2. Programación funcional

¿Por qué un capítulo sobre programación funcional? Los imperios de lenguajes y los reinos de *frameworks* surgen y caen, pero solo una cosa permanece: debemos mantener código. Esa es la parte que sin duda más tiempo requiere el proceso de desarrollo de software. Y es precisamente por eso por lo que nos debería preocupar crear un buen código, limpio, testeado y ante todo fácil de mantener. Y también porque React ha ido evolucionando al estilo funcional.

La programación funcional, en adelante PF, es uno de esos grandes temas como la inteligencia artificial, Internet, la ingeniería de software... que aparecen y desaparecen, pero que vuelven a estar de moda. Al margen de modas, cualquier *crafter* debería acercarse a esta área porque le puede proveer de herramientas muy útiles para su arsenal, incluso aunque no se abrace completamente. Como cuando Thorin *Escudo de Roble* (un orgulloso rey enano) encontró una espada élfica: no renunció a ella porque apreció su valor.

En este capítulo, vamos a presentar algunos conceptos de la PF y comprobará cómo encuentra conocimiento útil. Esto no se trata de presentar el último *framework* de moda o el último formato de configuración: se trata de técnicas de programación que le ayudarán a escribir mejor código en el día a día. Estos son temas que resisten el paso del tiempo y siguen siendo válidos y es por eso que merece la pena echarles un vistazo.

Nada de *monads*, *functors*, nada de mates abstractas. Solo código simple. Este capítulo presenta las piezas básicas de la programación funcional aplicada con JavaScript.

Funciones puras

Son funciones que para una entrada dada siempre devuelven el mismo resultado**,** y no pueden tener efectos secundarios o *side-effects*. Un ejemplo obvio:

```
function add(a, b) {
  return a + b;
}
```

Beneficios de las funciones puras

Debemos procurar que todas nuestras funciones sean puras porque:

- Son fáciles de testear.
- Facilitan la composición con otras funciones.
- Son cacheables o *memoizables*.
- Evita *side-effects*.
- Conducen a un código más limpio.

Y, por eso, ¡debemos aspirar a que todos los componentes React sean funciones puras! A continuación, desgranamos estas y otras propiedades.

Fácil de testear

La función solo tiene una entrada y su salida: no tiene que preocuparse de nada más. Los efectos secundarios hacen que el código sea más difícil de probar y mantener. ¿Por qué? Porque, para probar algo, probablemente necesite definir varios *mocks* o preparar un montón de objetos auxiliares. Sin embargo, una función que no depende de nadie es lo más simple para comprobar en un test.

Fácil de componer con otras

Como veremos más adelante, las funciones puras permiten la composición, especialmente cuando las funciones tienen un único parámetro. Es una técnica de reutilización de código.

Cacheable/*Memoizable*

Cuando una función es pura le da la garantía de que siempre devolverá lo mismo para un parámetro dado. Así, si ejecutamos la función una vez, podemos guardar o almacenar en una caché el resultado por si la volvemos a llamar con el mismo

Desarrollo *online*

Si no quiere ni puede instalar un entorno en su propio equipo, es más, si ni siquiera tiene un equipo propio, puede desarrollar aplicaciones React *online*, a través de distintas aplicaciones de desarrollo. Bastará con tener un ordenador que disponga de conexión a Internet y obviamente un navegador.

Codepen

Codepen, al igual que Jsfiddle, Codebin, Repl.it y otros muchos entornos de este tipo, nos permite crear pequeñas demos de *front-end*, donde, además de unir HTML, CSS y JavaScript puro, podemos añadir cualquier librería adicional que queramos y disponer de escenarios específicos para los *frameworks* más conocidos.

Además de las librerías, también se pueden aplicar transpilaciones o transformaciones como Babel, que es utilizada por React para que su sintaxis JSX y su ES6 pueda ejecutarse de forma segura en cualquier navegador.

Este es, por ejemplo, un entorno de React que puede utilizarse como plantilla para crear otros escenarios:

`https://codepen.io/pxai/pen/eaRaeb`.

Podemos ver un ejemplo en la figura 3.1.

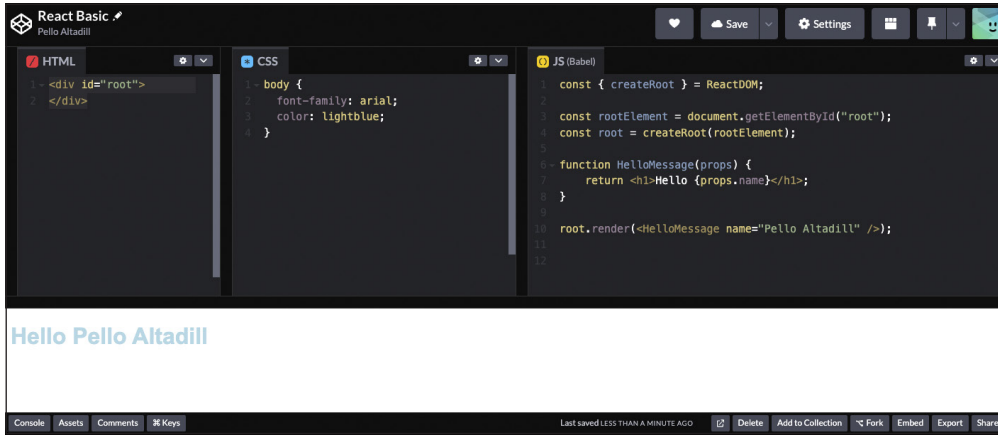

Figura 3.1. Aspecto de un escenario React cargado en Codepen.

Para configurar un entorno como este, podemos ver la configuración de JavaScript, como se ve en la figura 3.2. En este caso, se debe aplicar Babel para permitir opciones avanzadas de JavaScript, además las dos librerías `react` y `react-dom`.

App.js

Este es el componente `App.js` que a su vez tiene subcomponentes y al que le aplicamos estilos en un fichero llamada `App.css`. Antes del `return` que genera el contenido existen unas líneas que se encargan de varias cosas:

- `useState`: Se utiliza para gestionar valores del estado (`gender`, `country` y `people`), es decir, valores que cambian según lo que haga la aplicación o la interacción del usuario.
- `findPeople`: Una función que se encarga de llamar a una API (un recurso de la red) y guarda la respuesta en el estado.
- `useEffect`: Una función de React que se invoca automáticamente cuando se carga el componente.

Profundizaremos en el estado, `useEffect` y a las llamadas a la red en los próximos capítulos. Ahora vamos a centrarnos en la presentación y en ver cómo queda el componente.

Listado 4.2. Fichero `App.js`.

```
import { useState, useEffect, useCallback } from 'react';
import Person from './Person';
import SearchForm from './SearchForm';
import './App.css';
import axios from 'axios';

function App() {
  const [people, setPeople] = useState([]);
  const [gender, setGender] = useState();
  const [country, setCountry] = useState('US');

  const findPeople = useCallback(async () => {
    const url = `https://randomuser.me/api/?results=12&gender=${gender}&nat=${coun
try}`;
    const { data: { results } } = await axios.get(url);
    setPeople(results);
  }, [gender, country])

  useEffect(() => {
    findPeople();
  }, [gender, country, findPeople])

  const handleGender = (event) => {
    const selectedGender = event.target.value;
    setGender(selectedGender);
  }

  const handleCountry = (event) => {
```

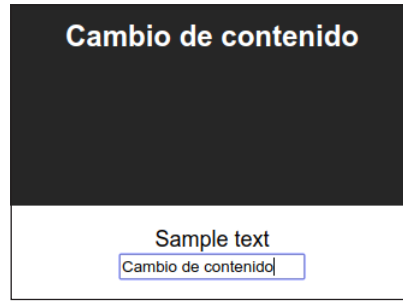

Figura 5.5. El título cambia en cuanto se escribe en el campo de texto.

Existen otras maneras de acceder a los campos de un formulario. Si solamente hay un campo, se puede acceder a través del evento de cambio, como en este caso.

Campos de selección múltiple

Hay campos de formulario que precisan un tratamiento más especial, como son las listas de selección múltiple y los *checkbox*. Estos últimos se pueden gestionar por separado, pero ahora se mostrarán agrupados y sus valores metidos en un único *array*.

Para las listas de selección múltiple, también se utilizará un *array*. En su caso, lo que se hace es crear un evento en el elemento `<select>`, para que, cada vez que haya un cambio, se actualice el *array* que contiene todos los elementos seleccionados.

Con los *checkbox*, si se agrupan, el tratamiento es más pesado, ya que hay que meter/sacar los seleccionados del *array* en el que se agrupan los valores. La ventaja es que, precisamente, se consigue guardar un conjunto de valores agrupados.

Lo vemos en el siguiente ejemplo.

App.js

Este es el componente único que contiene el formulario. Además del estado, por cada elemento del formulario tenemos unas funciones que se encargan de actualizar el estado según los eventos.

Listado 5.5. Fichero `App.js`.

```
import { useState } from 'react';
import './App.css';

function App() {
  const [state, setState] = useState({
```

callbacks de un lado a otro y también para compartir información entre componentes que no tienen por qué estar relacionados. La figura 7.4 lo resume de forma simple.

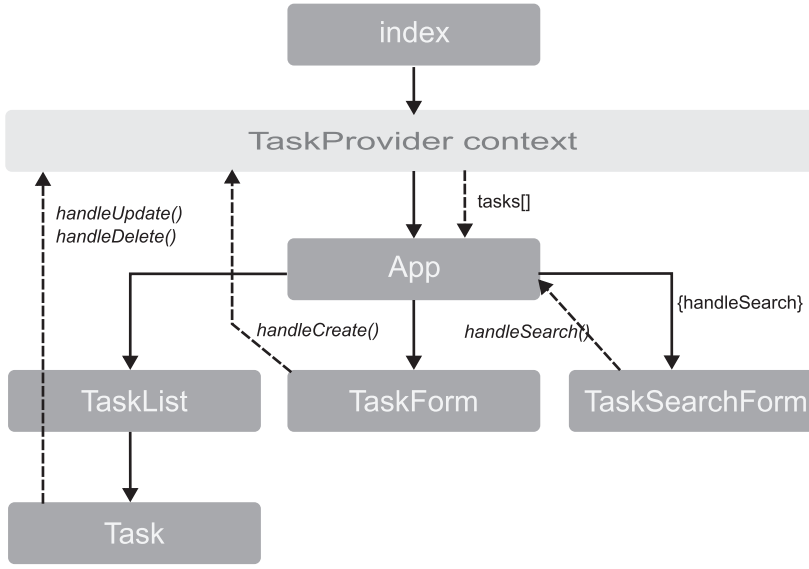

Figura 7.4. Esquema de la aplicación aplicando contexto.

Implementación

Lo que haremos en esta implementación es mover al contexto toda la información de las tareas y las funciones para modificarlas. De esa manera, no necesitaremos pasar información y *callbacks* de componente padres a hijos.

task.context.js

Empezaremos por definir un contexto que será compartido por todos los componentes. Este contexto define un *array* de tareas y una serie de funciones para gestionar ese *array*. Luego, como con todo contexto de React, se crea un *provider* o proveedor de contexto que usaremos para ofrecer ese contexto a toda la aplicación.

Listado 7.7. Fichero `task.context.js`.

```
import { createContext, useState } from 'react';
import initialTasks from '../initialTasks';

const addTaskToTasks = (tasks, taskToAdd) => [...tasks, taskToAdd];
const removeTaskFromTasks = (tasks, idToRemove) => tasks.filter(task => task.id !==
idToRemove);
```

Puesta en marcha

La ventaja es que se permite iniciar la aplicación como una aplicación React convencional:

```
npm start
```

Y lo que se consigue se ve en la figura 9.2.

Figura 9.2. Consola de ejecución de un proyecto React Native con el código QR.

Las opciones de menú resultan bastante obvias. La más cómoda para el desarrollo sería la opción `w`, que nos permite visualizar la aplicación en el navegador, tal y como se hace en una aplicación React convencional. Esta opción requiere descargarse unos paquetes extra que se nos indican en el momento (react-native-web, react-dom, @expo/webpack-config).

Si queremos ejecutar la aplicación en nuestro móvil lo podremos hacer mediante la *app* Expo. En el caso de que se vea que no se obtienen los últimos cambios en el móvil, se puede optar por la opción `r` para que la aplicación se reconstruya y se reinicie su ejecución.

App.js

Se trata del componente principal y también del inicio de la aplicación. Por tanto, no habría un fichero `index.js` en este caso. Además de las pantallas o componentes de la aplicación, también se pueden apreciar en algunos detalles específicos de React Native.

- **Estilos de React-Native:** Se importan elementos específicos de estilos para React Native.
- **Navegación:** La navegación en aplicaciones móviles es algo particular por las propiedades de los dispositivos. En React Native puede hacerse que cada ventana que se abre se acumule como una pila (*stack*), lo cual se hace utilizando `StackNavigator`. El aspecto de la pantalla inicial se aprecia en la figura 9.28.

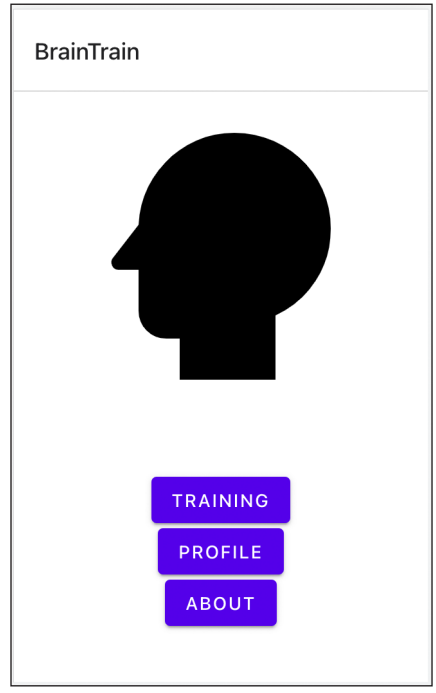

Figura 9.28. Aspecto de la pantalla inicial del proyecto.

Además de aplicación del contexto para el perfil del usuario, el método de navegación también requiere de una inicialización de un `NavigationContainer`, donde se introduce el `StackNavigator` y cada una de las pantallas a las que se puede acceder en este aplicación. En cierto modo se aproxima a la definición de rutas en React Router.

Listado 9.18. Fichero `App.js`.

```
import { StyleSheet } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack';
import HomeScreen from './screens/HomeScreen';
import AboutScreen from './screens/AboutScreen';
import ProfileScreen from './screens/ProfileScreen';
import TrainingScreen from './screens/TrainingScreen';
import { BrainTrainContext, contextValue } from './lib/context';

const Stack = createNativeStackNavigator();
export default function App() {
  return (
    <BrainTrainContext.Provider value={contextValue}>
      <NavigationContainer>
        <Stack.Navigator>
          <Stack.Screen
            name="Home"
            component={HomeScreen}
            options={{ title: 'BrainTrain' }}
          />
          <Stack.Screen name="Training" component={TrainingScreen} />
          <Stack.Screen name="Profile" component={ProfileScreen} />
          <Stack.Screen name="About" component={AboutScreen} />
        </Stack.Navigator>
      </NavigationContainer>
    </BrainTrainContext.Provider>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
});
```

Home.js

Esta es la pantalla inicial de la aplicación que presenta 3 botones para poder llegar a las pantallas de `Training`, `Profile` y `About` respectivamente. Al pulsar cada uno de los botones utilizamos la función `navigation.navigate` para poder cargar otra pantalla. Otra de las peculiaridades de la aplicación es que utiliza componentes de MaterialUI a través del paquete `@react-native-material`.

La guía que presenta todas las capacidades de esta librería, entre otras la parte visible del desarrollo web donde React ha ganado la partida a diferentes soluciones que compiten por ofrecer la mejor tecnología en la creación de nuevos proyectos

Nacido en el seno de Facebook pero puesto a disposición de todo el mundo, React está siendo utilizado por las compañías más importantes como Airbnb, Apple, Dropbox, Instagram, Netflix, Twitter, Tesla, Uber…, y se ha convertido en una de las apuestas más relevantes y recomendables de los últimos tiempos, ya que cuenta con un completo ecosistema de módulos, herramientas y componentes capaces de ayudar a construir cualquier desarrollo avanzado con relativamente poco esfuerzo.

React se distingue por ser una librería que permite la organización de una interfaz web en componentes reutilizables, aportando además una gestión más ágil de los elementos HTML mediante el uso de un DOM virtual. Facilita, además, desarrollar aplicaciones para móviles mediante la librería React Native, cuyo aprendizaje resulta trivial una vez que se asimilan los principios básicos que rigen esta completa solución de JavaScript.

Con esta obra práctica, el autor quiere presentar una visión de las numerosas posibilidades de desarrollo con React, desde aplicaciones *front-end* y para escritorio, *apps* para móviles e incluso aplicaciones *full-stack* con Next.js.

ANAYA MULTIMEDIA
www.anayamultimedia.es
ISBN 978-84-415-4773-5
2315188
9 788441 547735