Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de Aplicaciones Frontend con arquitectura Redux

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Miguel Coleto Muñoz

Tutor: Vicente Pelechano Ferragud

2017-2018

Resumen

En este proyecto se muestra la arquitectura Redux para la creación de nuevas aplicaciones. Se utilizan como ejemplo dos aplicaciones propias actualmente en producción. Se profundiza en su implementación y las refactorizaciones aplicadas. En concreto, una aplicación móvil para clientes de restaurants mediante la que se puede pedir y pagar, y un punto de venta para locales hosteleros. Se presenta la utilización de Redux en estos proyectos y las lecciones aprendidas. Se pretende dar una valoración personal sobre esta arquitectura y porqué está ganando popularidad en la industria del software.

Palabras clave: Arquitectura, Redux, Flux, React, Web, Javascript

Abstract

This project shows the Redux architecture for the creation of new applications. It uses two production ready applications of my own. We will be focusing on their implementation and applied refactors. These projects are, a mobile application for restaurants clients to order and pay your meals and a web sales point for restaurants owners and staff. We will show how Redux is used in these products and the learned lessons from them. We pretend to give a personal opinion about this architecture and why it is gaining popularity in the software industry.

Keywords: Architecture, Redux, Flux, React, Web, Javascript

Tabla de contenidos

Índice de figuras

1. Introducción

A medida que las tecnologías evolucionan nuestros ordenadores y teléfonos son más potentes. Esto ha ido provocando que las aplicaciones clásicas con arquitectura cliente/servidor deleguen más responsabilidad en el cliente. Conforme avanzamos, las aplicaciones frontales tienen más funcionalidades y bases de código más grandes.

Es clara la necesidad de establecer una estructura y lenguaje común a la hora de desarrollar estas aplicaciones para poder trabajar en equipo correctamente. Por ello, en 1988 se empezó a hablar de un patrón arquitectónico llamado Modelo Vista Controlador cuya relevancia sigue vigente hoy en día.

Sin embargo, las necesidades de nuestros proyectos cada vez son más complejas y la comunicación entre componentes se vuelve más confusa. Esto provoca que empiecen a aparecer fallos escondidos y que la arquitectura se vuelva rígida y difícil de mantener y evolucionar en nuestras aplicaciones.

El objetivo de este TFG es presentar una alternativa a las arquitecturas MVC[1] y derivadas con un enfoque basado en el flujo de datos unidireccional y con ideas del paradigma funcional reactivo, Redux. Veremos cómo funciona en algunos proyectos en producción y qué relevancia está teniendo en la industria.

a. Motivación

Desde que comencé a programar mi mayor aspiración siempre ha sido construir un producto que utilicen muchas personas. Para ello, es obligatorio adaptarse al público al que el proyecto se dirige, cada vez mas exigente gracias al rápido avance de la tecnología y la potencia de nuestros dispositivos de uso diario. Esperamos que las aplicaciones que frecuentamos funcionen de forma fluida y que se actualicen cada poco tiempo para ofrecernos nuevas funcionalidades.

Para que un equipo de desarrolladores sea capaz de lograr todos estos objetivos necesitan una arquitectura para facilitar la comunicación y realizar cambios sin afectar a todo el proyecto.

Cada proyecto tiene unas necesidades concretas que no permiten que exista una arquitectura perfecta para todos los casos. Por ello, este apartado del desarrollo de

[1] **MVC:** Modelo Vista Controlador

software me interesa mucho, ya que cuantos más patrones conozca, mejores decisiones podré tomar y en definitiva ser mejor ingeniero.

La decisión de realizar este trabajo sobre Redux viene dada por ser una nueva arquitectura que se sale de los patrones convencionales y está ganando popularidad en la industria rápidamente debido a que encaja perfectamente en ciertas metodologías de desarrollo. Actualmente he participado en el desarrollo de varios proyectos con esta arquitectura y encuentro interesante aportar mi experiencia con ella.

b. Objetivos

Con este trabajo se pretende:

Introducir la arquitectura Redux y proporcionar una visión general de cómo funciona el patrón. Redux es una nueva herramienta para el desarrollo con un enfoque distinto a las arquitecturas más populares.

Presentar la implementación de este patrón a través de una aplicación móvil y otra web, ambas pensadas para el mundo real y en producción en la actualidad.

c. Impacto esperado

Se espera dar visibilidad a Redux y que los posibles usuarios cuenten con una herramienta más en su repertorio para hacer frente a los desafíos que presenta la ingeniería del software.

Además, también se pretende plantear una reflexión sobre cómo lo establecido y lo más popular no tiene por que ser la solución adecuada para todos los problemas.

d. Estructura

En el capítulo 2 se hará una crítica al estado del arte. Se hablará de los inicios de la arquitectura Modelo Vista Controlador y los principales problemas que tiene la arquitectura.

En el capítulo 3 se expondrán los antecedentes de Redux y las diferencias que lo han hecho evolucionar. Se presentarán los principios claves sobre los que se construye la arquitectura, los componentes que la forman y las ventajas y desventajas que aportan a nuestra base de código.

Para apoyar la parte más teórica, en los capítulos 4 y 5, se expondrán dos casos prácticos de aplicaciones en producción que hacen uso de esta arquitectura. Se presentarán fragmentos de código comentados.

Por último, en el capítulo 6, se hablará de cómo se está adoptando esta arquitectura en la industria del software y se ha adaptado en otros lenguajes y entornos. En concreto, una fusión de algunos conceptos de Redux con una arquitectura más clásica y una librería que intenta portar por completo el patrón a otro lenguaje.

2. Estado del arte

Con el inicio del desarrollo de interfaces gráficas de usuario fue necesario establecer una forma de separar la lógica de negocio de las aplicaciones y la lógica de representación por pantalla, o lo que es equivalente, una transformación de un modelo mental que sirva al usuario para representar la información a un modelo que el ordenador sepa procesar.

Durante su estancia en el Xerox Palo Alto Research Center en los años 1978-79, Trygve Reenskaug implementó por primera vez el patrón *Modelo Vista Controlador* (*MVC*) para Smalltalk-76 (Reenskaug). Esta estructura fue de mucha utilidad para permitir al usuario la manipulación de programas en los que este modificaba cierta información desde varios contextos.

Esta arquitectura propone que cada componente gráfico de la interfaz tenga un par *Vista* y *Controlador*. Cada acción sobre la *Vista* resulta en una llamada al *Controlador* que actualiza el *Modelo* y en consecuencia la *Vista* recibe la modificación directamente y se actualiza.

El patrón ganó mucha popularidad y con su introducción en distintos *frameworks*[2] de desarrollo de aplicaciones web se convirtió en un estándar en la industria.

Por otra parte, *MVC* ha ido derivando en otros patrones arquitectónicos para adaptarse a ciertas carencias. Entre las más relevantes se encuentran *Modelo Vista Presentador* (*MVP*) y *Modelo Vista Vista-Modelo* (*MVVM*), pero en general, estos patrones son conocidos como *MV** ya que por norma general los componentes del Modelo y la Vista siempre están presentes.

Modelo Vista Presentador nace por la necesidad de hacer *testing* sobre las interfaces gráficas (Cejas, 2014). Generalmente el renderizado de los componentes que se visualizan por pantalla vienen dados por los *frameworks* y el desarrollador solo tiene acceso a las *APIs*[3] que estos exponen, dejando fuera la posibilidad de probarlos y dificultando esta fase del desarrollo. Este patrón propone que todas las acciones de la Vista deleguen en el Presentador, el cual esta desacoplado totalmente de la Vista y se comunican mediante una interfaz que permite el *mocking*[4] durante el *testing*. El objetivo es convertir a la Vista en un componente totalmente pasivo con cero lógica. El Presentador es un intermediario entre el Modelo y la Vista y los protege uno del otro. La

[2] **Framework:** Conjunto de librerías y artefactos que orientan el desarrollo y organización de un proyecto software.
[3] **API**: *Application Programming Interface.* Conjunto de funciones que una librería expone para ser utilizada.
[4] **Mock:** Técnica para facilitar las pruebas unitarias que consiste en crear un componente falso que aparentemente se comporta como uno original.

diferencia principal con *MVC* es que la *Vista* no tiene referencia del *Modelo* y por tanto es el *Presentador* el encargado de actualizar a esta.

Puesto que *MVP* propone muchos contratos entre la *Vista* y el *Presentador*. su implementación puede ser tediosa. *Modelo Vista Vista-Modelo* propone un patrón similar en el que la *Vista* delega todas las acciones en el *Modelo de Presentación* o *Vista-Modelo*, pero a diferencia de *MVP* este no guarda referencia a la *Vista*, si no que expone propiedades a la que la *Vista* se suscribe. De esta forma, se sigue obteniendo un componente totalmente pasivo, pero se reducen los contratos.

a. Crítica al estado del arte

El principal problema de las arquitecturas *MV** es que la comunicación entre componentes es bidireccional. Esto provoca que un cambio pueda afectar a muchas partes del conjunto lo que dificulta el entendimiento y depuración del proyecto. Escalar estos patrones puede ser un verdadero reto debido a este descontrol en el flujo de la aplicación.

Por otra parte, *MVC* no favorece la reutilización de código. Se propone que una *Vista* pueda tener varios *Controladores* para de esta forma reutilizarlos de forma parcial. El problema es que estos *Controladores*, aunque disminuyen en tamaño, pueden acoplarse a las *Vistas* con las que trabajan minando su flexibilidad y dificultando su mantenimiento. Se rompe el Principio de Responsabilidad Única de los principios *SOLID*.

Los principios *SOLID* son un conjunto de reglas introducidas por Robert C. Martin para la programación orientada a objetos (Oloruntoba, 2015), que, aplicados en un proyecto, aumentan la calidad y flexibilidad del proyecto:

- S – Principio de responsabilidad única (Single responbility principle): Un objeto solo debe tener una razón para cambiar, es decir, una sola responsabilidad.
- O – Principio de abierto/cerrado (Open/closed principle): Una clase debe estar abierta a la extensión, pero cerrada a la modificación.
- L – Principio de sustitución de Liskov (Liskov substitution principle): Los objetos deben ser reemplazables por instancias de sus subtipos sin alterar el funcionamiento del programa.
- I – Principio de segregación de interfaces (Interface segregation principle): Las clases que implementan interfaces deben poder implementar únicamente los métodos que vayan a usar.
- D – Principio de inversión de dependencia (Dependency inversión principle): Una clase ha de depender de abstracciones en vez de implementaciones.

b. Propuesta

En contraposición a los diseños arquitectónicos clásicos Facebook crea Flux (Facebook, 2014), un conjunto de patrones para lograr una comunicación unidireccional a lo largo de toda la base de código. Estos patrones se basan fuertemente en principios del paradigma funcional y reactivo.

Facebook libera Flux como unas guías de referencia con algunas implementaciones de ejemplo, pero al no existir una forma canónica rápidamente empiezan a surgir librerías que profundizan en estos patrones. La más famosa de todas ellas y la que más tracción ha tenido es Redux.

3. Solución propuesta: Redux

a. ¿Qué es Redux?

Redux es una librería para el manejo del estado de aplicaciones JavaScript (Redux). Fue creada por Dan Abramov en 2015.

A medida que la tecnología avanza las aplicaciones tienen más capacidades y podemos implementar más y mejores funcionalidades. Pero esto provoca que la complejidad de nuestros proyectos se dispare. Una de las razones por las que aumenta la dificultad del mantenimiento es el manejo del estado, tanto del modelo (información de terceros, datos creados localmente, etc.) como de la interfaz (paginación, listas, elementos seleccionados, etc.).

Redux intenta que el manejo de todo este estado sea predecible inspirándose en Flux, Elm, *CQRS*, *Event Sourcing* y principios del paradigma funcional.

Flux es un conjunto de patrones que Facebook presentó para acompañar a React (Facebook), su librería de componentes Web. Aplicados, dirigen el flujo de la aplicación de forma unidireccional.

Elm (Czaplicki, 2012) es un lenguaje de programación funcional. Los proyectos construidos con este lenguaje suelen adoptar la arquitectura *Elm Architecture* que propone un flujo de comunicación unidireccional entre sus componentes.

Command Query Responsibility Segregation (Fowler, 2011) y *Event Sourcing* (Fowler, 2005) son unos patrones arquitectónicos que someten el proyecto a ciertas restricciones, pero aumentan la riqueza del modelo y refuerzan el cumplimiento de los principios *SOLID*. Proponen que la lectura y escritura del modelo de datos se separe en dos subsistemas distintos y que las modificaciones se realicen mediante el lanzamiento de eventos de dominio.

Redux, además de Elm, se basa fuertemente en otros principios que comparten los lenguajes del paradigma funcional. En concreto:

- Inmutabilidad: La modificación de estructuras de datos se realiza mediante copia en vez de mutar el objeto original.
- Funciones puras: Ninguna función ha de lanzar efectos secundarios como llamadas a servicios web, consultas a bases de datos, etc. Gracias a este principio se consigue que el resultado de estas funciones sea determinista.

b. ¿Qué es Flux?

Flux es una arquitectura para desarrollar aplicaciones *Frontend*[5] creada por Facebook para complementar a React, librería para el renderizado de componentes web, también desarrollada por Facebook. Las aplicaciones de la empresa habían crecido tanto que era imposible seguir manteniéndolas con el patrón *Modelo Vista Controlador* incluso con *frameworks* para facilitarlo. La comunicación bidireccional de la arquitectura dificultaba la depuración y rastreo de errores.

Por ellos, Flux pretende proporcionar un flujo de datos unidireccional a lo largo de toda la aplicación (Figura 1), de forma que, cuando el usuario interactúa con la interfaz se propagan acciones que son recogidas por un único manejador central encargado de modificar el estado en los contenedores del estado de la aplicación.

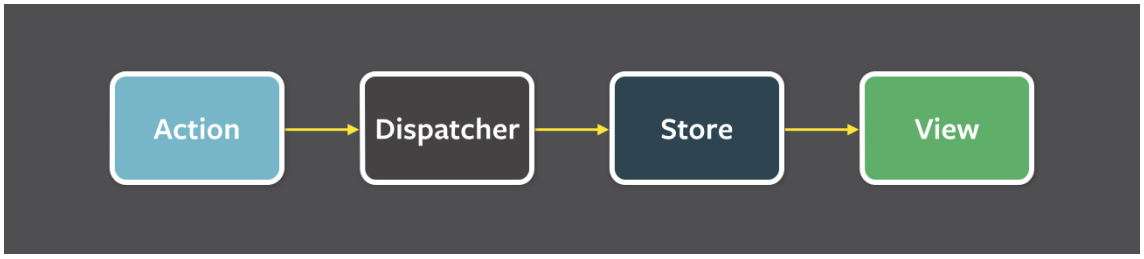

Figura 1: Flujo entre componentes de la arquitectura Flux.

No hay una implementación canónica de estos componentes. Facebook ofrece algunas implementaciones como referencia y existen muchas librerías hechas por la comunidad de desarrolladores que profundizan en estos patrones (Reflux, McFly, Fluxxor, Delorean, etc.)

c. ¿Qué es Elm?

Elm es un lenguaje de programación funcional que compila a Javascript, cuyo propósito principal es la construcción de aplicaciones web sin errores en tiempo de ejecución, referencias a *null* o a *undefined*. Compite con React puesto que ambos ofrecen una forma de desarrollar muy declarativa. El lenguaje está muy inspirado en otros lenguajes funcionales con tipado estático y se puede apreciar en su sintaxis similar a Haskell.

Como consecuencia del propio diseño del lenguaje, la comunidad empezó a organizar sus aplicaciones en base a un patrón que se ha bautizado como *The Elm*

[5] **Aplicaciones Frontend:** Programas que son destinados para el uso por parte de usuarios finales. Suelen contar con una interfaz gráfica y requieren de un servidor o *Backend* para funcionar.

Architecture. Esta estructura propone una comunicación unidireccional entre sus partes y sus conceptos se pueden portar a otros lenguajes de forma muy sencilla.

La arquitectura consta de 3 partes. Para explicarla nos apoyaremos en un ejemplo muy simple, un contador que ofrece dos botones para incrementar o disminuir una unidad:

Modelo: El estado de la aplicación. En este caso (Figura 2) es un número entero. Se especifica un alias sobre el tipo *Int* para facilitar la legibilidad. A continuación, se declara una variable llamada *model* del tipo *Model* (nuestro alias) y se inicia a 0.

```
type alias Model = Int

model : Model
model =
  0
```

Figura 2: Definición de un Modelo en Elm.

Actualización: Una función para actualizar el estado. Para definir como nuestro modelo puede cambiar debemos especificar los eventos del dominio a los que debe reaccionar. En la Figura 3, creamos un tipo unión *Msg* que únicamente puede tomar los valores *Increment* o *Decrement.* La función *update* toma como parámetros un evento y el modelo para devolver el nuevo estado de la aplicación actualizado.

```
type Msg = Increment | Decrement

update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      model + 1

    Decrement ->
      model - 1
```

Figura 3: Definición de un evento de dominio y una función de actualización en Elm.

Vista: Una función para renderizar el estado. Toma como parámetro el modelo del que obtiene los datos a representar y devuelve un fragmento HTML que puede lanzar eventos de dominio. La invocación de esta función se produce como consecuencia de cualquier actualización en el modelo.

La Figura 4 es un ejemplo de una función Vista.

```
view : Model -> Html Msg
view model =
  div []
    [ button [ onClick Decrement ] [ text "-" ]
    , div [] [ text (toString model) ]
    , button [ onClick Increment ] [ text "+" ]
    ]
```

Figura 4: Definición de una Vista en Elm.

Las ventajas de esta estructura son claras. Toda nuestra lógica de negocio queda recogida en las funciones de actualización que son muy faciles de probar y depurar. El estado de la aplicación queda representado de forma declarativa y se puede recuperar cualquier momento anterior a partir de los eventos de dominio que han sido lanzados.

d. CQRS – Command Query Responsibility Segregation

Es un patrón arquitectónico que pretende cumplir el *principio de responsabilidad única (SRP)* de *SOLID* con mayor exigencia que otras arquitecturas. Considera que la modificación y la consulta del estado de una aplicación son dos responsabilidades distintas y por ello se ha de construir un subsistema para cada una de estas tareas.

Por un lado, tenemos los *Comandos* que son peticiones para realizar operaciones de negocio que alteran el estado sistema. Por otra parte, están las *Consultas* que únicamente lo leen.

Cuando la dificultad del dominio crece lo suficiente, mantener un único modelo para consulta y escritura lleva a más complejidad y que el modelo no satisfaga ambas responsabilidades. Separando modelos podemos aplicar distinta lógica de negocio más específica.

CQRS suele ser una arquitectura muy adoptada en sistemas basados en microservicios ya que aporta la modularidad que estos sistemas buscan. La principal ventaja de la separación de subsistemas es que pueden evolucionar y escalar por separado.

Sin embargo, como con cualquier otro patrón de diseño, su implementación conlleva algunas contras. El principal problema de la separación de escritura y lectura es mantener ambos sistemas sincronizados y definir un protocolo de transacción para asegurar que no existan condiciones de carrera. Se ha de considerar si la arquitectura encaja con nuestra aplicación y si las ventajas que aporta su implementación son superiores a las dificultades que pueda presentar.

e. Event Sourcing

Una de las principales ventajas de la aplicación de *CQRS* es que los *Comandos* nos permiten incluir eventos de dominio para aumentar la riqueza de nuestro modelo. Gracias a estas acciones podemos mantener un historial con todos los sucesos que han alterado el estado de nuestra aplicación.

Por ejemplo, en una aplicación de *e-commerce* podríamos modelar el carrito de compra a base de eventos del tipo *ItemAdded*, *ItemRemoved*, *ClearCart*, *PayCart*, etc.

Event Sourcing nos proporciona mecanismos para deshacer y rehacer acciones, auditoría y reconstrucción del estado.

El propósito de este patrón es mantener un registro ordenado de los eventos que se propagan en la aplicación para poder conocer cómo se ha llegado al momento actual. *CQRS y Event Sourcing* suelen ir acompañados ya que este patrón ayuda al principal problema que nos planteaba el anterior: la sincronización de los sistemas de lectura y escritura. Para ello, debemos asegurarnos de que todos los cambios al estado de la aplicación pueden ser representados como objetos para que puedan ser guardados en secuencia.

Por otra parte, cualquier evento genera el almacenamiento de nuevos objetos lo que provoca que este patrón requiera de mucho espacio de almacenamiento y las consultas para reconstruir estados anteriores pueden convertirse en un proceso muy pesado.

También se han de considerar los eventos de dominio que requieren operaciones asíncronas y otros efectos secundarios ya que se han de manejar de forma distinta para no romper los mecanismos de atomicidad y mantener el orden correcto en el que los eventos se van recibiendo.

f. Principios de Redux

Redux pretende que el manejo del estado a lo largo de la aplicación sea predecible. Para ello, se han de imponer restricciones a la hora de actualizar el estado. Estas restricciones quedan recogidas en tres principios:

Única fuente de verdad. El estado de toda la aplicación se representa como un árbol (Figuras 20, 35 y 37) y, a diferencia de Flux, está almacenado en un único contenedor llamado *Store*. Todos los componentes de la interfaz extraen los datos de este objeto lo que hace que sea más fácil depurar la aplicación y proporciona la capacidad de guardar y restaurar el estado en cualquier momento.

Trasladándolo a un ejemplo cotidiano, es mucho más fácil consultar la cifra total de tus ahorros si están en una única cuenta bancaría, que si están repartidos en varias.

Estado protegido contra escritura. El contenedor que almacena el estado no tiene métodos para modificar el estado directamente. La única forma de cambiarlo es mediante la propagación de *Acciones* que describen qué ha pasado. El mecanismo para lanzar estos eventos esta centralizado lo que ayuda a que no existan condiciones de carrera y se almacenen en el orden correcto.

Por ejemplo, si tenemos un libro de firmas en una boda cuyas entradas están ordenadas cronológicamente, para asegurar ese orden a la hora de escribir en el libro, se establecen turnos para que dos personas no puedan escribir a la vez. Sin embargo, varias personas alrededor del libro si pueden leerlo al mismo tiempo.

Las modificaciones deben ser funciones puras. Este es un concepto que proviene de la programación funcional. Significa que una función no tiene efectos colaterales, es decir, una misma entrada produce siempre la misma salida sin afectar a otros contextos. Por tanto, para especificar cómo ha de cambiar el estado de la aplicación, se utilizan funciones puras llamadas *Reducers*.

g. ¿Cómo funciona?

Redux crea un flujo unidireccional de los datos en nuestras aplicaciones. Está compuesto principalmente por 3 elementos, como se puede observar en la Figura 5.

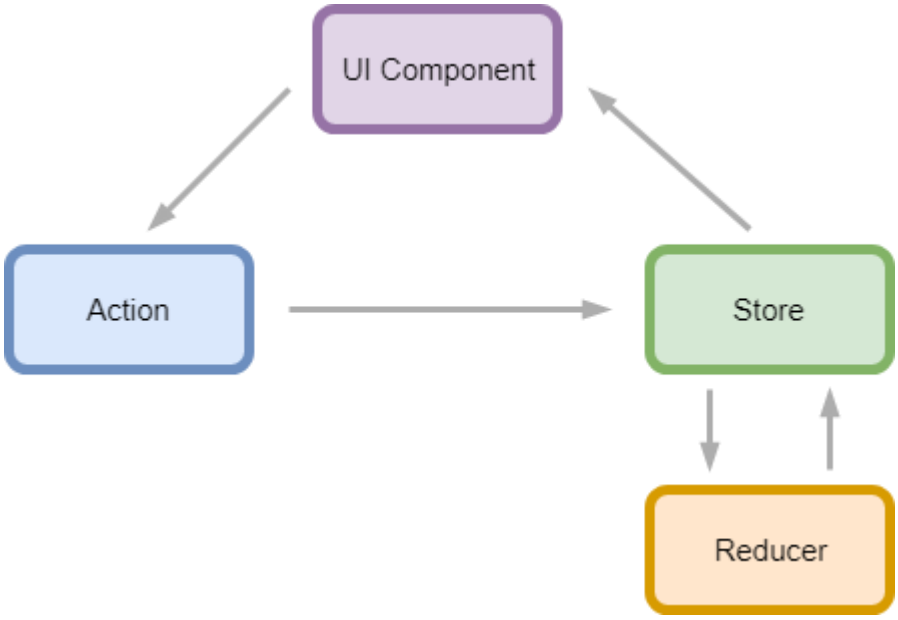

Figura 5: Flujo entre los componentes de la arquitectura Redux.

Acciones

Las acciones son portadores de la información que indica cómo se quiere modificar el estado. Se envían desde la aplicación hasta el *Store*.

Generalmente son objetos de Javascript que contienen un campo *type* que representa el tipo de la acción. Estos tipos son definidos como constantes *Strings.*

En la Figura 6 vemos un ejemplo de acción que expresa la intención de aumentar el contador del estado.

```
const action = {
  type: INCREASE_COUNTER
}
```

Figura 6: Definición de una Acción.

El resto de la estructura del objeto es libre. Se pueden incluir parámetros, errores, indicaciones para paginación… pero lo ideal es pasar la menor cantidad de información posible para facilitar el manejo de estos objetos.

En la Figura 7 ampliamos el ejemplo anterior indicando que la acción para aumentar el contador quiere sumar 5 unidades.

```
const actionWithParameter = {
  type: INCREASE_COUNTER,
  payload: 5
}
```

Figura 7: Definición de una Acción con un parámetro.

Reducers

Los *Reducers* son las funciones que se encargan de especificar cómo se ha de mutar el estado en base a las acciones recibidas. Son funciones puras que, dado un estado y una acción, devuelven un nuevo estado de forma determinista. Reciben un nombre propio porque comparten la misma estructura.

Para conseguir que actúen de forma determinista el estado debe ser Inmutable. Esto significa que, con cada operación se ha de devolver una copia del estado modificado.

La Figura 8 es un ejemplo de modificación inmutable de listas. Partiendo de una colección de números, queremos conseguir aquellos que son mayores que 2. La función *filter* deja la lista original inalterada y devuelve una nueva con el filtro aplicado.:

```
const arrayOfNumbers = [1,2,3]
const numsGt2 = arrayOfNumbers.filter(number => number > 2)

console.log(arrayOfNumbers)   // [1,2,3]
console.log(numsGt2)          // [3]
```

Figura 8: Ejemplo de una modificación inmutable.

Esta forma de manejar los datos proporciona varios beneficios como la predictibilidad que Redux busca. La mutación esconde cambios que crean efectos secundarios, por lo tanto, la inmutabilidad es más fácil de depurar y menos propensa a errores.

Continuando con nuestro ejemplo del contador, en la Figura 9 se observa como el *Reducer* encargado de modificarlo recibe el anterior estado (ponemos 0 como valor por defecto para indicar en qué valor empieza el contador) y la acción lanzada. Según el tipo de la acción se devuelve un nuevo estado u otro. También se hace uso de los parámetros que las acciones llevan asociados en la lógica de la modificación.

```
const reducer = (state = 0, action) => {
  switch (action.type) {
    case INCREASE_COUNTER:
      return state + action.payload
    case DECREASE_COUNTER:
      return state - action.payload
    case RESET_COUNTER:
      return 0
    default:
      return state
  }
}
```

Figura 9: Ejemplo de un *Reducer*.

Puesto que es irreal que el estado de nuestra aplicación se reduzca a un contador necesitamos componer los *Reducers* para cubrirlo y mantener las funciones lo más simple posible. Para ello, Redux proporciona una función de utilidad *combineReducers()*.

Store

El *Store* es el objeto que almacena el estado de la aplicación. Implementa el patrón de diseño *Observer* para comunicarse con los componentes encargados de renderizar la interfaz de forma reactiva.

El estado se puede consultar en cualquier momento gracias al método *getState()*. Para modificarlo se ha de llamar a *dispatch(action).*

Es de vital importancia que solo exista un *Store* en toda la aplicación para que se cumpla el principio de *Única fuente de verdad*. Siempre que se tenga que ampliar el estado se ha de hacer mediante la composición de *Reducers*.

Para crear un objeto *Store* se debe importar la función *createStore* que nos proporciona Redux a la que hay que pasarle por parámetros el *Reducer* raíz creado con *combineReducers*.

En la Figura 10 se presenta el ejemplo del contador entero:

- En las primeras líneas definimos las constantes de tipo *String* para definir los tipos de las acciones de la aplicación. Es una buena práctica extraer estos valores en constantes ya que pueden ser utilizados en varios *Reducers*. Si hay que cambiar el *String* del tipo por alguna razón de esta forma sólo ha de ser modificado en un lugar.
- Seguidamente definimos los 3 eventos que se pueden lanzar en esta aplicación: Aumentar el contador, disminuirlo o reiniciarlo.
- Definimos el *Reducer* encargado de manejar las modificaciones al contador.
- Creamos nuestra *Store*. Ya que esta aplicación tiene un único *Reducer* no es necesario hacer uso de *combineReducers* y podemos considerarlo como nuestro *Reducer* raíz.
- En este ejemplo nuestro equivalente a la interfaz gráfica es una suscripción al *Store* que con cada actualización imprime por consola el resultado.
- Para lanzar acciones y modificar el estado de la aplicación utilizamos el método *dispatch* del *Store* que provoca una impresión por consola.

```
import {createStore} from 'redux'

const INCREASE_COUNTER = "INCREASE_COUNTER"
const DECREASE_COUNTER = "DECREASE_COUNTER"
const RESET_COUNTER = "RESET_COUNTER"

const increaseAction = { type: INCREASE_COUNTER }
const decreaseAction = { type: DECREASE_COUNTER }
const resetAction = { type: RESET_COUNTER }

const counterReducer = (state = 0, { type }) => {
  switch (type) {
    case INCREASE_COUNTER:
      return state + 1
    case DECREASE_COUNTER:
      return state - 1
    case RESET_COUNTER:
      return 0
    default:
      return state
  }
}

const store = createStore(counterReducer)

const unsubscribe = store.subscribe(() => {
  console.log(store.getState())
})

store.dispatch(increaseAction)  // 1
store.dispatch(increaseAction)  // 2
store.dispatch(increaseAction)  // 3
store.dispatch(decreaseAction)  // 2
store.dispatch(resetAction)     // 0

unsubscribe()
```

Figura 10: Ejemplo de aplicación contador.

Otros componentes

Aunque la arquitectura es completamente viable con los componentes expuestos, esta estructura serviría únicamente en proyectos con una complejidad mínima.

Action Creators

Volviendo al ejemplo anterior, es posible que nuestros requisitos de negocio nos marquen que las modificaciones al contador no se deben hacer de unidad en unidad.

Podemos añadir un campo extra a las acciones de incremento y decremento para indicar la cantidad deseada. Pero ¿significa esto que debemos hacer infinitos objetos para cada cantidad posible? (INCREMENT_BY_1, INCREMENT_BY_2, INCREMENT_BY_1023, etc.)

Para poder producir acciones de forma genérica podemos hacer uso de *Action Creators*, esto son funciones cuyo único objetivo es devolver acciones.

Continuando el ejemplo del contador, en la Figura 11 observamos un *Action Creator* para crear eventos de incremento de *x* unidades:

```
const incrementActionCreator = (x) => ({
  type: INCREASE_COUNTER,
  payload: x
})
```

Figura 11: Ejemplo de un *Action Creator*.

Podríamos nombrar el parámetro de cualquier forma, pero es interesante que todas las Acciones de nuestra aplicación mantengan la misma estructura para su manejo en *Middlewares* como vamos a ver a continuación.

Middlewares

Los *Reducers*, como se ha especificado antes, solo pueden ser funciones puras. Esto les impide comunicarse con *API*s asíncronas, operar con bases de datos, implementar sistemas de *Logging* y *Crash Reporting*, etc.

Cómo realizar una aplicación que no pueda comunicarse con un servidor o con una base de datos hoy en día es impensable, para realizar estas acciones podemos incorporar *Middlewares* a nuestra *Store*.

Un *Middleware* es una función que es activada cuando una acción es lanzada antes de que llegue a los *Reducers*. Se utilizan para correr efectos secundarios como llamadas a servicios externos o implementar mecanismos de *logging*. Pueden devolver la acción original, una diferente o ninguna.

Para definir un *Middleware* en *Redux* se ha de definir una función que acepte un *Store*, otra función para continuar la ejecución y una acción por parámetros. Además, esta función debe estar *currificada* (Daza, 2016), es decir, debe acepar los parámetros como una secuencia de funciones que toman un único argumento y devuelven otra

función. Esta técnica proviene del paradigma funcional y permite la aplicación parcial de argumentos. Podemos llamar a la función con todos los argumentos y obtener un resultado o podemos proporcionarle un conjunto de estos y obtener una función que espera a tener el resto de parámetros.

En la Figura 12 podemos ver un ejemplo de *Middleware* que imprime por consola el tipo de acción, la fecha y los parámetros (si existen) cada vez que se lanza un evento.

```
const loggingMiddleware = store => next => action => {
  const date = new Date().toUTCString()
  console.log(`Dispatched action ${action.type} at
${date}`)

  if (action.payload) {
    console.log(`With payload ${action.payload}`)
  }

  return next(action)
}
```

Figura 12: Ejemplo de *Middleware*.

El funcionamiento de estas funciones es como una tubería, por eso la última sentencia de la función devuelve el resultado de aplicar el siguiente *Middleware* a la acción.

Selectores

Al conectar la *Store* con los componentes de la interfaz de usuario les otorgamos visibilidad completa del estado de la aplicación gracias al método *getState()*. Generalmente queremos evitar esto ya que no es responsabilidad de las vistas el navegar por el estado, únicamente deben renderizar la parte que se les indique.

Para separar las partes que queremos proporcionar a un componente podemos crearnos funciones *Selectoras*, que dado el estado devuelven una porción de este. También podemos incluir datos derivados en el resultado de estos métodos para aligerar la cantidad de información que almacenamos en el *Store*.

Por ejemplo (Figura 13), si en nuestro estado tenemos un carrito de compra junto con todos los productos que se venden podemos definir una función selector para obtener únicamente el carrito. Esta función acepta el estado por parámetros y se encarga de devolver el trozo correspondiente del carrito. De esta forma, la vista que haga uso de este selector no recibe más información que de la necesaria. Adicionalmente, el selector

incluye un apartado con el precio total del carrito ya que esta información se puede obtener a partir de los datos existentes en el estado y no es necesario mantener un campo más.

```
const state = {
  articles: [...],
  cart: [
    {id: "1", quantity: 2, price: 13},
    {id: "4", quantity: 1, price: 5.60}
  ]
}

const cartSelector = (state) => ({
  cart: state.cart,
  totalPrice: state.cart.reduce(
    (acc, item) => acc + item.price, 0.0
  )
})
```

Figura 13: *Ejemplo de un selector.*

h. Diferencias respecto a otras arquitecturas

La principal diferencia respecto a otros patrones arquitectónicos en aplicaciones *Frontend* es el flujo de datos dentro del sistema. En las arquitecturas basadas en *Modelo Vista Controlador* el controlador es el responsable de manejar los datos una vez le llegan, pero esos datos le llegan desde varios puntos (las vistas, el modelo, otros controladores e incluso el mismo componente). Esto no es ningún problema, pero se puede convertir en uno cuando la aplicación crece:

- Se vuelve fácil perder el control de quien es el responsable de provocar un cambio en el estado de un controlador.
- Las operaciones asíncronas (y efectos secundarios en general) provocan que los cambios se vuelvan indeterministas.
- La mutabilidad esconde los cambios por lo que crea efectos secundarios y puede volver el código propenso a errores.

En realidad, estos problemas se pueden resumir en uno solo: no existe una *Única fuente de verdad.* Esto es precisamente lo que Redux intenta combatir. Se establece un

flujo unidireccional de los datos de forma que cada componente solo recibe información por un punto. Las actualizaciones del estado se vuelven síncronas y se ha de utilizar el mecanismo propuesto para realizar estas transformaciones obligatoriamente porque el resto de puntos de acceso al estado son de solo lectura.

Si los datos entran en nuestro sistema en una única dirección conseguimos que rastrear el origen de los cambios sea muy fácil. Esto tiene un impacto muy beneficioso en la mantenibilidad del software, tanto a la hora de depurarlo y corregir fallos, como a su capacidad de evolución.

Trabajar sobre Redux nos proporciona previsibilidad lo que nos ayuda a evitar fallos en el diseño del sistema y podemos concentrarnos en programar nuevas funcionalidades, que es lo que el cliente final desea.

i. Desventajas de Redux

No existen soluciones perfectas en el desarrollo de software. Aplicar Redux en nuestros sistemas nos aporta muchas ventajas, pero nos somete a duras restricciones:

- El estado de la aplicación y sus cambios han de ser representados como objetos.
- Toda la lógica relativa a cambios del estado se ha de implementar como funciones puras.

Otra de las mayores quejas por parte de la comunidad de Redux es que es necesario mucho código que no aporta valor al sistema, pero que es necesario para el funcionamiento de la arquitectura. Hablamos de declaración de constantes para los tipos de las Acciones, duplicación de código para definir *Action Creators* y *Reducers*, etc.

Por otro lado, las modificaciones a un estado con mucha anidación, al ser este inmutable, puede llevar a código difícil de leer y mantener. Debemos mantenerlo lo más plano y simple posible para evitar situaciones como la que se muestra en la Figura 14.

Nuestro estado en esta ocasión contiene información de un restaurante. En el primer nivel del árbol podemos encontrar los pedidos de los clientes, un indicador de si el restaurante esta abierto o cerrado y el número total de mesas. En el siguiente nivel, dentro de las peticiones de los clientes se encuentran los pedidos organizados por numero de mesa. Dentro de cada mesa se encuentran los platos divididos por categoría de la carta del restaurante. Por último, cada plato tiene un id, un nombre y la cantidad de raciones que se han servido hasta el momento. Si quisiéramos añadir una ración de un plato para una mesa la modificación ha de ser inmutable. La variable *alteredRestaurant* es un ejemplo de ello.

```
const restaurant = {
  clientRequests: {
    "1": {
      "drinks": [
        {id: 3, name: "Beer", quantity: 4}
      ]
    },
    "2": {
      "appetizers": [
        {id: 1, name: "Onion rings", quantity: 2},
        {id: 2, name: "Cheese & bacon fries", quantity: 1}
      ],
      "drinks": [
        {id: 3, name: "Beer", quantity: 3}
      ]
    }
  },
  isOpen: true,
  totalTables: 7
}

// We want to add one more ration of fries for table number 2

const alteredRestaurant = {...restaurant,
  clientRequests: {...restaurant.clientRequests,
    "2": { ...restaurant.clientRequests["2"],
      "appetizers":
restaurant.clientRequests["2"]["appetizers"].map(appetizer =>
        (appetizer["id"] === 2) ? {...appetizer, quantity:
appetizer.quantity+1} : appetizer)
    }
  }
}
```

Figura 14: *Ejemplo de modificación inmutable de un estado con mucha anidación.*

Existen soluciones paliativas para estos problemas, pero generalmente pasan por el uso de librerías de terceros o la implementación de nuestros propios mecanismos auxiliares. En cualquier caso, es una decisión que se ha de tomar con cuidado porque puede tener un impacto muy grande en la mantenibilidad del sistema.

j. React y Redux

React es una librería de código abierto desarrollada por Facebook lanzada en el 2013 para la creación de interfaces de usuario web basadas en el desarrollo por componentes. Esto consiste en la creación de bloques pequeños de la interfaz (como un botón, un contenedor, etc.) con su propio estado interno, para después componerlos para crear interfaces completas.

React crea su propio *DOM* (*Document Object Model)* virtual (Facebook). Esta estructura de datos es usada por los navegadores webs para construir un árbol a partir del html y crear una representación visible. En el caso de React, crea su propia estructura con la que después compara y actualiza las diferencias eficientemente en el *DOM* del navegador.

Otra de las características principales de React es que es muy declarativo. Su sintaxis expresa qué ocurre, pero nos oculta cómo. Esto hace que el código sea muy legible y fácil de arreglar en caso de fallo.

React no asume el uso de ninguna otra dependencia, únicamente se centra en el desarrollo de la interfaz. Por esta razón, conjunta muy bien con Redux y con otras librerías para el manejo del estado global como MobX o Flux. React hace muy buen trabajo escondiendo la manipulación del DOM y la asincronía, pero el manejo del estado es responsabilidad del desarrollador. Por ello se suele acompañar de las librerías mencionadas.

Existe una librería llamada *react-redux* que aporta algunas utilidades para que la integración de ambas librerías sea mucho más sencilla:

Por una parte, se proporciona componente *Provider* al que le podemos pasar la *Store* por parámetros para hacerla accesible en toda la jerarquía de componentes de React.

En la Figura 15 podemos un ejemplo de uso del componente *Provider*. En primer lugar, definimos un *Reducer* sencillo con el que construimos un *Store*. El componente raíz *App* consta de un componente *Provider* que recibe *Store* que hemos definido. Todos los componentes que quedan envueltos por el proveedor tienen acceso a este *Store*, en este caso solo hay un componente llamado *FirstComponent*.

```
import {createStore} from 'redux'
import {Provider, connect} from 'react-redux'

const counter = (state = 0, { type }) => {
  switch (type) {
    case "INCREMENT":
      return state + 1
    case "DECREMENT":
      return state - 1
    default:
      return state
  }
}

const store = createStore(counter)

const App = () => (
  <Provider store={store}>
    <FirstComponent />
  </Provider>
)
```

Figura 15: Ejemplo de uso del componente *Provider*.

Por otra, para que un componente pueda acceder a la *Store* que *Provider* expone, debe ser decorado por la función de orden superior *connect()*. A partir de ese momento, el componente tiene acceso al método *dispatch()* pero no al estado, para ello es necesario pasar una función por parámetros a *connect()* a la que se suele llamar *mapStateToProps* que recibiendo un estado nos proporciona las partes que queremos.

Continuando el ejemplo anterior, en la Figura 16 observamos la definición del componente *FirstComponent* que consiste en un contenedor con dos textos. En uno de ellos se muestra el valor actual del contador que se extrae de los *props* del componente. Pero, puesto que el contador está en el *Store* de la aplicación, debemos conectar el componente al estado para que pueda acceder a él. Para ello definimos la función *mapStateToProps()* en la que indicamos que la propiedad *counter* de nuestro componente apunta a la del estado. Por último, hacemos uso de la función *connect()* y nuestro componente *FirstComponente* pasa a tener acceso al *Store* allí donde sea usado.

```
let FirstComponent = (props) => {
  return (
    <div>
      <p>First Component</p>
      <p>Global counter: {props.counter}</p>
    </div>
  )
}
const mapStateToProps = (state) => ({
  counter: state.counter
})

FirstComponent = connect(mapStateToProps)(FirstComponent)
```

Figura 16: Ejemplo de uso de connect en un componente.

Para evitar que todos los componentes queden conectados al *Store* y evitar el acoplamiento se propone una distinción entre componentes de presentación y contenedores.

Los componentes de presentación son aquellos que definen la manera en que se ve la interfaz y no tienen dependencias con el resto de la aplicación. Suelen ser definidos como funciones que reciben por parámetros todo lo necesario para su funcionamiento. Generalmente la composición de estas piezas se hace en otros componentes de presentación más grandes que a su vez componen otros más grandes aún, hasta que eventualmente acaban en un contenedor donde se llenan con información.

```
const Button = ({onClick, text}) => (
  <button style={{marginRight: 6}} onClick={onClick}>{text}</button>
)

const Counter = ({counter}) => (
  <p style={{color: 'blue'}}>Counter <strong>{counter}</strong></p>
)
```

Figura 17: Ejemplo de definición de dos componentes de presentación.

Los contenedores indican cómo funciona la interfaz, suelen estar decorados con *connect()* y sirven de fuente de datos para los componentes de presentación.

En la Figura 18 se muestra un ejemplo de contenedor que hace uso de los componentes de presentación creados en la Figura 17. Se definen las funciones *handleIncrement()* y *handleDecrement()* que serán llamadas cuando se haga click en los

botones. Este contenedor está conectado al *Store* y obtiene de él el contador que pasa a uno de los componentes hijos para que lo pinte en pantalla.

```
class CounterContainer extends React.Component {

  handleIncrement = () => {
    this.props.dispatch({type: "INCREMENT"})
  }

  handleDecrement = () => {
    if (this.props.counter > 0)
      this.props.dispatch({type: "DECREMENT"})
  }

  render() {
    return (
      <div>
        <Counter counter={this.props.counter} />
        <Button onClick={this.handleIncrement} text="Increment" />
        <Button onClick={this.handleDecrement} text="Decrement" />
      </div>
    )
  }
}

const mapStateToProps = (state) => ({
  counter: state.counter
})

const ConnectedCounterContainer =
    connect(mapStateToProps)(CounterContainer)
```

Figura 18: Ejemplo de definición de un componente contenedor.

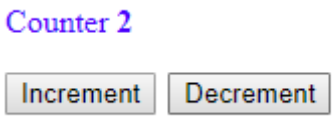

Figura 19: Resultado final del componente de la Figura 18.

Gracias a esta distinción se consigue separación de responsabilidades y reusabilidad ya que podemos usar el mismo componente de presentación con distintos contenedores.

4. Redux en una aplicación móvil para clientes de restaurantes

En España hay más de 79.000 restaurantes y 6 de cada 10 españoles con trabajo comen fuera de casa regularmente. Por otra parte, en 2016 España se convirtió en el país con más móviles por habitante y actualmente estamos entre los 5 países del mundo que más tiempo pasa *online* a través de estos aparatos.

En 2016, un grupo de alumnos de la ETSINF dimos comienzo a un proyecto bautizado COME, que pretende revolucionar el mundo de la hostelería gracias al uso de las nuevas tecnologías.

Generalmente, cuando comemos fuera de casa no queremos complicaciones. Esperamos un servicio rápido y poder pagar con tarjeta, pues en pleno 2017 solo aceptar efectivo en un negocio es una limitación. Si además, comemos fuera de casa entre semana porque trabajamos, lo ideal sería que al llegar al restaurante ya tengan mi plato de comida listo y caliente. O, por el contrario, si salimos a comer por ocio y queremos probar algo nuevo, tampoco queremos dar muchos rodeos hasta encontrar un sitio que nos guste.

Nuestro principal producto es una aplicación móvil para los clientes de restaurantes a través de la cual puedan pedir y pagar en sus locales habituales. De esta forma, tanto el restaurante como el cliente se ahorran dos de los tres principales tiempos de espera cuando comen fuera de casa: la espera hasta que el camarero les atiende y hasta que les cobran la cuenta.

Por otra parte, el restaurante consigue de agilizar su forma de trabajar logrando atender a más gente ofreciendo el pago con tarjeta. También ganan exposición para captar a más clientes el entrar en el mundo de Internet.

a. Requisitos

Registro de usuarios: Los usuarios han de introducir su email y una contraseña para quedar registrados en nuestra base de datos.

Identificación de usuarios: Los usuarios registrados pueden identificarse en la aplicación introduciendo el email y la contraseña con la que se registraron.

Identificación por Facebook: Los usuarios pueden utilizar su cuenta de Facebook para identificarse en la aplicación.

Cerrar sesión: Un usuario identificado puede cerrar su sesión en la aplicación para identificarse con otra cuenta.

Acceso anónimo: Un usuario puede entrar a la aplicación sin identificarse, pero las funcionalidades a las que puede acceder son limitadas.

Consultar lista de restaurantes: Se muestra al usuario la lista de restaurantes en los que puede pedir con COME. Cada local de la lista informa también de su ubicación y si esta abierto o cerrado en el momento actual.

Consultar la carta de un restaurante: El usuario puede visualizar la carta de un restaurante al completo dividida por categorías. Cada plato muestra su precio con IVA.

Consultar detalles de un plato: Se muestra al usuario una descripción más detallada del plato como sus ingredientes o algún comentario por parte del restaurante. También se visualiza la información de alérgenos del plato.

Añadir y quitar platos del carro: Un usuario identificado puede añadir y quitar los platos que desea comprar al carrito. Cada vez que se modifique se ha de mostrar al usuario cuantos elementos hay y que precio total tiene.

Pagar los elementos del carro: Cuando el cliente tiene clara su elección ha de proceder a pagar su pedido para que el restaurante empiece su preparación. Para ello el usuario ha de introducir los datos de su tarjeta de crédito, el email si se ha identificado por Facebook, y un comentario opcional para cocina.

Consultar historial de pedidos: Un usuario identificado puede consultar la lista de pedidos realizados anteriormente ordenados cronológicamente de más reciente a más antiguo acompañados de su importe.

Consultar detalles de pedidos pasados: Se muestran al usuario los detalles de un pedido como los platos que compró y el comentario que dejó a cocina.

b. Tecnologías usadas

Para desarrollar la aplicación usamos *React Native* (Facebook). Un *framework* desarrollado por Facebook que permite construir aplicaciones móviles con Javascript y React. La principal ventaja es que nos permite desarrollar la aplicación para Android e iOS con una sola base de código y gracias a ello hemos podido construir un Producto Mínimo Viable en muy poco tiempo.

Para acelerar este proceso más aún, utilizamos *Firebase* como nuestro *backend*. Este producto nos ofrece una base de datos y un sistema de identificación de usuarios

con integraciones con las redes sociales más populares. Todo configurado y listo para ser utilizado bajo unas *APIs* muy sencillas.

El uso de Redux en el proyecto viene justificado porque muchas partes del estado son utilizadas en varios puntos de la aplicación. Compartir esta información entre componentes hubiera sido una tarea muy costosa y el resultado sería propenso a errores. Sin embargo, gracias a Redux, todo el estado queda organizado en un punto central de forma que no es necesario comunicar componentes de la interfaz entre ellos.

Otra razón para usarlo es la necesidad de almacenar el estado entre sesiones para no descargar la carta de todos los restaurantes cada vez que se abra la aplicación, pues esto puede conllevar gastos en las tarifas de datos móviles de nuestros usuarios. Gracias a que el estado de la aplicación es un objeto serializable podemos guardarlo en la memoria local del teléfono y recargarlo en cada sesión.

c. Estructura del estado

La imagen que se muestra a continuación ilustra estéticamente la estructura del estado de la aplicación en forma de árbol.

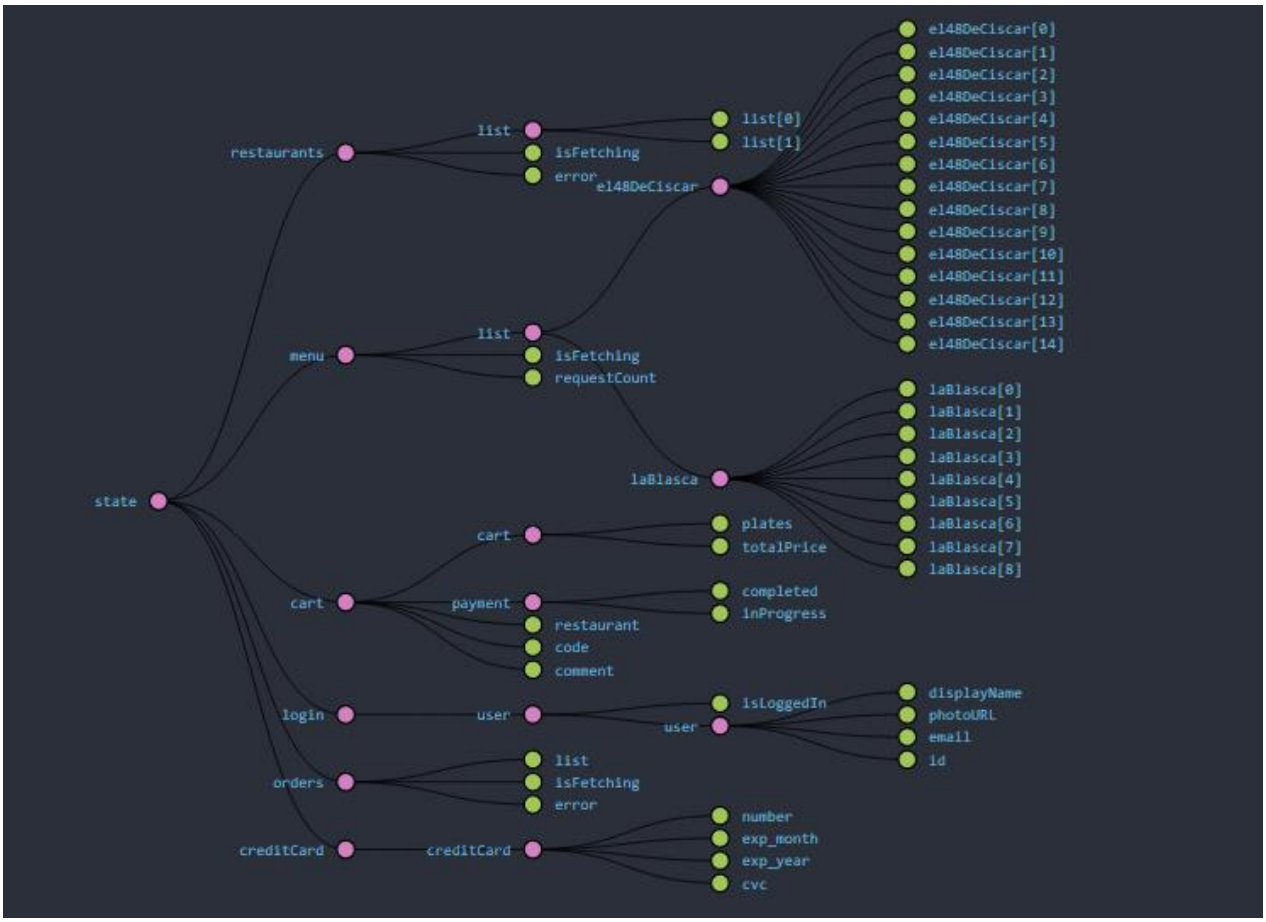

Figura 20: Representación en forma de árbol del estado de la aplicación para clientes de restaurantes.

Una representación más pragmática del estado es su visión en forma de documento JSON.

Figura 21: Representación JSON del estado de la aplicación para clientes de restaurantes.

d. Identificación y Registro

Saber si el usuario está identificado y consultar su información es un proceso que se produce varias veces a lo largo de la aplicación y se lleva a cabo en distintos componentes. Por ello, este suele ser uno de los mejores ejemplos en los que Redux encaja perfectamente.

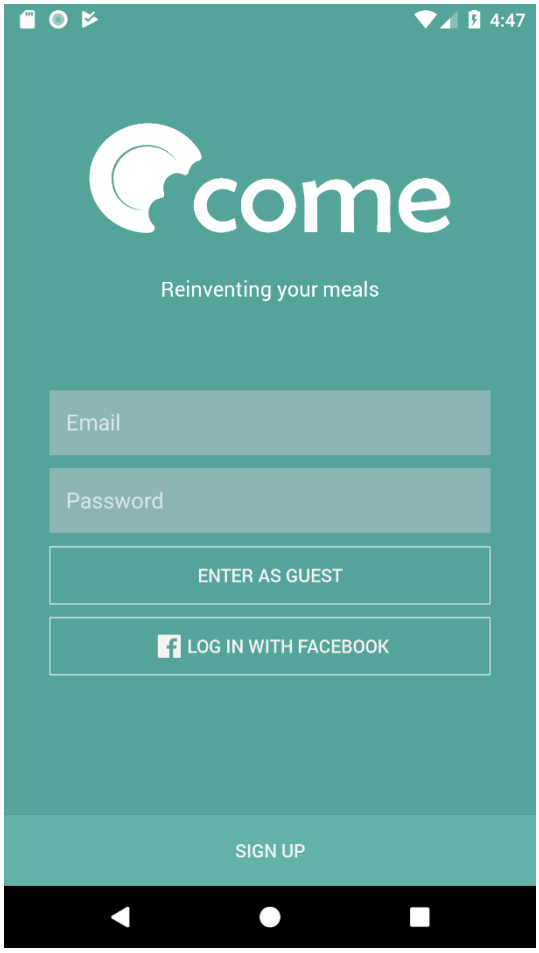

Figura 22: Pantalla para iniciar sesión en la aplicación para clientes de restaurantes.

En el objeto del estado global existe un apartado *login* con una variable *booleana* que indica si hay un usuario identificado y un objeto *user* que reúne toda la información necesaria sobre el mismo.

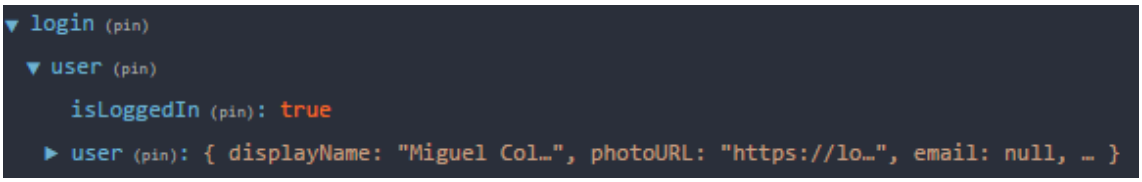

Figura 23: Apartado del estado con la información del usuario identificado.

Para identificar al usuario hay que realizar llamadas a la *API* de autentificación de *Firebase* como se ha hecho mención en el apartado de tecnologías utilizadas. Para ejecutar efectos secundarios utilizamos el *Middleware* "Thunk" (Redux). Esta librería nos permite crear *Action Creators* que devuelven una función en vez de una acción. Dentro de esta, podemos lanzar otras acciones. De esta forma, podemos retrasar el lanzamiento de la acción hasta que se haya completado un efecto secundario.

En el caso de las llamadas a servicios externos, la utilizamos para devolver una función que lanza una acción indicando que ha empezado un proceso y lanzar la llamada. Cuando el servidor nos devuelva el resultado disparamos otra acción indicando si ha sido satisfactorio o erróneo.

En la Figura 24 se muestra como se implementan estas llamadas. Se definen 3 *Action Creators*: uno para lanzar acciones en las que la identificación ha funcionado adecuadamente y se añade un usuario como parámetro, otro para manejar resultados erróneos que lanzan acciones con el error y, el más importante, aquel que hace uso de *Thunk* para lanzar la petición a la *API*.

Este último recibe un email y una contraseña por parámetros y devuelve una función (currificación) que la librería se encarga de llamar con los parámetros restantes (las funciones *dispatch()* y *getState()* del *Store* y un objeto *api* en el que tenemos definidas todas las llamadas que la aplicación puede hacer a servicios externos). Al ejecutar el *Action Creator* lo primero que se hace es lanzar una acción para indicar que se está llevando a cabo una petición. Seguidamente se hace la llamada y una vez obtenemos el resultado se llama a los otros creadores según corresponda para lanzar la acción que produzcan.

```
const loginSuccess = (user) => ({
  type: LOGIN_SUCCESS,
  payload: {
    displayName: user.displayName,
    photoURL: user.photoURL,
    email: user.email,
    id: user.uid
  }
})

const loginError = (result) => ({
  type: LOGIN_ERROR,
  payload: result
})

const emailLogin = (email, password) =>
  (dispatch, getState, api) => {
    dispatch({
      type: REQUEST_LOGIN
    })

    api.emailLogin(email, password)
      .then(user => {
          dispatch(loginSuccess(user))
        }, error => {
          dispatch(loginError(error))
        }
      )
}
```

Figura 24: *Action Creators* para la identificación de usuarios.

El *Reducer* para esta funcionalidad es muy simple. Parte de un estado por defecto en el que la variable que indica si hay alguien identificado está en *false* y el objeto del usuario vacío. Cuando recibe una acción del tipo *LOGIN_SUCCESS* cambiamos el indicador a *true* y rellenamos el usuario a partir del *payload*. En el caso de que se reciba *LOGIN_ERROR* únicamente añadimos un campo *error* al estado para que algún componente lo maneje debidamente.

```
const user = (
  state = {isLoggedIn: false, user: undefined},
  action
) => {
  switch (action.type) {

    case LOGIN_SUCCESS: {
      return {isLoggedIn: true, user: action.payload}
    }

    case LOGIN_ERROR: {
      return {...state, error: action.payload}
    }

    case LOGOUT_SUCCESS: {
      return {isLoggedIn: false, user: undefined}
    }

    default: {
      return state
    }

  }
}
```

Figura 25: *Reducer* encargado del tratamiento del usuario identificado en la aplicación.

Podemos observar como este *Reducer* también es el encargado de la lógica para cerrar sesión. Cuando se recibe una acción del tipo *LOGOUT_SUCCESS* borramos la información del usuario y ponemos el indicador a *false.*

e. Lista de restaurantes

Para mantener el estado lo más plano posible y facilitar cualquier modificación inmutable, dividimos la información de los restaurantes en dos ramas distintas del árbol. Por una parte, su información de presentación como el nombre, la localización, etc. Y por otro lado el menú.

Guardamos esta información en el estado global porque necesitamos saber en qué restaurante se está pidiendo, por lo que varios componentes de interfaz harán uso de estos datos. Además, esta información es bastante estática. Podemos aprovechar la capacidad de serialización del objeto del estado para almacenarlo entre sesiones de la app y ahorrar peticiones con el servidor.

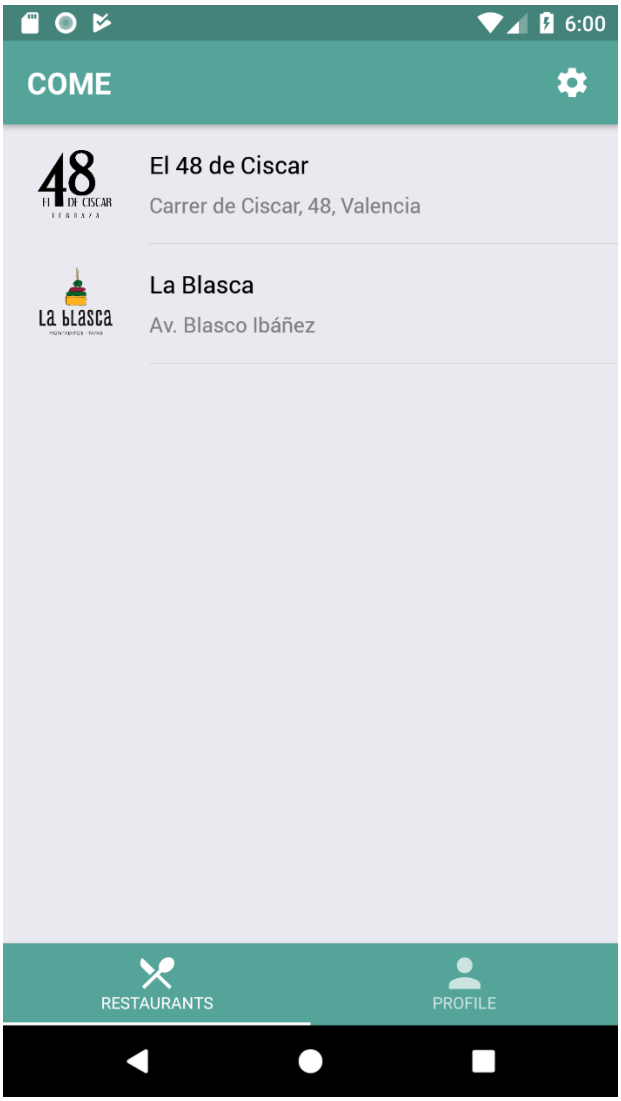

Figura 26: Pantalla para mostrar la lista de restaurantes asociados.

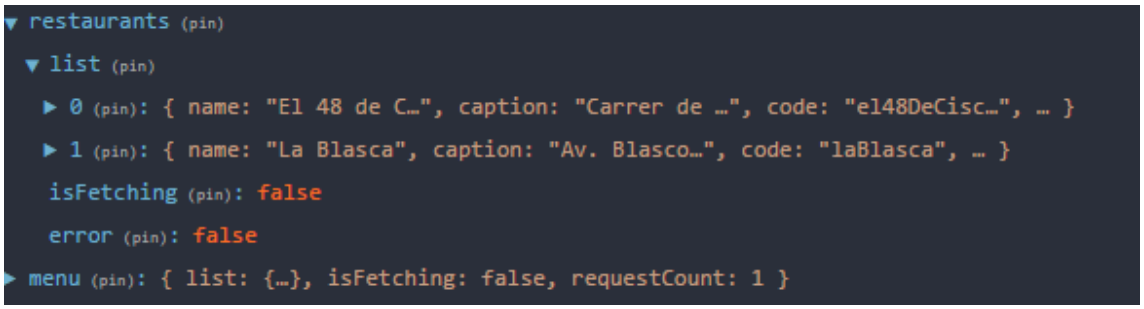

Figura 27: Apartado del estado para almacenar la lista de restaurantes.

Se puede observar como hemos creado una variable *isFetching* en el estado que nos indica si hay una petición al servidor en marcha para recuperar la lista de restaurantes. Hacemos uso de este indicador para dos cosas. La primera es mostrar un componente en la interfaz de usuario para notificar que se está realizando un trabajo en segundo plano y no bloquear la experiencia de uso. La segunda es no lanzar más llamadas de las necesarias. Únicamente realizaremos peticiones si la lista de restaurantes está vacía y no hay otra acción del mismo tipo en marcha.

En la Figura 28 podemos ver como se ha implementado este comportamiento en el *Action Creator* para pedir la lista de restaurantes. Las funciones creadas para hacer uso de "Thunk" deben devolver otra función a la que se le pasa por parámetros otras funciones. En este caso, gracias a *getState* tenemos acceso al estado de Redux donde podemos consultar la variable *isFetching* o la lista actual de locales. Estas comprobaciones se realizan en la función *shouldFetch()* que devuelve un *boolean* para indicar si se puede proceder a realizar la llamada. Si el resultado es positivo, se lanza una acción del tipo *LIST_RESTAURANT_REQUEST* para activar la variable en el estado que informa de que se están realizando llamadas. A continuación, se lanza la llamada al *API* y con el resultado hacemos uso de los otros *Action Creators* según el resultado sea correcto o erróneo.

```
const shouldFetch = (state) => {
  const restaurants = state.restaurants.list
  return !(restaurants.length > 0
     || state.restaurants.isFetching);
}

const restaurants = (response) => ({
  type: LIST_RESTAURANTS_SUCCESS,
  response: response
});

const restaurantError = (error) => ({
  type: LIST_RESTAURANTS_ERROR,
  message: error.message
      || 'Error while fetching restaurants'
});

const getRestaurants = () =>
  async (dispatch, getState, api) => {
    if (shouldFetch(getState())) {
      dispatch({
        type: LIST_RESTAURANTS_REQUEST
      });

      try {
        const response = await api.fetchRestaurants()
        dispatch(restaurants(response))
      } catch (error) {
        dispatch(restaurantError(error))
      }
    }
  }
```

Figura 28: *Action Creators* para la petición de la lista de restaurantes asociados.

Solo 3 tipos de Acciones son responsables de esta funcionalidad: *LIST_RESTAURANTS_REQUEST*, *LIST_RESTAURANTS_SUCCESS* y *LIST_RESTAURANTS_ERROR*. Por ello, el *Reducer* es también bastante sencillo. No obstante, para simplificar las operaciones que han de realizarse está dividido a su vez en 3 *Reducers* más pequeños que después se combinan gracias a *combineReducers*. Se puede observar un claro ejemplo de cómo una misma acción puede ser reutilizada en varios *Reducers*.

El *Reducer* encargado de la lista de restaurantes empieza con una colección vacía y únicamente responderá a los eventos *LIST_RESTAURANTS_SUCCESS* sustituyendo el estado actual por la lista que la acción trae por parámetros.

En el caso de la función responsable del indicador de operaciones en segundo plano se empieza con el valor negativo. Si recibe un evento del tipo *LIST_RESTAURANTS_REQUEST* pondrá el valor *true*, para el resto de casos devuelve *false.*

Por último, el indicador de errores empieza a *false* y cambia el valor según se reciban acciones del tipo *LIST_RESTAURANTS_SUCCESS* o *LIST_RESTAURANTS_ERROR.*

```
const list = (state = [], action) => {
  switch (action.type) {
    case LIST_RESTAURANTS_SUCCESS:
      return action.response;
    default:
      return state
  }
};

const isFetching = (state = false, action) => {
  switch (action.type) {
    case LIST_RESTAURANTS_REQUEST:
      return true;
    case LIST_RESTAURANTS_SUCCESS:
    case LIST_RESTAURANTS_ERROR:
      return false;
    default:
      return state
  }
};

const error = (state = false, action) => {
  switch (action.type) {
    case LIST_RESTAURANTS_SUCCESS:
      return false;
    case LIST_RESTAURANTS_ERROR:
      return true;
    default:
      return state
  }
};

export default combineReducers({
  list,
  isFetching,
  error
})
```

Figura 29: *Reducers* para manejar el estado relativo a la lista de restaurantes asociados.

Una vez obtenemos una lista válida de restaurantes hacemos uso de la librería "redux-persist" (rt2zz) que almacena localmente el estado a través de un *Middleware*, cada vez que el estado es actualizado lo persiste. Así, lo primero que se hace cuando se abre la aplicación es hidratar el *Store* con el último estado guardado.

f. Carrito de compra

Probablemente este es el fragmento del estado que más complejidad conlleva, y, aun así, limitarnos a usar funciones puras sumado a que no hay asincronía implicada en estos procesos hace que sea muy fácil de entender.

Una vez los usuarios han entrado en un restaurante pueden añadir y quitar elementos del menú a su carrito de compra.

Figura 30: Pantalla para consultar el menú de un restaurante y añadir productos al carro.

Figura 31: Apartado del estado para el carro de compra.

Las dos Acciones más importantes en este proceso son *ADD_PLATE* y *REMOVE_PLATE*. Ambas llevan un objeto Plato de nuestra base de datos como *payload*.

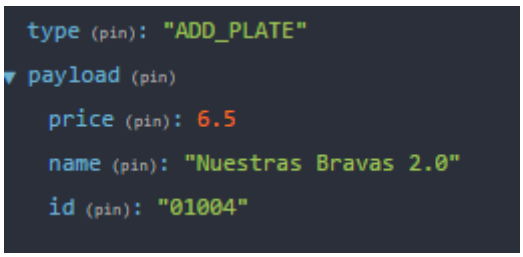

Figura 32: Ejemplo de acción del tipo *ADD_PLATE*.

El *Reducer* ha de encargarse de actualizar la lista de platos y el precio total de forma inmutable en base al *payload* de la acción. Para ello lo primero que hacemos es copiar la lista anterior con el método *slice()*. A continuación, buscamos el plato por nombre. A la hora de añadir, si el plato no existe entonces lo introducimos con cantidad 1. Si por el contrario estamos borrando, restamos 1 a la cantidad y si queda en 0 o menor entonces borramos el plato por completo de la lista. Por último, en ambos casos recalculamos el precio total con el método *reduce()*.

En la Figura 33 se explica el ejemplo con más detalle:

- El estado del carrito empieza con una lista de platos vacía y el precio total a 0.
- Antes de empezar el *switch* creamos una copia de la lista de platos con *slice()* y definimos algunas variables que necesitaremos más tarde.
- El primer paso tanto si se está añadiendo un plato como restando es encontrar el elemento en la lista con el método *find()* y guardar su índice con *indexOf()*.
- Si la acción es de tipo *ADD_PLATE* comprobamos si el resultado de *find()* existe. En tal caso, sumamos una unidad a la cantidad de ese plato. Por el contrario, si el plato no está en la lista, añadimos el parámetro de la acción a la lista con un campo adicional para marcar la cantidad empezando en 1.

- Si la acción es *REMOVE_PLATE* está asegurado que el plato se encuentre en la colección (de otra forma es imposible lanzar esta acción desde la interfaz). Se resta uno a su cantidad y acto seguido se comprueba si la nueva cantidad es menor o igual que 0, en tal caso se elimina de la lista.
- Por último, se recalcula el precio total del carrito gracias a la función *reduce()*.
- También se da la posibilidad a limpiar el carro con un evento del tipo *CLEAR_CART*.

```
const cart =
(state = {plates: [], totalPrice: 0}, action) => {
  let newPlates = state.plates.slice();
  let matchingPlate;
  let indexOfPlate;
  let newPrice;

  switch (action.type) {
    case ADD_PLATE:
      matchingPlate = newPlates
        .find(plate => plate.name === action.payload.name);
      indexOfPlate = newPlates.indexOf(matchingPlate);
      (matchingPlate)
         ? newPlates[indexOfPlate].quantity++
         : newPlates.push({quantity: 1, ...action.payload});

      newPrice = newPlates
        .reduce((sum, curr) => sum + (curr.price * curr.quantity), 0);
      return {plates: newPlates, totalPrice: newPrice};

    case REMOVE_PLATE:
      matchingPlate = newPlates
        .find(plate => plate.name === action.payload.name);
      indexOfPlate = newPlates.indexOf(matchingPlate);
      newPlates[indexOfPlate].quantity--;
      if (newPlates[indexOfPlate].quantity < 1)
        newPlates.splice(indexOfPlate, 1);

      newPrice = newPlates
        .reduce((sum, curr) => sum + (curr.price * curr.quantity), 0);
      return {plates: newPlates, totalPrice: newPrice};

    case CLEAR_CART:
      return {plates: [], totalPrice: 0};

    default:
      return state;
  }
}
```

Figura 33: *Reducer* para gestionar el carro de compra.

Puesto que esta fue la primera aplicación destinada a producción en la que usamos Redux somos conscientes con la experiencia que hemos adquirido en estos dos años que podemos reducir la complejidad de esta parte del estado más aún.

La modificación más evidente es eliminar el campo *totalPrice* del estado ya que se puede computar como un dato derivado a partir de la lista de platos y no es necesario mantenerlo explícitamente. Se puede crear una función *Selector* que a partir del estado calcule el precio total de la misma forma que en el *Reducer*. De esta forma, extraemos esa responsabilidad y la lógica se simplifica.

```
const cart = (state = [], action) => {
  let newPlates = state.slice();
  let matchingPlate;
  let indexOfPlate;

  switch (action.type) {
    case ADD_PLATE:
      matchingPlate = newPlates
        .find(plate => plate.name === action.payload.name);
      indexOfPlate = newPlates.indexOf(matchingPlate);
      (matchingPlate)
        ? newPlates[indexOfPlate].quantity++
        : newPlates.push({ quantity: 1, ...action.payload });
      return newPlates

    case REMOVE_PLATE:
      matchingPlate = newPlates
        .find(plate => plate.name === action.payload.name);
      indexOfPlate = newPlates.indexOf(matchingPlate);
      newPlates[indexOfPlate].quantity--;
      if (newPlates[indexOfPlate].quantity < 1)
        newPlates.splice(indexOfPlate, 1);
      return newPlates

    case CLEAR_CART:
      return []
  }
}

const totalPriceSelector = (state) => {
  const plateList = state.cart.cart.plates
  return plateList
    .reduce((sum, curr) => sum + (curr.price * curr.quantity), 0)
}
```

Figura 34: *Reducer* para gestionar el carrito simplificado con un *Selector*.

5. Redux en un punto de venta web para restaurantes

Es una realidad que la mayoría de los restaurantes en España siguen funcionando con lápiz y papel para anotar las comandas y hacer las cuentas posteriormente. Si además después han de pasarlo a una base de datos u hoja de cálculo supone el doble de trabajo.

Desde COME queremos ofrecer una plataforma con la que el personal del local pueda apuntar las comandas con la facilidad e inmediatez del papel y que se pueda integrar y consultar desde cualquier dispositivo. Aprovechando esta oportunidad, también ofrecemos en el mismo lugar la configuración de la carta y presencia en la aplicación para clientes detallada en el punto anterior. El nombre interno de este proyecto es ZAMPA.

Para este caso si existen numerosas aplicaciones en el mercado con muchas funcionalidades. Pero en su gran mayoría es necesario disponer de hardware especifico como impresoras de una cierta marca o iPads, etc. Nuestra principal ventaja es que somos capaces de integrarnos con cualquier hardware gracias al uso de estándares abiertos y tecnologías web.

a. Requisitos

Identificación: El personal de los restaurantes que utilicen la aplicación ha de poder identificarse.

Cierre de sesión: Los usuarios identificados pueden cerrar su sesión y cambiar de cuenta. Además, una persona no puede utilizar la misma cuenta en varios dispositivos a la vez debido a nuestro plan de comercialización del producto.

Consultar categorías del menú: Cualquier usuario identificado puede ver un listado con todas las categorías de la carta.

Activar y desactivar categorías: Cualquier usuario puede marcar una categoría como visible o invisible.

Modificación categorías: Solo el personal autorizado puede añadir, modificar, borrar y reordenar los elementos del menú.

Consultar platos de una categoría: De forma similar a las categorías, los platos pueden ser consultados por cualquier usuario.

Activar y desactivar platos: Similar al funcionamiento de las categorías.

Modificación de platos: Similar al funcionamiento de las categorías.

Consultar historial de comandas: Todos los usuarios de la aplicación pueden listar todas las comandas realizadas a través de la misma agrupadas por día.

Consulta de estadísticas: Cualquier usuario puede consultar:

- La cantidad de comandas servidas a lo largo del día
- La cantidad de dinero que hay en caja.
- La popularidad de los platos de cada categoría en los últimos 30 días.

Creación de nuevas comandas: Todos los usuarios son capaces de iniciar una comanda nueva siempre que la mesa que introduzca este libre.

Modificar platos a una comanda activa: Cuando un cliente pide un plato el camarero ha de poder registrarlo en la comanda correspondiente. También se da la posibilidad de eliminarlos o cambiar cantidades.

Modificar pagos de una comanda activa: Cuando se procede a pagar, los clientes pueden decidir hacerlo juntos o por separado. Para ello, los camareros añaden pagos a la comanda hasta que no quedan platos.

b. Tecnologías usadas

Este proyecto es una aplicación web desarrollada con React. El uso de Redux con React surge de forma natural, y como en el proyecto anterior, nos ayuda a reutilizar el estado global y organizar mejor el código. Otra de las razones por las que decidimos seguir con esta arquitectura fue porque queríamos utilizarla mejor habiendo aprendido de nuestra aplicación. Revisando el código observo la evolución a mejor pero también veo muchos puntos que podrían ser mejorados.

Respecto a otras tecnologías volvemos a hacer uso de "Thunk" como *Middleware* para llamadas asíncronas y "*redux-persist*" para almacenar partes del estado en memoria entre sesiones.

De las funcionalidades listadas hay algunas que no hacen uso de Redux en absoluto. Una de ellas es la consulta de estadísticas. No es necesario en este caso hacer uso de un estado global ya que esa información solo se va a visualizar en una pantalla concreta y no se va a reutilizar. Tampoco se puede obtener esa información por derivación de otras partes del estado, por ello, únicamente se hace una llamada al

servidor y se renderizan esos datos. Es importante saber cuándo no conviene aplicar Redux, porque, aunque aporta muchas ventajas, también estamos añadiendo complejidad al proyecto.

c. Estructura del estado

De nuevo, se muestran las representaciones en forma de árbol y documento JSON como recurso estético.

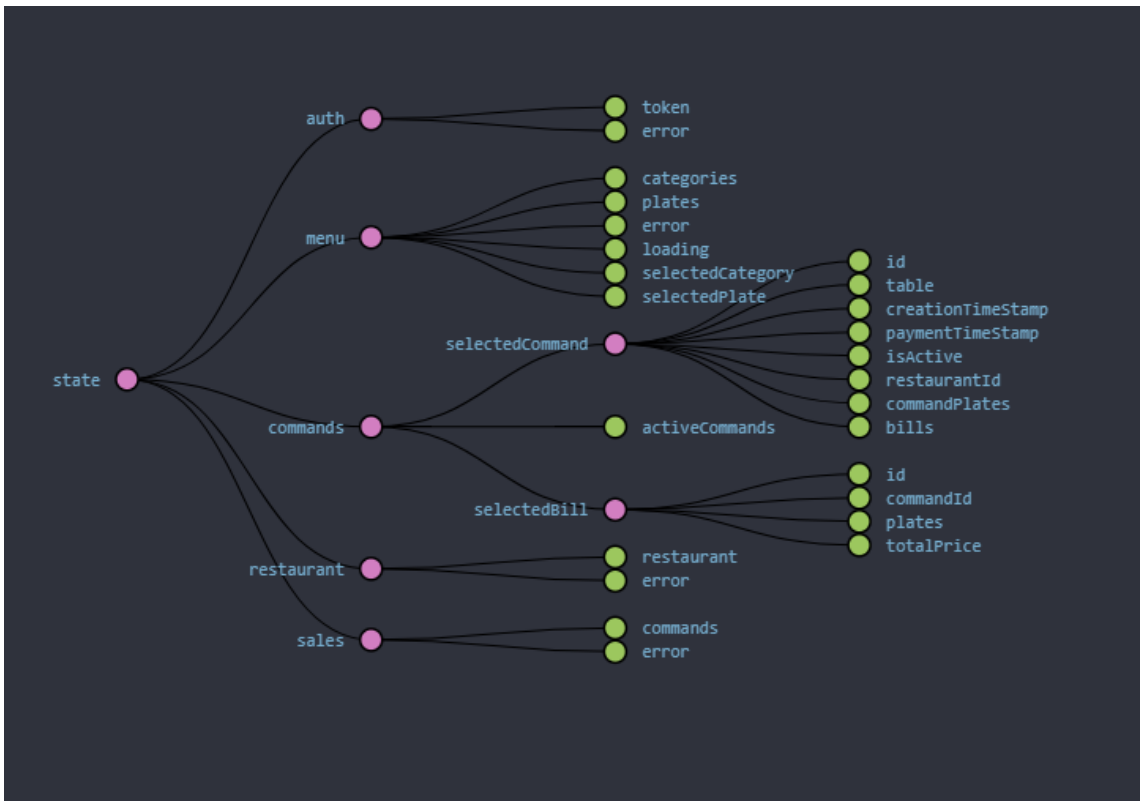

Figura 35: Representación en forma de árbol del estado del punto de venta web.

Figura 36: Representación JSON del estado del punto de venta web.

Se observan algunos campos que empiezan vacíos o a *null*. Esto es porque no se guarda la totalidad del estado entre sesiones para aligerar la velocidad de carga ya que se trae mucha información desde el servidor en cada apartado de la aplicación.

Un estado lleno queda reflejado en la siguiente figura:

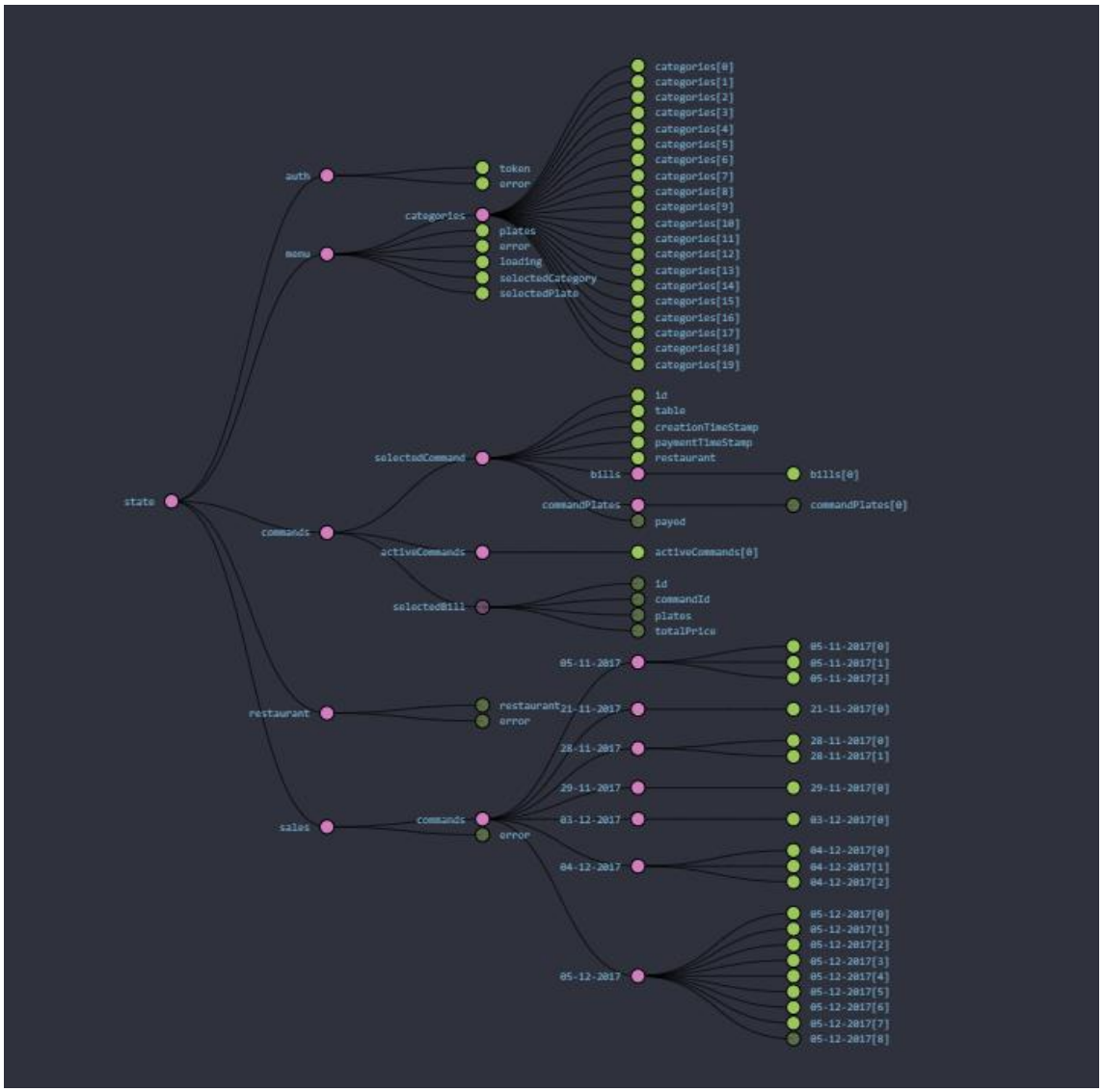

Figura 37: Representación en forma de árbol del estado completo del punto de venta web.

d. Identificación de usuarios

Para hacer uso de la aplicación los usuarios han de identificarse para que cuando haga peticiones al servidor se sepa cuál es su restaurante asociado y qué rol tiene.

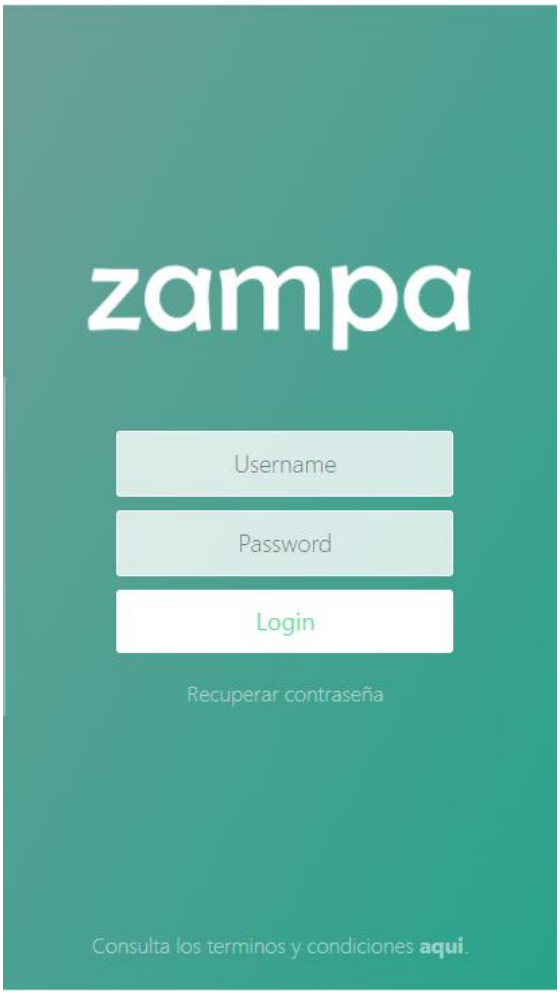

Figura 38: Pantalla para identificación de empleados del punto de venta web.

El mecanismo de autentificación que usamos en el servidor son *Json Web Tokens*. Gracias a ellos, podemos mandar un identificador cifrado asociado a un usuario, de esta forma en el servidor sabemos a qué usuario está vinculado y las acciones que incluyen este *token* en la petición se realizan en su nombre. Debemos guardar este *token* en el estado porque todas las llamadas al servidor lo deben incluir. También es un indicativo para saber si hay un usuario identificado, mientras no haya un *token* en el estado global mostramos la pantalla de identificación.

Desde el servidor podemos saber cuál es el último *token* generado para un usuario, de forma que, si un usuario se identifica en un nuevo dispositivo invalidamos el anterior obligando a que solo haya una sesión simultánea. Todas las llamadas al servidor desde un dispositivo invalidado devuelven el código HTTP 409. Para impedir que se sigan haciendo peticiones, al recibir una respuesta con ese código, cerramos la sesión del usuario automáticamente. Para ello, hemos implementado nuestro propio *Middleware* que comprueba todas las acciones en busca de este código.

Esta función busca en los parámetros de cada acción un *status*. En caso de que exista y sea igual a 409 se elimina el token del estado de la aplicación y se lanza una acción del tipo *LOGOUT*.

```
const sessionExpiredMiddleware = store => next => action => {
  if (action.payload &&
      action.payload.status &&
      action.payload.status === 409) {
        sessionStorage.removeItem("token")
        store.dispatch({type: LOGOUT})
  }

  next(action)
}
```

Figura 39: Middleware para la comprobación de sesiones únicas de empleados.

e. Modificar la carta

A diferencia de nuestra anterior aplicación, esta si requiere que se realicen modificaciones típicas de una aplicación de gestión. Esto provoca que se tenga que implementar lógica de negocio en cliente como vamos a ver a continuación y en otros requisitos.

Para facilitar las operaciones hemos creado un apartado en el estado global que apunta a la categoría sobre la que se están realizando modificaciones. También usamos esta estructura con los platos de una categoría.

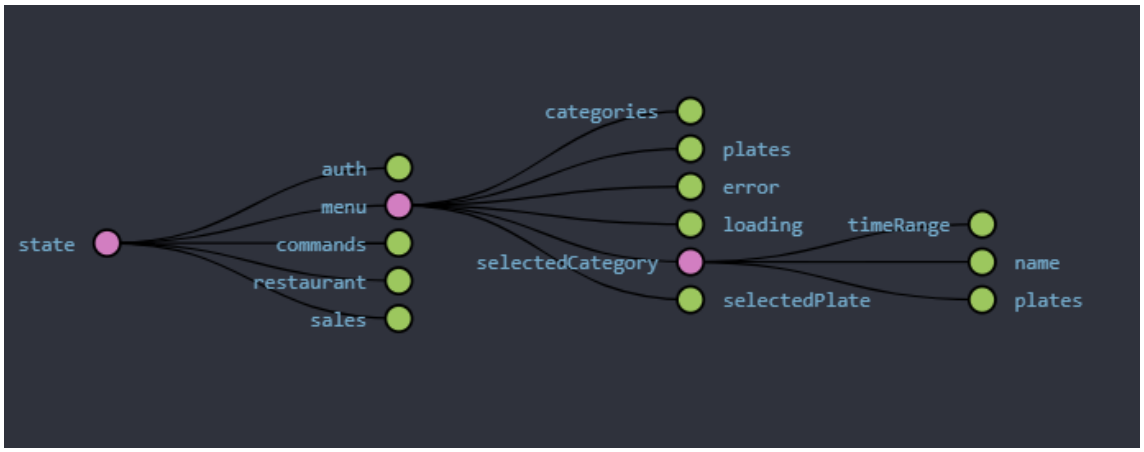

Figura 40: Representación en forma de árbol del fragmento del estado relativo a la modificación del menú de un restaurante.

Los reducers encargados de esa parte del estado solo atienden a las acciones del tipo *SELECT_CATEGORY* y *SELECT_PLATE*. Dado que las modificaciones son inmutables únicamente sustituimos el objeto anterior por su copia actualizada.

```
const selectedCategory = (state = null, action) => {
    switch (action.type) {
        case SELECT_CATEGORY:
            return action.payload
        default:
            return state
    }
}

const selectedPlate = (state = null, action) => {
    switch (action.type) {
        case SELECT_PLATE:
            return action.payload
        default:
            return state
    }
}
```

Figura 41: Reducers encargados de la categoría y el plato seleccionados para editar.

Para crear nuevos elementos hacemos uso de *Action Creators* que lanzan una acción del tipo *SELECT_CATEGORY/PLATE* que llevan como *payload* un objeto con los campos llenos con los valores por defecto. Si, por el contrario, se quiere editar un

elemento existente, utilizamos otros *Action Creators* que lanzan la misma acción, pero llevan el objeto seleccionado como parámetro.

```
export const selectCategory = (category) => ({
    type: SELECT_CATEGORY,
    payload: category
})

export const newCategory = () => ({
    type: SELECT_CATEGORY,
    payload: ({
        timeRange: {
            startTime: "00:00",
            endTime: "23:59"
        },
        name: "",
        plates: []
    })
})
```

Figura 42: Action Creators para seleccionar categorías existentes o crear nuevas.

En ambos casos, se muestra un modal al usuario para que edite los campos que desee. Al acabar, se lanza una acción indicando que sea ha acabado la edición o creación según corresponda para enviar esa información al servidor y que quede registrado en base de datos.

Para borrar elementos se sigue un procedimiento muy similar. Primero se actualiza el estado con el objeto que se quiere borrar y se pide una confirmación al usuario. Si este acepta, se manda directamente la actualización al servidor. En caso de que la respuesta sea satisfactoria se actualiza la lista de categorías/platos filtrando todos los elementos menos el eliminado.

Los *Action Creators* encargados de este proceso transforman los objetos para que correspondan con la forma aceptada por el servidor antes de lanzar la llamada y vuelven a normalizar el resultado.

El *Reducer*, como se puede ver a continuación, responde a los siguientes tipos de acciones:

- *CATEGORIES_SUCCESS*: Evento lanzado cuando la lista de categorías del menú de un restaurante se ha obtenido del servidor. Únicamente se sustituye la colección existente por la nueva.

- *CATEGORY_REMOVE_SUCCESS*: Acción lanzada cuando el servidor ha procesado correctamente la eliminación una categoría de la colección. Se devuelve una lista filtrada sin el elemento eliminado.
- *CATEGORY_EDIT_SUCCESS*: Esta acción se recibe al modificar una categoría con éxito en el servidor. Se devuelve la colección con el elemento en concreto modificado.
- *CATEGORY_CREATE_SUCCESS*: Evento recibido cuando el *backend* procesa correctamente la creación de una nueva categoría. Devolvemos la colección con el nuevo elemento incluido.

El *Reducer* encargado de gestionar los errores responde a todas las acciones que pueden incluir un error.

```
const categories = (state = [], action) => {
    switch (action.type) {
        case CATEGORIES_SUCCESS:
            return action.payload
        case CATEGORY_REMOVE_SUCCESS:
            return state.filter(item => item.id !== action.payload.id)
        case CATEGORY_EDIT_SUCCESS:
            return state.map(item =>
              (item.id === action.payload.id) ? action.payload : item)
        case CATEGORY_CREATE_SUCCESS:
            return [...state, action.payload]
        default:
            return state
    }
}

const error = (state = null, action) => {
    switch (action.type) {
        case CATEGORIES_FAIL:
        case CATEGORY_REMOVE_FAIL:
        case CATEGORY_EDIT_FAIL:
        case CATEGORY_CREATE_FAIL:
        case PLATE_CREATE_FAIL:
        case PLATE_EDIT_FAIL:
        case PLATE_REMOVE_FAIL:
            return action.payload.message
        case CLEAR_ERROR:
            return null
        default:
            return state
    }
}
```

Figura 43: *Reducers* encargados de la gestión de categorías del menú de un restaurante.

Para actualizar listas de forma inmutable utilizamos los métodos *map()*, que permite aplicar una operación a todos los elementos de una colección, y el operador de *destructuring*, que nos posibilita modificar colecciones con una sintaxis equivalente a la usada para su construcción.

f. Realización de comandas

Desde la aplicación ofrecemos la funcionalidad de creación de nuevas comandas para los camareros, de forma que puedan realizar pedidos a cocina desde sus teléfonos o cualquier dispositivo con acceso a internet.

En la pantalla principal se muestra la lista de mesas activas. Se puede entrar a consultar los detalles de estas comandas o crear nuevas. Estas funcionalidades sacan su información del apartado *activeCommands* del estado global. El *Reducer* correspondiente atiende a las acciones *ACTIVE_COMMANDS_SUCCESS* que indica que la llamada al servidor ha sido satisfactoria y *CREATE_NEW_COMMAND* que añade una nueva comanda vacía a la lista.

Para consultar los detalles y realizar modificaciones sobre una comanda, al igual que en la funcionalidad anterior, se ha creado un apartado *selectedCommand* (y *selectedBill*). La mayoría de las acciones relacionadas con esta parte tienen una complejidad elevada (+100 líneas). La inmutabilidad es en parte responsable, pero probablemente esto es una señal de que el estado no está bien organizado ya que mucha de esta lógica debería ser responsabilidad del servidor.

El problema es que, si realizamos estas modificaciones desde el *backend*, por ejemplo, añadir +1 a la cantidad de un plato, deberíamos devolver la comanda entera y esto provocaría que se hiciesen muchas renderizaciones innecesarias de contenido que se mantiene igual en la interfaz, impactando negativamente en el rendimiento. No es posible devolver partes más pequeñas desde el servidor porque el estado no está lo suficientemente separado. Por ello, todas las modificaciones se realizan en los *Action Creators* y todos llaman a un único *endpoint* de la *API* para editar las comandas y que quede registrado en base de datos, pero no esperan su resultado.

Actualmente nos encontramos realizando una refactorización sobre esta parte. Las lecciones aprendidas en proyectos posteriores a este sumado al uso de librerías como *Redux ORM* nos están permitiendo simplificar el estado para su posterior separación.

6. Impacto en la industria del Software

A la luz de las ventajas que Redux aporta a la hora de construir arquitecturas escalables y mantenibles, son varias las empresas que han decidido apostar por la librería y construir sus productos en torno a ella. Algunas de ellas son Spotify, Airbnb, Netflix, Twitter, etc.

Por otra parte, algunas de las ideas que Redux propone han calado más allá del ecosistema de Javascript y se está intentando portar a otros lenguajes y entornos de *frontend* modernos como Kotlin o Swift en el ámbito del desarrollo de aplicaciones móviles (Valiente, 2018). Empresas importantes como Microsoft han adoptado estos patrones en pequeños equipos que, a su vez, gracias a sus filosofías de *open sourcing* cuentan sus experiencias y ayudan a mejorar estas implementaciones.

Como Redux aprovecha que Javascript es un lenguaje no tipado, las librerías que lo imitan en otros lenguajes no llegan a ser totalmente fieles. Por ello, hay desarrolladores que deciden únicamente adaptar ciertas partes de la arquitectura en sus proyectos junto con otras que dirigen el desarrollo hacia un flujo unidireccional y reactivo. Estas arquitecturas toman el nombre de Modelo Vista Intento (Dorfmann, 2016).

- El Intento es una función que coge el *input* del usuario y lo pasa a la función *model*. Generalmente estos *inputs* son Acciones o Comandos.
- El Modelo es otra función que recoge el *output* del Intento para realizar los procesos de negocio necesarios de nuestra aplicación. Cualquier alteración en el Modelo ha de ser inmutable.
- La Vista es la función que recoge el resultado de la función Modelo y lo muestra por pantalla.

El flujo de comunicación entre estas funciones es unidireccional gracias a la reactividad y el patrón Observador.

a. Caso Job and Talent

Una empresa que utiliza una adaptación de este patrón es, por ejemplo, el equipo de iOS de *Job and Talent* (Recuenco, 2017). Han integrado en su arquitectura Model Vista Vista-Modelo un artefacto al que llaman Vista-Estado. De esta forma, sus vistas son el resultado de llamar a una función con el Vista-Estado que a su vez recibe por

parámetros el Vista-Modelo (equivalente al *Store* de Redux), siendo este último su única fuente de verdad en la arquitectura.

$$App = UI(viewState(domainState))$$

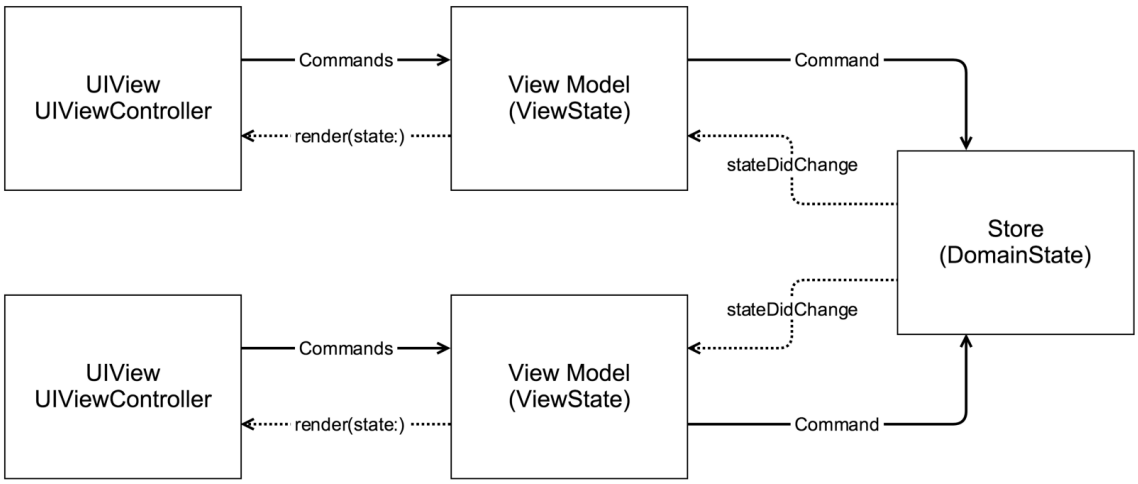

Figura 44: Arquitectura Job & Talent para aplicaciones iOS.

Por otra parte, a diferencia de Redux y debido a las limitaciones de Swift (lenguaje de programación usado en aplicaciones para iOS) dividen el Vista-Modelo en varios contenedores que almacenan una propiedad del estado. Las modificaciones al estado son mutables pero quedan reunidas en estos contenedores y únicamente se llevan a cabo a través de transacciones, evitando así que otras mutaciones se lleven a cabo antes de que la actual haya acabado.

Para modelar estados se puede hacer uso de los enumerados de Swift, de forma que sean exclusivos. Esto sirve, por ejemplo, para representar el estado de un proceso en segundo plano.

```
enum ProcessState {
  case idle
  case loading
  case loaded([Result])
  case error
}
```

Figura 45: Ejemplo de *ViewState* de la arquitectura Job & Talent.

En el *Store* encargado de este estado incluimos el proceso sobre el que se quiere observar el progreso para representar a través del estado. Se hace uso de los

mecanismos de modificación mutables para alterar de forma atómica y evitar condiciones de carrera que den lugar a estados inconsistentes.

Por ejemplo, en el *Store* mostrado a continuación, se expone un método para iniciar una tarea en segundo plano. Al ejecutarlo, se modifica el estado de forma atómica y pasa a indicar que se esta realizando un trabajo. Seguidamente, se ejecuta la tarea con un *callback* que será ejecutado cuando termine, volviendo a escribir el estado con el resultado de la tarea.

```
class ProcessStore : Store<ProcessState> {
  private let service: BackgroundService

  init(service: BackgroundService) {
    self.service = service
    super.init(initialState: .idle)
  }

  func startProcess() {
    super.writeState { state = .loading }
    service.execute { result in
      super.writeState { state = .loaded(result) }
    }
  }
}
```

Figura 46: Ejemplo de *Store* de la arquitectura Job & Talent.

Los *ViewModels* por tanto solo deben suscribirse a estos *Stores* de forma que son notificados de cualquier cambio en el estado y lo propagan hacia sus *ViewStates* que se encargan de transformar el resultado de la lógica de dominio en un estado representable por la interfaz.

b. Caso de implementación propia

Como se ha podido observar durante toda esta exposición, los principios sobre los que Redux se construyen son muy sencillos. Por ello, he realizado una prueba de concepto portando los patrones a Kotlin, el nuevo lenguaje oficial de Android.

Debido a las libertades que tiene Javascript por ser un lenguaje no tipado no es posible imitar el funcionamiento al completo. Por ello, otras librerías que intentan simular Redux suelen utilizar estructuras más parecidas a Flux donde existen varios contenedores en vez uno único. En mi librería, a la que he bautizado "Duck", he intentado portar lo más fielmente posible las bases de la arquitectura sacrificando a cambio comodidad para el usuario final.

En primer lugar, tenemos la clase *Store* que recibe una colección de *Reducers* por constructor. También guardamos una lista de suscriptores. Esta clase ofrece método para suscribirse, cancelar la suscripción y lanzar acciones. De forma interna, cuando se lanza un objeto acción primero se pasa a los *Middlewares* que están implementados con el patrón Cadena de Responsabilidad. Si la acción llega al final de esta cadena se pasa entonces a los *Reducers.*

Los *Reducers* se organizan en una estructura de tipo *Map<String, Any>* donde el *String* es un identificador que se les otorga para habilitar la suscripción parcial del estado y *Any* se refiere al tipo del estado que albergan, que deberá ser especificado por el usuario en algún punto de la aplicación.

```
interface Reducer<S, in A: Action> : Duck {
    val identifier: String
    val initialState: S

    fun reduce(state: S, action: A): S
}
```

Figura 47: Contrato a cumplir por los *Reducers* en la librería Duck.

Todos los *Reducers* tienen una función *reduce* que acepta el estado actual y una acción para devolver un nuevo estado. En el *Store* se pasa la acción lanzada a todos los *Reducers* para que se ejecuten y se construye un nuevo mapa de forma inmutable con los resultados para posteriormente notificar a los suscriptores.

```
internal fun reduceAction(action: Action) {
        state = reducers
                .map { it as Reducer<Any, Action> }
                .map { Pair(
                   it.identifier,
                   it.reduce(state[it.identifier]!!, action)
                )}
                .toMap()

        subscribers
                .forEach { it.onStateChanged() }
}
```

Figura 48: Mecanismo del *Store* para propagar acciones a todos los *Reducers*.

Por último, todos los Suscriptores se adhieren a la interfaz correspondiente que les da acceso a una referencia al *Store* de solo lectura y expone un método para que sean notificados de cualquier cambio.

```
interface Subscriptor {
    val store: Store
    fun onStateChanged()
}
```

Figura 49: Contrato a cumplir por los suscriptores del *Store*.

Para ilustrar como se puede hacer uso de la librería he creado una aplicación para jugar al Ahorcado a través del cliente de mensajería Telegram.

El estado del juego se puede representar mediante tres *Reducers*: La palabra original a adivinar, los intentos hasta el momento actual y el estado de la palabra que los jugadores ven.

El primero únicamente atiende a las acciones de tipo *NewWord* sustituyendo la palabra actual por una nueva que porta el objeto acción.

```
class OriginalWordReducer : Reducer<String, HangmanActions> {
    override val identifier: String = "originalWord"
    override val initialState: String = ""

    override fun reduce(state: String, action: HangmanActions): String =
        when (action) {
            is HangmanActions.CurrentWordActions.NewWord ->
                action.payload
            else -> state
        }

}
```

Figura 50: *Reducer* encargardo de manejar la palabra original en el ejemplo del juego del ahorcado.

El segundo maneja las acciones correspondientes a un intento fallido de mostrar una letra de la adivinanza, añadiendo dicha letra a una lista de fallos, y aquellas que cambian la palabra original por una nueva, limpiando la lista de fallos.

```
class GuessesReducer : Reducer<List<String>, HangmanActions> {
    override val identifier: String = "guesses"
    override val initialState: List<String> = emptyList()

    override fun reduce(state: List<String>,
                        action: HangmanActions): List<String> =
            when (action) {
                is GuessesActions.IncorrectGuess ->
                    state + action.payload
                is HangmanActions.CurrentWordActions.NewWord ->
                    emptyList()
                else -> state
            }
}
```

Figura 51: *Reducer* encargado del estado de adivinanzas fallidas del juego del ahorcado.

Por último, el *Reducer* encargado de variar el estado actual de la palabra por adivinar maneja las acciones para cambiar la palabra actual por una nueva y los intentos de adivinación correctos. En el primer caso se sustituye cada carácter del *String* por guiones bajos para ocultar la palabra original. Por cada intento correcto se sustituyen las posiciones de la palabra por el carácter correcto.

```
class CurrentWordReducer : Reducer<String, HangmanActions> {
    override val identifier: String = "currentWord"
    override val initialState: String = ""

    var originalWord: String = ""

    override fun reduce(state: String,
                        action: HangmanActions): String =
            when (action) {
                is CurrentWordActions.NewWord -> {
                    originalWord = action.payload.toLowerCase()
                    "_".repeat(action.payload.length)
                }
                is HangmanActions.GuessesActions.Guess -> {
                    val wordPositions =
                            findCharacterPositionsInString(
                                    action.payload.first(),
                                    originalWord
                            )

                    replaceCharacters(
                            originalWord,
                            state,
                            action.payload.first(),
                            0,
                            wordPositions
                    )
                }
                else -> state
            }
}
```

Figura 52: *Reducer* encargado del estado de la palabra a adivinar en el juego del ahorcado.

Los *Reducers* aceptan acciones del tipo *HangmanActions*. Esto se debe a que la librería intenta aprovechar un sistema de tipos en vez de *Strings* para indicar el tipo de las acciones. Para ello, se crea una jerarquía cerrada de herencia donde se recogen todas las clases disponibles para ser lanzadas.

```
sealed class HangmanActions : Action {

    sealed class CurrentWordActions : HangmanActions(),
PayloadAction<String> {
        class NewWord(override val payload: String) : HangmanActions(),
PayloadAction<String>
    }

    sealed class GuessesActions : PayloadAction<String> {
        class IncorrectGuess(override val payload: String) :
HangmanActions(), PayloadAction<String>
        class Guess(override val payload: String) : HangmanActions(),
PayloadAction<String>
    }

}
```

Figura 53: Jerarquía de acciones posibles para el juego del ahorcado.

Aquí se puede observar como quizás este es uno de los puntos donde se pierde comodidad respecto a Redux original. No obstante, usar un sistema de tipos es más escalable que utilizar datos primitivos y nos proporciona mejores herramientas.

Por último, el *Store* tiene un único suscriptor que se encarga de lanzar acciones con los inputs que llegan desde el chat y reacciona a los cambios en el estado publicando otros mensajes. Se guardan referencias locales al estado que solo son actualizadas cuando se detectan diferencias ya que trabajar directamente con los datos del estado puede ser incomodo debido a que sea ha de especificar el tipo de ese fragmento cada vez que se accede a él. Como posible mejora se pueden crear funciones Selectoras para reducir el tamaño de la función *onStateChanged()* y ofrecer una forma más cómoda de leer el estado.

Por ejemplo, cada vez que se reciba desde el chat un input para intentar adivinar una letra de la palabra, se llamará al método *guess()* con la letra en cuestión. Si la letra se encuentra en la lista de fallos se contestará con un mensaje en el chat indicando que ya ha sido usada. Si no ha sido usada y hay menos de 5 fallos se lanzará un evento *Guess* que provocará una llamada al *onStateChanged()* que actualizará las referencias locales al estado que hayan cambiado y notificará al chat de la nueva situación (*handleLostGame(), handleLostLife()* y *status()).*

```
class HangmanController(override val store: Store) : Subscriptor {

    var guesses: List<String> = emptyList()
    var currentState: String = ""
    var originalWord: String = ""

    override fun onStateChanged() {
        if (guesses != store.state["guesses"] as List<String>) {
            guesses = store.state["guesses"] as List<String>
            if (guesses.size >= 5) handleLostGame()
        }

        if (originalWord == store.state["originalWord"]
          && currentState == store.state["currentWord"] as String) {
            if (!guesses.contains(lastLetter)) handleLostLife()
        } else {
            originalWord = store.state["originalWord"] as String
            currentState = store.state["currentWord"] as String
            status()
        }
    }

    fun guess(letter: String) {
        lastLetter = letter
        when {
            guesses.contains(letter) -> bot.sendMessage(chatId,
"${letter.toUpperCase()} ya ha sido usada")
            guesses.size < 5 -> store.dispatch(Guess(letter))
            else -> handleLostGame()
        }
    }

    fun newGame(word: String) {
        guesses = emptyList()
        lastLetter = ""
        store.dispatch(NewWord(word))
    }
}
```

Figura 54: Suscriptor del *Store* del juego del ahorcado.

7. Conclusiones

La arquitectura del software es uno de los puntos más importantes a la hora de iniciar un proyecto porque tiene un impacto crítico en el mantenimiento del producto y su evolución. Debemos ser conscientes de que no existe una solución perfecta y por ello debemos dedicar el tiempo necesario a investigar qué conviene en nuestro proyecto.

En el campo de arquitecturas para aplicaciones de usuario o *frontend* la arquitectura más conocida es Modelo Vista Controlador. Este patrón tiene limitaciones que son solucionadas en parte por derivaciones que introducen otros artefactos como Presentadores o Vistas-Modelo. En cualquier caso, todas estas arquitecturas siguen padeciendo problemas por su flujo de datos bidireccional. Como ingenieros del software es nuestra responsabilidad conocer la mayor cantidad de herramientas para usar la que mejor convenga en cada caso.

Redux y Flux aparecen como respuesta desde equipos con necesidades específicas a estas arquitecturas Modelo Vista Controlador. Proponen flujos de datos unidireccionales apoyados sobre otros patrones que favorecen la reactividad. Toman como base muchos principios del paradigma de programación funcional como funciones puras o inmutabilidad. Pero, como cualquier otra arquitectura, tampoco es perfecta porque requiere que el proyecto se someta a fuertes restricciones.

La implementación de Redux en una aplicación es muy sencilla ya que la arquitectura consta de pocos artefactos y está lista para ser utilizada en productos en producción como se ha observado a través de los ejemplos expuestos.

Por último, el impacto que han tenido estos patrones en la industria del software ha sido grande y muy positivo. Además, la llegada de nuevos lenguajes modernos propicia estos cambios gracias a las herramientas que proporcionan. Empresas como Microsoft, Facebook o Netflix ya están usando Redux en algunos proyectos y muchas *startups* nacen directamente aplicando esta arquitectura desde el principio.

8. Relación del trabajo con los estudios cursados

Durante la carrera hemos cursado varias asignaturas que hablan sobre la estructura de proyectos y su arquitectura. Pero, en concreto, las dos asignaturas que mas relación tienen con este trabajo son Lenguajes, Tecnologías y Paradigmas de Programación (LTP), y Diseño de Software (DDS).

En LTP pudimos ver como existen aproximaciones más allá del paradigma imperativo y la orientación a objetos. Se dio una introducción al paradigma declarativo funcional con Haskell y desde entonces mi interés por esta rama no ha hecho mas que crecer. Redux está fuertemente influenciado por este paradigma y gracias a mis conocimientos previos me resultó muy fácil entender sus principios.

Diseño de Software nos presentó algunos Patrones de Diseño de Software y la importancia de disponer de herramientas para facilitar la comunicación entre desarrolladores. Entre estos patrones se profundizó en algunos arquitectónicos como MVC. Gracias a esta asignatura he podido identificar y entender como Redux esta implementado a nivel interno y he podido realizar mi propia implementación en otro lenguaje.

Considero que este trabajo esta alineado con los estudios que he cursado. La rama de Ingeniería del Software nos prepara para diseñar y construir soluciones apropiadas. Redux es una herramienta más para añadir a nuestro repertorio y es totalmente apto para aplicaciones en producción.

9. Trabajos futuros

Debido a que las aplicaciones que se presentan son productos de una *startup* en fase de validación tanto del modelo de negocio como del producto, la velocidad a la que debemos realizar cambios sobre ellos no nos ha permitido realizar pruebas automáticas de ningún tipo. Me hubiera gustado poder hablar sobre *testing* en Redux ya que es una de sus principales ventajas debido a su facilidad, pero no tener ejemplos reales sobre los que apoyarme le resta veracidad y tampoco tengo una opinión real al no haber realizado pruebas por mí mismo.

Por otra parte, quiero seguir mejorando mi implementación de Redux en Kotlin ya que es el lenguaje con el que me encuentro trabajando ahora mismo. Creo que algunos de los conceptos que usa la arquitectura son muy interesantes y no hace falta implementarla por completo para aprovechar sus ventajas. Esto junto a la potencia de los lenguajes modernos tipados como Kotlin nos permite construir mejores soluciones. El código de esta librería está abierto en mi cuenta de Github y espero conseguir colaboración para continuar con su desarrollo.

Por último, Redux es una arquitectura muy reciente y todavía puede evolucionar. De hecho, en su comunidad aún hay mucha controversia sobre las mejores formas de usarla y como solucionar algunos de los problemas que plantea. Con esto quiero decir que todavía me queda mucho camino por recorrer y mejorar la forma en la que hago uso de esta herramienta.

10. Referencias

Cejas, Fernando. 2014. Architecting Android... The clean way? [En línea] 03 de 09 de 2014. [Citado el: 20 de Abril de 2018.] https://fernandocejas.com/2014/09/03/architecting-android-the-clean-way/.

Czaplicki, Evan. 2012. Elm. [En línea] 2012. [Citado el: 20 de Abril de 2018.] http://elm-lang.org/.

Daza, Yeison. 2016. Currying en JavaScript: Funciones con superpoderes. *Medium*. [En línea] 20 de Junio de 2016. [Citado el: 21 de Abril de 2018.] https://medium.com/entendiendo-javascript/currying-en-javascript-funciones-con-superpoderes-1c8760c728a.

Dorfmann, Hannes. 2016. Model View Intent on Android. *Hannes Dorfmann*. [En línea] 4 de Marzo de 2016. [Citado el: 30 de Abril de 2018.] http://hannesdorfmann.com/android/model-view-intent.

Facebook. 2014. Flux. [En línea] 2014. [Citado el: 20 de Abril de 2018.] https://facebook.github.io/flux/.

—. React. [En línea] [Citado el: 20 de Abril de 2018.] https://reactjs.org/.

—. React Native. [En línea] [Citado el: 21 de Abril de 2018.] https://facebook.github.io/react-native/.

—. Virtual DOM and Internals. *React*. [En línea] [Citado el: 21 de Abril de 2018.] https://reactjs.org/docs/faq-internals.html.

Fowler, Martin. 2011. Command Query Resposibility Segregation. *Martin Fowler*. [En línea] 11 de Julio de 2011. [Citado el: 20 de Abril de 2018.] https://martinfowler.com/bliki/CQRS.html.

—. 2005. Event Sourcing. *Martin Fowler*. [En línea] 12 de Diciembre de 2005. [Citado el: 20 de Abril de 2018.] https://martinfowler.com/eaaDev/EventSourcing.html.

Oloruntoba, Samuel. 2015. S.O.L.I.D: The first 5 principles of object oriented design. [En línea] 18 de Marzo de 2015. [Citado el: 17 de Junio de 2018.] https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design.

Recuenco, Luis. 2017. iOS Architecture: A State Container based approach. *Medium*. [En línea] 16 de Octubre de 2017. [Citado el: 30 de Abril de 2018.] https://jobandtalent.engineering/ios-architecture-an-state-container-based-approach-4f1a9b00b82e.

Redux. Redux. [En línea] [Citado el: 20 de Abril de 2018.] https://redux.js.org.

—. Thunk. [En línea] [Citado el: 21 de Abril de 2018.] https://github.com/reduxjs/redux-thunk.

Reenskaug, Trygve. Notes and Historical documents. [En línea] [Citado el: 20 de Abril de 2018.] http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

rt2zz. Redux Persist. [En línea] [Citado el: 22 de Abril de 2018.] https://github.com/rt2zz/redux-persist.

Valiente, César. 2018. Unidirectional data flow on Android using Kotlin: The blog post. *Medium*. [En línea] 14 de Marzo de 2018. [Citado el: 30 de Abril de 2018.] https://proandroiddev.com/unidirectional-data-flow-on-android-the-blog-post-part-1-cadcf88c72f5.