

Full Stack Developer

Programación reactiva
RxJS y Redux

Quedan rigurosamente prohibidas, sin la autorización escrita de los titulares del Copyright, bajo las sanciones establecidas en las leyes, la reproducción total o parcial de esta obra por cualquier medio o procedimiento, comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares de ella mediante alquiler o préstamo públicos. Diríjase a CEDRO (Centro Español de Derechos Reprográficos, www.cedro.org) si necesita fotocopiar o escanear algún fragmento de esta obra.

INICIATIVA Y COORDINACIÓN

DEUSTO FORMACIÓN

COLABORADORES

Realización:

ISCA Training & Consulting S.L.

Elaboración de contenidos:

Pedro Jiménez Castela

Desarrollador senior Full Stack JavaScript, TypeScript, Angular, React, NodeJS, MongoDB y JS Testing (Jest y Cypress).

Instructor en formación oficial MongoDB y NodeJS.

Instructor del CFTIC de la Comunidad de Madrid.

Certificado como: MongoDB DBA n.º 582-916-826 y MongoDB DEV n.º 560-512-224.

Supervisión técnica y pedagógica:

MG AGNESI TRAINING

Coordinación editorial:

MG AGNESI TRAINING

© MG AGNESI TRAINING, S.L.

Barcelona (España), 2022

Primera edición: septiembre 2022

ISBN: 978-84-19142-31-3 (Obra completa)

ISBN: 978-84-19142-44-3 (Programación reactiva RxJS y Redux)

Depósito Legal: B 3409-2022

Impreso por:

SERVINFORM

Avenida de los Premios Nobel, 37

Polígono Casablanca

28850 Torrejón de Ardoz (Madrid)

Printed in Spain

Impreso en España

Esquema de contenido

1. Introducción a Redux

- 1.1. Conceptos generales sobre Redux
- 1.2. Implementación de Redux en un proyecto

2. *Store, reducers, actions* y suscripción a cambios

- 2.1. *Store, reducers y actions*
- 2.2. *Dispatch* de acciones en Redux
- 2.3. Suscripciones en Redux

3. Conectando Redux con React DevTools

- 3.1. Redux en React
- 3.2. Conectando Redux con React DevTools
- 3.3. Suscripción al estado

4. Asincronía en Redux

- 4.1. Escalando el estado de Redux en React
- 4.2. Asincronía en Redux

5. Programación reactiva, concepto de *observable* y *observer*

- 5.1. Librería RxJS y el concepto de *observable* y *observer*
- 5.2. *Observables* y *observer* en JavaScript con RxJS

6. Estrategias de uso con React

- 6.1. Librería RxJS y React
- 6.2. *Observable, observer* y *subject* en React

7. Estrategias de uso con Angular

- 7.1. Librería RxJS y Angular
- 7.2. *Observable, observer* y *subject* en Angular

Introducción

A medida que las aplicaciones escalan incorporando centenares de requisitos, se amplía la complejidad de las relaciones entre los diferentes componentes que constituyen la interfaz de usuario, motivo por el que el manejo del estado de la aplicación es una de las tareas críticas de un proyecto.

En este módulo vamos a aprender cómo gestionar el estado en las aplicaciones Single Page Application, comenzando por los fundamentos del patrón Redux, uno de los más utilizados, a través de la librería del mismo nombre.

Continuaremos aprendiendo a utilizar Redux de manera práctica en JavaScript plano para seguidamente comprender su implementación en proyectos React, *framework* en el que se usa a menudo en procesos asíncronos.

También aprenderemos a utilizar la programación reactiva para el manejo de estado y otras funcionalidades a través de la librería RxJS, en sus implementaciones tanto para proyectos React como para proyectos Angular.

1. Introducción a Redux

Comenzamos esta primera unidad relacionada con la programación reactiva con una introducción a Redux, una de las librerías más empleadas para este propósito en las aplicaciones avanzadas.

En primer lugar, describiremos las principales características de Redux y sus usos, poniendo especial atención en la importancia de la gestión del estado de una aplicación, en la que se consiguen resultados eficientes con este patrón.

Posteriormente, generaremos un primer proyecto en JavaScript plano para comenzar a instalar Redux, preparar los archivos y posteriormente implementar el patrón.

1.1. Conceptos generales sobre Redux

Redux es una librería JavaScript de código abierto para el manejo del estado en aplicaciones creada por Dan Abramov y Andrew Clark, disponible para cualquier desarrollo o *framework*, pero especialmente relacionada con el uso de React, ya que parte de una librería anterior de Facebook denominada Flux y sus desarrolladores y comunidad están muy relacionados con React.

Como ya se ha detallado y practicado anteriormente, las aplicaciones modernas necesitan una implementación e interrelación de numerosos componentes de interfaz de usuario que contienen, de manera encapsulada, su lógica y su capa de presentación. Hemos visto que esos componentes pueden compartir sus datos en diferentes relaciones o patrones (jerarquía padre-hijo, empleo de servicios, clases e interfaces, etc.), pero, en todos los casos, cuando la aplicación escala en complejidad, tendremos cientos de valores que deberán utilizar los componentes con diferente casuística. Será, en resumen, un estado de la aplicación mutable y complejo.

Lo que persigue Redux es disponer de un sistema de almacenamiento fiable del estado de la aplicación, lo que se denomina como “fuente de la verdad”, y un mecanismo de acceso y modificación de ese estado también fiable y seguro, proporcionando un servicio al que los componentes se pueden suscribir para actualizar esos cambios de manera ágil; y es aquí donde interviene la programación reactiva.



Para saber más

En realidad, Redux está a medio camino entre una librería y un patrón de arquitectura; puede llegar a utilizarse sin su instalación, aunque exigiría un mayor esfuerzo de desarrollo.

Las ventajas de Redux son múltiples en cuanto a fiabilidad, predictibilidad, consistencia y mantenimiento; pero, a cambio, exige cumplir con el patrón que implementa, para lo cual es necesario tener claros los conceptos o elementos que componen el uso de esta librería.

Estos elementos están bien definidos y se pueden resumir en el siguiente esquema (Figura 1.1)

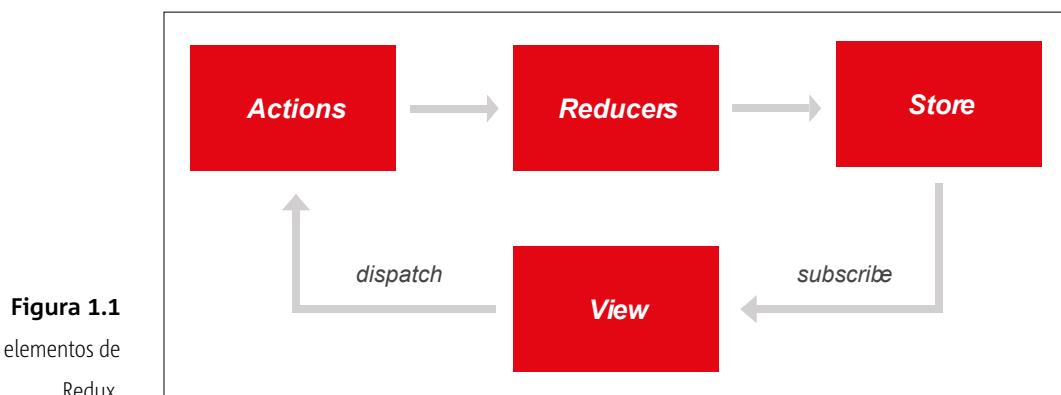


Figura 1.1

Esquema de elementos de Redux.

Los componentes o elementos detallados en el esquema determinan su implementación y presentan las siguientes características:

- **Store:** como su nombre traducido del inglés indica, se trata del almacén del estado de nuestra aplicación, un objeto con la forma que necesite el desarrollo, de manera que sus propiedades y tipos de datos permitan un acceso fiable a sus valores y pueda ser fácilmente escalable.
- **View:** las vistas serán los componentes de interfaz de usuario que, a partir de la suscripción al estado, podrán obtener los valores del estado para implementar la lógica necesaria de acuerdo con estos. A diferencia de un acceso a datos en un modelo tradicional, los valores en Redux son leídos con esas suscripciones, pero no serán mutados directamente; la actualización de datos se realizará desde la vista, lanzando los eventos con un proceso *dispatch* que llamará a las *actions*.
- **Actions:** también, como su nombre indica, las *actions* son objetos donde se define qué acciones se van a producir en el cambio del estado, para construir una colección de cambios en el estado totalmente controlados y disponibles para ser lanzados desde las vistas mediante los *dispatch*.

- **Reducers:** finalmente, los *reducers* son funciones que reciben el estado y una *action*, y lo devuelven modificado de acuerdo con las instrucciones de la *action*. Estas funciones son invocadas en cada *dispatch* y, al retornar el nuevo estado, actualizan este y provocan que las vistas suscritas al estado actualicen sus valores e implementen la lógica programada con esos cambios.

Como ocurre siempre en programación, la mejor manera de comprender estos conceptos una vez conocidos es su implementación en código, para lo cual comenzaremos a desarrollar a continuación un proyecto práctico con Redux.

1.2. Implementación de Redux en un proyecto

Aunque, como ya se ha detallado, Redux se usa habitualmente en React, de cara a una mejor comprensión inicial es interesante usar sus elementos en un proyecto con JavaScript plano, en el que veremos de forma limpia cada uno de ellos.

Para ello, vamos a crear en cualquier ubicación de nuestro equipo un directorio *practica1*, lo abrimos con Visual Studio Code y, una vez cargado, creamos un directorio *js* con un archivo *main.js*, y, en la raíz, un archivo *index.html* en el que añadimos el siguiente código:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Redux</title>
    <script src="https://unpkg.com/redux@latest/dist/redux.min.js"></script>
  </head>
  <body>
    <h1>Stock Control</h1>
    <h2>Nuevos artículos</h2>
    <input type="text" id="sku" placeholder="sku">
    <input type="text" id="name" placeholder="name">
    <input type="text" id="stock" placeholder="stock">
    <button id="addButton">Añadir</button>

    <script src=".//js/main.js"></script>
  </body>
</html>
```

Como podemos observar, en primer lugar hemos añadido un archivo redux.min.js a través de un servicio CDN, de manera que no será necesario instalar Redux en este proyecto para simplificarlo. A continuación, hemos añadido varios elementos “input” en los que recogeremos los valores de un objeto de artículo y su *stock*, y finalmente hemos añadido el *script* con nuestro archivo main.js, que es donde podremos utilizar y poner en práctica más adelante los beneficios de Redux.

Otra alternativa en la implementación de Redux es su instalación a través de npm con el siguiente comando:

```
npm install redux --save
```

Esta fórmula es más común en su uso en *frameworks*, fundamentalmente React, y será complementada con otras librerías de implementación, como veremos más adelante.



Resumen

- Redux es una librería JavaScript de código abierto para el manejo del estado en aplicaciones complejas. Incorpora un *store* de almacenamiento de propiedades para ser consumidas por los componentes de manera fiable y escalable.
- Redux se puede implementar en los proyectos JavaScript a través de un servicio CDN o bien mediante un paquete instalado por npm, y puede usarse en cualquier aplicación con independencia del *framework* que se utilice.

2. *Store, reducers, actions* y suscripción a cambios

Continuamos con el aprendizaje de Redux a través de la declaración de los diferentes elementos que componen el patrón y el uso de su *store* mediante la suscripción de los componentes.

En primer lugar, implementaremos de manera práctica en un proyecto JavaScript plano los elementos de *store*, *reducers* y *actions*, realizando procesos de *dispatch* para comprobar cómo se almacena el estado.

Posteriormente, generaremos un bloque de código mediante suscripción a los cambios del estado y podremos comprobar en la vista la modificación que producen esas suscripciones.

2.1. *Store, reducers y actions*

Llega el momento de implementar en código los elementos de Redux, para lo cual vamos a abrir nuestro proyecto *practica1* en Visual Studio Code, abrimos el archivo *main.js* y, en primer lugar, añadimos el primer elemento, el *store* de almacenamiento de los datos del estado:

```
const store = Redux.createStore();
```

En este sencillo proceso, simplemente llamamos la atención acerca de que el *store* usa la librería Redux para, mediante el método “*createStore()*”, crear una instancia donde se almacenará el estado. Este método recibe como argumento el siguiente de los componentes que hay que desarrollar, el *reducer*. Modificamos, por tanto, el código de la siguiente forma:

```
const stockReducer = (state, action) => {  
  return state;  
}  
  
const store = Redux.createStore(stockReducer);
```

Como vemos, los *reducers* son funciones que reciben el estado y una acción, y devuelven el estado modificado. De esta manera, como son pasados al *store*, cada vez que sean invocados, actualizarán sus valores. En este caso, el reducer para cambiar el *stock* (por eso se denomina así) no realiza ningún cambio en el estado, por lo que debemos desarrollar una *action* para introducirla en su cuerpo. Por tanto, vamos a añadir en el código una primera *action* y modificar el *reducer* de la siguiente forma:

```
const stockReducer = (state, action) => {

  const { type, payload } = action;

  switch (type) {
    case 'ADD_STOCK':
      return [...state, payload];
    default:
      return [];
  }
}

const addStockAction = (sku, name, stock) => {
  return {
    type: 'ADD_STOCK',
    payload: {sku, name, stock}
  }
}

const store = Redux.createStore(stockReducer);
```

Ahora podemos observar la forma de la función de tipo *action*: dispone de unos parámetros definidos para recibir los valores que serán usados en la acción y devuelve un objeto. Este objeto consta de una propiedad “*type*”, donde se pasa un *string* para identificar el tipo de acción que se va a realizar, y un *payload* donde se pasan los valores que se van a usar en la acción (en este caso, provenientes de los parámetros).

Una vez que tenemos nuestra primera *action*, es incorporada a la función *reducer*, en nuestro caso denominada “*stockReducer*”. La forma habitual es usar una estructura “switch” que evalúa el “*type*” de cada *action* para llevar a cabo en su bloque correspondiente la modificación en el estado a partir de los valores que proporciona cada una de las *actions* que tuviéramos. En nuestro caso, la *action* “*ADD_STOCK*” provoca que se añada al estado un objeto con las tres propiedades y valor de su *payload*.



Para saber más

Dependiendo del tipo de acción que llevemos a cabo en la modificación del estado, el *payload* tendrá mayor o menor complejidad. En algunos casos puede tratarse de una propiedad primitiva cuando la acción es muy sencilla.

Ahora llega el momento de cerrar el ciclo Redux mediante el uso de *dispatch* para generar una action y obtener el estado. Para simular este ciclo, añadimos el método “*dispatch()*” con la acción y “*getState()*” sobre el *store* para obtener el estado en la consola de la siguiente forma:

```
const stockReducer = (state, action) => {
  const { type, payload } = action;
  switch (type) {
    case 'ADD_STOCK':
      return [...state, payload];
    default:
      return [];
  }
}

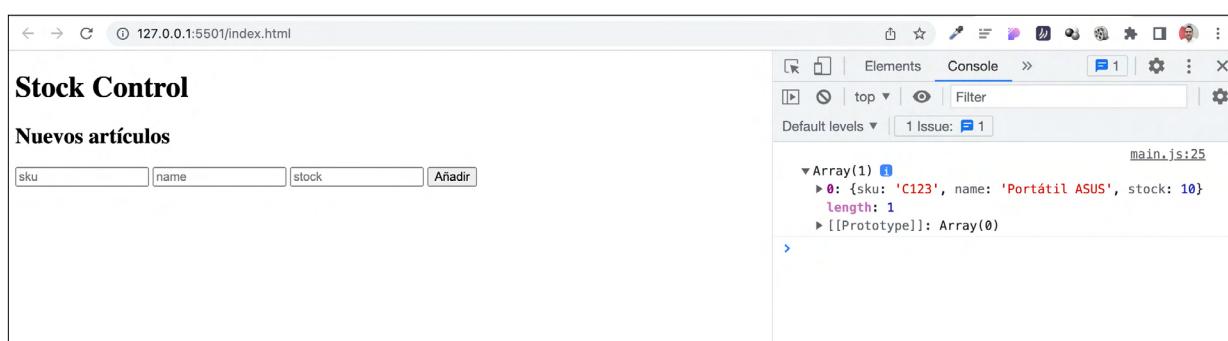
const addStockAction = (sku, name, stock) => {
  return {
    type: 'ADD_STOCK',
    payload: {sku, name, stock}
  }
}

const store = Redux.createStore(stockReducer);
store.dispatch(addStockAction('C123', 'Portátil ASUS', 10));
console.log(store.getState());
```

Ahora mismo, de manera programática, hemos lanzado el “*dispatch()*” con la *action* para añadir un objeto a nuestro estado, empleando la invocación de “*addStockAction()*”; y, posteriormente, en un “*console.log()*” obtenemos del *store* el estado de la aplicación con el método “*getState()*”. Podemos comprobarlo en el navegador: abrimos el archivo index.html con la extensión Live Server de Visual Studio Code, y en la consola de las herramientas de desarrollador (Figura 2.1) veremos el valor del estado.

Figura 2.1

Salida por consola del estado de una aplicación mediante Redux.



Podemos repetir el proceso *dispatch* tantas veces como necesitemos, pero, como veremos a continuación, el disparo de las acciones se llevará a cabo desde eventos de los componentes lógicos de las vistas.

2.2. Dispatch de acciones en Redux

En el código anterior hemos lanzado de manera programática el *dispatch* de nuestra acción, pero lo normal es que esté enlazado a eventos de nuestra aplicación. Aunque en nuestro proyecto, por tratarse de JavaScript plano, no disponemos de componentes, sí que podemos usar nuestros elementos HTML de *input* para, una vez pulsado el botón “añadir”, obtener en JavaScript sus valores e introducirlos en la acción. Para ello, vamos a crear una función en nuestro archivo main.js que se encargue de este proceso modificando el código de la siguiente forma:

```
const stockReducer = (state, action) => {

  const { type, payload } = action;

  switch (type) {
    case 'ADD_STOCK':
      return [...state, payload];
    default:
      return [];
  }
}

const addStockAction = (sku, name, stock) => {
  return {
    type: 'ADD_STOCK',
    payload: {sku, name, stock}
  }
}

const store = Redux.createStore(stockReducer);

// store.dispatch(addStockAction('123', 'Laptop', 10));

const skuInput = document.getElementById('sku');
const nameInput = document.getElementById('name');
const stockInput = document.getElementById('stock');

document.getElementById(' addButton').addEventListener('click', () => {
  store.dispatch(addStockAction(skuInput.value, nameInput.value, stockInput.value));
  skuInput.value = '';
  nameInput.value = '';
  stockInput.value = '';
  console.log(store.getState());
})
```

Ahora podemos añadir varios conjuntos de valores a través de los campos del formulario HTML e ir actualizado el estado, comprobando de nuevo en la consola la evolución del estado (Figura 2.2).

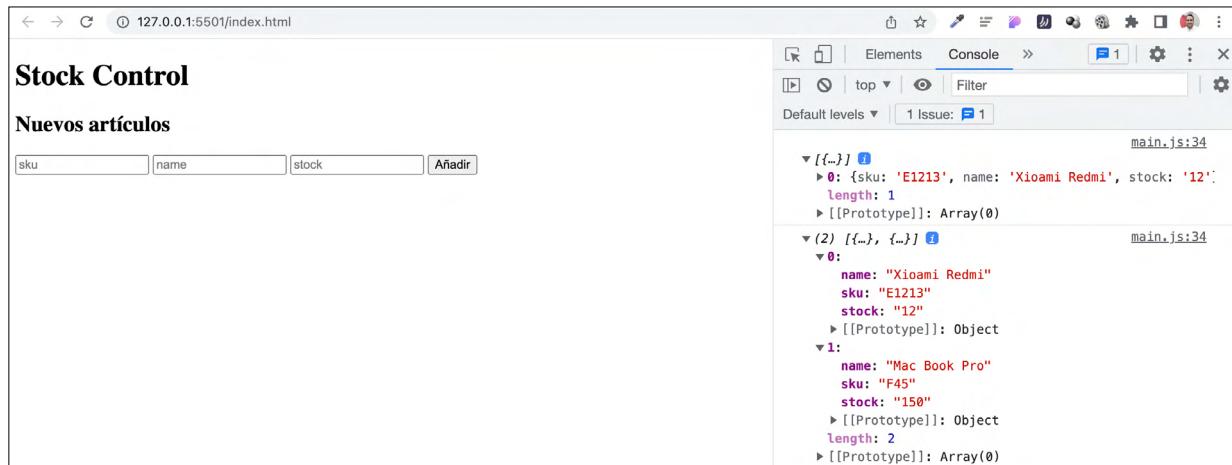


Figura 2.2
Salida por consola del estado Redux a partir de los *inputs* de un formulario.

Otra de las ventajas ya comentadas de Redux es su escalabilidad: en cualquier momento podremos modificar las *actions* y los *reducers* para extender propiedades o funcionalidad, e igualmente podremos tener diferentes *actions* según la lógica de cada actualización del estado que tengamos que implementar.

2.3. Suscripciones en Redux

La manera en que se consume el estado almacenado en Redux nos aporta otra de las grandes ventajas de esta librería: los componentes se pueden suscribir al estado, de forma que, cuando el mismo es notificado, recibirán los nuevos valores y podrán implementar una determinada lógica. Estas suscripciones permiten que cada cambio en el estado suponga un evento que se puede asociar a una función que ejecutará unos determinados bloques de código.

Podemos comprobarlo de manera práctica en nuestro proyecto: añadimos en el archivo index.html una estructura de tabla que renderice los objetos del estado, de manera que, cuando se produzcan cambios en este, ejecute una función que recorra el *array* de objetos del estado y los implemente en una tabla para que el usuario los visualice.

Para ello, en primer lugar, modificamos el código de index.html con la tabla de la siguiente forma:

```

<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Redux</title>
    <script src="https://unpkg.com/redux@latest/dist/redux.min.js"></script>
  </head>
  <body>
    <h1>Stock Control</h1>
    <h2>Nuevos artículos</h2>
    <input type="text" id="sku" placeholder="sku">
    <input type="text" id="name" placeholder="name">
    <input type="text" id="stock" placeholder="stock">
    <button id="addButton">Añadir</button>
    <h2>Listado Stock</h2>
    <table>
      <thead>
        <tr>
          <th>SKU</th>
          <th>Nombre</th>
          <th>Stock</th>
        </tr>
      </thead>
      <tbody id="stockList">
      </tbody>
    </table>
    <script src=".js/main.js"></script>
  </body>
</html>

```

A continuación, en el archivo main.js vamos a utilizar el método “subscribe” sobre el objeto *store* pasándole una función “showStock()” que se invocará cuando haya cambios en el estado, y en esta función usaremos los valores del estado para construir la tabla y, por tanto, renderizarla. Modificamos el código de la siguiente forma:

```

const stockReducer = (state, action) => {

  const { type, payload } = action;

  switch (type) {
    case 'ADD_STOCK':
      return [...state, payload];
    default:
      return [];
  }
}

```

```

}

const addStockAction = (sku, name, stock) => {
  return {
    type: 'ADD_STOCK',
    payload: {sku, name, stock}
  }
}

const store = Redux.createStore(stockReducer);

const skuInput = document.getElementById('sku');
const nameInput = document.getElementById('name');
const stockInput = document.getElementById('stock');

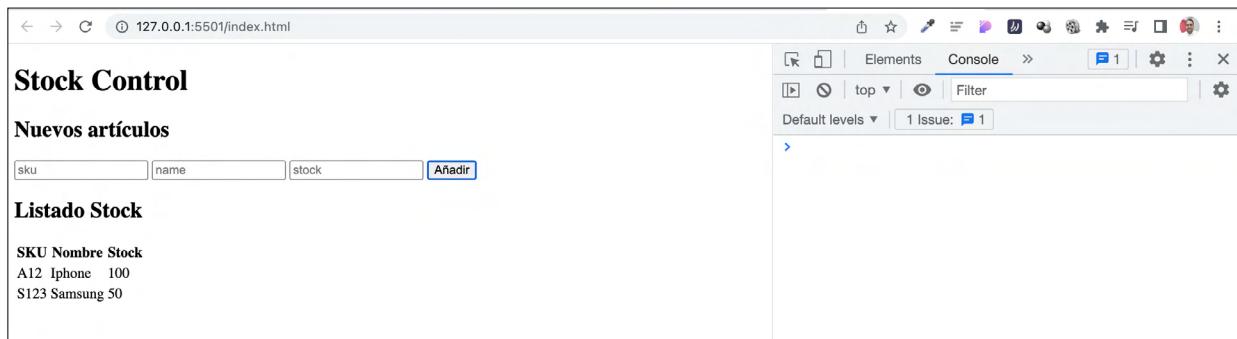
document.getElementById('addButton').addEventListener('click', () => {
  store.dispatch(addStockAction(skuInput.value, nameInput.value, stockInput.value));
  skuInput.value = '';
  nameInput.value = '';
  stockInput.value = '';
})

const showStock = () => {
  const items = store.getState();
  let stockList = document.getElementById('stockList');
  stockList.innerHTML = '';
  items.forEach(item => {
    let row = document.createElement('tr');
    let sku = document.createElement('td');
    let name = document.createElement('td');
    let stock = document.createElement('td');
    sku.appendChild(document.createTextNode(item.sku));
    name.appendChild(document.createTextNode(item.name));
    stock.appendChild(document.createTextNode(item.stock));
    row.appendChild(sku);
    row.appendChild(name);
    row.appendChild(stock);
    stockList.appendChild(row);
  })
}

store.subscribe(showStock);

```

Al ejecutar de nuevo nuestra aplicación, podemos comprobar en pantalla que se van actualizando los objetos en la tabla (Figura 2.3).



Pese a la simplicidad de nuestra práctica, hemos visto el ciclo completo de Redux, actuando finalmente en la vista de nuestro archivo index.html, en el que se actualizan automáticamente los registros introducidos en el *store*, y hemos implementado la parte de programación reactiva de Redux en esta suscripción.

Figura 2.3
Actualización de un elemento de tabla mediante el uso de suscripción en Redux.



Resumen

- Redux implementa los elementos *store*, *reducers* y *actions* para controlar el almacenamiento del estado y la actualización de los valores de este, con una colección de funciones para desarrollar y métodos de la librería diseñados para llevar a cabo el ciclo de manejo de estado de las aplicaciones.
- Las vistas consiguen su actualización de cambios en el estado a partir de suscripciones al *store* que podrán invocar funciones cuando cambie el estado en este como consecuencia del disparo de acciones.

3. Conectando Redux con React DevTools

Llega el momento de integrar Redux en React, uno de los *frameworks* en los que más utilizada es esta librería de manejo de estado de una aplicación, y para la cual dispone de más soporte y herramientas adicionales.

En primer lugar, implementaremos de manera práctica Redux en una aplicación React mediante el uso de las librerías específicas y con la suscripción de componentes al estado.

Posteriormente, instalaremos las herramientas de depuración para conseguir una integración total de la librería y su patrón, obteniendo información de todo el proceso de actualización del estado.

3.1. Redux en React

Existen diferentes maneras de instalar Redux en React, dependiendo de la arquitectura de directorios y archivos que necesitemos implementar; pero una de las más utilizadas es la propuesta por la herramienta Create-React-App, que implementa una estructura a partir de una plantilla o *template*.

Vamos a comprobarlo de manera práctica, para lo cual abrimos en cualquier ubicación de nuestro equipo una consola o terminal y escribimos el siguiente comando para generar el proyecto con la plantilla Redux:

```
npx create-react-app practica2 --template redux
```

Una vez finalizado el proceso, el proyecto se ha creado en el directorio `practica2` en Visual Studio Code y podemos observar que la estructura de archivos dispone de dos directorios, `app` y `features`, dentro del directorio `src` (Figura 3.1).

Veamos el contenido de ambos directorios: en el caso de `app`, el archivo que albergará el `store` de Redux; y, en el caso de `features`, un directorio por cada funcionalidad que haya que almacenar en el estado. Cada uno de esos directorios dispondrá de un archivo para las `actions` y los `reducers`, y otro con el componente.

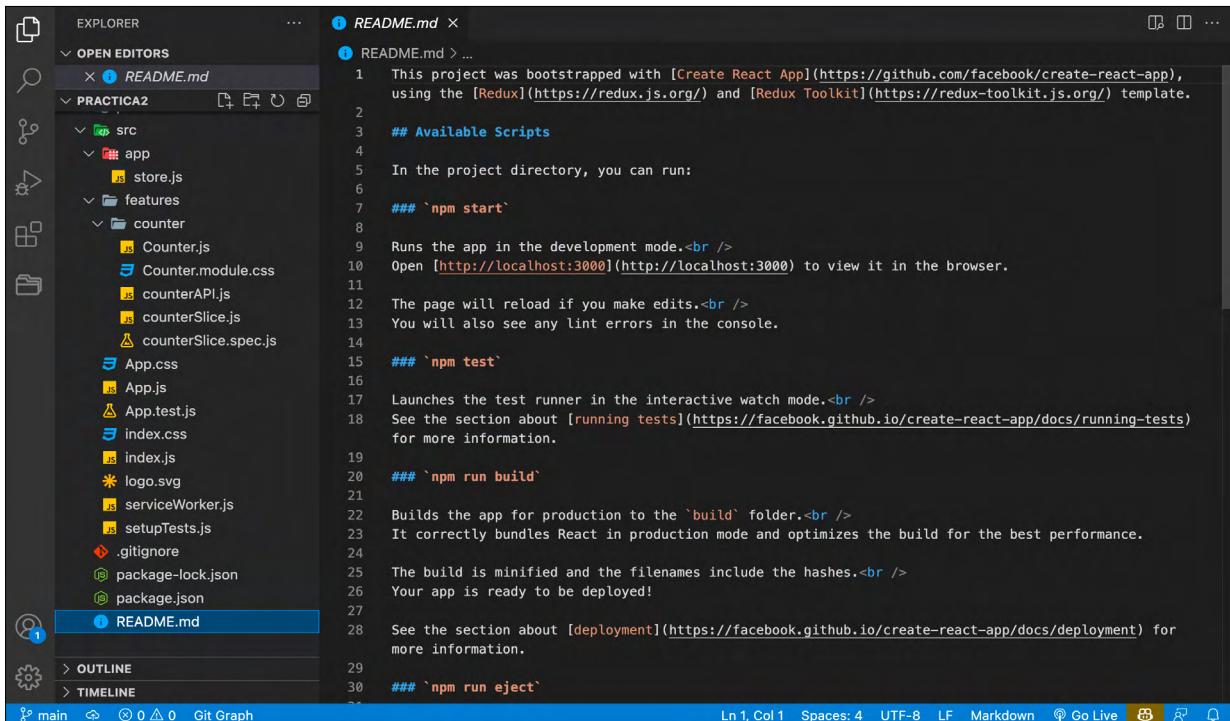


Figura 3.1

Estructura de una aplicación React con Redux a partir de su plantilla en Visual Studio Code.

Para comprender mejor su funcionamiento, vamos a replicar el estado almacenado en la práctica anterior relacionado con un *stock* de artículos, para lo cual eliminamos el directorio *counter* ubicado en *features* y vamos a crear un nuevo directorio llamado *stock*, en el que comenzamos creando otro nuevo directorio denominado *stockSlice.js* que contendrá los *reducers* y las *actions*. Añadimos el siguiente código en este:

```
import { createSlice } from '@reduxjs/toolkit';

const initialState = [];

export const stockSlice = createSlice({
  name: 'stock',
  initialState,
  reducers: {
    addItem: (state, action) => {
      state.push(action.payload);
    }
  }
});

export const { addItem, removeItem } = stockSlice.actions;

export const selectStock = (state) => state.stock;

export default stockSlice.reducer;
```

Como vemos, ya no se implementa el patrón Redux de manera directa, sino que usamos la librería *toolkit* para obtener un método “createSlice” que genera un objeto “stockSlice”. Este contiene el nombre del estado, el estado inicial en “initialState” y, en la propiedad “reducers”, un objeto con las diferentes actions que empleará el *reducer*.

Posteriormente, y para que puedan ser llamadas las *actions* desde los componentes, se exportan desde la propiedad “actions” de este objeto “stockSlice” y, de la misma forma, se exporta como “default” el *reducer* en la propiedad del mismo nombre. Para concluir, se exporta una función “selectStock” que devuelve el estado, para que los componentes que se quieran suscribir a este reciban los datos en cada actualización.

Con esta sintaxis incluimos el estado inicial, *actions* y *reducers*, así que el siguiente paso es definir el *store*; para ello, modificamos el código en el archivo store.js de la siguiente forma:

```
import { configureStore } from '@reduxjs/toolkit';
import stockReducer from '../features/stock/stockSlice';

export const store = configureStore({
  reducer: {
    stock: stockReducer,
  },
});
```

Con esto ya tenemos Redux implementado en la aplicación, pero para que se cargue en ella, necesitamos implementarlo en el archivo de punto de entrada, de manera que modificamos el archivo index.js para usar el componente “Provider” de la librería React-Redux de la siguiente forma:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import { store } from './app/store';
import { Provider } from 'react-redux';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
)
```

```

</React.StrictMode>,
document.getElementById('root')
);

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();

```

De esta forma ya tenemos el almacén de estado listo, así que vamos a generar un primer componente para actualizarlo. En el directorio stock generamos un componente en el archivo Stock.js con el siguiente código:

```

import React, { useState } from 'react';
import { useDispatch } from 'react-redux';
import { addItem } from './stockSlice';

export default function Stock() {
  const dispatch = useDispatch();
  const itemObject = {
    sku: '',
    name: '',
    stock: ''
  }
  const [item, setItem] = useState(itemObject);

  const handleChange = (e) => {
    setItem({
      ...item,
      [e.target.name]: e.target.value
    });
  }

  const handleSubmit = (e) => {
    e.preventDefault();
    dispatch(addItem(item));
    setItem(itemObject);
  }

  return (
    <>
      <h1>Stock Control</h1>
      <h2>Nuevos artículos</h2>
      <form onSubmit={handleSubmit}>
        <input type="text"
          name="sku"
          value={item.sku}

```

```

        placeholder="sku"
        onChange={handleChange}/>
      <input type="text"
        name="name"
        value={item.name}
        placeholder="name"
        onChange={handleChange}/>
      <input type="number"
        name="stock"
        value={item.stock}
        placeholder="stock"
        onChange={handleChange}/>
      <button type='submit'>Añadir</button>
    </form>
  )
}

```

Como podemos observar, se trata de un formulario que obtiene las tres propiedades de un objeto de artículo, y cuando se pulsa en el botón “Añadir”, utiliza el hook “useDispatch()” para añadir el objeto con la acción “addItem()”. Para implementar este componente en la aplicación, modificamos el archivo App.js del modo habitual:

```

import React from 'react';
import ListStock from './features/stock/ListStock';
import Stock from './features/stock/Stock';

function App() {
  return (
    <div className='container'>
      <Stock />
    </div>
  );
}

export default App;

```

Ahora nos queda añadir unos estilos CSS, para lo cual modificamos el archivo index.css con el siguiente código empleado en otros proyectos:

```

@import url('https://fonts.googleapis.com/css?family=Lato&display=swap');

* {
  padding: 0;
  margin: 0;
}

```

```
    box-sizing: border-box;
}

body {
    font-family: 'Lato', sans-serif;
    color: #718096;
}

.container {
    max-width: 940px;
    margin: 0 auto;
    padding: 1rem;
}

form {
    max-width: 440px;
    width: 100%;
    margin: 2rem auto;
}

.row {
    display: flex;
    flex-direction: column;
}

.row-buttons {
    display: flex;
    justify-content: flex-end;
}

label {
    width: 100%;
    display: inline-block;
    margin-bottom: 0.5rem;
    color: #545353;
}

input {
    width: 100%;
    font-size: 1rem;
    padding: 0.5rem 1rem;
    margin: 0.5rem 0;
    border: none;
    box-shadow: inset 0 2px 4px 0 rgba(0,0,0,0.08);
    color: #6a6a6a;
    font-family: 'Lato', sans-serif;
}

button {
    font-size: 1rem;
    padding: 0.5rem;
```

```

margin: 1rem 0;
color: white;
background-color: #008489;
border: #008489 solid 1px;
cursor: pointer;
transition: background-color 400ms ease-in-out;

}

button[disabled] {
background-color: #6a6a6a;
cursor: not-allowed;
}

table {
width: 100%;
margin-bottom: 24px;
}

th {
padding: 10px 20px;
background-color: #008489;
color: white;
text-align: left;
}

td {
padding: 10px 20px;
border: 1px solid #008489;
}

```

A continuación, levantamos nuestra aplicación con el comando “npm start” y podemos comprobar en el navegador la aplicación (Figura 3.2).

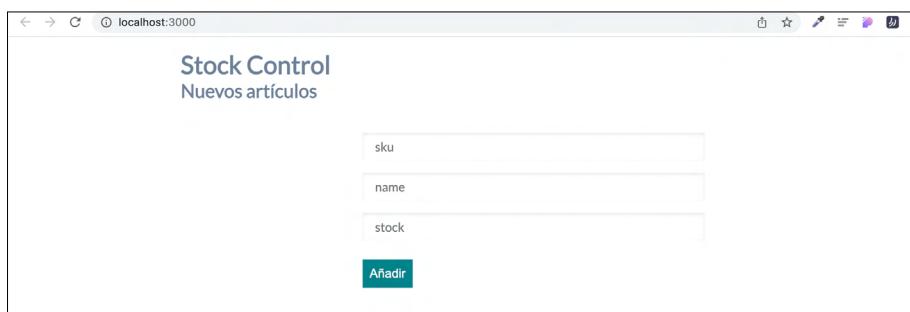


Figura 3.2
Componente con Redux en una aplicación React en el navegador.

Podemos comprobar el funcionamiento del componente, pero, como no disponemos en las herramientas React Dev Tools del acceso a Redux, no podemos evaluar cómo se actualiza el estado, salvo que uti-

lizáramos procesos de depuración. Así que, a continuación, veremos cómo añadir una nueva herramienta para este propósito.

3.2. Conectando Redux con React DevTools

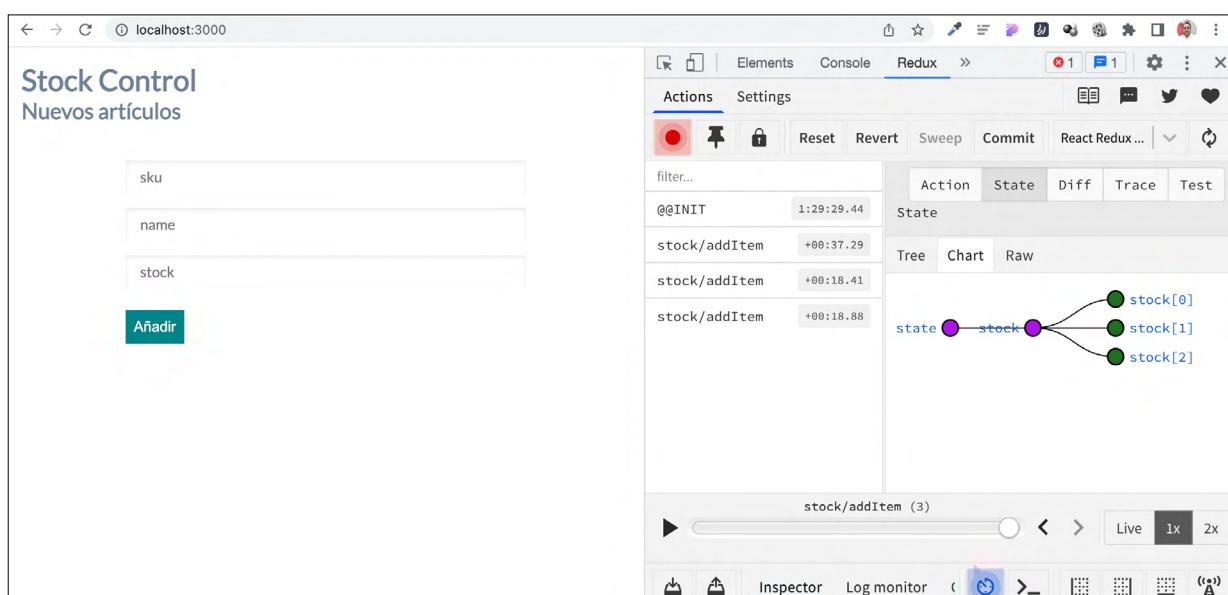
Figura 3.3
Instalación de la extensión Redux DevTools en Chrome.

Para controlar en todo momento Redux y su estado, disponemos de la extensión para Chrome Redux DevTools. Simplemente introduciendo este término en el buscador, accederemos a la primera coincidencia y la instalaremos (Figura 3.3).



Figura 3.4
Información del estado en la extensión Redux DevTools en Chrome.

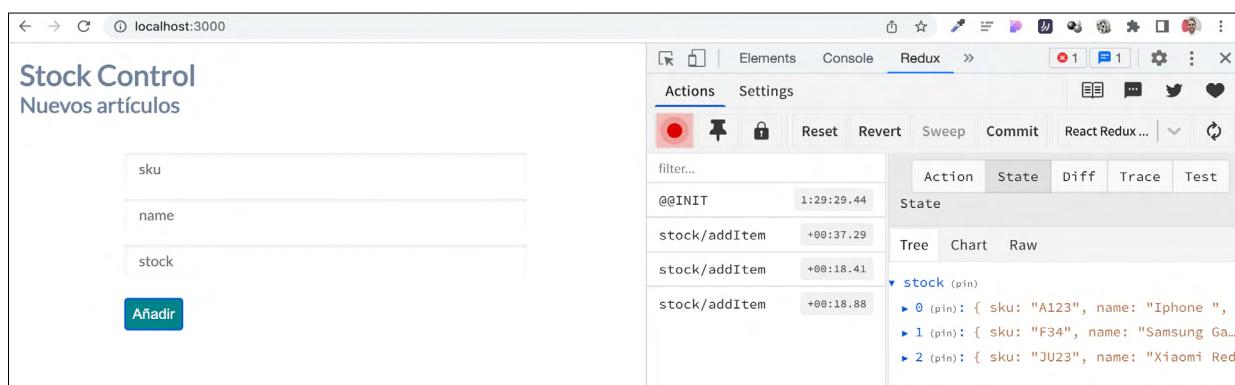
Ahora podemos usar el formulario de nuestra aplicación para añadir varios artículos, y podremos comprobar que, en la pestaña Redux de las herramientas de desarrollador, estas DevTools nos proporcionan toda la información sobre el estado, incluyendo la secuencia temporal (Figura 3.4).



Como normalmente el estado de una aplicación se convierte en un objeto complejo, las Redux DevTools nos proporcionan también una versión gráfica de la información (Figura 3.5), muy útil para comprender los modelos de datos implementados.

Figura 3.5

Información gráfica del estado en la extensión Redux DevTools en Chrome.



3.3. Suscripción al estado

Para completar el ciclo Redux, podemos usar el estado para que los componentes puedan suscribirse y emplear los datos en la vista o bien en la lógica de cada uno de ellos.

Para comprobarlo, vamos a crear un nuevo componente que utilice los artículos almacenados en el Store Redux y renderizarlos en una tabla. Para ello, en el directorio stock creamos el archivo ListStock.js con el siguiente código:

```
import React from 'react';
import { useSelector } from 'react-redux';
import { selectStock } from './stockSlice';

export default function ListStock() {
  const stock = useSelector(selectStock);

  return (
    <table>
      <thead>
        <tr>
          <th>SKU</th>
          <th>Nombre</th>
          <th>Stock</th>
        </tr>
      </thead>
      <tbody>
        {stock.map(item => {
          return (
            <tr>
              <td>{item.sku}</td>
              <td>{item.name}</td>
              <td>{item.stock}</td>
            </tr>
          )
        })
      </tbody>
    </table>
  );
}
```

```

        <tr key={item.sku}>
          <td>{item.sku}</td>
          <td>{item.name}</td>
          <td>{item.stock}</td>
        </tr>
      )
    )}
</tbody>
</table>
)
}

```

Como vemos, importamos “selectStock”, la función que devuelve el estado, y la pasamos al hook “useSelector()”; de esta forma, el componente queda suscrito a los cambios del estado y provoca la actualización automática del componente. A continuación, añadimos el componente al archivo App.js de la siguiente forma:

```

import React from 'react';
import ListStock from './features/stock/ListStock';
import Stock from './features/stock/Stock';

function App() {
  return (
    <div className='container'>
      <Stock />
      <ListStock />
    </div>
  );
}

export default App;

```

Ahora podemos comprobar en la aplicación (Figura 3.6) que el componente es actualizado automáticamente con cada cambio en el estado.

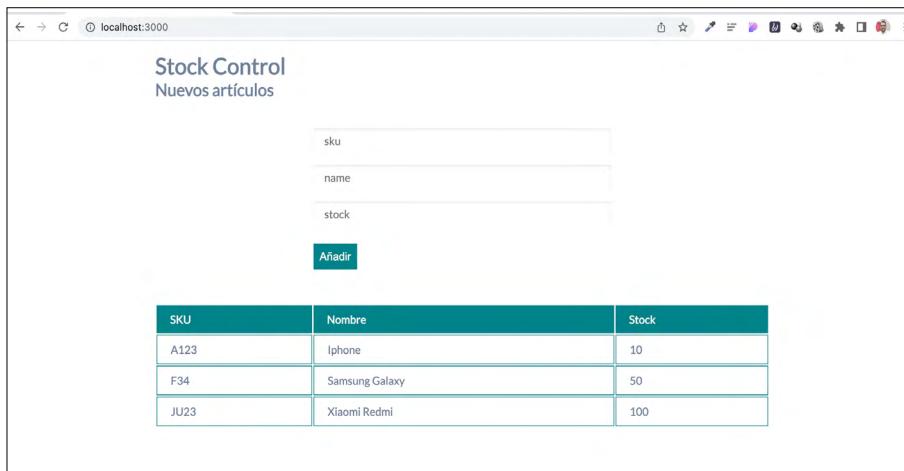


Figura 3.6

Componente React suscrito al estado Redux en el navegador.

Podemos además establecer bidireccionalidad en la comunicación entre los componentes y el estado. Por ejemplo, vamos a añadir una *action* a Redux para eliminar un artículo por su código SKU; para ello, modificamos el archivo stockSlice.js de la siguiente forma:

```
import { createSlice } from '@reduxjs/toolkit';

const initialState = [];

export const stockSlice = createSlice({
  name: 'stock',
  initialState,
  reducers: {
    addItem: (state, action) => {
      state.push(action.payload);
    },
    removeItem: (state, action) => {
      const index = state.findIndex(item => item.sku === action.payload.sku);
      state.splice(index, 1);
    }
  }
});

export const { addItem, removeItem } = stockSlice.actions;

export const selectStock = (state) => state.stock;

export default stockSlice.reducer;
```

Ahora vamos a implementar en el componente ListStock.js la llamada a la *action*, de nuevo con el *dispatch* asociado a una función que podemos introducir en cada fila de un objeto. Modificamos el código del archivo de la siguiente forma:

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { selectStock, removeItem } from './stockSlice';

export default function ListStock() {
  const dispatch = useDispatch();
  const stock = useSelector(selectStock);

  const handleRemoveItem = sku => {
    dispatch(removeItem({sku}));
  }

  return (
    <table>
      <thead>
        <tr>
          <th>SKU</th>
```

```

        <th>Nombre</th>
        <th>Stock</th>
        <th></th>
    </tr>
</thead>
<tbody>
{stock.map(item => {
    return (
        <tr key={item.sku}>
            <td>{item.sku}</td>
            <td>{item.name}</td>
            <td>{item.stock}</td>
            <td onClick={() => handleRemoveItem(item.sku)}>Eliminar</td>
        </tr>
    )
})
</tbody>
</table>
)
}

```

Y, finalmente, comprobamos en el navegador (Figura 3.7) que ahora podemos disparar la acción para eliminar un objeto del estado y, al mismo tiempo, el componente suscrito recibe y ejecuta la actualización.

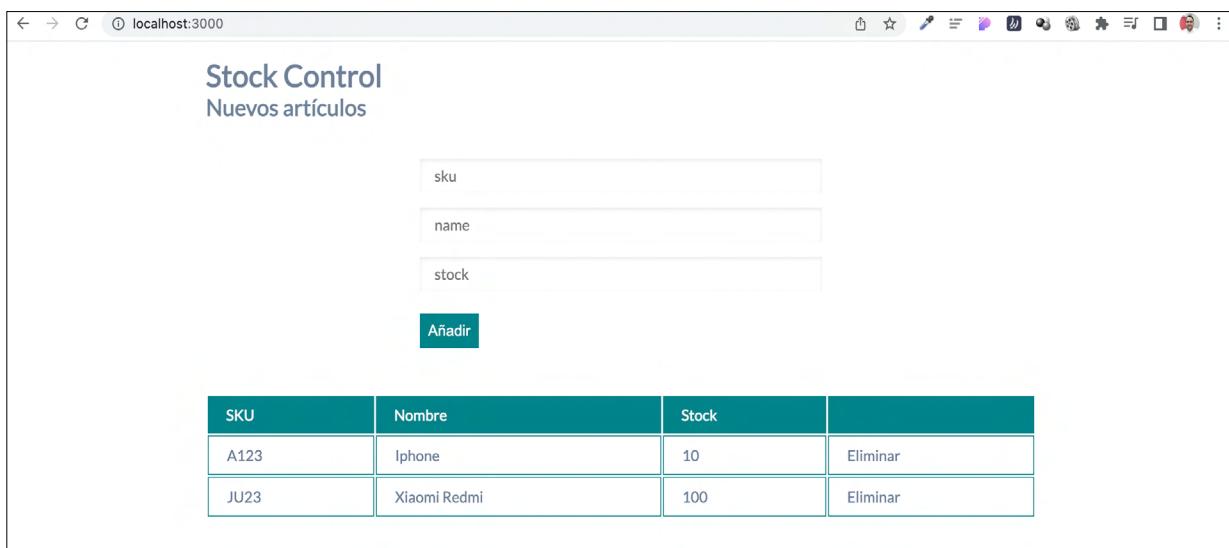


Figura 3.7 Componente React con comunicación bidireccional al estado Redux en el navegador.

Con la implementación de Redux en React, conseguimos un mecanismo fiable de almacenamiento complejo para todo tipo de soluciones y requisitos en nuestras aplicaciones.



Resumen

- Una de las formas de instalar Redux en React es mediante la plantilla de la herramienta Create-React-App, que implementa una estructura de directorios y archivos para implementar los elementos de *store*, *reducers* y *actions*.
- El desarrollo de los elementos de *store*, *reducers* y *actions*, así como la suscripción al estado de Redux, se lleva a cabo con las librerías adicionales React Redux y Redux Toolkit.
- La depuración del estado implementado por Redux en React se puede llevar a cabo de manera sencilla mediante la extensión para navegadores Redux DevTools.

4. Asincronía en Redux

Continuamos con el desarrollo con Redux, un almacén de estado para todo tipo de aplicaciones, pero especialmente diseñado para aquellas cuya complejidad requiere de patrones confiables y escalables.

Comenzaremos la unidad escalando el estado de nuestra aplicación para comprobar que, con pequeños pasos de refactorización del estado, podemos cumplir con los requisitos necesarios para la evolución de nuestras soluciones.

Posteriormente, implementaremos asincronía en Redux, condición imprescindible en las aplicaciones, puesto que siempre tendremos comunicaciones con latencia en el uso de datos.

4.1. Escalando el estado de Redux en React

Una vez que tenemos implementado Redux en una aplicación React, es muy sencillo escalar el estado para añadir nuevas propiedades que controlar y almacenar. Por ejemplo, en nuestro proyecto podemos necesitar disponer de una propiedad booleana que indique si la aplicación se encuentra en un proceso asíncrono, con el fin de que pueda ser consumida por los componentes para usarla en una lógica que informe al usuario final.

Para comprobarlo, vamos a abrir nuestro proyecto del directorio práctica2 en Visual Studio Code, y comenzaremos por cambiar el nombre al estado en el archivo store.js, modificando el siguiente bloque:

```
...
reducer: {
  stockState: stockReducer,
},
...
```

A continuación, vamos a modificar la implementación de *actions* y *reducers*; por tanto, cambiamos todo el código del archivo stockSlice.js por el siguiente:

```

import { createSlice } from '@reduxjs/toolkit';

const initialState = {
  stock: [],
  loading: false,
}

export const stockSlice = createSlice({
  name: 'stockState',
  initialState,
  reducers: {
    addItem: (state, action) => {
      state.stock.push(action.payload);
    },
    removeItem: (state, action) => {
      const index = state.stock.findIndex(item => item.sku === action.payload.sku);
      state.stock.splice(index, 1);
    },
  },
});

export const { addItem, removeItem } = stockSlice.actions;

export const selectStock = (state) => state.stockState.stock;
export const selectLoading = (state) => state.stockState.loading;

export default stockSlice.reducer;

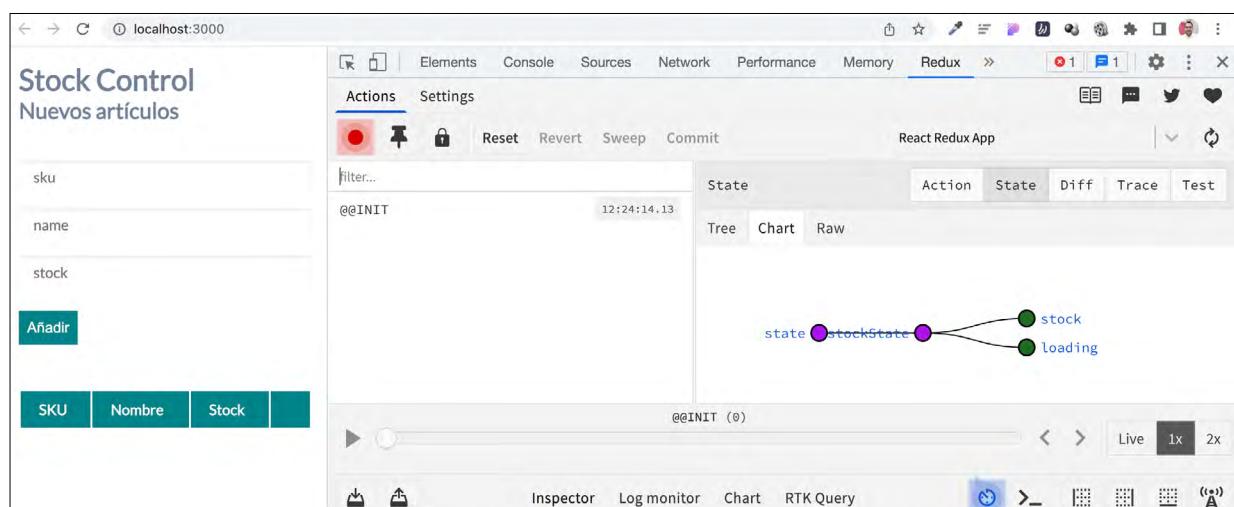
```

Comprobamos ahora que nuestro estado es un objeto con dos propiedades, el *stock* con el conjunto de datos de artículos y una booleana denominada “loading” que podrá ser suscrita en cualquier componente mediante la función “selectLoading”.

Al iniciar nuestra aplicación, gracias a las Redux DevTools podemos comprobar el nuevo árbol de estado de la aplicación (Figura 4.1).

Figura 4.1

Árbol de estado de la aplicación en las Redux DevTools.





Para saber más

Los almacenes de estado como el usado en el patrón-librería Redux son denominados “fuentes de la verdad”, ya que su implementación debe cumplir siempre que los valores proporcionados sean exactamente los que tiene la aplicación en ese momento.

Ahora podremos utilizar esta propiedad “loading” en los siguientes pasos para almacenar el estado correspondiente a peticiones asíncronas en nuestra aplicación.

4.2. Asincronía en Redux

Prácticamente todas las aplicaciones tienen procesos asíncronos, comunicaciones con API para recibir o enviar datos con los que cumplir con las reglas de negocio; por tanto, la integración de esos procesos con Redux es esencial, ya que nuestro almacén de estado debe contener en todo momento los valores de cada propiedad en ese momento temporal.

Vamos a simular en nuestra aplicación un proceso asíncrono por el que, desde nuestro componente de formulario, se envía el artículo; este objeto sería enviado a una API y con la respuesta correcta deberá ser incluido en el *store* de React. Como no disponemos de API, usaremos “setTimeout” para simular la latencia de la operación; por tanto, vamos a crear en el directorio stock un archivo stockAPI.js con el siguiente código:

```
export function fetchStock(item) {
  return new Promise((resolve) =>
    setTimeout(() => resolve({ item }), 1500)
  );
}
```

Con esta simulación de un servicio de llamada a API, vamos a implementar el manejo de asincronía en Redux. Para ello, en el archivo stockSlice.js usaremos una nueva función que utiliza el método “createAsyncThunk()”, el cual se encarga de llamar a la función de la API y retornar la respuesta. Al manejar la asincronía de la operación, la *callback* empleada se implementa con “async-await”:

```
export const stockAsync = createAsyncThunk(
  'stock/fetchStock',
  async (item) => {
    const response = await fetchStock(item);
    return response.item;
  }
);
```

Una vez añadida esta función, la usaremos en el reducer de manera diferente a los anteriores, ya que dispondrá de un estado “pending” y “full”

filled”, que durante el proceso asíncrono nos permite cambiar el estado en primer lugar en la propiedad “loading” y posteriormente recibirá la respuesta con el artículo y actualizará el estado de “stock”. Modificamos definitivamente todo el código de stockSlice.js de la siguiente forma:

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';
import { fetchStock } from './stockAPI';

const initialState = {
  stock: [],
  loading: false,
}

export const stockAsync = createAsyncThunk(
  'stock/fetchStock',
  async (item) => {
    const response = await fetchStock(item);
    return response.item;
  }
);

export const stockSlice = createSlice({
  name: 'stockState',
  initialState,
  reducers: {
    addItem: (state, action) => {
      state.stock.push(action.payload);
    },
    removeItem: (state, action) => {
      const index = state.stock.findIndex(item => item.sku === action.payload.sku);
      state.stock.splice(index, 1);
    },
    extraReducers: (builder) => {
      builder
        .addCase(stockAsync.pending, (state) => {
          state.loading = true;
        })
        .addCase(stockAsync.fulfilled, (state, action) => {
          state.loading = false;
          state.stock.push(action.payload);
        });
    },
  },
);

export const { addItem, removeItem } = stockSlice.actions;

export const selectStock = (state) => state.stockState.stock;

export default stockSlice.reducer;
```

Ahora vamos a modificar el componente suscrito al estado para comprobar el uso de la asincronía y la propiedad “loading”. Cambiamos el código de Stock.js por el siguiente:

```
import React, { useState } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { stockAsync, selectLoading } from './stockSlice';

export default function Stock() {
  const dispatch = useDispatch();
  const itemObject = {
    sku: '',
    name: '',
    stock: ''
  }
  const loading = useSelector(selectLoading);
  const [item, setItem] = useState(itemObject);

  const handleChange = (e) => {
    setItem({
      ...item,
      [e.target.name]: e.target.value
    });
  }

  const handleSubmit = (e) => {
    e.preventDefault();
    dispatch(stockAsync(item));
    setItem(itemObject);
  }

  return (
    <>
      <h1>Stock Control</h1>
      <h2>Nuevos artículos</h2>
      <form onSubmit={handleSubmit}>
        <input type="text"
              name="sku"
              value={item.sku}
              placeholder="sku"
              onChange={handleChange}/>
        <input type="text"
              name="name"
              value={item.name}
              placeholder="name"
              onChange={handleChange}/>
        <input type="number"
              name="stock"
              value={item.stock}
              placeholder="stock"/>
    </>
  );
}
```

```

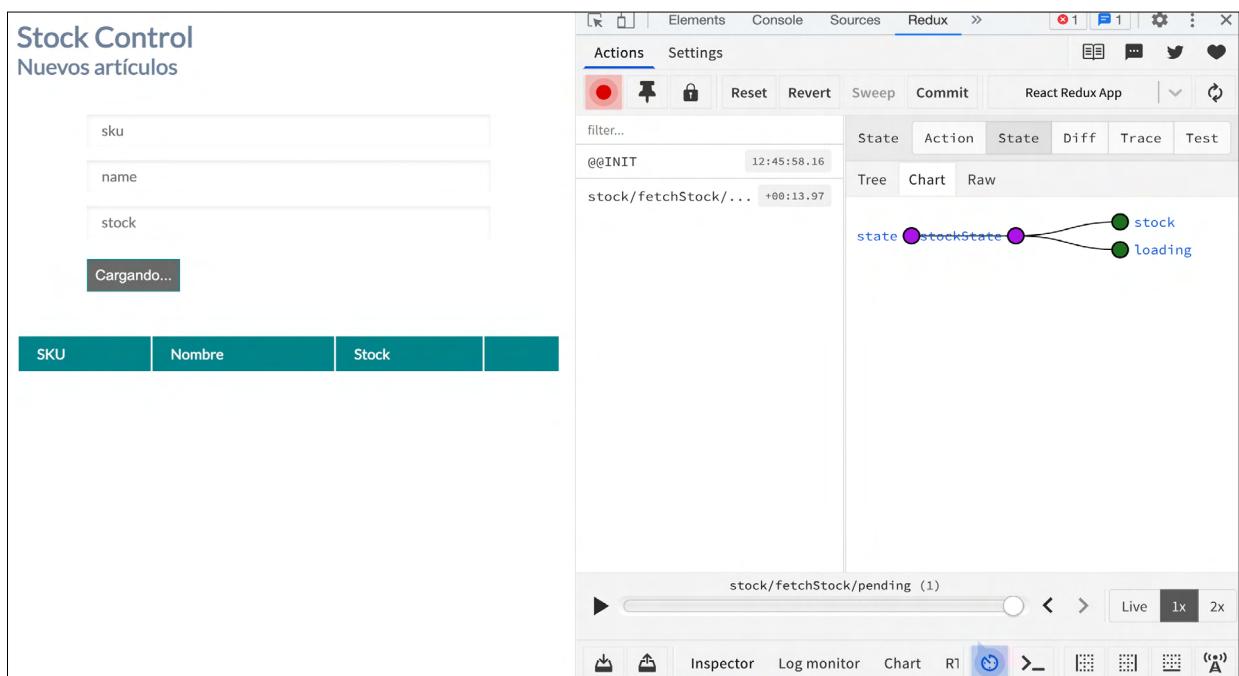
        onChange={handleChange}/>
      {loading ?
        <button disabled>Cargando...</button>
        :
        <button type="submit">Añadir</button>
      }
    </form>
  )
}

```

Ahora, cuando se produzca un envío del objeto tras pulsar en el botón “Añadir”, veremos que se implementa la lógica a partir del valor de la booleana “loading” en pantalla (Figura 4.2).

Figura 4.2

Implementación de asincronía en Redux en el navegador.



Con este mecanismo dispondremos de la solución a cualquier proceso asíncrono cuyos datos necesiten estar disponibles en el estado para ser suscritos por cualquier componente.



Resumen

- Gracias a los objetos de estado, el escalado de almacenamiento en Redux para cualquier nuevo requisito de la aplicación es fácilmente implementable en React manteniendo la funcionalidad del desarrollo existente.
- La asincronía en Redux en aplicaciones React se puede implementar gracias al uso del método “createAsyncThunk()” de la librería adicional Toolkit y su empleo en los *reducers*.

5. Programación reactiva, concepto de *observable* y *observer*

Llega el momento de conocer la programación reactiva de la mano de otra librería, RxJS, una alternativa a Redux empleada especialmente por Angular enfocada a la suscripción más que al estado y que está teniendo un éxito enorme en multitud de aplicaciones complejas.

Comenzaremos la unidad describiendo qué es RxJS y cuáles son sus principales características. Posteriormente, utilizaremos RxJS en un proyecto JavaScript plano para poder implementar los conceptos de *observable* y *observer*.

5.1. Librería RxJS y el concepto de *observable* y *observer*

RxJS, cuyas iniciales provienen de unir reactividad (“Rx”) con JavaScript (“JS”), es una librería para este lenguaje que implementa la programación reactiva a través del manejo de una sucesión de eventos a los que es posible suscribirse gracias al patrón *observable-observer*.

En otras palabras, lo que buscan el patrón *observable-observer* y la programación reactiva es reaccionar ante el flujo de los eventos producidos, de manera que la producción de un evento en un punto de la aplicación pueda ser recogida en otro punto sin que estas dos partes estén conectadas mediante la invocación de funciones o métodos. En este sentido, en el patrón contaremos con tres elementos principales que se pueden describir de la siguiente manera:

- **Observable:** es el objeto que puede ser observado, del que se puede obtener un flujo de valores a partir de los eventos enlazados, que provocan que contenga una colección de eventos futuros.
- **Observer:** es el encargado de observar; se trata de un objeto que incluye una colección de funciones *callback* que recogerán el flujo de datos contenido en el *observable*.
- **Subject:** es el emisor de eventos disparado por la lógica de la aplicación, capaz de crear el flujo de datos incorporados al *observable* para que, en la suscripción a este, el *observer* pueda recoger los valores.



Para saber más

Aunque pudiera parecer que Redux y RxJS sirven para lo mismo porque implementan programación reactiva, Redux está más enfocada al manejo del estado, mientras que RxJS tiene un propósito más amplio y general.

En este plano puramente teórico, es difícil obtener una aproximación de lo que realmente conseguimos con el patrón, con lo cual es siempre interesante llevar a cabo dos pasos: en primer lugar, la implementación en un proyecto JavaScript plano para poner en práctica la sintaxis de estos elementos; y, posteriormente, y para una mayor comprensión práctica, implementarlo en una aplicación con componentes con una lógica de negocio.

5.2. *Observables* y *observer* en JavaScript con RxJS

Para comprender los elementos anteriormente expuestos, vamos a crear en cualquier ubicación de nuestro equipo un directorio denominado `practica3` y lo abrimos en Visual Studio Code. Creamos un archivo `index.html` y un directorio en la raíz denominado `js`, en el que añadimos el archivo `main.js`.

Ahora, en `index.html`, añadimos el siguiente código con el enlace al CDN de la librería RxJS y un *script* con el archivo JavaScript de nuestro proyecto:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>RxJS</title>
    <script src="https://unpkg.com/rxjs@^7/dist/bundles/rxjs.umd.min.js"></script>
  </head>
  <body>
    <script src=".//js/main.js"></script>
  </body>
</html>
```

A continuación, vamos a escribir el código más sencillo que se puede implementar con este patrón: la instancia de un *observable* en una constante “observable”, cuyo flujo de datos es creado en el propio objeto con su método “next”, que devolverá diferentes valores numéricos. Añadimos el siguiente código en el archivo `main.js`:

```
const { Observable } = rxjs;
const numberObservable = new Observable((observer) => {
  observer.next(5);
  observer.next(10);
  observer.next(15);
});
```

Una vez que tenemos el *observable*, en el propio archivo podemos suscribirnos a él con su método “*subscribe*”, que recibe como argumento el *observer*, es decir, una función *callback* que recibirá como argumento los datos del flujo del *observable* para, en este sencillo caso, imprimirlas por la consola. Completamos el código de la siguiente forma:

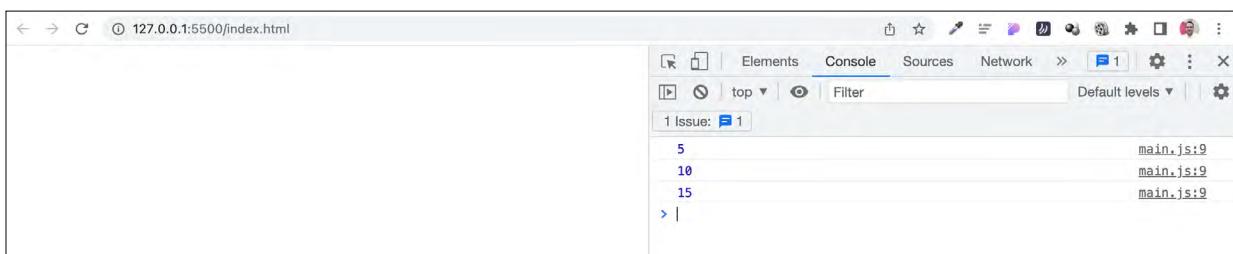
```
const { Observable } = rxjs;
const numberObservable = new Observable((observer) => {
  observer.next(5);
  observer.next(10);
  observer.next(15);
});

numberObservable.subscribe(value => {
  console.log(value)
});
```

Si abrimos index.html con Live Server, podremos comprobar en la consola del navegador (Figura 5.1) que el *observer* resuelve cada evento recogido en la suscripción y contenido en el *observable*.

Figura 5.1

Resultado en consola de la suscripción a un *observable* RxJS.



La siguiente implementación de RxJS es también muy básica, pero en ella vamos a reflejar mejor el *observer* como un conjunto de *callbacks* que permiten leer de nuevo los valores del flujo, capturar un error en caso de que se produzca y conocer cuándo ha finalizado el flujo. Además, vamos a implementar **asincronía**, ya que este patrón se usa frecuentemente cuando tenemos este tipo de procesos. Cambiamos todo el código del archivo main.js por el siguiente:

```
const { Observable } = rxjs;

const observable = new Observable(subscriber => {
  subscriber.next(5);
  subscriber.next(10);
  subscriber.next(15);
```

```

    setTimeout(() => {
      subscriber.next(20);
      subscriber.complete();
    }, 2000);
  });

console.log('Antes de la suscripción');
observable.subscribe({
  next: value => console.log(value),
  error: err => console.error(err),
  complete: () => console.log('Suscripción finalizada')
});
console.log('Después de la suscripción');

```

Podemos comprobar por la consola del navegador (Figura 5.2) que se

Figura 5.2

Resultado en consola de la suscripción completa a un *observable* RxJS.

produce el flujo esperado y además se notifica que este ha finalizado gracias al método “complete()” en el *observable*. Además, el *observer* ahora es un objeto pasado a la suscripción con las diferentes funciones *callback* para cada caso.



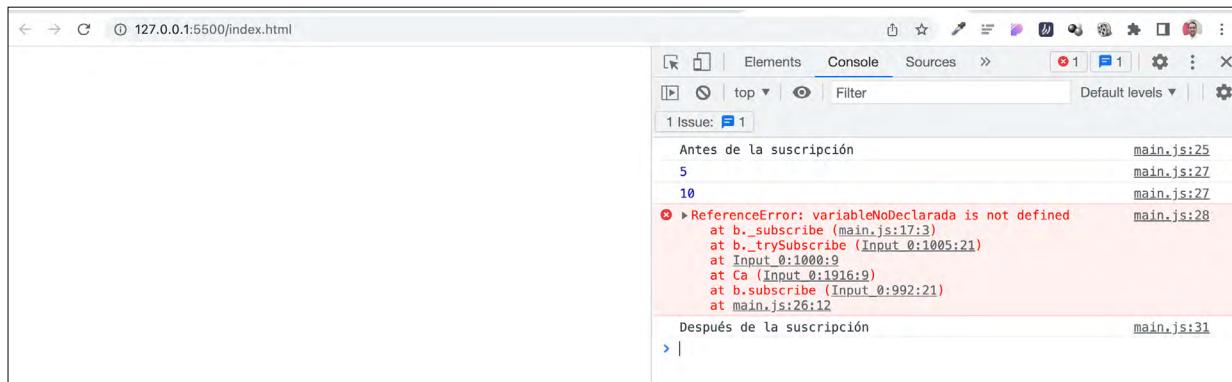
Para comprobar también el caso de que se produzca un error en la lógica que genera el flujo de datos del *observable*, podemos modificar este de la siguiente forma:

```

const observable = new Observable(subscriber => {
  subscriber.next(5);
  subscriber.next(10);
  variableNoDeclarada++;
  subscriber.next(15);
  setTimeout(() => {
    subscriber.next(20);
    subscriber.complete();
  }, 2000);
});
...

```

Y podemos comprobar en la consola (Figura 5.3) que la función *callback* que recibe el error lo captura, y que el resto del flujo tras el error no llega a producirse.



Las posibilidades de la librería RxJS son enormes: además del patrón *observable-observer*, dispone de decenas de operadores y una colección de métodos para trabajar la programación reactiva en nuestras aplicaciones. Veremos más adelante cómo implementar en su uso en componentes esta potente solución de *software*.

Figura 5.3

Captura de error en consola de la suscripción a un *observable* RxJS.



Resumen

- RxJS es una librería para el lenguaje JavaScript que implementa la programación reactiva a través del manejo de una sucesión de eventos a los que, gracias al patrón *observable-observer*, es posible suscribirse.
- El *observable* es un objeto con una sucesión de eventos en flujo al cual es posible suscribirse para, mediante el *observer*, obtener los datos o eventos producidos gracias a la colección de funciones *callback* de este último.

6. Estrategias de uso con React

Aunque la manera habitual de introducir programación reactiva en aplicaciones React es mediante el uso de Redux, es posible y una buena alternativa introducir esta funcionalidad a través de RxJS.

En esta unidad vamos a comenzar por crear un proyecto con React e instalar la librería RxJS para implementar un caso de uso habitual en la programación reactiva: que un componente aislado se pueda suscribir a cambios en un objeto de estado.

Posteriormente, desarrollaremos de forma práctica el concepto de *observable*, *observer* y *subject* desde un servicio centralizador de datos enlazable con los diferentes componentes de la aplicación.

6.1. Librería RxJS y React

Como ya sabemos, RxJS es una librería de implementación para JavaScript y es totalmente agnóstica del *framework* que haya que utilizar, además de disponer de una documentación muy abundante y de calidad (Figura 6.1).

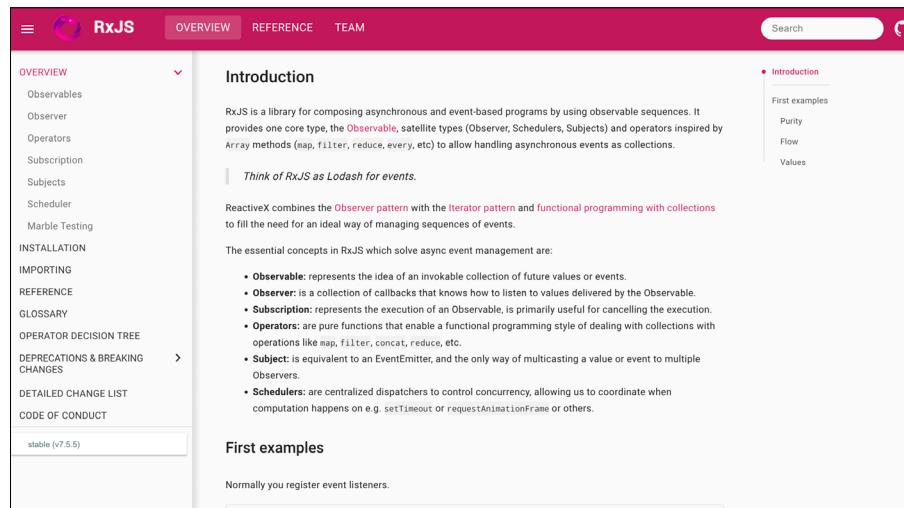
The screenshot shows the RxJS documentation website. The top navigation bar includes links for 'OVERVIEW', 'REFERENCE', and 'TEAM'. A search bar is located at the top right. The main content area is titled 'Introduction' and contains text about RxJS being a library for composing asynchronous and event-based programs using observable sequences. It defines terms like Observable, Observer, Operators, Subscription, Subject, Scheduler, and Marble Testing. It also covers Installation, Importing, Reference, Glossary, Operator Decision Tree, Deprecations & Breaking Changes, Detailed Change List, and Code of Conduct. A sidebar on the right lists 'First examples', 'Purity', 'Flow', and 'Values'. A note at the bottom left says 'stability (v7.5.5)'.

Figura 6.1

Sitio web con documentación de RxJS.

La forma de implementarla en React es mediante su instalación con el gestor de paquetes npm. Para comprobarlo, vamos a crear en cualquier ubicación de nuestro equipo un nuevo proyecto React llamado practica4 con el siguiente comando:

```
npm create-react-app practica4
```

A continuación, abrimos el directorio en Visual Studio Code y, en la terminal integrada, instalamos RxJS con el siguiente comando:

```
npm i rxjs --save
```

Una vez que tenemos nuestro proyecto creado, comenzamos a trabajar en una propuesta que consiste en la simulación de una tienda de comercio electrónico. Dispondremos en la pantalla principal de una rejilla con tarjetas de artículos que, clicando en un botón, podrán ser añadidos a la simulación de una cesta.

Antes de la creación de la primera vista, vamos a crear un servicio para almacenar los artículos que se añaden a la cesta. Comenzamos, por tanto, creando un directorio llamado services en el directorio src, y dentro de este, creamos un archivo itemsService.js con el siguiente código:

```
let items = [];

export function addItem(item) {
    items.push(item);
}
```

Como vemos, se trata de un archivo muy sencillo en el que simplemente disponemos de un *array* para almacenar los elementos añadidos a la cesta y una función para añadir elementos al mismo.

A continuación, creamos un directorio llamado pages en src y, dentro del mismo, un archivo GridItems.js donde vamos a añadir, a partir de un *array* de artículos, una serie de tarjetas con la presentación de estos datos y un botón que permita añadir estos artículos a una cesta. Para ello, añadimos el siguiente código:

```
import React, {useState} from 'react'
import { addItem } from '../services/itemsServices'

export default function ItemsGrid() {

  const [items, setItems] = useState([
    {sku: 'A123', name: 'Pace', brand: 'Adidas', price: 40},
    {sku: 'N456', name: 'Court', brand: 'Nike', price: 65},
```

```

        {sku: 'NW67', name: 'Essential', brand: 'New Balance', price: 55},
    ])

const handleAddItem = (sku) => {
    addItem(sku);
}

return (
    <div className="container flex">
        {items.map(item => {
            return (
                <div className="card" key={item.sku}>
                    <p>{item.brand} {item.name}</p>
                    <p>{item.price}</p>
                    <div className="buttons-row">
                        <button onClick={() =>
                            handleAddItem(item.sku)}>Add</button>
                    </div>
                </div>
            )
        })}
    </div>
)
}

```

Ahora, añadimos el componente “ItemsGrid” al componente raíz, modificando el archivo App.js de la siguiente forma:

```

import ItemsGrid from './pages/ItemsGrid';

function App() {
    return (
        <>
            <ItemsGrid />
        </>
    );
}

export default App;

```

Y, finalmente, añadimos los estilos CSS al archivo global, index.css, de la siguiente forma:

```
@import url('https://fonts.googleapis.com/css?family=Lato&display=swap');
```

```
* {
  padding: 0;
  margin: 0;
  box-sizing: border-box;
}

body {
  font-family: 'Lato', sans-serif;
  color: #718096;
}

.container {
  max-width: 940px;
  margin: 0 auto;
  padding: 1rem;
}

.container.flex {
  display: flex;
  justify-content: space-around;
}

.card {
  width: 440px;
  border-radius: 0.5rem;
  box-shadow: 2px 2px 4px lightgray;
  margin: 1rem;
  padding: 1rem;
}

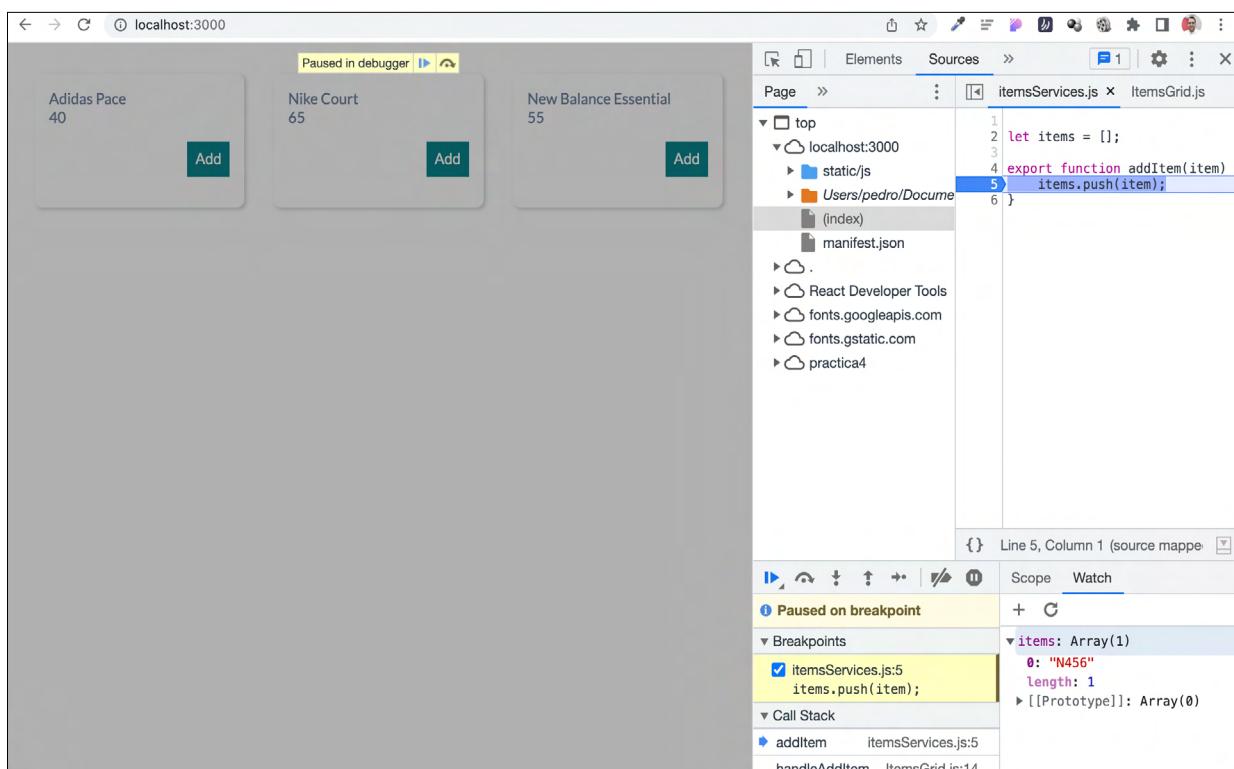
.buttons-row {
  display: flex;
  justify-content: flex-end;
}

button {
  font-size: 1rem;
  padding: 0.5rem;
  margin: 1rem 0;
  color: white;
  background-color: #008489;
  border: #008489 solid 1px;
  cursor: pointer;
  transition: background-color 400ms ease-in-out;
}
```

Ahora ya podemos levantar nuestra aplicación con el habitual comando “npm start” y, en el navegador, comprobar que se renderizan las tarjetas; y, si depuramos el código del archivo del servicio, itemsService.js, comprobamos que se obtienen los artículos en el *array* correspondiente (Figura 6.2.).

Figura 6.2

Comprobación de *array* con las herramientas de desarrollador en el navegador.



6.2. *Observable, observer y subject* en React

Supongamos ahora un caso de uso común y para el que hemos desarrollado esta simulación de web de comercio electrónico. Necesitamos que se muestre en una barra de menú un elemento de cesta con el número de artículos añadidos. El problema que se nos presenta es que, por la configuración de las vistas, los menús suelen ser elementos estáticos que se cargan al iniciar la aplicación y no reciben los eventos de usuario, ya que estos, como en nuestro caso, se producen en la interacción con los componentes de pantalla.

En estos casos es donde RxJS y la programación reactiva son enormemente útiles, porque, con el patrón *observable*, *observer* y *subject*, un componente se puede suscribir a los cambios de una variable de un servicio o de otro componente sin necesidad de implementar una jerarquía padre-hijo o invocar una función.

En nuestro proyecto, vamos a implementar el *subject* y el *observable* en el servicio itemsService.js, modificando su código de la siguiente forma:

```
import { Subject } from 'rxjs';

let items = [];
const counterSubject = new Subject();

export function addItem(item) {
    items.push(item);
    counterSubject.next({counter: items.length});
}

export function getCounterItems() {
    return counter.asObservable()
}
```

Como vemos, añadimos una variable “counterSubject” que será el *subject* y, por tanto, una instancia de esta clase de RxJS. Este *subject* permite usar los métodos del *observable* (en este caso, “next()”) para que, cada vez que se invoque la función “addItem”, devuelva en ese flujo de datos un objeto con la cantidad de artículos actualizados.

El segundo paso es declarar el *observable*. Al disponer de un *subject*, lo que hacemos es crear una función que devolverá un *observable* a partir del *subject*, empleando para ello el método “asObservable()”. El propósito de este código es que cualquier componente que use la función que devuelve el *observable* se suscriba a los cambios emitidos por el sujeto, obteniendo los datos de cuántos artículos se han añadido.

Esos cambios se producirán cada vez que sea invocada la función “addItem”, momento en que “next()” emite un nuevo dato en el flujo o *stream* del *observable*. Ya tenemos la implementación en el servicio, así que ahora vamos a crear un nuevo componente de menú de cabecera. Creamos un directorio llamado components en src y añadimos el archivo HeaderMenu.js con el siguiente código:

```
import React, { useEffect, useState } from 'react'
import { getCounterItems } from '../services/itemsServices'

export default function HeaderMenu() {

    const [counter, setCounter] = useState(0)
```

```

useEffect(()=> {
    getCounterItems()
        .subscribe({
            next: data => setCounter(data.counter)
        })
},[counter])

return (
    <nav>
        <div className="basket">
            cesta
            <div className="counter">{counter}</div>
        </div>
    </nav>
)
}

```

Como podemos comprobar, usamos el *hook* “useEffect()” para, al iniciarse el componente, invocar la función que nos devuelve el *observable* del servicio. De esta manera, con el método “subscribe” nos podemos suscribir a los cambios y utilizar el objeto “observer” para que, cuando haya cambios, utilice los datos recibidos para actualizar una variable “counter” que emplearemos en la plantilla.

El siguiente paso es añadir este componente al componente principal, para lo cual modificamos el archivo App.js de la siguiente manera:

```

import HeaderMenu from './component/HeaderMenu';
import ItemsGrid from './pages/ItemsGrid';

function App() {
    return (
        <>
            <HeaderMenu />
            <ItemsGrid />
        </>
    );
}

export default App;

```

Ahora añadimos los estilos CSS de este menú en el archivo index.css con los siguientes bloques de código:

```

nav {
  background-color: white;
  box-shadow: 0 2px 4px 0 rgba(0,0,0,0.08);
  height: 60px;
  display: flex;
  justify-content: flex-end;
  align-items: center;
  padding: 0 1rem;
}

.basket {
  background-color: #008489;
  padding: 0.25rem 1rem;
  border-radius: 0.25rem;
  color: white;
  margin-right: 1rem;
  position: relative;
}

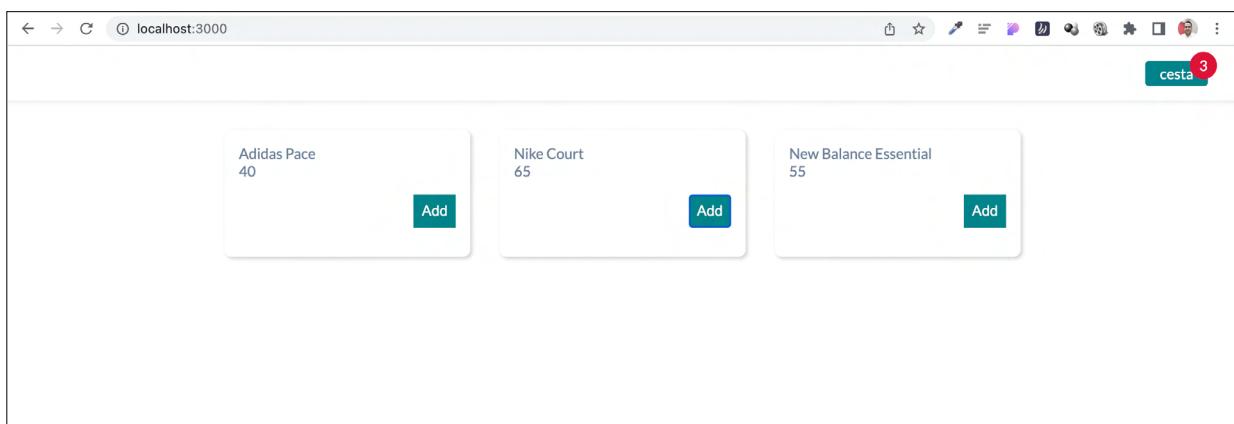
.basket .counter {
  background-color: crimson;
  width: 30px;
  height: 30px;
  border-radius: 30px;
  display: flex;
  justify-content: center;
  align-items: center;
  position: absolute;
  top: -10px;
  right: -10px;
}

```

Figura 6.3

Comprobación de la implementación de programación reactiva con RxJS en el navegador..

Finalmente, podemos comprobar la funcionalidad esperada en el navegador (Figura 6.3), que actualiza la cesta cada vez que se añade un artículo.





Resumen

- Los proyectos desarrollados con React permiten implementar la librería RxJS mediante el gestor de paquetes npm, de manera que se puedan desarrollar sus aplicaciones de programación reactiva en la comunicación entre componentes.
- Con el patrón *observable*, *observer* y *subject* de RxJS, un componente se puede suscribir a los cambios de una variable de un servicio o de otro componente sin necesidad de implementar una jerarquía padre-hijo o invocar una función.

7. Estrategias de uso con Angular

El *framework* Angular ha apostado definitivamente por el uso de la librería RxJS para la implementación de programación reactiva en el desarrollo de aplicaciones, incluyéndola de manera nativa en sus proyectos.

En esta unidad vamos a aprender a utilizar e implementar RxJS en Angular, comenzando por la generación de un proyecto con un servicio para centralizar los datos que hay que manejar por la aplicación.

Posteriormente, desarrollaremos de nuevo y de forma práctica el concepto de *observable*, *observer* y *subject* en el servicio, para la suscripción de componentes reactivamente a los cambios.

7.1. Librería RxJS y Angular

Mientras que en React pueden existir dudas sobre en qué parte de la aplicación implementar RxJS, en Angular, por su propia definición, no cabe duda de que el servicio es el lugar idóneo, al ser el elemento de la aplicación centralizador de los datos de una entidad.

Para comprobarlo de manera práctica, vamos a desarrollar un proyecto simulando una pantalla de sala de chat, donde un usuario anfitrión puede añadir a otros usuarios que serán representados por un rectángulo. Para ello, comenzaremos por implementar un servicio con esos datos de usuario.

Por tanto, comenzamos generando un nuevo proyecto Angular, para lo cual completamos el siguiente comando en cualquier ubicación de nuestro equipo:

```
ng new practica5
```

Al completar este comando, Angular CLI nos consulta si queremos instalar Routing, y pulsamos Intro para confirmar. Seguidamente, nos pide seleccionar los estilos, y de nuevo pulsamos Intro para confirmar CSS.

A continuación, abrimos el proyecto en Visual Studio Code y vamos a generar el servicio escribiendo el siguiente comando en la terminal integrada:

```
ng generate service users
```

De momento, en el archivo creado users.service.ts simplemente añadimos con el siguiente código un *array* de usuarios y unos métodos para añadirlos y retornarlos.

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class UsersService {

  users: Array<string> = ['Pilar(Anfitrión)'];

  constructor() { }

  addUser(user: string): void {
    this.users.push(user);
  }

  getUsers(): Array<string> {
    return this.users;
  }

}
```

A continuación, vamos a crear un componente de sala de chat, para lo cual completamos el siguiente comando en la terminal:

```
ng generate component chatRoom
```

Antes de trabajar en el componente, como vamos a usar un formulario, debemos añadir el correspondiente módulo en app.module.ts, modificando su código para añadir las siguientes líneas:

```
...
import { ReactiveFormsModule } from '@angular/forms';
...

imports: [
  BrowserModule,
```

```
    AppRoutingModule,
    ReactiveFormsModule
  ],
...
```

Una vez generado, comenzamos a trabajar en la clase del componente añadiendo la recuperación de los datos de usuarios desde el servicio, el objeto para un campo *input* para establecer nuevos usuarios y el uso del método para añadir un usuario en el servicio. Completamos el siguiente código en el archivo chat-room.component.ts:

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';
import { UserService } from '../users.service';

@Component({
  selector: 'app-chat-room',
  templateUrl: './chat-room.component.html',
  styleUrls: ['./chat-room.component.css']
})
export class ChatRoomComponent implements OnInit {

  users: Array<string> = [];
  form: FormGroup = new FormGroup({});

  constructor(private userService: UserService) { }

  ngOnInit(): void {
    this.users = this.userService.getUsers();
    this.form = new FormGroup({
      user: new FormControl('')
    })
  }

  addUser() {
    this.userService.addUser(this.form.value);
    this.form.reset();
  }
}
```

A continuación, añadimos la vista en el archivo chat-room.component.html con el siguiente código:

```
import class="container">
<form [FormGroup]="" (ngSubmit)="addUser()">
  <input type="text"
    placeholder="Nuevo usuario"
```

```

        formControlName="user">
      <button type="submit">Añadir</button>
    </form>
    <div class="grid">
      <div class="screen" *ngFor="let user of users">
        <p>{{user}}</p>
      </div>
    </div>
  </div>

```

Como siempre en Angular, una vez creada una pantalla, la añadimos al módulo de *routing*. Para ello, añadimos en el archivo app-routing.module.ts la ruta modificando el código:

```

import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ChatRoomComponent } from './chat-room/chat-room.component';

const routes: Routes = [
  {path: '', component: ChatRoomComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }

```

En el siguiente paso, mantenemos la etiqueta de salida en el componente raíz, modificando el archivo app.component.html de la siguiente forma:

```
<router-outlet></router-outlet>
```

Y, finalmente, añadimos los estilos CSS en el archivo styles.css con los siguientes bloques:

```

@import url('https://fonts.googleapis.com/css?family=Lato&display=swap');

* {
  padding: 0;
  margin: 0;
  box-sizing: border-box;
}

```

```
body {
  font-family: 'Lato', sans-serif;
  color: #718096;
}

.container {
  max-width: 1260px;
  margin: 0 auto;
  padding: 1rem;
}

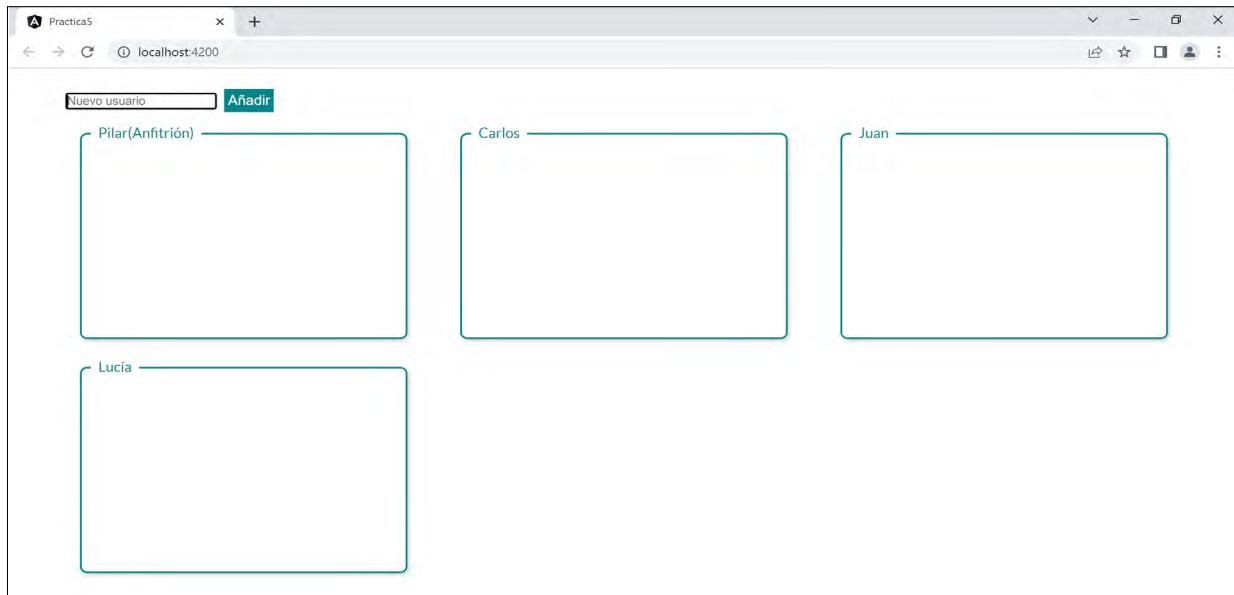
.grid {
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;
}

.screen {
  width: 360px;
  height: 240px;
  border: 2px solid #008489;
  border-radius: 0.5rem;
  box-shadow: 2px 2px 4px lightgray;
  margin: 1rem;
  padding: 1rem;
  position: relative;
}

.screen p {
  position: absolute;
  padding: 0 0.5rem;
  top: -12px;
  left: 10px;
  background-color: white;
  color: #008489;
}

button {
  font-size: 1rem;
  padding: 0.2rem;
  margin: 0.5rem;
  color: white;
  background-color: #008489;
  border: #008489 solid 1px;
  cursor: pointer;
  transition: background-color 400ms ease-in-out;
}
```

Para comprobar la aplicación, la levantamos en local con el comando habitual, “ng serve -o”. Podemos añadir usuarios y verificar que se van renderizando en pantalla (Figura 7.1).



De nuevo, si necesitamos un componente adicional de menú con información del número de usuarios conectados, puesto que necesita estar en una implementación diferente a donde se producen las interacciones, necesitaremos RxJS para suscribirnos a los cambios. En el siguiente apartado, vamos a ver cómo se implementa en el servicio.

Figura 7.1

Vista principal de la aplicación Angular en el navegador.

7.2. *Observable, observer y subject* en Angular

A continuación, vamos a introducir un *subject* y *observable* en el servicio Angular. Vamos a comprobar que la implementación es similar a cualquier otro *framework* o incluso JavaScript plano, con la diferencia de que aporta el tipado de datos de TypeScript. Modificamos el archivo users.service.ts de la siguiente forma:

```
import { Injectable } from '@angular/core';
import { Observable, Subject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class UsersService {
```

```

private users: Array<string> = ['Pilar(Anfitrión)'];
private countUserSubject = new Subject();

constructor() { }

addUser(data: any): void {
  this.users.push(data.user);
  this.countUserSubject.next({usersNumber: this.users.length});
}

getUsers(): Array<string> {
  return this.users;
}

getUsersNumber(): Observable<any> {
  return this.countUserSubject.asObservable()
}

}

```

Una vez que tenemos el *subject* y *observable* devuelto en la función “getUsersNumber”, vamos a crear un componente de menú, para lo cual completamos en la terminal:

```
ng generate component nav
```

Una vez generado, en la clase del componente del archivo nav.component.ts implementamos la lógica para suscribirnos al servicio y usar el observer para obtener el número de usuarios conectados. Cambiamos el código por el siguiente:

```

import { Component, OnInit } from '@angular/core';
import { UsersService } from '../users.service';

@Component({
  selector: 'app-nav',
  templateUrl: './nav.component.html',
  styleUrls: ['./nav.component.css']
})
export class NavComponent implements OnInit {

  usersNumber: number = 1;
}

```

```

constructor(private userService: UsersService) { }

ngOnInit(): void {
  this.userService.getUsersNumber()
    .subscribe({
      next: data => this.usersNumber = data.usersNumber
    })
}

}

```

En la plantilla, archivo nav.component.html, añadimos los elementos con la interpolación del número de usuarios mediante el siguiente código:

```

<nav>
  <p>Usuarios conectados {{usersNumber}}</p>
</nav>

```

Nuestro nuevo componente, al ser estático, lo implementamos con su etiqueta directamente en el archivo app.component.html, cambiando el código por el siguiente:

```

<app-nav></app-nav>
<router-outlet></router-outlet>

```

Y, finalmente, añadimos un bloque de estilos CSS en el archivo styles.css:

```

nav {
  background-color: white;
  box-shadow: 0 2px 4px 0 rgba(0,0,0,0.08);
  height: 60px;
  display: flex;
  justify-content: flex-end;
  align-items: center;
  padding: 0 1rem;
}

```

Ahora ya podemos comprobar en el navegador (figura 7.2) que se actualiza la información del número de usuarios conectados según los añadimos, gracias a la suscripción al observable retornado por el servicio.

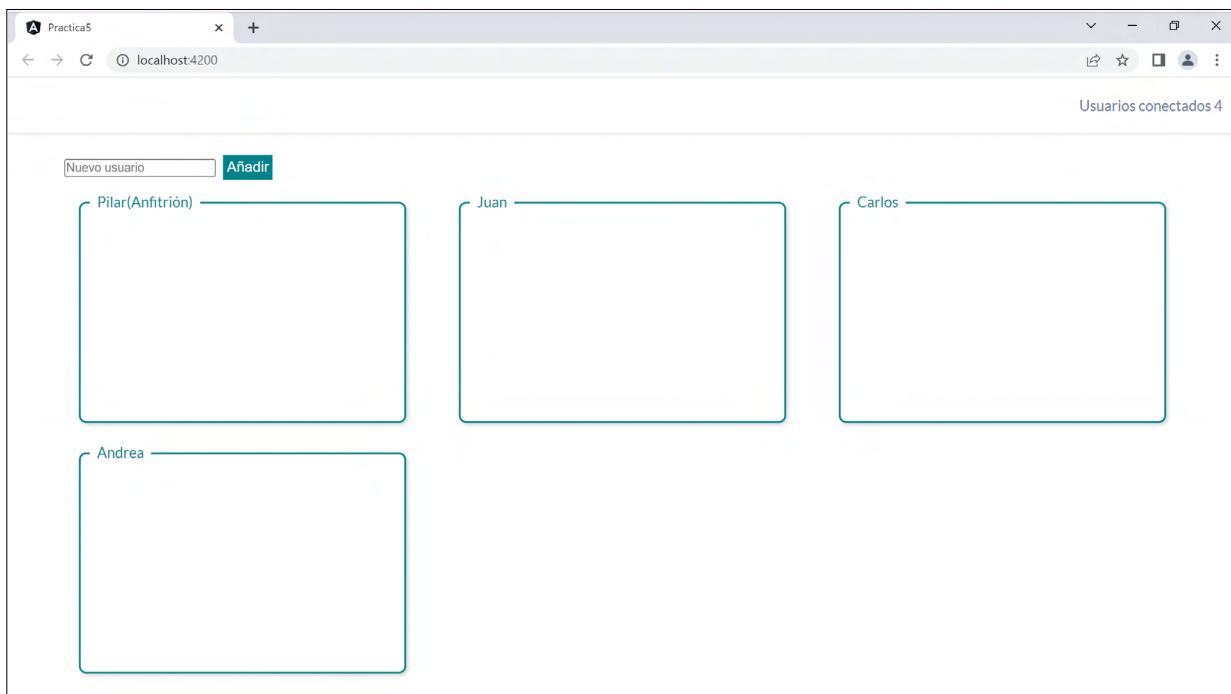


Figura 7.2 Actualización de valores por suscripción en Angular en el navegador.

Las posibilidades de implementación de RxJS son infinitas, y pueden ser la solución a muchos problemas ocasionados por los requisitos de funcionalidad que afecten a varios componentes de nuestras aplicaciones.



Resumen

- Los proyectos desarrollados con Angular incorporan de manera nativa la librería RxJS, de manera que se puedan desarrollar sus aplicaciones de programación reactiva en la comunicación entre componentes.
- El patrón *observable*, *observer* y *subject* de RxJS es incorporado en Angular de manera similar a otros *frameworks*, con la excepción de que aporta el tipado fuerte de datos de TypeScript.

Índice

Esquema de contenido	3
Introducción	5
1. Introducción a Redux	7
1.1. Conceptos generales sobre Redux	7
1.2. Implementación de Redux en un proyecto	9
Resumen	11
2. Store, reducers, actions y suscripción a cambios	12
2.1. Store, reducers y actions	12
2.2. Dispatch de acciones en Redux	15
2.3. Suscripciones en Redux	16
Resumen	20
3. Conectando Redux con React DevTools	21
3.1. Redux en React	21
3.2. Conectando Redux con React DevTools	28
3.3. Suscripción al estado	29
Resumen	33
4. Asincronía en Redux	34
4.1. Escalando el estado de Redux en React	34
4.2. Asincronía en Redux	36
Resumen	40
5. Programación reactiva, concepto de observable y observer	41
5.1. Librería RxJS y el concepto de observable y observer	41
5.2. Observables y observer en JavaScript con RxJS	42
Resumen	46
6. Estrategias de uso con React	47
6.1. Librería RxJS y React	47
6.2. Observable, observer y subject en React	51
Resumen	55

7. Estrategias de uso con Angular	56
7.1. Librería RxJS y Angular	56
7.2. <i>Observable, observer y subject</i> en Angular	61
Resumen	65