **getAllTodosSuccess**, y si se produce algún error, lo cual lo detectamos con **catchError**, llamamos a **getAllTodosError** con el elemento de **error** por argumento para tratarlo posteriormente.

Nótese que utilizamos la cláusula **of**, y esto es debido a que en nuestra definición del **TodoState** en el fichero **todo.reducer.ts** definimos la propiedad **error** como **any** ya que inicialmente no sabíamos cómo sería. Con la cláusula **of** nos permite mapear (asignar) la respuesta **err** a nuestra propiedad **error** mediante un observable.

Después configuraremos estos efectos en los **imports** del **app.module** para terminar la configuración.

Creemos dentro de la carpeta **effects** un fichero **index.ts** con lo siguiente:

```
index.ts X
src > app > todos > effects > index.ts > ...
1 import { TodosEffects } from './todos.effects';
2
3
4 export const EffectsArray: any[] = [TodosEffects];
5
```

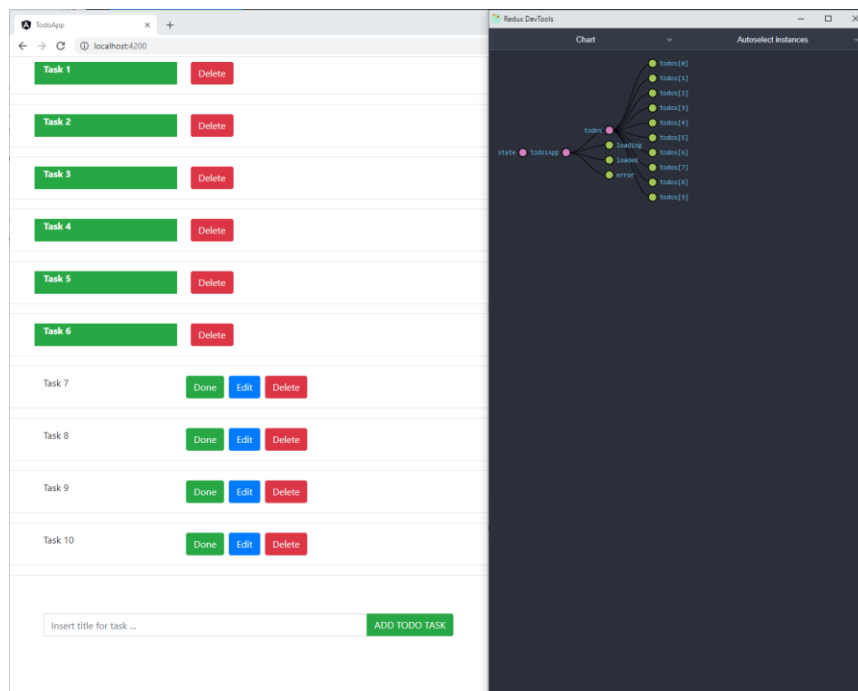
Vamos al **app.module.ts** :

```
app.module.ts X
src > app > app.module.ts > ...
2 import { NgModule } from '@angular/core';
3
4 import { AppRoutingModule } from './app-routing.module';
5 import { AppComponent } from './app.component';
6 import { TodoModule } from './todos/todo.module';
7 import { StoreModule } from '@ngrx/store';
8 import { todoReducer } from './todos/reducers/todo.reducer';
9 import { StoreDevtoolsModule } from '@ngrx/store-devtools';
10 import { environment } from 'src/environments/environment';
11 import { appReducers } from './app.reducer';
12 import { EffectsModule } from '@ngrx/effects';
13 import { EffectsArray } from './todos/effects';
14
15 @NgModule({
16   declarations: [
17     AppComponent
18   ],
19   imports: [
20     BrowserModule,
21     AppRoutingModule,
22     TodoModule,
23     // StoreModule.forRoot({todos: todoReducer}),
24     StoreModule.forRoot(appReducers),
25     EffectsModule.forRoot(EffectsArray),
26     StoreDevtoolsModule.instrument({
27       maxAge: 25,
28       logOnly: environment.production
29     })
30   ],
31   providers: [],
32   bootstrap: [AppComponent]
33 })
34 export class AppModule { }
35
```

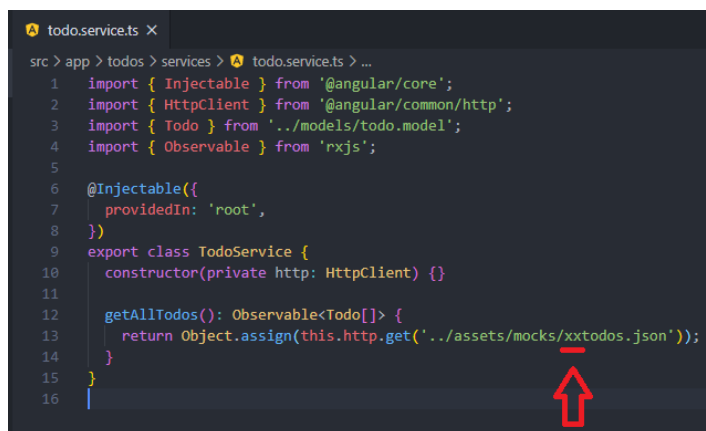
Al fichero **todo-list.component.ts** lanzamos la llamada al efecto:

```
todo-list.component.ts X
src > app > todos > todo-list > todo-list.component.ts > ...
2 import { Store } from '@ngrx/store';
3 import { AppState } from 'src/app/app.reducer';
4 import { getAllTodos } from '../actions';
5 import { Todo } from '../models/todo.model';
6 import { TodoService } from '../services/todo.service';
7
8 @Component({
9   selector: 'app-todo-list',
10   templateUrl: './todo-list.component.html',
11   styleUrls: ['./todo-list.component.css'],
12 })
13 export class TodoListComponent implements OnInit {
14   todos: Todo[] = [];
15
16   constructor(
17     private store: Store<AppState>
18   ) // private todoService: TodoService
19   {}
20
21   ngOnInit(): void {
22     // this.store.select('todos').subscribe((todos) => (this.todos = todos));
23
24     this.store.select('todosApp').subscribe((todosResponse) => {
25       this.todos = todosResponse.todos;
26     });
27
28     this.store.dispatch(getAllTodos());
29
30     // this.todoService.getAllTodos().subscribe((todos) => this.todos = todos);
31   }
32 }
33
```

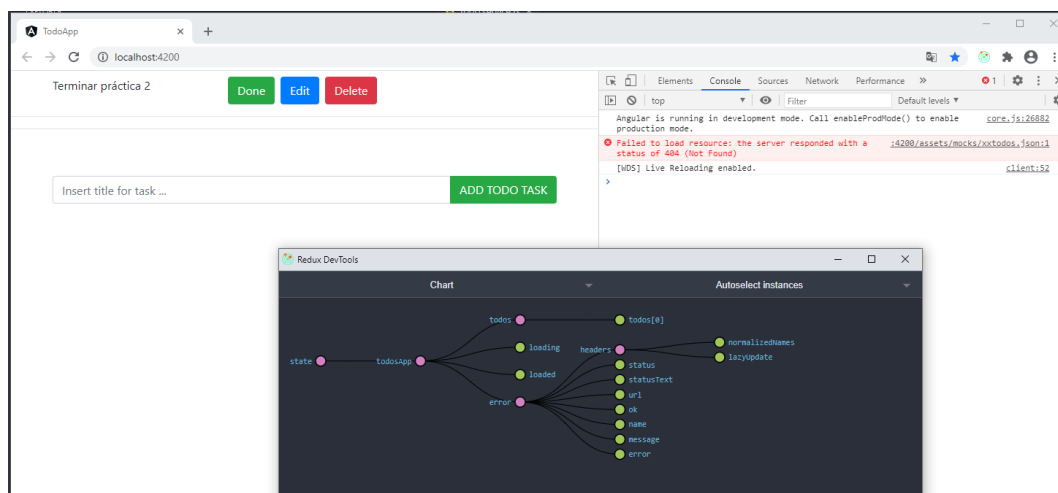
Fijémonos **Redux dev-tools** al cargar que tenemos todos los datos:



Y ahora, por ejemplo, forcemos un error:



Vemos la ejecución:



Veremos que dentro de **error** tenemos todas las propiedades, por ejemplo, dentro de **status** tendríamos el código de error 404.

Podríamos utilizar esta información para mostrar a la vista un poco de información del tipo de error al usuario.

Por ejemplo, en el fichero **todo.reducer.ts** en la acción **getAllTodosError**, podríamos modificar la línea:

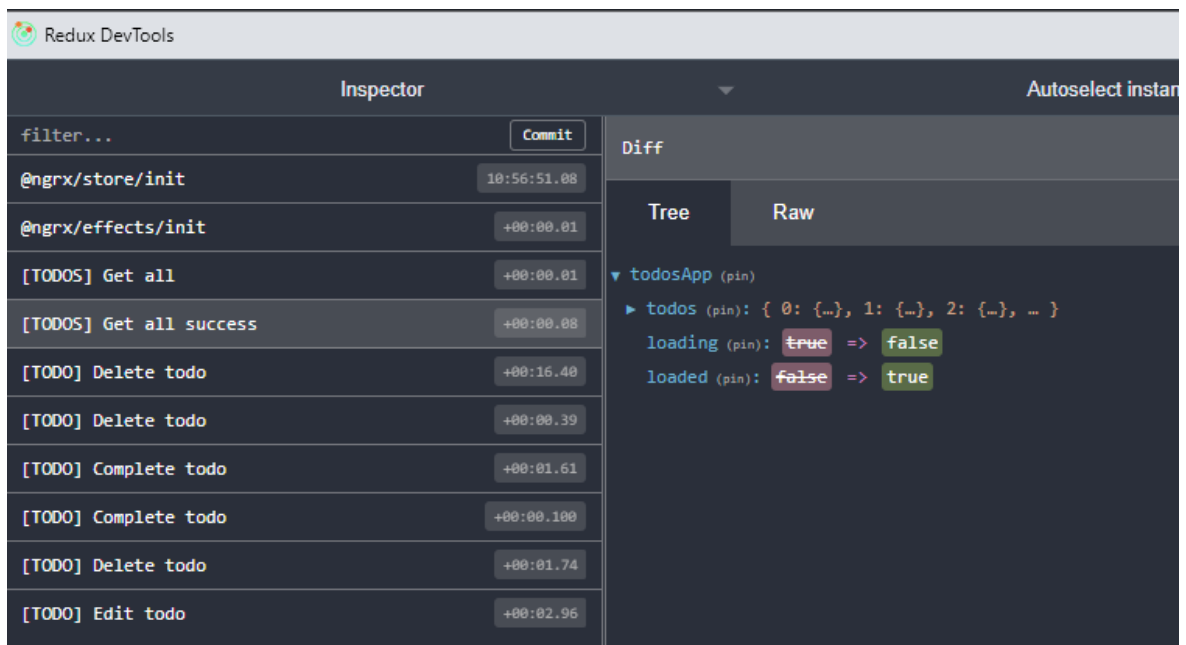
```
error: payload
```

Por algo así:

```
error: {
  url: payload.url,
  status: payload.status,
  message: payload.message
}
```

Es decir, de las propiedades de error que nos devuelve la llamada que hemos visto en **Redux dev-tools** anteriormente, coger las que nos interesen, de manera que podríamos utilizarlas para mostrar un poco de información al usuario en la vista para que sepa que está pasando.

También podríamos utilizar las variables **loading** y **loaded** para, por ejemplo, gestionar un **spinner** mientras esperamos la petición.



Redux DevTools

Inspector

filter... Commit

@ngrx/store/init 10:56:51.08

@ngrx/effects/init +00:00.01

[TODOS] Get all +00:00.01

[TODOS] Get all success +00:00.08

[TODO] Delete todo +00:16.40

[TODO] Delete todo +00:00.39

[TODO] Complete todo +00:01.61

[TODO] Complete todo +00:00.100

[TODO] Delete todo +00:01.74

[TODO] Edit todo +00:02.96

Autoselect instance

Diff

Tree Raw

▼ todosApp (pin)

▶ todos (pin): { 0: {...}, 1: {...}, 2: {...}, ... }

loading (pin): true => false

loaded (pin): false => true

Mostramos la estructura de ficheros final de nuestro proyecto:

