ESTUDIO SOBRE REACT Y SU ECOSISTEMA

Interfaces de Usuario, 2022-2023

Ingeniería en Informática

Intensificación en Ingeniería del Software

Universidad de Murcia

Grupo 11:

Alberto Gálvez Gálvez – alberto.galvezg@um.es

Mario Rodríguez Béjar – mario.rodriguezb1@um.es

Tabla de contenido

1 Resumen

JavaScript ha dominado el mundo del desarrollo web durante los últimos años, y como a todo aquello que se encuentra en auge o de moda, le han surgido distintas alternativas y opiniones en forma de librerías y frameworks de Javascript, gestores de paquetes, librerías de componentes y demás. Por ello, se ha realizado un estudio sobre React, la librería más de moda, y su ecosistema, viendo sus características más importantes, ventajas y desventajas respecto a sus principales competidores, el uso de la librería de componentes Material UI, el desarrollo de un caso práctico y distintas mejoras que se pueden aplicar a los proyectos desarrollados en esta tecnología.

Como conclusiones, podemos determinar que React es una gran herramienta de desarrollo de interfaces webs, ya que tiene un gran rendimiento, es fácil de aprender y dispone de características que hacen que que el desarrollo con esta herramienta sea veloz y simple. Además, su uso acompañado de la librería de componentes Material UI amplifica las bondades mencionadas. Sin embargo, también se ha concluido que React en proyectos de mediana o gran complejidad pueden sufrir de distintos problemas, como poca mantenibilidad del código y difícil testeabilidad (debido entre otros, al no tipado de Javascript, que React no fuerce adoptar un arquitectura) . Frente a esto último, presentamos distintas alternativas (arquitectura hexagonal, uso de Typescript, uso de RxJs) que puede mejorar sustancialmente los proyectos desarrollados con React.

2 Introducción

Es innegable que la sociedad actual se encuentra inmersa en una imparable transformación digital, siendo el software uno de los pilares básicos en los que se sustenta dicho proceso. En este proceso, como en todo aquel que está en auge, surge una enorme amalgama de opiniones, alternativas, conceptos y modas de toda índole, que a veces hacen que se pierda la perspectiva del objetivo de la ingeniería de software, que es el de producir productos de alta calidad con el menor tiempo y coste posible.

Esta desviación se puede observar en el surgimiento de una infinidad de herramientas de desarrollo, librerías de Javascript, gestores de paquetes, librerías de componentes y demás, que en esencia hacen lo mismo, pero que han producido que se desvíe la atención del desarrollo web hacia las tecnologías y su idiosincrasia, en lugar de hacia cuestiones de mayor interés como son, entre otras, la ingeniería de requisitos, la usabilidad, la accesibilidad y la experiencia de usuario.

Así pues, el objetivo de este trabajo es el de centrarnos en una tecnología web puntera y en cierta medida estandarizada como React (librería del lenguaje de programación Javascript) a la que acompañaremos con Material UI (librería de componentes para React) para el desarrollo de interfaces de usuario. Para ello, ahondaremos en los conceptos básicos de React, su ecosistema y Material UI, así como desarrollaremos un prototipo con las tecnologías mencionadas. Además, también se ilustrarán tecnologías,

arquitecturas y técnicas que pueden complementar y mejorar considerablemente los proyectos desarrollados con React.

Las distintas partes en las que se divide nuestro estudio son:

- Características y conceptos clave de React: se tratan las características más destacables de React, como son los gestores de paquetes como NPM, el lenguaje JSX, los conceptos de Virtual DOM, state, renderizado, componente y hook.
- Material UI: se explica esta famosa librería de componentes.
- Caso práctico: se ilustra un prototipo web desarrollado para este trabajo, así vinculando muchos de los conceptos desarrollados en los apartados anteriores.
- Comparativa con otros lenguajes para el prototipado: se compara React frente a sus principales competidores para el prototipado, Angular y Vue, así justificando nuestra decisión por React.
- Problemas y mejoras para proyectos en React: se muestran algunos de los principales problemas que sufren los proyectos desarrollados con React y se proponen posibles soluciones.
- Conclusiones y vías futuras: en el que se recapitula lo desarrollado a través de los demás apartados y se proponen vías futuras.

3 Características y conceptos clave de React

3.1 NPM y Yarn

Para la programación en React es algo esencial el "Packet Manager" (1), una herramienta que facilita a los desarrolladores la compartición y reúso del código gracias a la posibilidad de utilizar componentes desarrollados por la comunidad, encontrando soluciones realmente útiles para problemas muy comunes como la división de secciones de páginas web con *Routers*, envío y tratamiento de peticiones HTTP con *Axios*. Para realizar esta tarea encontramos principalmente dos grandes competidores:

- NPM
- YARN

Ambos ofrecen la funcionalidad mencionada, sin embargo, cada uno tiene unas prestaciones concretas. Por su parte, Yarn, apareció como un sustituto a la única opción antiguamente existente, ofreciendo varias mejoras deseadas por usuario descontentos con NPM, sin embargo, ambos han escalado juntos como resultado de la competitividad entre ellos.

En la práctica haremos uso de NPM, ya que es el más ampliamente extendido y, por tanto, más amigable para nuevos usuarios debido a la gran cantidad de documentación y ejemplos existentes.

Finalmente, se va a hacer una guía básica de comandos en npm necesarios para cualquier programador de React:

- **npx create-react-app my-app**, comando usado para la creación de un proyecto básico de React. NPX es una herramienta que acompaña a NPM y que se encarga facilitar la ejecución de paquetes de npm.

- **npm install router**, comando usado para la instalación de un paquete a través de npm, en este caso de **Router**. Este comando instala el paquete únicamente en el proyecto actual.
- **npm -g install axios**, comando usado para instalar de forma global un paquete en el sistema en vez de tener que hacerlo de forma local para cada uno de los distintos proyectos que tengas.
- **npm uninstall axios**, comando usado para desinstalar un paquete existente.
- **npm start,** comando que ejecutaría nuestra aplicación en modo de desarrollo. Así, podemos navegar a *http:localhost:3000* en cualquier navegador para obtener una vista previa de nuestra aplicación en vivo. La página se recargará automáticamente cada vez que detecte cualquier cambio de código en los archivos fuente. Las advertencias y errores que surjan también se pueden ver en la consola.

3.2 State

El state o estado de React es un objeto integrado de React que contiene información de un componente. El state puede cambiar con el tiempo; cada vez que cambia, el componente se vuelve a renderizar. El cambio de estado puede ocurrir como respuesta a una acción del usuario o eventos generados por el sistema y estos cambios determinan el comportamiento del componente y cómo se representa. En los componentes y hooks comentaremos ahondaremos más sobre este concepto.

3.3 Componentes

Un **componente** es uno de los bloques o piezas básicas de React. Así, cualquier aplicación en React y en muchas otras tecnologías web, estará formada por estos. Por ejemplo, una barra de búsqueda, una publicación, un menú o un listado pueden verse como componentes individuales y, fusionando todos estos, podemos obtener un componente padre, que sería la interfaz del usuario final para una página en concreto de una web. Los componentes en React básicamente devuelven un fragmento de código JSX que indica qué es lo que se debe representar en la pantalla.

Tipos de componentes. En React ha distintos tipos de componentes, cada uno con distinttas características. Los distintos tipos son:

Componentes sin estado. Se definen como funciones en Javascript que no tienen ni trabajan con estado. Los únicos datos con los que trabajan este tipo de componentes es con las props recibidas (de ser pasadas) (2).

Ejemplo de componente dummy o sin estado en React

```
const componente = list.length > 1 ?
      [
          <li className="mi-clase">{list[0]}</li>,
          <li className="mi-clase">{list[1]}</li>
```

```
      ]
    :
    null;
```

Componentes de clase. Se tratan de clases que extienden de React.Component, tienen un estado que definen y actualizan. Además, a cada cambio tanto en props como en su estado llaman al método render (3).

Ejemplo de componente de clase con estado en React

```
import { Component } from 'react';
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { currState: true }
  }
  render() {
    <div>
      <p>Hello, World!</p>
<button onClick={() => this.setState({ currState:
this.state.currState })}>
          Click me
</button>
    </div>
  }
}
```

Componentes funcionales. Existentes a partir de la versión 16.4, son como los componentes de clase pero definidos como una función en lugar de como una clase (4).

Ejemplo de componente funcional escrito como función con estado en React

```
function MyComponent(props) {
  const [currState, setCurrState] = useState(true);
  return (
    <div>
      <button onClick={() => setCurrState(false)}>
                Click me
      </button>
    </div>
  );
}
```

Ejemplo de componente funcional escrito como constante con estado en React

```
const MyComponent = (props) => {
```

```
  return (
    <div>
      <p>Hello, World</p>
      <button onClick={() => setCurrState(false)}>
                Click me
      </button>

    </div>
  );
}
```

Ventajas de los componentes funcionales. Como se puede observar en las figuras vistas arriba, los componentes funcionales son más cortos y simples, lo que hace que sean más fáciles de desarrollar, entender y testear. Además, los componentes de clase pueden resultar bastante confusos, esto debido a los usos de *this*, los cuales en Javascript son bastante confusos, lo cual en los componentes funcionales no ocurre. Además, el rendimiento de los componentes funcionales es mejor, debido a optimizaciones en cuanto a comprobaciones y reservas innecesarias de memoria (5).

Componentes de orden superior. HOC por las siglas en inglés de higher-order components, se tratan de es una técnica avanzada en React para la reutilización de la lógica de componentes. Los HOCs no son parte de la API de React, son un patrón que surge de la naturaleza composicional de React. El funcionamiento de esta técnica es básicamente la de que un componente envolvuelva otros compontenes a través del uso de parámetros. Cabe destacar que este HOC no modifica el componente de entrada, ni usa herencia para copiar su comportamiento (6) .

Ejemplo de High Orden Component en React

```
export default function Hoc(HocComponent){
    return class extends Component{
        render(){
            return (
                <div>
                    <HocComponent></HocComponent>
                </div>
            );
        }
    }
}
```

3.4 JSX

En **React** encontramos una sintaxis única para la especificación de elementos de la interfaz de usuario, **JSX** (7). Se trata de un lenguaje de etiquetas que ofrece una fusión

entre **HTML** y la potencia de **Javascript**, permitiendo incluir sentencias en este lenguaje.

Este lenguaje consigue aunar la lógica con los elementos de la interfaz de usuario para la creación de **componentes** (explicados en la sección 2.3), eliminando la complejidad añadida de tener que tratar la vista y la lógica desde ficheros separados, es decir HTML y CSS por un lado, y, por otro, JavaScript. Pese a todo lo comentado, **JSX** no es algo que sea obligatorio usar con el uso de **React**, pudiendo separar la lógica y vista como se comento anteriormente.

Para poder entender mejor este concepto observemos el ejemplo que encontramos a continuación, donde podemos ver el uso de condicionales y variables con valor dinámico en armonía con código **HTML**.

Ejemplo de un elemento de HTML cuya clase depende de una variable de javascript llamada "click"

```
<ul className={ click ? "nav-menu active":"nav-menu" }>
```

También, podemos ver en el siguiente ejemplo como se puede **restringir** la visualización de un elemento de la vista según el valor de una variable.

Ejemplo de Código en el que se protege un link de la barra de navegación dependiendo del tipo de usuario actual

```
{usuario == "Medico" &&
<li className="nav-item">
<NavLink onClick={handleClick} className="nav-link"
to="/myInfo" >Ver Citas</NavLink>
</li>
}
```

Otro ejemplo sería el cargar **componentes** en la vista basándose en los datos obtenidos en una variable de nuestro programa, como podemos ver en el ejemplo inferior.

Ejemplo de Código en el que se carga un componente desarrollado por nosotros llamado "SideNav" y se hace uso de una colección de datos para instanciar componentes

```
<div className="CitasView-Container">
     <SideNav />
     <div className="CitasPanel">
          {data.map((cita) => <CitaPanel
          key={String(cita.citaId)} cita={cita} />)}
     </div>
</div>
```

3.5 Virtual DOM

En este apartado vamos a hablar del **Virtual DOM** (8), una característica propia de **React**, que pretende incrementar el rendimiento del **DOM real**.

Primero, vamos a explicar a que nos referimos con DOM real. DOM significa **Document Object Model** y este maneja los objetos contenidos en un documento **HTML**, tratándolos como un **árbol** y permitiéndonos crear, agregar o eliminar elementos de él a través de las distintas operaciones que ofrece.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device
    <meta http-equiv="X-UA-Compatible" content=
    <title>Estructura de árbol del DOM</title>
  </head>
  <body>
    <h1>Estructura de árbol del DOM</h1>
    <h2>Aprende sobre el DOM</h2>
  </body>
</html>
```

Fig. 1. Código HTML

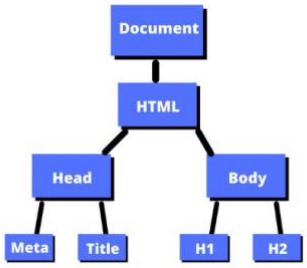

Fig. 2. Árbol construido a partir del documento HTML

Para poder entenderlo mejor podemos observar la *figura 2*, en la que observamos como el DOM **interpreta** el código HTML de la *figura 1*.

Una vez visto esto, el Virtual DOM es una representación del DOM que busca **mejorar su rendimiento**. Cada vez que haya cambios en nuestra aplicación el virtual DOM es actualizado en vez del DOM real.

La mejora de rendimiento mencionada se consigue a la hora de la **actualización del DOM** real ante cambios en el estado de algún nodo del árbol, ya que en vez de realizar los cambios sobre este se realiza el siguiente proceso:

1. Se crea un nuevo Virtual DOM con los nuevos estados.
2. Se compara ese Virtual DOM con el anterior.

3. Se calcula el mejor método para hacer los cambios sobre el DOM real.

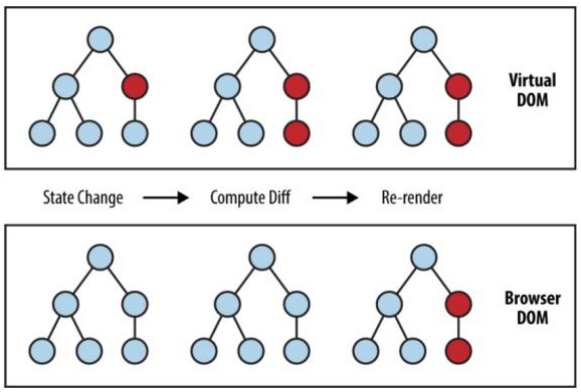

Fig. 3. DOM Virtual

3.6 Renderizado

Los elementos son los bloques más pequeños de las aplicaciones de React y representan la interfaz web. A diferencia de los elementos del DOM de los navegadores, los elementos de React son objetos planos, y su creación de bajo costo. Uno podría confundir los elementos con el muy conocido concepto de "componente", del que hablamos en el apartado Componentes. Sin embargo, estos elementos son los que constituyen los componentes (9).

Los elementos de React son inmutables, es decir, una vez se crea crea un elemento, no se pueden cambiar sus hijos o atributos sin producir un renderizado. React necesita tomar el control total del DOM). Ahora, este DOM virtual no se compone de elementos HTML como lo hace el DOM real, sino de elementos React.

Lo que estamos viendo es un objeto JavaScript normal y simple y no un nodo DOM. Entonces, ahora, entendemos que el llamado DOM virtual no se parece en nada al DOM real, sino que es un árbol de objetos React (JavaScript) que representan la interfaz de usuario en ese momento.

Si un componente padre en React cambia (digamos, porque su estado o accesorios cambiaron), React recorre todo el árbol por este elemento padre y vuelve a renderizar todos los componentes. Si su aplicación tiene muchos componentes anidados y muchas interacciones, sin saberlo, está sufriendo un gran impacto en el rendimiento cada vez que cambia el componente principal (asumiendo que es solo el componente principal que desea cambiar) (10).

Es cierto que el renderizado no hará que React cambie el DOM real porque, durante la reconciliación, detectará que nada ha cambiado para estos componentes. Pero, todavía es tiempo de CPU y memoria desperdiciada, y se sorprenderá de lo rápido que se acumula (9).

3.7 Hooks

Explicación. Un Hook es una función de javascript que permite crear/acceder al estado y a los ciclos de vida de React y que, para asegurar la estabilidad de la aplicación, debe de utilizarse siguiendo dos reglas básicas (11):

4. Debe de ser llamado en el nivel superior de la aplicación. Un hook nunca debe de llamarse dentro de bucles, condicionales o funciones anidadas, ya que el orden de llamada de los hooks debe de ser siempre el mismo para asegurar que el resultado sea predecible durante la renderización. Este uso únicamente en el nivel superior es lo que asegura que el estado interno de React se preserve correctamente entre diferentes llamadas al mismo hook.
5. Debe de llamarse en funciones o en otros hooks personalizados de React. Un hook nunca debe de ser llamado fuera de una función de React o de otro hook personalizado, de forma que la lógica de estado del componente sea cláramente visible desde el resto del código para el scope establecido por React.

Finalidad. Como la finalidad última de los hooks es simplificar la lógica actual, React proporciona únicamente un set reducido, con la flexibilidad para responder ante diversas situaciones del ciclo de vida de una aplicación y la posibilidad de construir también los nuestros propios (12).

Hooks para el ciclo de vida de un componente. React proporciona tres hooks básicos que responden a las necesidades habituales para implementar un ciclo de vida en un componente de clase (12):

useState. Este hook nos devuelve un valor con un estado mantenido entre renderizados (color) y una función que es necesaria para actualizarlo (setColor).

Ejemplo de uso del hook useState para la gestión del estado de un componente en React

```
function FavoriteColor() {
  const [color, setColor] = useState("red");

  return (
    <>
      <h1>My favorite color is {color}!</h1>
      <button
        type="button"
        onClick={() => setColor("blue")}
      >Blue</button>
    </>
  )
}
```

useEffect. Este hook nos permite agregar efectos secundarios a un componente funcional dado, es decir, nos permite realizar modificaciones en nuestra lógica una vez se haya realizado el renderizado, de la misma forma que los métodos de ciclo de vida componentDidMount, componentDidUpdate y componentWillUnmount en los componentes de clase.

Ejemplo de uso del hook useEffect para le gestión del ciclo de vida de un componente en React

```
function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  });

  return <h1>I've rendered {count} times!</h1>;
}
```

useContext. El contexto en React es una forma de pasar datos entre diferentes componentes sin necesidad de realizar una cascada manual de props (props drilling). Este hook es útil, por ejemplo, cuando queremos crear temas o localizaciones, que deben de ser globales para todo el árbol de componentes y puede ser engorroso tener que propagarlos para todos y cada uno de los componentes.

Ejemplo de uso del hook useContext para la gestión del contexto de un componente en React

```
const UserContext = createContext();

function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <UserContext.Provider value={user}>
      <Component2 />
    </UserContext.Provider>
  );
}

function Component2() {
  const user = useContext(UserContext);

  return (
    <>
      <h2>{`${user}`}</h2>
```

```
    </>
  );
}
```

Hooks personalizados. Cuando queremos compartir lógica entre funciones de JavaScript, podemos extraemos esta lógica en otra; y como los componentes y Hooks son funciones de Javascript, estos pueden llamar a otros hooks y funciones, así como pasar parámetros entre ellos.

Ejemplo de creación de un hook personalizado

```
import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {  const [isOnline,
setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID,
handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID,
handleStatusChange);
    };
  });

  return isOnline;
}
```

Ventajas de los hooks (12)**.**

- Reduce el número de conceptos necesarios en el desarrollo de aplicaciones React, de forma que no se nos hace necesario cambiar continuamente entre funciones, clases, HOCs o elementos para realizar tareas semejantes; Los hooks nos ofrecen homogeneidad en el ecosistema.
- Simplifican el ciclo de vida de React se ha simplificado en gran manera con el uso de los hooks , de modo que todos los métodos de ciclo de vida de los componentes de clase se resumen en un único hook `useEffect` que actúa como los tres.
- Refuerza el principio base de React de evitar las mutaciones, ya que cambiar el estado de un componente de clase es lo mismo que mutar su propiedad state, al igual que es necesario realizar un “binding” para las funciones que gestionan

eventos, aproximaciones que incrementan notablemente la complejidad y reducen la predictibilidad del componente.

- Introduce en el núcleo de React la posibilidad de trabajar con un patrón Redux sin necesidad de dependencias adicionales a través del hook useReducer. Este hook, catalogado en la categoría de "hooks adicionales", es siempre recomendable cuando el estado de la aplicación sea demasiado complejo y con una gran cantidad de anidación, ya que el reducer acepta una función del tipo (state, action) => newState devolviendo el estado actual y una función de "dispatch", lo cual nos permite emular las funcionalidades disponibles actualmente en las librerías redux y react-redux tan utilizadas para solucionar el problema de gestión del estado o de cascada de propiedades.

3.8 Routers

En este apartado vamos a hablar sobre uno herramienta muy extendida de React, **Routers** (13), que es de gran utilidad para el particionamiento de una página web en secciones distintas a través del enrutado de páginas, algo esencial en cualquier página web actual. Esta herramienta nos proporciona los siguientes componentes (14):

- **Route**, componente usado para indicar que acción debe realizarse al navegar a una dirección concreta, si la dirección concuerda con el "path" entonces se cargará el componente envuelto por Route.
- **Switch**, Componente contenedor donde deben de estar contenidos los distintos Route, que se encargará de iterará sobre ellos para encontrar la dirección adecuada, la primera ruta que coincida será la que se usará en caso de que haya dos que coincidan con la url de entrada.
- **Link**, componente que tiene el funcionamiento de un <a> de HTML, redirigiendo a la dirección indicada en el atributo "to".
- **BrowserRouter**, componente que indica a React el uso de los demás componentes en aquello que se encuentre envuelto por él.

Lo primero que se debe hacer es indicar el uso de esta herramienta, para ello debemos de indicar en la raíz de nuestro DOM de React a través del componente **BrowserRouter** que nuestra aplicación hará uso del enrutamiento.

Ejemplo de uso de uso de BrowserRouter para indicar el uso de rutas en nuestro proyecto

```
ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  document.getElementById("root")
);
```

Tras esto ya podremos realizar el enrutamiento que consideremos oportuno e indicar que componentes queremos que sean renderizados en función de la url introducida.

Además, aclarar que componentes como *Switch*, *Route*, *BrowserRouter* son componentes con lógica cuya adición no supone cambios en el aspecto visual de la interfaz del usuario, el único que la modifica es *Link*, el cual es traducido a un elemento HTML *<a>*.

Ejemplo de uso de los componentes incluidos en la librería React Router para el redireccionamiento dentro de la página

```
<div>
    <nav>
        <ul>
           <li>
              <Link to="/">Home</Link>
           </li>
           <li>
              <Link to="/about">About</Link>
           </li>
           <li>
              <Link to="/users">Users</Link>
           </li>
        </ul>
    </nav>

    <Switch>
          <Route path="/about">
            <About />
          </Route>
          <Route path="/users">
            <Users />
          </Route>
          <Route path="/">
            <Home />
          </Route>
    </Switch>
</div>
```

En el ejemplo de la parte superior en caso de recibir una url con el siguiente formato *www.miPaginaWebEjemplo.com/about* se renderizará el componente **</About>**.

4 Material UI

Material UI (15) es una librería open-source la cual provee de numerosos componentes prediseñados enfocados a la creación de interfaces de usuario. El objetivo de esta librería es acelerar la creación de interfaces de usuario en React, permitiendo a nuevos

y experimentados desarrolladores reducir el tiempo empleado en la creación de interfaces de usuario, pudiendo así enfocar sus esfuerzos y tiempo en otros aspectos.

Además, hacer uso de librerías como esta u otras como sería Boostrap para HTML y CSS, no significa carecer de marca propia o no poder personalizar los componentes empleados para que tengan un aspecto personalizado o de marca, ya que MUI ofrece ofrece multitud de atributos a especificar para indicar con detalle el funcionamiento concreto que deseemos o incluso modificar el estilado de los componentes a través de lenguaje CSS.

Para ilustrar como se haría uso de esta librería, tomemos el siguiente ejemplo. Imaginemos que queremos añadir valoraciones en una aplicación de recomendación de restaurantes, el proceso sería el siguiente:

1. Buscar **rating** en la documentación de Material UI.
2. Valorar las distintas posibilidades.
3. Decidir un diseño.
4. Observar el código asociado en el ejemplo de la documentación.
5. Adaptar el código a nuestro Proyecto a través de las múltiples posibilidades para personalizar estos componentes como por ejemplo el uso de la propiedad **sx**, la cual tienen todos sus componentes y nos permite incluir código CSS y javascript, o la posibilidad de crear “themes”, estilos que se aplican sobre los componentes modificando sus atributos por defecto.

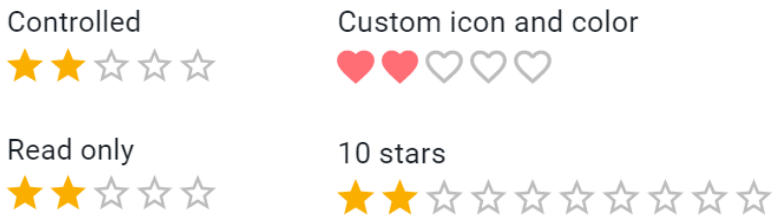

Fig. 4. Distintos estilos de Rating MUI

Ejemplo de Código en el que se emplean componentes de la libreria de MUI como “Typography” o “StyledRating”

```
<Typography component="legend">Custom icon and
color</Typography>
<StyledRating
  name="customized-color"
  defaultValue={2}
  getLabelText={(value: number) => `${value}
Heart${value !== 1 ? 's' : ''}`}
  precision={0.5}
```

```
  icon={<FavoriteIcon fontSize="inherit" />}
  emptyIcon={<FavoriteBorderIcon fontSize="inherit" />}
/>
```

Fig. 5. Ilustración 1. Estilo de MUI seleccionado

Finalmente, hay que comentar que se ha decidido emplear Material UI frente a otros competidores como puede ser Material Design debido a la gran documentación existente, el diseño de los componentes que ofrece, así como la gran madurez de esta librería, que existe prácticamente desde la aparición de React y, por tanto, tendrá un gran mantenimiento en el futuro.

5 Caso práctico

Hemos desarrollado un prototipo de la web de Formandera. Este, ha sido desarrollado con React y Material UI, arquitectura hexagonal (en cierta medida), Typescript y RxJS (para justificación de el uso de los tres últimos conceptos mencionados, ver el apartado Problemas y mejoras para proyectos en React).

5.1 Barra de navegación

Fig. 6. Barra de navegación escritorio completa

Fig. 7. Barra de navegación dimensiones de pantalla reducidas

En este caso nos encontramos con la barra de navegación, en la cual hemos conseguido hacer un diseño fiel al prototipo que diseñamos en la práctica 2 gracias a la gran flexibilidad proporcionada por la librería de Material UI.

En este podemos ver el uso de componentes de la librería Material UI como puede ser:

- Toolbar, componente que representa la barra de navegación, la cual hemos podido estilar a nuestro gusto para adaptarlo al prototipo haciendo uso de la propiedad ***styled,*** con el que le hemos podido establecer el tamaño de nuestra barra de navegación a través de media querys.

Ejemplo de código en el que se modifica el estilado de un componente a través de la propiedad "styles" de Material UI

```
const StyledToolbar = styled(Toolbar)(({ theme }) => ({
    alignItems: 'flex-start',
    paddingTop: theme.spacing(1),
    paddingBottom: theme.spacing(1),
    paddingLeft:0,
    paddingRight:0,

    // Override media queries injected by
theme.mixins.toolbar
    '@media all': {
      minHeight: 128,
    },
    '@media (max-width: 970px)': {
        minHeight: 20,
      },
  }));
```

- **Box**, componente básico que representa un contenedor y que hemos empleado para incluir las distintas secciones de la barra de navegación, así como, para indicar que cosas debemos de mostrar en caso de las especificaciones de anchura de la pantalla con el uso de los breakpoints establecidos por Material UI y la propiedad *display* dentro del parámetro ***sx,*** donde se indica el estilado css.

Ejemplo del código y componentes empleados para la creación de los links de la barra de navegación

```
<Box sx={{ flexGrow: 1, justifyContent:"center" ,
display: { xs: 'none', md: 'flex' },  alignSelf: 'flex-
end' }}>
      {pages.map((setting) => (
```

```
          <Link to={setting.url}
          style={{textDecoration:'none', color:
          '#FFFFFF'}}>
              <MenuItem key={setting.name}
              onClick={handleCloseUserMenu}>
                      <Typography textAlign="center">
                      {setting.name}
                      </Typography>
               </MenuItem>
          </Link>
        ))}
</Box>
```

- **Menu y Menuitems**, componentes que nos permite agrupar los distintos links en caso de que no tengan suficiente espacio para mostrarlos en un menú de hamburguesa.

Ejemplo del código en el que se muestra un ejemplo para la creación de un menú desplegable haciendo uso de los componentes de Material UI.

```
<Menu
  sx={{ mt: '45px' }}
  id="menu-appbar"
  anchorEl={anchorUser}
  anchorOrigin={{
            vertical: 'top',
            horizontal: 'right',}}
  keepMounted
  transformOrigin={{
            vertical: 'top',
            horizontal: 'right',}}
  open={Boolean(anchorUser)}
  onClose={handleCloseUserMenu}>

    {iniciarCuenta.map((setting) => (
        <MenuItem key={setting}
        onClick={handleCloseUserMenu}>
          <Typography textAlign="center">
              {setting}
          </Typography>
       </MenuItem>
    ))}
    {settings.map((setting) => (
        <MenuItem key={setting}
        onClick={handleCloseUserMenu}>
```

```
            <Typography textAlign="center">
               {setting}
             </Typography>
           </MenuItem>
       ))}
    </Menu>
```

- **createTheme**, una funcionalidad proporcionada por Material UI que nos permite personalizar los estilos básicos para todo aquello que este envuelto por el componente **ThemeProvider**, con el tema creado.

Ejemplo del código en le que se cambián las propiedades básicas de CSS de los componentes de Material UI a través de la creación de un “theme” personalizado

```
const theme = createTheme({
    typography: {
        button: {
          textTransform: 'none'
        }
      },
    palette:{
        action:{
            hover:"#474b4e",
        }
    },
    breakpoints: {
        values: {
          xs: 0,
          sm: 600,
          md: 970,
          lg: 1200,
          xl: 1536,
        },
      },
})
```

En este caso podemos ver que hemos cambiado el color por defecto a la hora de hacer *hover* sobre algún elemento, hemos eliminado la funcionalidad que hacia que el texto de los botones se pusiera en mayúscula, además, se han modificado los valores por defecto de los breakpoints de Material UI.

5.2 Pantalla de Inicio

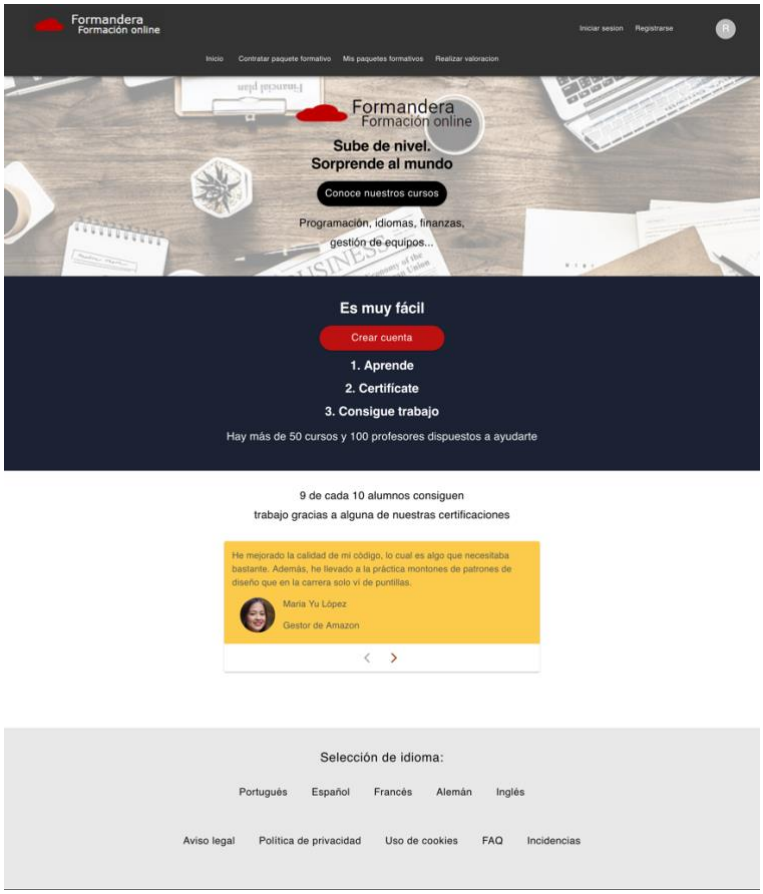

Fig. 8. Pantalla de inicio de Formandera

Otro componente que hemos realizado es la pantalla de inicio, en la cual se hace uso de componentes simples de Material UI y un simple manejo de estado para la opinión de los clientes. Esos componentes son:

- **Typography**, componente usado para el estilado de textos donde podemos indicar en qué tipo de textos queremos que sea traducido, a un párrafo, una cafecera etc.

Ejemplo de componente usado para el estilado de textos

```
<Typography variant="h1" component="h1" sx={{ fontSize:
30, fontWeight: 'bold' }}>
         Sube de nivel.
</Typography>
```

- **Container** y **Box**, componentes empleados para envolver otros componentes y estructurar la página.

Ejemplo del código en el que se muestra el uso de los componentes que ilustran la pantalla de inicio

```
 <Container component="main" maxWidth="xl" sx={{
justifyContent: 'center', textAlign: 'center',
maxHeight: "600px", display: "flex", backgroundColor:
"#1A2233" }}>
    <Box sx={{ alignItems: "center", display: "flex",
    flexDirection: "column", flexGrow: 2,
    justifyContent: "center", alignContent: "center",
    marginTop:"2rem", marginBottom:"2rem" }}>
        <Typography variant="h1" component="h1" sx={{
        fontSize: 30, fontWeight: 'bold', color:
        "white", marginBottom: "1rem", marginTop: "1rem"
        }}>
            Es muy fácil
        </Typography>

        <PersonalisedButton isBlack={false} text="Crear
        cuenta" onClick={() => { }} minHeight={true} />

        <Typography variant="h2" component="h2" sx={{
        fontSize: { xs: "20px", md: "25px" },
        fontWeight: "bold", color: "white", marginTop:
        "1rem" }}>
            1. Aprende
        </Typography>
        <Typography variant="h2" component="h2" sx={{
        fontSize: { xs: "20px", md: "25px" },
        fontWeight: "bold", color: "white", marginTop:
        "1rem" }}>
            2. Certifícate
        </Typography>
        <Typography variant="h2" component="h2" sx={{
        fontSize: { xs: "20px", md: "25px" },fontWeight:
        "bold", color: "white", marginTop: "1rem",
        marginBottom: "1rem" }}>
```

```
            3. Consigue trabajo
          </Typography>
          <Typography variant="subtitle1" component="p"
          sx={{ fontSize: { xs: "18px", md: "22px" },
          marginBottom: "1rem", color: "white" }}>
            Hay más de 50 cursos y 100 profesores
            dispuestos a ayudarte
          </Typography>
      </Box>
  </Container>
```

- **Card**, componente que permite mostrar información haciendo uso de una estructura predeterminada, contando con título, cuerpo, encabezado etc. Empleado para mostrar las opiniones de los distintos clientes de la página.

5.3 Componente formulario

Motivación. La aplicación web de Formentera se caracteriza por tener un gran número de formularios (inicio de sesión, registro, crear paquete, formativo, realizar valoración, etc). Así, si no afrontamos este hecho mediante una perspectiva ingenieril, seguramente incurramos en código duplicado, el cual puede dar lugar a poca mantenibilidad y testeabilidad, errores, inconsistencias entre el diseño de los distintos formularios, etc.

Por tanto, para evitar estos problemas mencionados, deberíamos abordar los formularios, mediante la aplicación de patrones de diseño y buenas prácticas. Entonces, nuestra solución propuesta y desarrollada es la aplicación de los patrones, factoría y estrategia.

Solución propuesta. Abajo podemos ver un uso del componente desarrollado en el componente LoginFormComponent. Como podemos ver, se indican todos los campos necesarios para la creación de el formulario que queremos:

- Tres callback (campos onChange, onSubmit y onClear) en los cuales determinan las accion a realizarse frente a cambios en cualquier campo del formulario, el hacer click en el botón submit y el hacer click en el botón limpiar formulario.
- Un objeto (campo data) en el cual se volcará la información de los campos del formulario de acuerdo a la entrada del usuario.
- Un objeto (campo ui) en el cual se indican los campos que queremos que tenga el formulario. Para cada uno de estos campos, indicando la información de este, como es el nombre que queremos que se muestre en la interfaz, el nombre que tiene el campo en el objeto sobre el que se vuelca la información, si el campo es obligatorio, el tipo de input, etc.

Ejemplo de uso de un componente FormComponent en el componente LoginFormComponent

```
<FormComponent
            onSubmit={()=>handleLogin(loginState)}
```

```
            onClear={()=>{console.log('Clear hecho')}}
            onChange={(fieldName: any, fieldValue: any)
=> {
              setLoginState({...loginState, [fieldName
as keyof typeof loginState]: fieldValue})
            }}
            data={loginState}
            ui={{
              fields: [
                {
                  name: 'username',
                  type: FormInputComponentType.Text,
                  props: {
                    name: 'username',
                    required: true,
                    label: 'Usuario'
                  }
                },
                {
                  name: 'password',
                  type:
FormInputComponentType.Password,
                  props: {
                    name: 'password',
                    required: true,
                    label: 'Contraseña'
                  }
                }
              ],
              object: loginState
            }}
          />
```

En los fragmentos de código de abajo, podemos observar como se define en nuestro modelo el formulario. Como podemos ver, al final se trata de un array de input fields, el objeto sobre el que se vuelcan los cambios y los textos que tendrán el botón de submit y el de limpiar campos.

Tipo FormDefinition, el cual define la estructura del componente formulario

```
export type FormDefinition =  {
    fields: Array<FormField>
    object: Object
    submitText: string
    clearText?: string
```

```
}

export type FormField = {
    name: string
    type: FormInputComponentType
    props: FormInputComponentData
}
```

Enumerado FormInputComponentType, el cual define los distintos tipos de inputs que admite el componente FormInputComponent

```
export const enum FormInputComponentType {
    Text,
    Password,
    Date,
    Time,
    DateTime,
}
```

En el siguiente fragmento de código, podemos encontrar el componente formulario como tal, el cual consiste en renderizar los distintos componentes mediante el uso de los patrones, factoría y estrategia, así como renderizar un botón para enviar el formulario y otro (en caso de haberse indicado un texto para él) para limpiar el formulario.

Componente FormComponent

```
export type FormComponentProps = {
  onSubmit: () => void
  onClear?: () => void
  onChange: (fieldName: any, fieldValue: any) => void
  data: Object
  ui: FormDefinition
}

export const FormComponent:
React.FC<FormComponentProps> = ({onSubmit, onClear,
onChange, data, ui}: FormComponentProps) => {

  return (
    <>
      <Box component='form' sx={{display: 'block'}}
noValidate>

        {/* Renderizamos todos los elementos del
formulario indicados en ui.fields */}
```

```
        {ui.fields.map( (e) => (
          <Box key={e.name} sx={{m: 3}}>
            {FormInputComponentFactory
              .getInputFieldView(e.type)
              .getView(onChange, e.props)
            }
          </Box>
        ))}

        {/* Renderizamos un botón para el submit con el
texto indicado en ui.submitText
            También renderizamos uno para limpiar el
form si se ha indicadado el parámetro cleartext en
ui.clearText */}
        <Box component='span' sx={{display: 'block' ,
mt: 3, alignItems: 'center'}}>
            <PersonalisedButton onClick={onSubmit}
text={ui.submitText} isBlack={true} minHeight={false}
margin={1} />
            {(ui.clearText !== undefined &&
ui.clearText !== '')
              ? <PersonalisedButton onClick={onClear!}
text={ui.clearText!} isBlack={true} minHeight={false}
margin={1} />
              : null
            }
        </Box>

      </Box>
    </>
  )

}
```

En los dos siguientes fragmentos de código podemos encontrar por una parte, la factoría que nos devuelve el tipo de campo que hemos indicado, y por otra parte la interfaz que obliga a cualquier nuevo tipo de campo que se quiera incluir en la aplicación, implementar el método rénder, el cual devuelve un componente de React con la interfaz asociada al tipo de campo.

Clase FormInputComponentFactory, factoría que devuelve una instancia de la clase que indica el tipo de input indicado

```
export default class FormInputComponentFactory {
```

```
    public static getInputFieldView(type:
FormInputComponentType): FormInputComponent {
        switch(type){
            case FormInputComponentType.Text:
                return new TextFormInputComponent()
            case FormInputComponentType.Password:
                return new PasswordFormInputComponent()
            case FormInputComponentType.Date:
                return new DateFormInputComponent()
            case FormInputComponentType.Time:
                return new TimeFormInputComponent()
            case FormInputComponentType.DateTime:
                return new DateTimeFormInputComponent()
            default:
                return new TextFormInputComponent()
        }

    }

}
```

Interfaz FormInputComponent que define lo que tienen que aplicar los nuevos inputs que deseen crearse

```
export default interface FormInputComponent {
    getView(onChange: (fieldName: any, fieldValue: any)
=> void, formInputComponentData:
FormInputComponentData): ReactNode | null
}

export type FormInputComponentData = {
    name: string
    required: boolean
    label?: string
}
```

En el siguiente fragmento de código, podemos ver una implementación concreta de un tipo de campo, en este caso un campo de tipo fecha. Como se puede ver, lo que hacemos es aplicar la interfaz ForMInputComponente, indicando la interfaz que mostrará este tipo de input. Esta interfaz hace uso de React y Material UI como se puede ver.

Clase DateFormInputComponent. Se trata de un ejemplo de clase que implementa la interfaz FormInputComponent

```
export default class DateFormInputComponent implements
FormInputComponent {

  private value = new Date()

  getView(onChange: (fieldName: any, fieldValue: any)
=> void, formInputComponentData:
FormInputComponentData): ReactNode | null {
    return (<>
      <LocalizationProvider
dateAdapter={AdapterDateFns}>
        <MobileDatePicker
            label={formInputComponentData.label}
            inputFormat="MM/dd/yyyy"
            value={this.value}
            onChange={ (e) =>
              onChange(formInputComponentData.name,
this.value)
            }
            renderInput={
              (params: JSX.IntrinsicAttributes &
TextFieldProps) =>
                <TextField
required={formInputComponentData.required} {...params}
/>}
          />
      </LocalizationProvider>

    </>)
  }

}
```

Resumen. El desarrollo de este componente ha supuesto que si queremos desarrollar o implementar un nuevo tipo de input en nuestra aplicación (uno típico como puede ser para indicar fechas o u horas, o uno menos estándar y complejo para nuestra aplicación) tengamos tan solo que crear una clase que implemente el método rénder (es decir, aplicar la interfaz FormInputComponent, mostrando su vista en JSX), darlo de alta en el enumerado de tipos de inputs (FormInputFieldType) y en la factoría que se encarga de crearlos (FormInputFieldComponentFactory).

De esta forma, si queremos mostrar un nuevo formulario en nuestra aplicación, tan solo deberemos instanciar al componente FormInputComponent indicando sus

parámetros necesarios (los campos que va a tener, el objeto sobre el que se vuelcan la información indicada por el usuario, etc), abstrayéndonos de toda la complejidad de que tiene un formulario, como es la gestión de los eventos sobre los campos, el indicar cada uno de los componentes con su respectiva información y estilos, etc.

6 Comparativa con otros lenguajes para el prototipado

Con la creciente necesidad de las empresas y pequeños negocios de posicionarse en Internet como medio indispensable para el crecimiento, el desarrollo web y móvil ha cobrado una gran importancia, resultando en multitud de alternativas para realizar el desarrollo. Muy conocidos son React de Facebook (ahora Meta), Angular de Google o Vue (creado por uno de los creadores de Angular.

Ante la aparición de tantas alternativas, la crispación sobre cuál de ellas se debe usar es mayor por parte de desarrolladores y empresas, ya que cada uno cuenta con sus propias especificaciones, preferencias y demás particularidades.

En nuestro caso, hemos hecho uso de React con Material UI frente a sus principales competidores. Para ilustrar los motivos de nuestra preferencia por React para este trabajo, vamos a realizar una comparativa con respecto a las siguientes características:

- Autosuficiencia
- Curva de aprendizaje
- Comunidad
- Desempeño
- Lenguaje
- Estructura de la aplicación
- Madurez y estabilidad

6.1 Angular

Como principal competidor de React encontramos a Angular (16), un framework desarrollado por Google, que en comparación con React cuenta con una **curva de aprendizaje** mayor, por lo que está enfocado a desarrolladores veteranos y que cuenten con experiencia en el mundo del desarrollo web. Por su parte, React es un lenguaje mucho más sencillo de aprender y con una curva de aprendizaje menos pronunciada, siendo mucho más simple poder formar a alguien con pocos o nulos conocimientos de desarrollo web 22.

Angular, al ser un framework, cuenta con un ecosistema más robusto, incluyendo herramientas de terminal, librerías de interfaz gráfica, herramientas de Side rendering como Angular Universal, una especificación de cómo realizar una estructuración estandarizada y adecuada del proyecto, entre otros. Por otro lado, React, al ser una librería, dispone de un ecosistema más liviano, pero menos completo. Para ser una tecnología que aspira a la robustez de Angular, React se nutre de la comunidad a través de librerías de terceros, por lo que su **autosuficiencia** es mucho menor.

Un aspecto de relevancia también es la **madurez** de las herramientas, impidiendo un inesperado abandono por el distribuidor, falta de comunidad para encontrar ayuda etc.

algo que en caso de React y Angular es irrefutable, contando con buenas comunidades y el apoyo de dos de las más grandes empresas a nivel mundial.

Ambas tecnologías son OpenSource, por lo que son tecnologías en las que el apoyo de la **comunidad** es muy importante al crecer con las aportaciones de los desarrolladores en gran parte. En este aspecto React parece contar con la ventaja, cosa que podemos afirmar tras ver las estadísticas obtenidas en el año 2021 sobre que lenguaje elegirían los desarrolladores.

Fig. 9. Gráfico de StackOverflow sobre popularidad de los lenguajes web

Continuando, si hablamos **rendimiento** de React supera a Angular gracias al uso de estructuras como el Virtual DOM, comentado en el apartado X, o la existencia de un enlazado de datos unidireccional, es decir, que la información solo puede pasar en una única dirección, del padre al hijo dando lugar a un código más limpio y eficiente a la hora del renderizado que otros como Angular que tienen un enlazado de datos bidireccional (17).

Luego, a la hora de declarar componentes de la interfaz de usuario Angular nos ofrece el uso del **lenguaje** Typescript con plantillas HTML, permitiendo hacer uso de variables condicionales, entre otras funcionalidades. Por su parte, React hace uso del lenguaje JSX, comentado en la sección X.

Finalmente, en cuanto a la **estructuración de un proyecto** encontramos una gran diferencia entre Angular y React, ya que Angular nos ofrece una guía sobre una estructura estandarizada y como seguirla, en cambio, React tiene una estructura menos marcada, encontrando numerosas alternativas y opciones plantadas por la comunidad, pero sin encontrar opiniones por parte del propio React.

En conclusión, ambos son lenguajes muy instaurados en la actualidad y secundados por dos de las más grandes empresas tecnológicas, por lo que son buenas opciones para cualquier tipo de proyecto. Sin embargo, Angular se adaptaría mejor a un proyecto de mayor envergadura con trabajadores con cierta experiencia, mientras que React se adapta mejor a todo tipo de proyectos, pero para gran parte de la comunidad puede

resultar menos escalable en grandes proyectos comparado con Angular, al menos si no se hacen uso de mejoras como las comentadas en el apartado 6.

6.2 Vue

Otro competidor que cada vez más lucha por el puesto de mejor herramienta de desarrollo web es Vue (18), un framework desarrollado por uno de los creadores de React que ha buscado unificar Angular y React incluyendo características de ambas permitiendo una mayor flexibilidad a la hora de decidir cómo realizar ciertas acciones, resultando en una **curva de aprendizaje** inferior a React.

En este caso, al igual que Angular, es un framework, y como tal cuenta con un ecosistema más robusto, con gestor de proyectos como Vue CLI, un "packet manager" para la compartición de componentes como Bit, un enrutador como Vue Router etc. contando con mucha más **autosuficiencia**.

Al igual que React, Vue podemos decir que a día de hoy es una herramienta **madura**, no obstante, React cuenta con una mayor comunidad y una gran empresa como es Facebook mientras que Vue no esta respaldado por una gran empresa sino por una persona de renombre.

Continuando, Vue también es OpenSource, por lo que la **comunidad** detrás de este es un factor de relevancia, antiguamente Vue estaba muy recargado en comparación con las otras alternativas. Sin embargo, en estos últimos años ha estado en auge, lo que le coloca en una buena posición, pero detrás de Angular y React.

	stars 🌟	forks 🍽	issues ⚠
react	95112	17903	454
angular.js	58401	28928	583
vue	93210	13694	179
angular	35933	8700	2339

Fig. 10. Estadisticas de los lenguajes de desarrollo web

Algo en lo que tienen bastantes similitudes React y Vue es en el **rendimiento**, contando los dos con el uso del Virtual DOM, no obstante, en Vue se optó por el uso de enlace de datos bidireccional, que como comentamos anteriormente con Angular, complica las cosas debido a las numerosas interacciones para grandes aplicaciones que se generan.

Un buen punto a favor de Vue es su flexibilidad, encontrando diversas posibilidades para el **lenguaje** a emplear, contando con HTML, CSS y Javascript como una opción,

plantillas HTML o JSX como React permitiendo al usuario decidir aquella en la que tiene mayor maestría.

Para finalizar, la **estructuración** de proyectos es un aspecto en el que ambos comparten la libertad por parte del creador y la falta de un estándar claro de cómo llevar a cabo.

En conclusión, Vue es una opción muy adecuada para la realización de productos mínimos viables, así como, proyectos de pequeña complejidad debido a su fácil aprendizaje y sintaxis familiar. Por otro lado, React sería una decisión menos acertada en estos casos debido a que necesita una gran cantidad de código que suele dar frutos en proyectos de mayor envergadura, aunque en nuestro caso con el uso de Material UI se ha conseguido un gran rendimiento para realizar un proyecto pequeño.

6.3 Comparativa general

-	React	Angular	Vue
Autosuficiencia	Necesita otras librerías y componentes	Tiene un ecosistema propio	Tiene un ecosistema propio
Curva de Aprendizaje	Media	Alta	Baja
Comunidad	Activa y numerosa	Activa y notable	Activa y en crecimiento
Desempeño	Alto rendimiento con Virtual DOM	Inferior con el uso de Real DOM	Alto rendimiento con Virtual DOM y lazy loading
Lenguaje	JSX	Plantillas HTML, CSS, Typescript	Plantillas HTML, CSS, Javasript o JSX
Estructura	Poco estandarizada con varias alternativas planteadas por la comunidad	Uso de una arquitectura MVC especificada en la propia documentación	Poco estandarizada con varias alternativas planteadas por la comunidad
Madurez	Alta	Alta	Media-Alta

Table 1. Tabla de comparación de Angular, React y Vue

7 Problemas y mejoras para proyectos en React

7.1 Arquitectura hexagonal

Motivación. En el área del desarrollo nos enfrentamos a sistemas cada vez más complejos, siendo algunas de las mayores pesadillas las aplicaciones de software, la infiltración de la lógica del negocio en el código de la interfaz de usuario, así como el acoplamiento de las distintas partes de una aplicación (por ejemplo, entre lógica de negocio y detalles de infraestructura como la base de datos). Por tanto, las aplicaciones requieren de una estructura sólida que facilite su creación, mantenimiento y crecimiento en el futuro. Debido a esto, el concepto de arquitectura en el ámbito del software, que nos permita separar las responsabilidades mediante capas y definiendo reglas de dependencias entre ellas, adquiere una importancia cada vez mayor. Sin embargo, React no destaca por establecer o fomentar arquitecturas, así como estructuras de directorios. Esto, supone una gran velocidad y facilidad de desarrollo en etapas primerizas de un proyecto, pero en etapas más avanzadas supone que los proyectos puedan caracterizarse por poca mantenibilidad, testeabilidad, reutilización, flexibilidad frente a cambios e independencia de otros sistemas.

Explicación (19)**.** Frente a esto, presentamos la Arquitectura Hexagonal, cuyo motivo de creación es la de permitir que una aplicación sea usada de la misma forma por usuarios, programas, pruebas automatizadas y scripts, y que pudiera ser tanto desarrollada como probada de forma aislada de sus eventuales dispositivos y bases de datos en tiempo de ejecución. Esta arquitectura, también llamada arquitectura de puertos y adaptadores propone separar nuestra aplicación en distintas capas o regiones, cada una de ellas con su propia responsabilidad permitiendo así que evolucionen de manera aislada, y que cada una de ellas sea testeable e independiente de las demás. Para conseguir esta independencia de capas se utiliza el concepto de puertos y adaptadores. Un puerto no es más que un concepto lógico mediante el cual se define un punto de entrada y salida de la aplicación. La función del adaptador es la de implementar la conexión con ese puerto y otros servicios externos. De esta forma podremos tener múltiples adaptadores para un mismo puerto. Existen dos clases de puertos y adaptadores, primarios y secundarios. La diferencia entre ellos radica en quién desencadena o está a cargo de la conversación. En el caso de los puertos y adaptadores primarios, es el usuario quien, mediante la interfaz de usuario, realiza una solicitud a la aplicación. Por ejemplo, a través de una petición HTTP un usuario puede solicitar un listado. Por otro lado, en los puertos y adaptadores secundarios, la acción es desencadenada por la aplicación. Por ejemplo, podría realizarse una solicitud de persistencia en base de datos proveniente de una acción de un adaptador primario. Cabe destacar que la forma de hexágono no tiene nada que ver con su número de lados sino más bien con su representación, ya que cada lado representa un puerto hacia dentro o hacia fuera de la aplicación. Las distintas capas de la arquitectura hexagonal son:

Capa de dominio (20). Es la capa central del hexágono y contiene las reglas de negocio. En ella podemos encontrar los modelos de datos y sus restricciones. Esta capa no conoce cómo se estructurará, se guardará y se recuperará la información del repositorio. Simplemente expondrá una serie de interfaces (puertos) que serán adaptados en la capa de infraestructura para cada caso concreto de implementación de esa persistencia. Dicho de otra manera, nuestro núcleo puede ser descrito como una API con unos contratos bien especificados. Definiendo puertos o puntos de entrada e interfaces (adaptadores) para que otros módulos (UI, BBDD, Test) puedan implementarlas y comunicarse con la capa de negocio sin que ésta deba saber el origen de la conexión.

Capa de aplicación (20). En ella se definen los casos de uso pensando en las interfaces disponibles en el hexágono de la aplicación y no en alguna de las tecnologías disponibles que podamos utilizar. En esta capa también se adaptan las distintas peticiones que recibe la aplicación a través desde la capa de infraestructura. Por ejemplo, un caso de uso aceptará los datos de entrada que provienen de la capa de infraestructura y realizará las acciones necesarias para devolverle unos datos de salida.

Capa de infraestructura (20). .Se trata de la capa exterior del hexágono y corresponde a las implementaciones o adaptaciones de las interfaces o puertos de las demás capas. Normalmente esta capa corresponde al framework, pero también contiene librerías de terceros, SDKs o cualquier otro código externo a la aplicación. La capa de infraestructura implementa los servicios definidos en la capa de aplicación (adaptadores secundarios) así como todo lo relativo a la interacción con el usuario (adaptadores primarios). Dicho de otra manera, esta capa obtendrá unos datos de entrada que serán utilizados para solicitar el caso de uso correspondiente en la aplicación y devolverá unos datos de salida.

Comunicación entre capas (20). Tal y como hemos comentado, la Arquitectura Hexagonal propone que nuestro dominio sea el núcleo de las capas y que este no se acople a nada externo. En lugar de hacer uso explícito y mediante el principio de inversión de dependencias nos acoplamos a contratos (interfaces o puertos) y no a implementaciones concretas. De esta manera consigue desacoplar capas de nuestra aplicación permitiendo que evolucionen de manera aislada.

7.2 Typescript

Una buena mejora a la hora de desarrollar un proyecto de envergadura media o alta sería el uso de Typescript (21), un superset de Javascript, el cual añade grandes ventajas como las siguientes:

- **Tipado de datos**, permitiendo la detección de fallos en **compile time** en vez de tener que esperar a **ejecutar** el programa. Además, proporciona una mayor verbosidad con respecto a los errores producidos.
- Características típicas de los lenguajes orientados a objetos (OOP), como son las **interfaces**, **clases**, **uniones**, **herencia, clases abstractas**, etc.

- Mayor legibilidad del código debido al tipado de datos.
- Ofrece **sugerencias** y **autocompletado** de código, ya que, al contar con el tipado de datos sabe con qué clase estamos trabajando, y por tanto, sus métodos y atributos.

Así, cabe mencionar que, todo código **Javascript** es válido para **Typescript**, así como todo código Typescript se transforma a Javascript al compilarse. Esto, supone que la migración de un tipo de proyecto a otro sea viable y sin tantas complicaciones como si la migración fuese a otro lenguaje

A su vez, también debemos mencionar que no son todo ventajas (22), ya que, al añadir el tipado de datos, conceptos de OOP y demás, se incurre dedicar un tiempo añadido a obtener códigos más extensos, así no siendo ideal su uso para el desarrollo de un prototipo. Además, un factor a tener en cuenta es el uso de librerías de terceros con Typescript, ya que la existencia de los tipos empleados aun no es un estándar, por lo que puede resultar en un gran entorpecimiento (ya que la mayoría de las librerías de Javascript no tienen su respectiva versión en Typescript).

En conclusión, Typescript es una herramienta que puede hacer que proyectos desarrollados con Javascript sean más escalables y robustos frente a posibles fallos o incongruencias, así como facilita la mantenabilidad, legibilidad y testeabilidad del código.

7.3 RxJs

RxJS (Por sus siglas en Inglés, "Reactive Extensions for JavaScript") es una librería para programación reactiva usando obvservables que hacen más fácil la creación de código asincrono o basado en callbacks (23).

RxJS proporciona una implementación del tipo Observable, el cual es necesitado hasta que el tipo de dato sea parte del lenguaje y hasta que los navegadores ofrezcan un soporte. La librería también proporciona funciones de utilería para la creación y trabajo con observables. Dichas funciones de utilería pueden ser usadas para:

- Convertir código existente para operaciones asíncronas en observables.
- Iterar a través de valores en un flujo de datos.
- Mappear valores en tipos de datos diferentes.
- Filtrar flujos de datos.
- Composición de múltiplos flujos.

Ejemplo de uso de la librería RxJS

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

// Create an Observable out of a promise
const data = from(fetch('/api/endpoint'));
// Subscribe to begin listening for async result
data.subscribe({
  next(response) { console.log(response); },
  error(err) { console.error('Error: ' + err); },
```

```
    complete() { console.log('Completed'); }
  });
```

8 Conclusiones y vías futuras

React ha demostrado ser una gran herramienta de prototipado y desarrollo de proyectos web debido a su facilidad de aprendizaje en comparación a otras alternativas, gran documentación, amplia comunidad, gran rendimiento debido al concepto de virtual DOM, y un rico ecosistema de gestores de paquetes, librerías y demás. Sin embargo, como toda herramienta, hay que utilizarla en contextos en los que sea apropiado su uso.

Así, quizás React no sea la mejor opción para un desarrollo que deba seguir estrictamente ciertos patrones de diseño o arquitectónicos típicos de un proyecto de gran envergadura, ya que la herramienta de por sí no fuerza a seguir ninguna estructura o convención (como si hace Angular por ejemplo). Pero frente a esto, tal como hemos comentado en el apartado Problemas y mejoras para proyectos en React, podemos solucionar estos problemas forzando el uso de una arquitectura, una buena estructura de directorios y llevando a cabo buenas prácticas, recayendo la responsabilidad en los equipos de desarrollo.

Tampoco quizás, la combinación de React y Material UI sea el mejor conjunto de herramientas para muy veloz y adecuado para reunirse con clientes e ir adaptando el prototipo con ellos, siendo por ejemplo Axure RP mejor en este aspecto debido a su uso más visual de Drag and Drop. Pero desde luego que esto no significa que React no sea una tecnología adecuada para el desarrollo de prototipos, ya que el paso de prototipo a producto final es más rápido que por ejemplo Axure RP, debido a que esta última tan solo proporciona prototipos desechables.

En cuanto a vías futuras relacionadas con el ámbito de nuestro trabajo están:

- Un estudio en profundidad de las características más potentes de Typescript.
- La utilización de la herramienta de programación reactiva RxJs para la comunicación entre distintas partes de la aplicación sin acoplarnos a un framework de Javascript concreto, sino al lenguaje Javascript.
- El uso de otra librería de componentes como FUSE o Fluent UI.
- La utilización de alguna librería de Testing para React como Jest. (24)

9 Bibliografía

1. **Santos, Anjali Ariscrisnã y André.** *ImaginaryClouds.* [En línea] 5 de 5 de 2022. [Citado el: 1 de 12 de 2022.] https://www.imaginarycloud.com/blog/npm-vs-yarn-which-is-better/.

2. **Palacios, Adan.** Tipos de componentes en React Js. *DevTo.* [En línea] 17 de 5 de 2019. https://dev.to/ajpalacios/tipos-de-componentes-en-react-js-4epg.

3. **Alonso, Adrián.** Tipos de Componentes en ReactJS. *Medium.* [En línea] 12 de 2017. https://adrianalonsodev.medium.com/tipos-de-componentes-en-reactjs-f387a6f8e2b7.

4. **Facebok OpenSource.** *ReactJS.* [En línea] https://es.reactjs.org/docs/components-and-props.html.

5. **Cuarterolo, Franco.** *DevTo.* [En línea] 25 de 4 de 2021. https://dev.to/cuarte4/componentes-de-clase-o-funcionales-4c1c.

6. **Wieruch, Robin.** *Robin Wieruch.* [En línea] 19 de 4 de 2019. https://www.robinwieruch.de/react-higher-order-components/.

7. **Facebook OpenSource.** *ReactJs.* [En línea] [Citado el: 1 de 12 de 2022.] https://reactjs.org/docs/introducing-jsx.html.

8. **Fernández, Gerardo.** *Medium.* [En línea] 21 de 6 de 2019. [Citado el: 1 de 12 de 2022.] https://latteandcode.medium.com/y-eso-del-virtual-dom-de-react-qué-es-3feed6366925.

9. **Ankush.** *GeekFlare.* [En línea] 7 de 10 de 2022. https://geekflare.com/es/react-rendering/.

10. **Facebook OpenSource.** *ReactJS.* [En línea] https://es.reactjs.org/docs/rendering-elements.html.

11. **OpenSource, Facebook.** *ReactJS.* [En línea] https://reactjs.org/docs/hooks-intro.html.

12. **Huet, Pablo.** *OpenWebinars.* [En línea] 14 de 10 de 2020. https://openwebinars.net/blog/react-hooks-que-son-y-que-problemas-solucionan/.

13. **Remix Software.** *ReactRouter.* [En línea] Remix Software. [Citado el: 1 de 12 de 2022.] https://reactrouter.com/en/6.4.4/start/tutorial.

14. **Guaña, Juan C.** *FreeCodeCamp.* [En línea] FreeCodeCamp. [Citado el: 1 de 12 de 2022.] https://www.freecodecamp.org/espanol/news/la-hoja-de-trucos-de-react-router-todo-lo-que-necesitas-saber/.

15. **Google.** *MaterialUI.* [En línea] Google. [Citado el: 1 de 12 de 2022.] https://mui.com/material-ui/.

16. **Raval, Nihar.** *Radix.* [En línea] [Citado el: 1 de 12 de 2022.] https://radixweb.com/blog/react-vs-angular.

17. **Barros, Aryclenio.** *Dev.* [En línea] 3 de 6 de 2022. [Citado el: 1 de 12 de 2022.] https://dev.to/aryclenio/unidirectional-and-bidirectional-data-flow-the-ultimate-front-end-interview-questions-guide-pt-1-5cnc.

18. **Nowak, Maja.** *Monterail.* [En línea] [Citado el: 1 de 12 de 2022.] https://www.monterail.com/blog/vue-vs-react.

19. **Salguero, Edu.** *Medium.* [En línea] 22 de 6 de 2022. https://medium.com/@edusalguero/arquitectura-hexagonal-59834bb44b7f.

20. **Novoseltseva, Ekaterina.** *Apiumhub.* [En línea] 20 de 4 de 2022. https://apiumhub.com/es/tech-blog-barcelona/arquitectura-hexagonal/.

21. **Chacón, José Luis.** *Profile.* [En línea] [Citado el: 1 de 12 de 2022.] https://profile.es/blog/que-es-typescript-vs-javascript/.

22. ***STX Next.*** **[En línea] https://www.stxnext.com/blog/typescript-pros-cons-javascript/.**

23. Google LLC. *Angular.* [En línea] 2020. https://docs.angular.lat/guide/rx-library.

24. [En línea] https://dev.to/ajpalacios/tipos-de-componentes-en-react-js-4epg.