Centro Integrado de Formación Profesional

AVILÉS

Principado de Asturias

UNIDAD 5:
BIBLIOTECAS Y FRAMEWORKS. REACT

DESARROLLO WEB EN ENTORNO CLIENTE

2º CURSO

C.F.G.S. DISEÑO DE APLICACIONES WEB

REGISTRO DE CAMBIOS

Versión	Fecha	Estado	Resumen de cambios
1.0	22/11/2024	Aprobado	Primera versión
1.1	08/01/2025	Aprobado	Versión completa
1.2	04/02/2025	Aprobado	Añadidas dependencias en pruebas unitarias
1.3	09/02/2025	Aprobado	Añadido hook useRef
1.4	26/02/2025	Aprobado	Corregido formato

ÍNDICE

UNIDAD 5: BIBLIOTECAS Y FRAMEWORKS: REACT

5.1 BIBLIOTECAS Y FRAMEWORKS DE ACTUALIZACIÓN DINÁMICA

En la unidad 1 ya se presentaron las bibliotecas y frameworks más utilizadas, tales como Node.JS, jQuery, React, Angular y Vue.JS. A continuación, se verán otras herramientas que, aunque menos populares, también pueden ser de utilidad en el desarrollo de la parte de cliente.

5.1.1 PLUGINS JQUERY

La comunidad jQuery ha ido desarrollando durante su existencia multitud de plugins, desde pequeños selectores de ayuda a widgets de interfaz de usuario a gran escala. En su momento, estos se podían encontrar en el repositorio https://plugins.jquery.com/ aunque ahora se recomienda acudir al repositorio **npm** buscando plugin jQuery:
https://www.npmjs.com/search?q=keywords:jquery-plugin

Utilizar un plugin jQuery es tan simple como añadir su referencia detrás de la de jQuery en el head de la página:

```
<head>
   <script src="./Scripts/jquery.min.js"></script>
    <script src="./Scripts/jquery.plugin.js"></script>
</head>
```

Por ejemplo, el plugin jquery-form es un ejemplo de script que ayuda a convertir una tarea complicada en algo más sencillo. Su función básica es convertir un formulario convencional en un formulario AJAX (se verá posteriormente). Por ejemplo, la siguiente porción de código realiza esa operación con **miform**:

```
$(document).ready(function () {
  $('miform').ajaxForm();
});
```

Otro plugin interesante es jQuery UI. En este caso, no es un plugin al uso, sino una biblioteca completa de plugins, la cual contiene una serie de componentes básicos de interacción y widgets completos para ayudar a mejorar la experiencia Web hasta aproximarse a una aplicación de escritorio. Por ejemplo, se puede hacer que una lista sea ordenable con solo añadirle un método **sortable** a la lista en sí. Cuando se le añade este comportamiento, los elementos de la lista se pueden mover y reordenar.

En las versiones anteriores, las opciones de visualización se daban una serie de opciones en la llamada al método; en las últimas, se hace aplicando ciertos estilos a cada elemento de la lista.

En la siguiente porción de código puede verse cómo hacer una lista ordenable mediante jQuery-UI.

```
<html>
  <head>
    <link rel="stylesheet"
      href=https://code.jquery.com/ui/1.14.1/themes/base/jquery-ui.css />
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.7.1/jquery.min.js">
    </script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jqueryui/1.14.1/jquery-
ui.min.js"></script>
    <style>
      #sortable {list-style-type: none; margin: 0; padding: 0; width: 60%;}
      #sortable li {margin: 0 3px 3px 3px; padding: 0.4em; padding-left: 1.5em;
        font-size: 1.4em; height: 18px;}
      #sortable li span {position: absolute; margin-left: -1.3em;}
    </style>
  </head>
  <body>
    <ul id="sort-container">
      <li class="ui-state-default">
        <span class="ui-icon ui-icon-arrowthick-2-n-s"></span>Juan</li>
      <li class="ui-state-default">
        <span class="ui-icon ui-icon-arrowthick-2-n-s"></span>Paula</li>
      <li class="ui-state-default">
        <span class="ui-icon ui-icon-arrowthick-2-n-s"></span>Jorge</li>
      <li class="ui-state-default">
        <span class="ui-icon ui-icon-arrowthick-2-n-s"></span>Rita</li>
      <li class="ui-state-default">
        <span class="ui-icon ui-icon-arrowthick-2-n-s"></span>Pedro</li>
      <li class="ui-state-default">
        <span class="ui-icon ui-icon-arrowthick-2-n-s"></span>Sara</li>
    </ul>
    <script>
      $(function () {
        $("#sort-container").sortable({
          opacity: 0.5,
          cursor: "move",
          axis: "y",
        });
      });
    </script>
  </body>
</html>
```

Además, jQuery UI incluye un conjunto de widgets robustos de interfaz de usuario que funcionan de forma parecida a elementos de aplicaciones de escritorio. Por ejemplo, el widget Dialog permite crear un cuadro de diálogo de forma que se pueda arrastrar y soltar.

Al igual que antes se consiguió hacer una lista ordenable, se puede aplicar a un div el método dialog para convertirlo en un cuadro de diálogo. Se puede ver un ejemplo a continuación:

```
<html>
  <head>
    <link
      rel="stylesheet"
      href=https://code.jquery.com/ui/1.14.1/themes/base/jquery-ui.css />
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jqueryui/1.14.1/jquery-
ui.min.js"></script>
    <script>
      $(function () {
        $("#dlg").dialog();
      });
    </script>
    <title>Ejemplo de Dialog</title>
  </head>
  <body>
    <div id="dlg" title="Diálogo básico">
      <p>
        Mi diálogo básico. Esta ventana se puede mover, redimensionar y cerrar
        mediante el icono 'x'
      </p>
    </div>
  </body>
</html>
```

Otro plugin interesante es Magnify, el cual permite que con una imagen pequeña y la misma con mayor tamaño y resolución se puede generar una lupa que permita ampliación sobre la imagen más pequeña. En el siguiente código se puede ver un ejemplo:

```
<html><head>
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/magnify/2.3.3/css/magnify.css">
 <script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/magnify/2.3.3/js/jquery.magnify.min.js">
</script></head>
  <body>
    <img src="product.png" class="zoom" data-magnify-src="product-large.png">
    <script>
        $(document).ready(function() {
          $('.zoom').magnify();
        });
        </script>
 </body>
</html>
```

5.1.2 EXPRESS

Express es un framework web para Node.JS el cual permite desarrollar aplicaciones web estables en poco tiempo. Además, es muy ligero, es decir, no parte de demasiadas premisas y pone a disposición material suficiente como para ahorrar tiempos de programación. Hay que matizar que, a diferencia de otras herramientas que se verán en este apartado, Express no solo se centra en la parte de desarrollo en entorno cliente, sino que también se ubica en la parte de servidor. Esto tiene como ventaja el hecho de que no es necesario aprender un lenguaje nuevo para desarrollar una aplicación web de forma íntegra, con conocer JavaScript es suficiente si se usa NodeJS + Express.

Entre otras cosas, Express puede hacer lo siguiente:

- APIs basadas en JSON.
- Aplicaciones web de una sola página (SPA)
- Aplicaciones web que se ejecutan en tiempo real

Entre sus ventajas están las siguientes:

- Reduce el tiempo necesario para crear aplicaciones
- Contiene modelos como enrutamientos y capas para vistas de forma que no es necesario volver a programarlos
- La comunidad de desarrolladores usa este framework sin parar dedicándose a su mantenimiento y a probar código, que además es muy estable.

Express se ha inspirado en Sinatra, un framework web ligero para el lenguaje Ruby. Es compatible con los motores para plantillas y lo que se conoce como enrutamiento. Además, la forma de pasar información a las vistas es muy similar a la que sigue Sinatra con Ruby.

La instalación de Express se realiza mediante la siguiente orden:

```
npm install -g express
```

En este ejemplo se usa el modificador -g lo que provoca que Express se instale de forma global pudiendo ser ejecutado desde cualquier comando independientemente de dónde esté.

Una vez instalado Express, ahora se puede construir y poner en marcha un sitio muy básico. La herramienta que permite crear el esqueleto de una web es Express Generator. Se lanza del siguiente modo:

```
npm install -g express-generator
```

También aquí se usa el modificador -g por las mismas razones que en el caso anterior. Para ver que se ha instalado correctamente, se lanza la siguiente orden:

```
express --version
```

la cual muestra la versión instalada de Express.

Ahora, para crear una aplicación de ejemplo, basta con lanzar la orden **express** con el nombre del directorio destino y, por ende, de la aplicación. Además, se le puede dar un motor de vistas a medidas, a elegir entre uno de los siguientes según muestra la ayuda: ejs | hbs | hjs | jade | pug | twig | vash

En la documentación oficial se decanta por Pug, pero el más utilizado actualmente es EJS (Embedded JavaScript). Estos motores de vistas no son más que un conjunto de plantillas que permite escribir HTML de forma más sencilla y legible. Por tanto, la orden a ejecutar es la siguiente:

```
express --view=ejs miapli
```

Esto crea un directorio miapli ahí donde se haya ejecutado. Ahora hay que entrar en ese directorio e instalar las dependencias necesarias. Estas vienen definidas en el archivo package.json dentro del apartado dependencies.

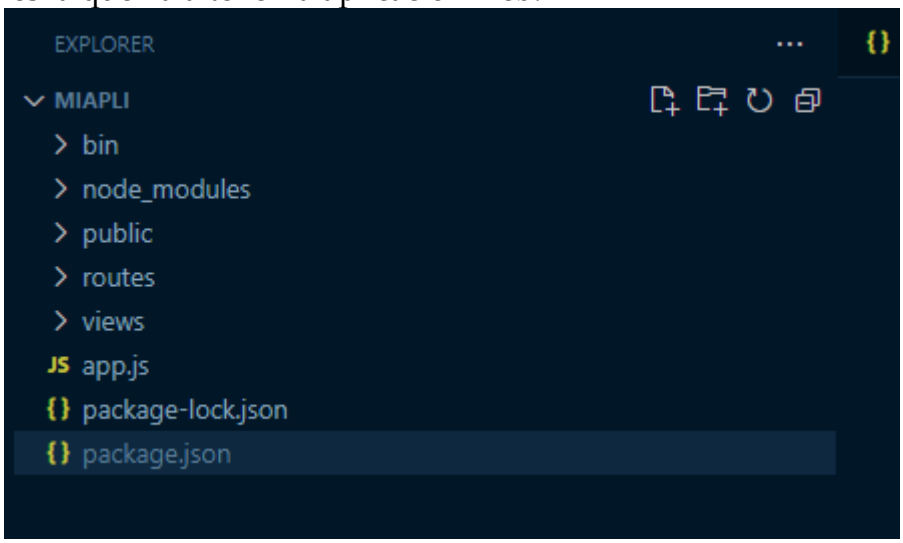
```
cd miapli && npm install
```

Al terminar de instalar las dependencias, es bastante probable que algunas hayan quedado obsoletas. El problema de esto es que de ahí surgen vulnerabilidades que pueden ser importantes (las etiquetadas como high). Al terminar, el sistema propone fijarlas lanzando la siguiente orden:

```
npm audit fix --force
```

Es posible que se actualicen unas y a su vez queden otras sin actualizar, por lo que probablemente haya que lanzarlo más de una vez hasta que muestre 0 vulnerabilities.

Al terminar, si se abre el directorio con Visual Studio Code, se puede ver la siguiente estructura, que es la que va a tener la aplicación web:

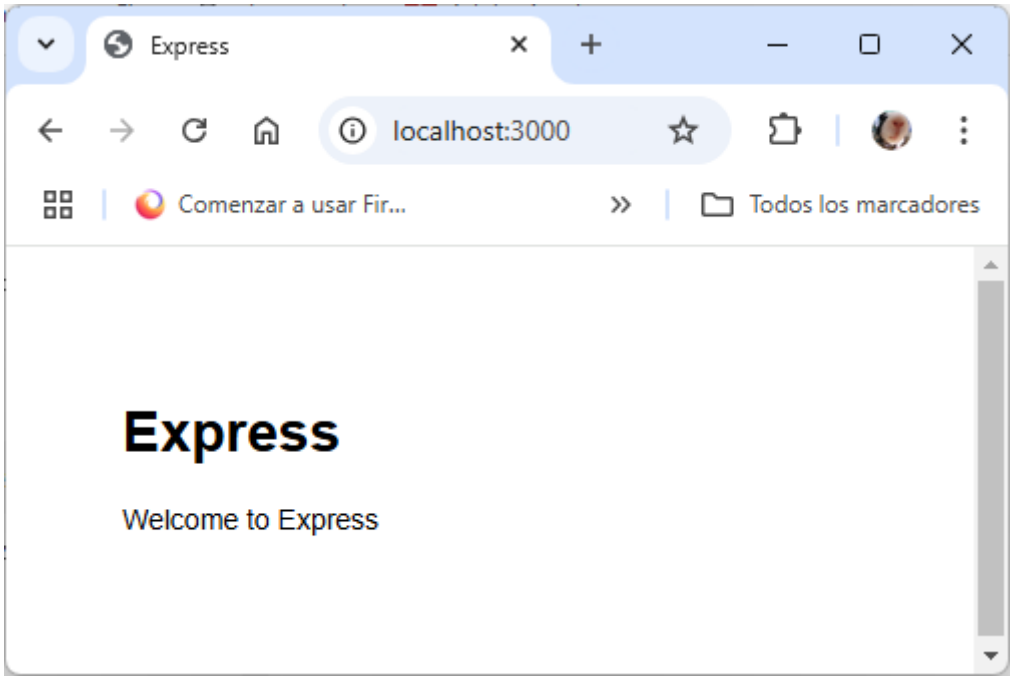


1. Estructura de aplicación Express

Una vez terminada la instalación, la aplicación se ejecuta con el siguiente comando (en MS-DOS):

```
set DEBUG=miapli:* & npm start
```

A partir de este momento, se inicia la aplicación, la cual puede ser consultada accediendo a http://localhost:3000 (probablemente se muestre un aviso del Firewall en Windows).



2. Aplicación de ejemplo de Express en ejecución

La aplicación en ejecución va a registrar los accesos que se produzcan:

```
D:\Codigo\miapli>set DEBUG=myapp:* & npm start

> miapli@0.0.0 start
> node ./bin/www

GET / 200 164.075 ms - 170
GET /stylesheets/style.css 200 2.472 ms - 111
GET /favicon.ico 404 6.987 ms - 982
```

3. Registro de accesos en aplicación Express

En cuanto a la estructura de una aplicación Express, se pueden ver los siguientes elementos:

- bin: Contiene el código del archivo que inicia la aplicación.
- app.js: El archivo que sirve de inicio a la aplicación. Es del que se parte al lanzar el comando de ejecución en combinación con bin.
- package.json: Proporciona información sobre la aplicación incluyendo las dependencias requeridas

- node_modules: Los módulos del nodo que se han definido en package.json incluyendo sus dependencias
- public: Este directorio sirve la aplicación a la web. En él se encuentran hojas de estilo, código JavaScript e imágenes. No contiene lógica alguna de la aplicación, simplemente es un modelo que se usa para garantizar la seguridad de las aplicaciones Express.
- routes: Una ruta define las páginas a las que debe responder la aplicación. Por ejemplo, si se necesita que la aplicación contenga una página "Acerca de" podría configurarse una ruta about. En este directorio es donde estaría esta declaración además de otras.
- views: En este directorio se encuentran las vistas, lo que en esencia determina el frontal de la aplicación. En la imagen se puede ver el código HTML de la vista principal (index.ejs). Es interesante ver cómo usa HTML convencional unido a una serie de código incrustado (similar a lo que se hace en PHP, ASP o JSP) con contenido dinámico.

views > index.ejs > ...

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' /
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
  </body>
</html>
```

Aquí la única variable dinámica es **title**. ¿Dónde se puede encontrar su asignación? Basta con acceder al archivo **routes/index.js** para observar la siguiente porción de código:

```
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});
```

Dado que lo que se muestra al acceder a la aplicación no es ni más ni menos que a la ruta principal (index) este archivo contendrá la codificación necesaria para su representación; en él se puede ver cómo proporciona un parámetro **title** a la página que luego la vista **index.ejs** recoge y ubica donde corresponda. Hay que tener en cuenta que no responde a los cambios en tiempo real, por lo que, si se cambia algo en el código, hay que parar la aplicación (con Control-C) y volverla arrancar.

El equipo que desarrolló Express.js ha lanzado un framework minimalista llamado koa que promueve un enfoque modular y moderno. En esta URL se puede ver cómo empezar a trabajar con él: https://koajs.com/#introduction

5.1.3. OTROS FRAMEWORKS Y BIBLIOTECAS

Además de los ya citados anteriormente y los descritos en el apartado anterior, a continuación, se citan otros frameworks y bibliotecas que no son tan populares, pero pueden ser de utilidad ya sea como sustitutos de los ya vistos o como complementos:

- Lit. Se trata de una biblioteca minimalista creada por Google que se basa en la especificación estándar de los Web Components vistos anteriormente. Es muy ligero y se enfoca en la construcción de componentes reutilizables sin imponer una estructura rígida. Utiliza la API de Web Components en lo que se refiere a componentes personalizados y Shadow DOM, lo que implica que sus componentes pueden utilizarse en combinación con otras tecnologías. En la siguiente URL se explica cómo comenzar a construir aplicaciones con Lit: https://lit.dev/docs/getting-started/
- NextJS. Framework sobre React cuyo objetivo es simplificar la creación de aplicaciones web ya que ofrece renderizado híbrido, optimiza el rendimiento, permite enrutamiento basado en archivos e incluso crear APIs dentro de un mismo proyecto. Su objetivo no es sustituir a React sino complementarlo. Tiene una particularidad que lo hace muy interesante y es que tiene renderizado del lado del servidor (SSR – Server-side Rendering) lo que no implica que sea una tecnología de servidor, sino que genera las páginas llamando a código en Node.JS orientada a la visualización en la página. La diferencia con una aplicación React pura es que su contenido se genera totalmente en el cliente. Se pueden ver sus primeros pasos en el siguiente enlace: https://nextjs.org/docs
- Alpine.js. Biblioteca JavaScript que ofrece la naturaleza reactiva y declarativa de otras como Vue.JS o React. Es muy ligerita y permite agregar interactividad básica a proyectos web, además de más sencilla que las ya citadas. Para ver una primera aproximación de instalación y uso: https://alpinejs.dev/start-here
- Svelte.js. Framework para convertir interfaces de usuario de forma sencilla y eficiente. En lugar de ejecutar su lógica en el navegador, Svelte compila el código en componentes JavaScript, por lo que aquel ya recibe código optimizado y eficiente sin necesidad de un virtual DOM ni similares, ya que actualiza el DOM directamente. Tiene una sintaxis declarativa que combina HTML, CSS y JavaScript en un solo archivo de componente; además, también incorpora reactividad. Primeras instrucciones de uso: https://svelte.dev/docs
- Astro. Otro framework de desarrollo web cuya consigna es la mejora de rendimiento por usar "menos JavaScript en el cliente". Se centra en contenido estático y aplicaciones web ligeras. Puede convivir con componentes de otras bibliotecas y frameworks como React, Vue o Svelte pero solo envía el JavaScript necesario para ellos dejando el resto de la página como HTML estático. Básicamente, lo que hace es compilar el código en HTML estático durante el proceso de construcción. Los componentes se renderizan en servidor (como Next) y si es necesario añadir interactividad, ya se hace en el cliente. Para comenzar a usarlo: https://docs.astro.build/en/getting-started/
- Otros: SolidJS, Qwik, Mihtril.js, Stencil.js, Ember.js, Riot.js, Restify (para API REST) etc.

5.2 INTRODUCCIÓN A REACT

React es una biblioteca de código abierto para el desarrollo de aplicaciones web que también sirve para crear sistemas de información para otras plataformas como dispositivos móviles u ordenadores personales. Su principal característica es el uso de un DOM virtual donde se realizan todos los cambios en la interfaz de usuario que luego se aplican al DOM real acelerando enormemente la ejecución de aplicaciones.

A continuación, se indican algunos de los puntos fuertes de React para ser elegida en desarrollo de aplicaciones:

- Es una biblioteca más bien ligera si se compara, por ejemplo, con Angular
- La división de la interfaz de usuario en componentes reutilizables
- Uso directo de JavaScript y JSX para el diseño de interfaces

Muchas de las características de React tienen su base en elementos de la programación funcional y de la declarativa. Cuando se desarrolla software, lo convencional es establecer un flujo en el que hay un inicio y un fin y encargarse de establecer el control sobre ese flujo. Las sentencias de programación son órdenes que se ejecutarán en la secuencia de ese flujo.

En la siguiente tabla se detallan los distintos tipos de programación a nivel general:

Tipo de programación	Descripción
Imperativa	Programación mediante un flujo de instrucciones
Declarativa	Expresamos qué se quiere, no cómo debe computarse.
Estructurada	Se estructura el flujo con anidamientos evitando el uso de go to
Orientada a objetos	Cada objeto es un contenedor tanto de sus propiedades como de su propio comportamiento
Lógica	Basada en reglas de inferencia que permite generar respuestas
Funcional	Se trata de desarrollar mediante **funciones puras**.

Una función pura:

1. Siempre devuelve el mismo resultado para los mismos parámetros.
2. No tiene efectos secundarios (side-effect). La función no puede modificar nada, ni realizar ninguna otra acción.

En programación funcional se evita la utilización de variables, ya que estas son mutables y su modificación lleva a efectos secundarios.

Al igual que SQL, el motor de la base de datos toma la declaración de la sentencia y devuelve los datos, **React** proporciona un lenguaje declarativo que permite encapsular los elementos **HTML** y sus datos y **React se encarga de** reproducir el **HTML final.** Se apoya en una copia ligera del DOM, Virtual DOM, permite que pueda realizar la comparación entre dos estados del DOM antes de que éste se muestre finalmente. Esa comparación se llama diffing. Cuando React evalua el Virtual DOM y comprueba los componentes del DOM real afectados, se encarga de refrescar únicamente estos, ahorrando tiempo de ejecución en el renderizado.

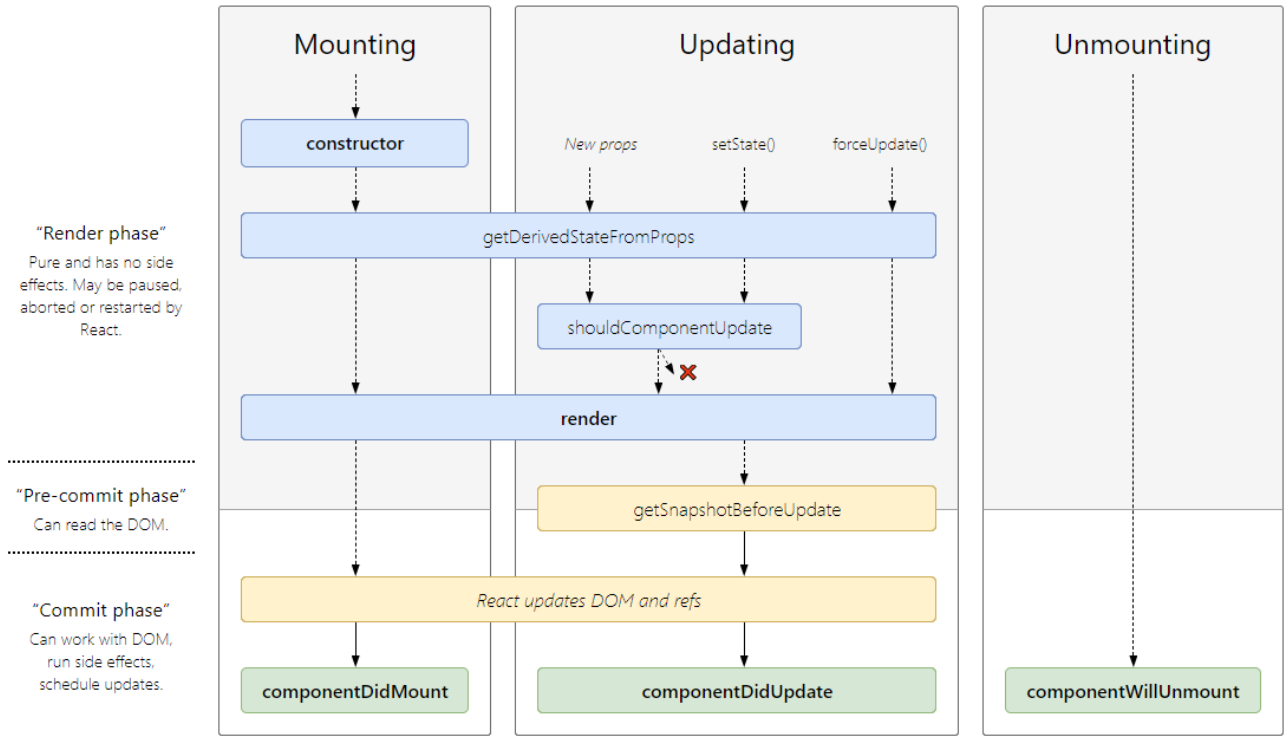
En React hay que pensar en términos declarativos; es decir, analizar los estados del componente y hacer que el interfaz reaccione ante los cambios de estado. Las fases de ese diseño son:

1. Identificar los estados visuales
2. Determinar qué hace que esos estados varíen.
3. Mantener el estado en "alguna memoria"
4. Conectar el cambio de estado con el evento que lo maneja.

Posibles estados:

- Vacío. El formulario está vacío
- Rellenando. El formulario tiene datos
- Enviando. Los datos se están enviando
- Validado. Datos enviados correctamente.
- Error. Mensaje de error en rellenando.

El ciclo de vida de React se puede ver en la siguiente figura:



4. Ciclo de vida en React

Cada **componente** de **React** (o elemento generado por él) pasa por tres fases en el ciclo de vida:

1. **Montaje.** El nuevo componente se crea y se inserta en el DOM. Sólo sucede una vez.
2. **Actualización.** Cuando las **props** (propiedades) del componente se modifican o el **state** se actualiza, el elemento se tiene que volver a mostrar con las modificaciones.
3. **Desmontaje.** El elemento se elimina del DOM.

La utilización de componentes funcionales (recomendado) hace que el ciclo de vida del componente quede oculto al desarrollador.

5.2.1 INSTALACIÓN DE ENTORNO

Una de las distintas formas de instalación de React se basa en el uso del entorno de ejecución Node.js, por lo tanto, sería un requisito previo imprescindible. Su gestor de paquetes es **npm**, el cual se utilizará para realizar aquellas operaciones que requieran una cierta automatización. Otra opción es utilizar yarn como gestor de paquetes, el cual dispone de opciones equivalentes a **npm**. Con el gestor de paquetes elegido, se instalaría el paquete **create-react-app** y luego se lanzaría este comando con el nombre de la aplicación a crear. El problema es que esa herramienta ya es obsoleta y ha cesado su desarrollo desde 2023, por lo que el equipo de React desaconseja su uso.

Un gestor de paquetes moderno y que supone una evolución de **npm** es **pnpm** ya que es más rápido y eficiente que el primero. Para crear un primer proyecto de React (una miniaplicación funcional que muestra un mensaje "holamundo"), se utilizará pnpm combinado con Vite como herramienta de construcción. Los pasos a seguir se muestran a continuación:

1. Instalar **pnpm**:

```
npm install -g pnpm
```

2. Se crea la aplicación con el gestor y vite utilizando la plantilla react

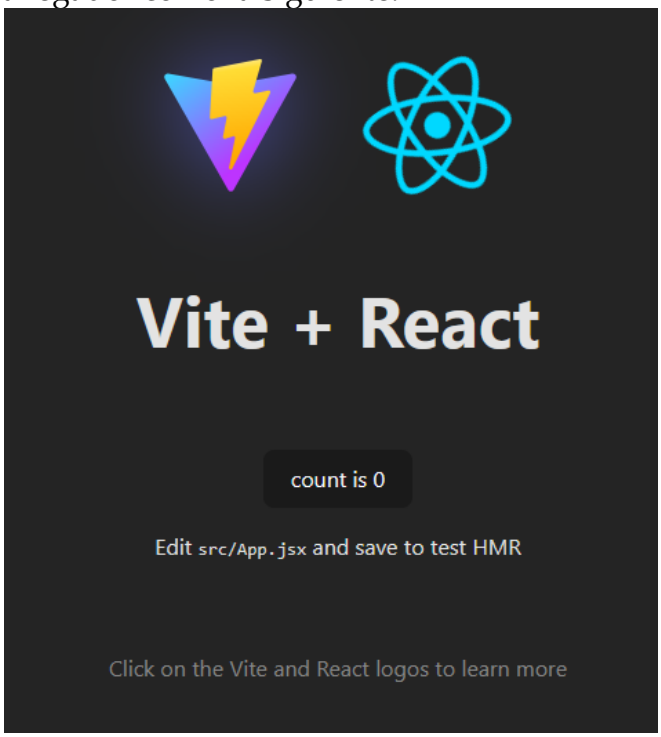
```
pnpm create vite hola-mundo -- --template react
```

Lanzará un asistente en el que inicialmente se ajustará React como framework con variante JavaScript.

Al terminar, ejecutará directamente el servidor mostrando lo siguiente en consola:

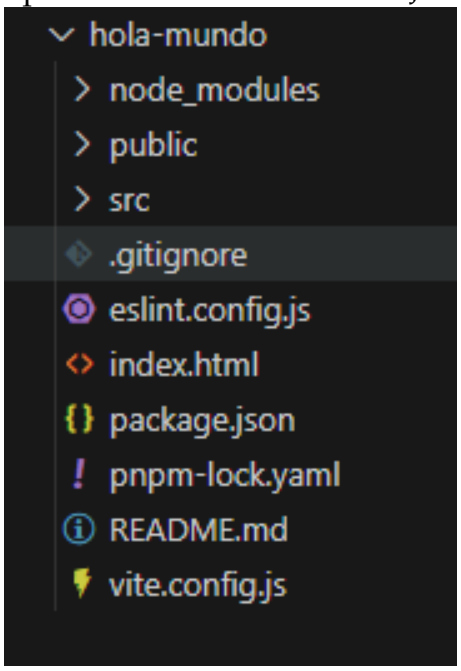
```
  VITE v7.2.2  ready in 884 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

Que por lo que se puede ver, la aplicación ya está arrancada y funcionando en la URL dada. Daría una salida en el navegador como la siguiente:



5. Página de presentación de Vite + React

Una vez terminado, se puede ver una estructura con algunos elementos comunes con la creada mediante Express como **node_modules**, **public** y **package.json**. En public se ubica la página inicial index.html además de otros elementos típicos de una aplicación web (favicon, logos, robots.txt, archivo de manifiesto). En src es donde se localiza el código fuente a editar para desarrollar la aplicación. Ahí se pueden encontrar archivos JavaScript y CSS principalmente.



6. Estructura de aplicación React

5.2.2 COMPONENTES BÁSICOS

A continuación, se enumeran y detallan los ficheros de código esenciales en un proyecto React.

main.js

Es el fichero que inicia la aplicación React y que incluye el único componente que tiene en este caso: App. Básicamente, el inicio de React no es más que una llamada a renderizar el elemento raíz <App /> y se le indica en qué elemento de la página lo va a cargar, en este caso, un elemento HTML cuyo id es root.

```jsx
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

7. Fichero main.jsx

El componente **React.StrictMode** se añade sobre <App /> para que sea posible detectar y depurar problemas durante el desarrollo del proyecto.

index.html

Este fichero, que se encuentra en public, contiene el HTML donde se van a cargar los componentes React. Según cómo lo define el archivo main.jsx, el componente raíz se debería cargar dentro de un elemento cuyo id sea root. Esto es sencillamente parte de la plantilla HTML donde puede verse este elemento.

```html
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>hola-mundo</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

8. index.html

App.jsx

Fichero que define el único componente React que se utiliza el cual devuelve un contenido estático mediante un **return**. Aunque pareciera que devuelve HTML, en realidad está combinado con JSX, lenguaje que se utiliza en React para poder introducir código, eventos, etc.

```jsx
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'

function App() {
  const [count, setCount] = useState(0)

  return (
    <>
      <div>
        <a href="https://vite.dev" target="_blank">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank">
          <img src={reactLogo} className="logo react" alt="React logo" />
        </a>
      </div>
      <h1>Vite + React</h1>
      <div className="card">
        <button onClick={() => setCount((count) => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.jsx</code> and save to test HMR
        </p>
      </div>
      <p className="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
    </>
  )
}

export default App
```

9. Contenido de App.js

Básicamente, es una función JavaScript que devuelve contenido, es decir, un componente funcional. Este es el estilo propuesto por el equipo de desarrollo de **React** y la comunidad, es decir, funcional en contraposición al orientado a objetos, aunque también se puede utilizar en la creación de componentes heredando de la clase **Component**.

App.css

Representa los estilos aplicados al componente.

```css
#root {
  max-width: 1280px;
  margin: 0 auto;
  padding: 2rem;
  text-align: center;
}

.logo {
  height: 6em;
  padding: 1.5em;
  will-change: filter;
  transition: filter 300ms;
}
.logo:hover {
  filter: drop-shadow(0 0 2em #646cffaa);
}
.logo.react:hover {
  filter: drop-shadow(0 0 2em #61dafbaa);
}

@keyframes logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}

@media (prefers-reduced-motion: no-preference) {
  a:nth-of-type(2) .logo {
    animation: logo-spin infinite 20s linear;
  }
}
```

10. Extracto de App.css

vite.svg

El logotipo en formato de imagen vectorial que se muestra desde el componente App de esta aplicación. Una imagen vectorial SVG se define mediante XML y tiene la particularidad de que se puede escalar sin perder calidad.

5.2.3 EJECUCIÓN DEL PROYECTO

Ahora se puede ejecutar el proyecto en modo desarrollo lanzando en la consola la siguiente orden:

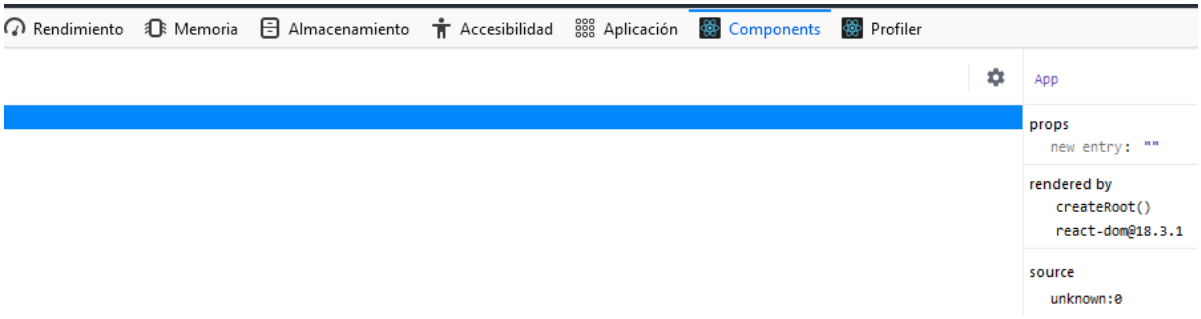
```
vite dev
```

Lo que hace esta orden (entre otras cosas) es "transpilar" el JSX de React. ¿Y qué es transpilar? No, no se trata de transportar un pilar o de ponerse detrás; la transpilación no es ni más ni menos que la conversión de un código JavaScript (o de otros lenguajes de script como TypeScript o JSX) a JavaScript compatible con un navegador o especificación. En definitiva, como JSX no es entendido por el navegador, se traduce a JavaScript para que pueda funcionar.

De esta forma, se ha podido desarrollar en un lenguaje que aquel no conoce y la biblioteca ya se encarga de traducirlo.

Al arrancar la web, ya va dando pistas de lo que se puede hacer. La propia página indica que se puede editar el archivo **App.js** y que basta con guardarlo para que se recargue de forma automática, es decir, lo que se consideraría un comportamiento reactivo.

A partir de aquí, para mejorar la depuración de aplicaciones React, es interesante instalar una extensión de navegador conocida como React Developer Tools. Una vez instalada, si se abren las herramientas de depuración, aparecen nuevas pestañas relacionadas con **React: Components y Profiler**



11. Nuevas pestañas de React Developer Tools

Por otra parte, para trabajar con Visual Studio Code, la extensión **ES7+ React/Redux/React-Native snippets** es muy útil ya que permite introducir código de forma automática utilizando atajos. En esta URL se puede consultar la lista completa:
https://github.com/r5n-dev/vscode-react-javascript-snippets/blob/HEAD/docs/Snippets.md

5.2.4 EL VIRTUAL DOM DE REACT

React dispone de un mecanismo que facilita mucho la renderización de elementos del DOM. Hay que tener en cuenta que cuando se produce un cambio en el DOM, a la hora de renderizarlo, los navegadores lo que hacen es refrescar la página al completo. En páginas sencillas, esto no tiene por qué suponer un problema, pero si la página dispone de unos cuantos componentes y tiene una cierta complejidad, sí que se puede notar la pérdida de rendimiento y quien más lo va a hacer es el usuario final.

El **Virtual DOM de React** no deja de ser una copia del DOM para su manejo con la biblioteca. Si se piensa en cambios en el DOM como los citados anteriormente, este Virtual DOM optimizará el refrescado de la página agrupando elementos para que no sea necesario tener que volver a generar todo el HTML. React, por tanto, marcará aquellos nodos que puedan ser modificables o sobre los que se haya previsto una modificación (lo que se conoce como Dirty Nodes) y se lo notificará al Virtual DOM para que el renderizado sea más sencillo y eficiente.

Para ilustrar el concepto de Virtual DOM se verá un ejemplo en el que se va a utilizar la biblioteca sin necesidad de instalarla, simplemente referenciando a repositorios externos de forma que se pueda ejecutar este código incluso en un sistema de ficheros sin necesidad de recurrir a un servidor web.

Para ello, en primer lugar, se creará una página index.html con el siguiente código:

```
<body>
    <div id="boton"></div>
    <!-- Bibliotecas React -->
    <script src="https://unpkg.com/react@18/umd/react.development.js"
crossorigin></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>
    <!-- Código propio -->
    <script src="boton.js"></script>
</body>
```

Esta página tiene un único componente **HTML,** un ** ** identificado con el **id boton.** En ese punto se pone a funcionar un elemento **<button>** apoyándose en **React.** Se cargan los scripts de las bibliotecas de React y se tiene el código que se encargará de manipular el componente deseado, en este caso un botón. Como se ve en el código, se hace referencia a un archivo llamado **boton.js** que deberá estar al mismo nivel que el **index.html** en el árbol de directorios. Su código se muestra a continuación:

```
const Boton = () => {
  const [color, setColor] = React.useState("hsl(0, 50%, 50%)");
  // Se extrae la función del handler
  const handleClick = () => {
    setColor(`hsl(${Math.random() * 360}, 80%, 50%)`);
  };
  return React.createElement(
    "button",
    {
      onClick: handleClick,
      style: {
        backgroundColor: color,
        padding: "10px 20px",
        border: "none",
        borderRadius: "5px",
        cursor: "pointer",
      },
    },
    "Cambia el color"
  );
};
```

La función representa un **componente React: Boton.** Esta función conforma y devuelve un **elemento de React. El elemento es el objeto que** recoge y representa las características que se desean del elemento **DOM.** El método **createElement** de React es muy similar al que proporciona el objeto Document. La gran diferencia es que el elemento creado en React se vinculará al Virtual DOM y será manejado por la biblioteca.

A continuación de la función, se introduce el siguiente código:

```
const nodoRaiz = document.getElementById('boton');
const raiz = ReactDOM.createRoot(nodoRaiz);
const boton = React.createElement(Boton);
raiz.render(boton);
```

El código realiza los siguientes pasos:

1. Tomar el elemento del **DOM** real donde se mostrará la **aplicación**.
2. Crear el elemento raíz del **DOM Virtual** que se mostrará en el **DOM Real**
3. Crear el **Elemento** a partir del componente**.**
4. Renderizar el elemento en el contenedor del **DOM Real** (en este caso el nodoRaiz) utilizando el proceso de **diferenciado (diffing).**

5.3 EL LENGUAJE JSX

JSX no es más que una extensión de sintaxis de JavaScript para escribir aplicaciones React. Básicamente, lo que hace es crear componentes React que luego materializa en una página web. En esencia, permite una comunicación más sencilla entre HTML y JavaScript. El siguiente código ilustra un ejemplo muy básico de uso de JSX:

```
const elemento = <h1>Hola, mundo</h1>;
```

Como puede verse, es una mezcla de HTML y JavaScript sin ser ninguna de los dos lenguajes en su forma estricta. El siguiente ejemplo va un poco más allá:

```
const nombre = 'Juan Diego';
const elemento = <h1>Hola, {nombre}</h1>;
```

En este caso, se personaliza el nombre al que saludar y se integra dentro del código de marcado incluyendo la variable entre llaves.

El uso de JSX, al igual que el de un diseño funcional, es opcional en React, pero al resultar tan cómodo, se usa en prácticamente la totalidad de proyectos React.

Hay que tener en cuenta que JSX es mucho más estricto que HTML. Por ejemplo, hay que cerrar las etiquetas **
. Además, un componente no puede devolver múltiples etiquetas JSX sino que hay que envolverlas en un elemento padre compartido (por ejemplo, un **div) o en un envoltorio vacío como puede verse en el siguiente ejemplo:

```
const AcercaDe = () => {
  return (
    <>
      <h1>Acerca de</h1>
      <p>
        Hola.
        <br />
        ¿Cómo vas?
      </p>
    </>
  );
};
export default AcercaDe;
```

Otra restricción de JSX: si se generan varios elementos en un componente tales como listas, filas de tabla, etc. usando un bucle, se debe asignar una clave a cada uno de esos elementos incluyendo un atributo **key**. Si no se hace, el propio generador de la aplicación React advertirá con un mensaje de aviso como el siguiente:
Warning: Each child in a list should have a unique "key" prop.

La página oficial de **React** ofrece un convertidor de HTML a JSX en el siguiente enlace:
https://transform.tools/html-to-jsx

5.3.1 COMENTARIOS

Dentro de JSX también se pueden incluir comentarios.

Se distinguen dos tipos:

- Comentarios propios de JavaScript. Los propios del lenguaje, multilínea y de una sola línea
- Comentarios en componentes React:

```
const Comentarios = () => {
    // Comentario de una sola línea
    return (
        <div>
            {/*
            * Comentario multilínea dentro de un componente
            *
            */}
            <h1>Título</h1>
            {/* Un comentario de una sola línea en JSX solo puede ir así */}
            <p>Contenido</p>
        </div>
    );
}
export default Comentarios;
```

Como puede verse, si se quiere incrustar código JavaScript en un componente JSX simplemente basta con encerrarlo entre llaves.

5.3.2 VARIABLES

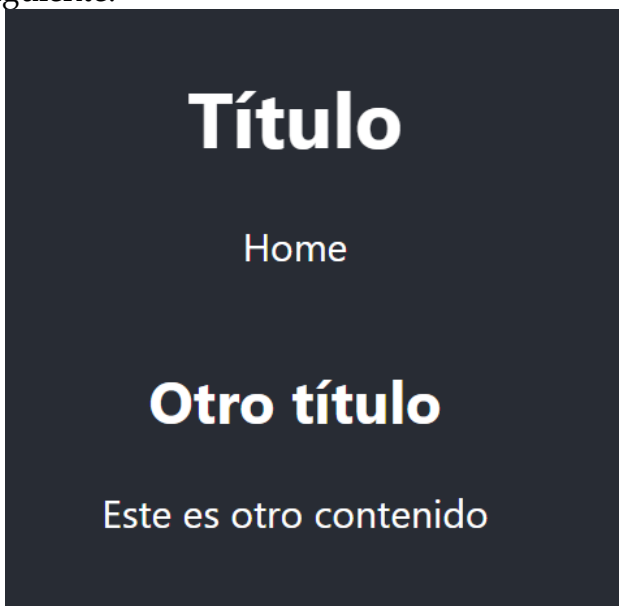
Dentro de JSX se pueden incluir variables que alteran la apariencia y comportamiento de un componente. Además de las propiedades y estado que se verán posteriormente, dentro de un componente se permite definir variables que pueden ser utilizadas a conveniencia.

```
const Variables = () => {
    const titulo = "Este es el título";
    const contenido = "Y esto el contenido";
    return (
        <>
            <h1>{titulo}</h1>
            <p>{contenido}</p>
        </>
    )
}
export default Variables;
```

Incluso se pueden definir dentro de variables lo que serían partes concretas de un componente y luego utilizar esas variables al devolver el componente mediante **return**:

```
const Variables2 = () => {
    const encabezado = (
        <header>
            <h1>Título</h1>
            <p>Home</p>
        </header>
    );
    const contenido = (
        <main>
            <h2>Otro título</h2>
            <p>Este es otro contenido</p>
        </main>
    );

       return (
        <>
            {encabezado}
            {contenido}
        </>
    )
}
export default Variables2;
```

Este código muestra lo siguiente:



12. Variables en JSX

5.4 COMPONENTES

La potencia de React, tal como se ha mencionado anteriormente, radica en la facilidad de creación de componentes. En este apartado se verá cómo crear un primer componente a partir de la aplicación de ejemplo que se crea de forma automática. Cabe recordar que en una arquitectura React, el componente principal del cual "cuelgan" los demás es **App.jsx**, es decir, se puede ver como una estructura jerárquica cuyo nodo raíz es precisamente el fichero citado. Para crear componentes, lo adecuado es crear un directorio llamado **components** (o componentes) donde ubicarlos; de esta forma, será más fácil tener localizados estos elementos. El directorio debe ir dentro de **src**.

El componente de este ejemplo, en un alarde de creatividad, tendrá como nombre **HolaMundo.jsx**. Es importantísimo tener en cuenta que el nombre del componente debe comenzar siempre con mayúscula. Dentro de este fichero hay que crear una función con el mismo nombre que el componente (**HolaMundo**), que, aunque no es obligatorio, sí es conveniente. Además, lo importante es que cumpla la misma regla que el componente, es decir, que comience por mayúscula. Y, tal como se vio anteriormente, basta con hacer un return del código HTML del componente:

```
import React from "react";
const HolaMundo = () => (
  <div>
    <h1>Hola, mundo</h1>
    <h2>Nuevo componente básico de React</h2>
  </div>
);
export default HolaMundo;
```

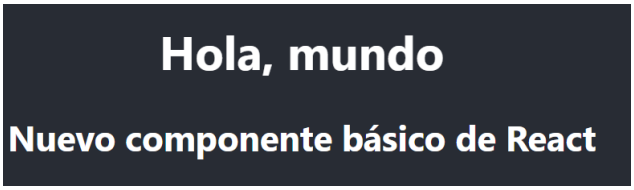
En este caso, se exporta la función con default del mismo modo que se vio en el apartado de módulos tratado en una unidad anterior. Para poder insertar el componente, primero hay que ir al principal (**App.jsx**) e importarlo:

```
import HolaMundo from "./components/HolaMundo";
```

Y la inserción del componente es muy similar a la que se lleva a cabo en los **Web Components**, es decir, colocar la etiqueta correspondiente, que en este caso es **HolaMundo**. Para probarlo, se puede colocar justo antes de la etiqueta de cierre de **App.jsx**:

```
<HolaMundo />
```

Al visualizar la página de ejemplo de React aparecerá el nuevo componente tal cual fue diseñado.



13. Componente básico en React

Ahora podría añadirse en el fichero **HolaMundo.jsx** otro componente con el código que aparece a continuación:

```
export const AdiosMundo = () => (
  <div>
    <h3>Adiós</h3>
  </div>
);
```

Como se puede ver, aquí se exporta directamente y no como **default**, ya que, como se vio en su momento, solo puede exportarse una variable/clase/función como default. Al no ser el elemento programático por defecto, la importación no se hace igual; en este caso, se rodea el nombre de la función con llaves:

```
import HolaMundo, { AdiosMundo } from "./components/HolaMundo";
```

Y ahora simplemente basta con insertar el componente del mismo modo que se hizo con el anterior: etiqueta al canto donde se quiera colocar (por ejemplo, detrás del de HolaMundo):

```
<AdiosMundo />
```

En cualquier caso, no es una buena práctica incluir más de un componente en un fichero; lo adecuado es tener un archivo por cada componente diseñado. Se propone como ejercicio crear el fichero correspondiente al nuevo control.

Por otra parte, no es obligatorio el uso de **divs** para rodear el código visto hasta ahora; React proporciona un artefacto conocido como **React.Fragment** que permite realizar esta operación. Su funcionamiento es sencillísimo, colocar una etiqueta vacía **<React.Fragment>** y otra de cierre **</React.Fragment>**. En cualquier caso, la biblioteca permite usarlos sin nada, es decir, como etiquetas vacías. Por ejemplo, el control HolaMundo quedaría así:

```
const HolaMundo = () => (
  <>
    <h1>Hola, mundo</h1>
    <h2>Nuevo componente básico de React</h2>
  </>
);
```

O en su versión extendida, tanto etiqueta de apertura como de cierre quedarían así:

```
<React.Fragment>
    <h1>Hola, mundo</h1>
    <h2>Nuevo componente básico de React</h2>
</React.Fragment>
```

BUCLES

Se puede aprovechar que los componentes se crean con JavaScript para poder automatizar la generación de sus elementos. Por ejemplo, definir un array de elementos simples, de objetos y generar varios elementos iterando sobre él.

Para ello, se puede utilizar el método **map** definiendo un array con datos y en el componente iterar generando varios elementos usando estos datos.

```
const Bucles = () => {
    const personajes = [
        {nombre: "Seong Gi-hun", edad: 52},
        {nombre: "Hwang Jun-ho", edad: 33},
        {nombre: "Hwang Front-man", edad: 54}
    ];
    return(
        <>
            <h1>Personajes</h1>
            <ul>
                {
                    personajes.map((p) =>
                        <li key={p.nombre}><b>{p.nombre}</b>: {p.edad} años</li>)
                }
            </ul>
        </>
    )
};
export default Bucles;
```

React obliga a introducir el atributo key en los elementos generados; es una forma de identificar los elementos que van cambiando y volver a renderizar aquello que haya cambiado. Hay que procurar asignar un valor específico (un identificador o el índice del elemento).

5.5 FORMATOS PARA ENVÍO Y RECEPCIÓN DE INFORMACIÓN

5.5.1 PROPS

React proporciona unos artefactos que sirven para pasar información de un control a otro; son conocidos como **props**. Estas props o propiedades son muy flexibles y pueden fluir en todos los sentidos dentro de la jerarquía de componentes, es decir, un componente que contiene a otro puede enviarle el valor de esa prop y también recibirlo. Para ilustrar el uso de props, a continuación, se llevará a cabo un ejemplo de utilización. La idea es muy simple: crear un componente que muestre un saludo personalizado. El componente se llamará **Saludar.jsx** y tendrá la siguiente codificación:

```
const Saludar = (props) => (
  <div>
    <h2>
      Hola, {props.nombre}, me he enterado de que tiene {props.edad} años
    </h2>
  </div>
);
export default Saludar;
```

La forma de pasar las **props** es muy muy simple: añadirlas como argumento de la función del componente. Ese argumento **props** no deja de ser un objeto al que se le pueden dar las propiedades que se desee. En este caso, el componente trabajará con dos propiedades, nombre y edad. Ambas se colocan en el código de saludo envolviéndolas con llaves. ¿Y cómo se le pasan? Pues como atributos propios del componente. El código a continuación ubicará dos componentes de saludo personalizados cada uno con sus propiedades correspondientes:

```
<Saludar nombre="Juan Diego" edad="55" />
<Saludar nombre="Carmen" edad="25" />
```

Si se omite alguna propiedad, el sistema no dará ningún error, pero simplemente no mostrará la información correspondiente. Este ejemplo es muy sencillo, pero tiene un problema: la asignación de atributos es completamente estática (damos una cadena concreta de nombre y otra de edad). ¿Se puede pasar el contenido de una variable? Pues sí, de una forma ya vista, es decir, envolviendo la variable entre llaves. El código siguiente ilustra cómo se pueden pasar las dos variables asignadas y declaradas en el código de **app.jsx**:

```
function App() {
  const nombreUsuario = "Juan Diego";
  const edadUsuario = 54;
  return (
        ...
        <Saludar nombre={nombreUsuario} edad={edadUsuario} />
      </header>
    </div>
);}
```

Todo esto está muy bien, pero lo normal en cualquier tipo de desarrollo es trabajar con objetos. Si **props** no deja de ser un objeto, no hay ninguna razón por la cual no pueda contener otros

objetos. Por ejemplo, supóngase que ahora se agrupan los datos anteriores en un objeto usuario y, además, se le añade un color favorito. Este código sustituiría a las líneas de asignación de variables en **app.jsx**:

```
const usuario = {
    nombre: "Juan Diego",
    edad: 55,
    color: "Verde"
}
```

En este caso, se crea un objeto usuario con las propiedades dadas y se hace en formato JSON. Pues bien, este será el objeto que se pasará al completo mediante **props** al componente debiendo este lidiar con él. La forma de hacerlo es la esperada, es decir, props.nombreobjeto.propiedad. A la hora de la propiedad al componente, se le puede dar un nombre de atributo al argumento (en este caso, **userInfo**) y ya se encargará el componente de procesarlo. En el mismo app.js, se modificaría el componente de este modo:

```
<Saludar userInfo={usuario} />
```

Por último, no queda más que introducir la lógica interna del componente para que tenga en cuenta este nuevo parámetro. Quedaría de este modo:

```
const Saludar = (props) => (
  <>
    <h2>
      Hola, {props.userInfo.nombre}, me he enterado de que tiene{" "}
      {props.userInfo.edad} años y de que su color favorito es el{" "}
      {props.userInfo.color}
    </h2>
  </>
);
```

De esta forma, se pasa el objeto y se trata en el destino como un todo, lo cual permite mucha mayor flexibilidad y estructuración del código además de reutilización.

Pero no solo se pueden pasar objetos y variables mediante **props**, también funciones. Esto proporciona una flexibilidad enorme a la hora de enviar manejadores de eventos personalizados. Por ejemplo, supóngase que el control tiene un botón al que se le quiere asociar un manejador de eventos de nombre **saludarFn**. Para asociar el evento se utilizará la forma tradicional declarativa distinta de la vista hasta ahora con **addEventListener**, es decir, incluir el atributo onClick al botón:

```
const Saludar = (props) => (
  <>
    <button onClick={props.saludarFn}>Si me pinchas ¿no sangro?</button>
  </>
);
```

Lo siguiente es definir la función, pero eso se hará en la página en la que se insertará el control (app.js en este caso) justo antes del return:

```
const saludarFn = () => {alert('Hola, mundo')}
```

A continuación, se modifica el control para pasarle entre sus **props** la función:

```
<Saludar userInfo={usuario} saludarFn={saludarFn} />
```

Pero no se vayan todavía, aún hay más. No solo se puede personalizar el manejador de eventos, sino que también puede recoger variables en su llamada. Lo primero, cambiar la forma de la función manejadora:

```
const saludarFn = (nombre) => {
    alert("Hola, " + nombre);
};
```

Y ahora, modificar el código del control para que llame al manejador personalizado. Aquí hay que tener en cuenta de que debe hacerse mediante una función anónima para que responda al evento. La solución más corta pasa por utilizar una función arrow o expresión lambda:

```
<>
    <button onClick={() => props.saludarFn(props.userInfo.nombre)}>
      Si me pinchas ¿no sangro?
    </button>
</>
```

Este ejemplo es muy interesante porque ilustra muy bien la comunicación paternofilial: el padre le envía al hijo una función que recoge un nombre y este se dedica a llamarla con un parámetro suyo propio (el **userInfo**).

Viendo en perspectiva lo codificado, hay que reconocer que es bastante potente, pero también que puede resultar pesado estar haciendo referencia constantemente a **props** desde el componente. Pues bien, la desestructuración ya vista anteriormente puede ser muy útil en estos casos. En este ejemplo se desestructura **props** y ahora basta con hacer referencia a los nombres de las propiedades, lo que hace que la sintaxis sea más clara:

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

  const { userInfo, saludarFn } = props;

  return (
    <>
      <button onClick={() => saludarFn(userInfo.nombre)}>
        Si me pinchas ¿no sangro?
      </button>
    </>
  );
}
```

Es más, se puede ir un paso más allá desestructurando y extrayendo el nombre de **userInfo**.

```
const { nombre } = userInfo;
```

quedando el botón así:

```
<button onClick={() => saludarFn(nombre)}>
```

Esta última opción es idónea para casos en los que no se proporcione una de las propiedades. Por ejemplo, supóngase que se crea un objeto sin nombre:

```
const usuario = {
    edad: 55,
    color: "Verde",
};
```

La idea es que el sistema lidie con esto y una posibilidad es añadir un valor por defecto en la desestructuración anterior:

```
const { nombre = "Anónimo" } = userInfo;
```

5.5.2 ESTADO Y HOOKS

El **state** o **estado** permite controlar y modificar el componente durante su ejecución y refleja los cambios que se hacen en el mismo. A diferencia de las **props**, el estado **sí** se puede y debe modificar para poder controlar la apariencia y comportamiento del componente.

En el momento en el que se modifica el estado, el componente se vuelve a renderizar; es decir, su contenido se vuelve a generar. En el caso de componentes de clase, el método render se invoca automáticamente. Por tanto, el estado es la palanca con la que se puede hacer que cambie un componente.

En el apartado del virtual DOM ya se mencionó el **hook useState**. Un **hook** es una función reutilizable que se puede introducir en los componentes funcionales. Todo componente ha de tener unos valores de estado definidos inicialmente. Para ello se utiliza **useState**, hook que permite generar un elemento del estado como **nombre**, un método para modificarlo (**setNombre**) y un valor por defecto ('Alicia'). Lo mismo se puede hacer con el valor de estado **datos**, se inicia con un array vacío. Todo esto se puede ver en el siguiente componente justo antes del return:

```
import { useState } from "react";
function App() {
  const [nombre, setNombre] = useState("Alicia");
  const [datos, setDatos] = useState([]);
  return (
    <>
      <header>
        <h1>{nombre}</h1>
      </header>
      <main>...</main>
    <>
  );
}
```

React cuida que se haga buen uso del estado, por eso no se puede cambiar su valor directamente y siempre hay que hacerlo mediante el método que se ha creado, en este ejemplo, **setNombre** y **setDatos**.

En el siguiente ejemplo se juega con el estado mediante un componente simple con un contador definido con userState como parte del estado del componente. Al pulsar el botón se incrementa su valor y, por tanto, del estado, con lo que App se vuelve a renderizar y se pueden apreciar los cambios.

```
import { useState } from "react";
function App() {
  const [contador, setContador] = useState(0);
  return (
    <>
      <h1>Estado simple: {contador}</h1>
      <button onClick={() => setContador(contador + 1)}>
        Cambio de estado
      </button>
    </>
  );
}
```

También se pueden agrupar valores de estado en forma de objeto y así no tienen que definirse por separado:

```
import { useState } from "react";
function App() {
  const [estado, setEstado] = useState({
    nombre: "Beetlejuice",
    datos: [],
  });
  return (
    <>
      <header>
        <h1>{estado.nombre}</h1>
      </header>
      ...
    </>
  );
}
```

Se puede modificar cualquiera de los valores de estado mediante desestructuración:

```
setEstado({
      ...estado, nombre:"Bob"
})
```

El siguiente ejemplo es más elaborado. El botón llama a un método que genera un número aleatorio. Este número modifica el estado de varias formas:

- Se añade a un array dentro del estado que luego se muestra como lista en el return
- Se modifica una variable de estado que contiene la hora
- El título del componente varía según si el número generado es par o impar
- Permite sacar un color de un array para cambiar el fondo

```
import { useState } from "react";
function App() {
  const [estado, setEstado] = useState({
    titulo: "Por defecto",
    hora: new Date().toLocaleTimeString(),
    numero: 0,
    numeros: [],
  });
  const cambiarEstado = () => {
    let numero = Math.round(Math.random() * 4);
    let numeros = estado.numeros;
    numeros.push(numero);
    setEstado({
      hora: new Date().toLocaleTimeString(),
      numeros: numeros,
      numero: numero,
      titulo: numero % 2 === 0 ? "Número par" : "Número impar",
    });
    console.log("cambiarEstado> ", estado);
  };

  const colores = ["red", "yellow", "green", "blue", "orange"];

  return (
    <div className="App" style={{ backgroundColor: colores[estado.numero] }}>
      <header>
        <h1>
          {estado.titulo} - {estado.numero}
        </h1>
      </header>
      <div className="App-intro">
        <div>{estado.hora}</div>
        Pulsa el botón para cambiar el estado
        <div>
          <button onClick={cambiarEstado}>Cambiar estado</button>
        </div>
        <div>
          Números generados:
          <ul>
            {estado.numeros.map((n) => (
              <li key={n}>{n}</li>
            ))}
          </ul>
        </div>
      </div>
    </div>
  );
}
```

¿Qué diferencia hay entre propiedades y estados? Las propiedades (**props**) se usan como parámetros iniciales de un componente y no se deben cambiar dentro de él. En cierto modo, se podría considerar que son propiedades de solo lectura de un componente. Si cambian, es porque un componente padre las ha reasignado, lo que provocará que el componente se vuelva a renderizar. Por otra parte, el estado contiene la situación de un componente en cada momento, por lo que cada vez que cambia hace que el componente vuelva a renderizarse. El estado puede cambiar por eventos que suceden dentro del propio componente, por acciones de usuario, datos recibidos, etc.

5.5.3 EJEMPLO: VISOR DE IMÁGENES

Como ejemplo sencillo, se realizará un componente completo que sirva de visor de imágenes. Se trata de un componente que recibe un array de nombres de ficheros de imágenes (que habrá que ubicar en **public/images**). El componente se llamará Visor y se sustentará en dos ficheros, **Visor.css** y **Visor.jsx**. Ambos ficheros se ubicarán en la ruta **src/components.** El fichero de estilo del componente contendrá el siguiente código:

```
.visor {
    width: 300px;
    height: 200px;
    border: 0;
}
.imagen {
    width: 100%;
    height: 100%;
    display: block;
}
.botones {
    display: flex;
    justify-content: space-between;
    width: 100%;
}
.botones button {
    flex: 1;
    margin: 5px;
    padding: 10px;
    cursor: pointer;
}
.flip-horizontal {
    transform: rotateY(180deg);
}
```

Básicamente son clases para el propio visor, la imagen contenida, la botonera, estilo de botones y una clase para girar horizontalmente una imagen que será usada en combinación con la biblioteca de iconos Font Awesome la cual servirá para los botones.

Font Awesome se puede utilizar en proyectos React de distintas formas; la más adecuada consiste en usar el paquete oficial instalando las dependencias necesarias:

```
pnpm install --save @fortawesome/fontawesome-svg-core @fortawesome/free-solid-svg-icons @fortawesome/react-fontawesome
```

El archivo Visor.js comienza importando los módulos necesarios y, en el caso de Font Awesome los distintos iconos. Después, se instancia el componente al completo:

```
import { useRef } from "react";
import './Visor.css'
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome';
import { faPlay, faForward, faBackward } from '@fortawesome/free-solid-svg-icons';
const Visor = (props) => {
    let indiceActual = 0;
    const visorRef = useRef(null);
    const imagenes = props.imagenes;
    const mostrarImagen = () => {
        if (visorRef.current) {
            const path = "/images/" + props.imagenes[indiceActual];
            visorRef.current.src = path;} }
    const siguiente = () => {
        indiceActual++;
        if (indiceActual == imagenes.length) indiceActual = 0;
        mostrarImagen();}
    const anterior = () => {
        indiceActual--;
        if (indiceActual == -1) indiceActual = imagenes.length-1;
        mostrarImagen();}
    const primera = () => {
        indiceActual = 0;
        mostrarImagen();}
    const ultima = () => {
        indiceActual = imagenes.length-1;
        mostrarImagen();}
  setTimeout(() => {mostrarImagen();}, 0);
    return (
    <div className="visor">
        <img ref={visorRef} className="imagen" /><br />
        <div className="botones">
            <button onClick={siguiente}><FontAwesomeIcon icon={faPlay} /></button>
            <button onClick={ultima}><FontAwesomeIcon icon={faForward} /></button>
            <button onClick={anterior}><FontAwesomeIcon icon={faPlay}
className="flip-horizontal" /></button>
            <button onClick={primera}><FontAwesomeIcon icon={faBackward} /></button>
        </div>
    </div>
    );
}
export default Visor;
```

Básicamente, el componente consta de los siguientes elementos:

1. Recibe unas props que en realidad simplemente van a constar del array de rutas de imágenes fuentes del visor.
2. Un índice que representa el de la imagen que se debe ver en el visor en base al array pasado mediante props. Las imágenes se almacenan en public/images
3. El propio array de imágenes obtenido de props.
4. Un método **mostrarImagen()** el cual representará la imagen dada por el índice actual. Para acceder al **VirtualDom** se usa un hook useRef que permite referenciar a un elemento mediante el atributo ref. De esta forma se puede hacer referencia al componente img que contiene la imagen y modificarla mediante código.
5. Cuatro métodos: **siguiente(), anterior(), primera() y ultima()** para navegar por las distintas imágenes en el visor y que corresponden a los manejadore de evento para sus botones correspondientes.
6. **setTimeout** contiene el código lanzado en el renderizado del componente por primera vez.
7. Se devuelve el código del control completo con elementos **img** para la imagen y cuatro **buttons**.

Bien, como ya se ha visto, ahora habría que ir al archivo **App.jsx**, importar el visor, crear un array de imágenes construido con nombres de ficheros, añadir el control y darle la propiedad imágenes. La parte principal del código quedaría así:

```
import './App.css';
import Visor from './components/Visor';
function App() {
  const imagenesSrc = new Array(
    "imagen1.png",
    "imagen2.png",
    "imagen3.png",
    "imagen4.png",
    "imagen5.jpg",
    "imagen6.jpg",
    "imagen7.webp"
  );
  return (
    <div className="App">
      <header className="App-header">

      <Visor imagenes={imagenesSrc} />

      </header>
    </div>
  );
}
export default App;
```

5.6 RENDERIZADO CONDICIONAL

Dentro de la propia función o del **return** es posible mostrar elementos de forma condicional de distintas maneras:

- return condicional. Se trata de que dentro del código JavaScript se haga un return u otro según unas condiciones. Se rige por los condicionales del lenguaje, por lo que puede usarse tanto if-else como switch-case:

```
if(condicion)
        return <Componente1 />
else
        return (
            <div>
                <Componente1 />
            </div>
    )
}
```

- if simple en el return. Dentro del return se pueden aplicar condicionales con && en caso de que se quiera que se muestre algo según una condición

```
return (
    <>
    <h1>Mostrando info</h1>
    {this.state.rows.length>0 &&
        <table>
            <tr>...</tr>
        </table>
    }
    </>
)
```

- Operador ternario en return. También se puede aplicar en este caso:

```
return(
    <>
    <h1>Datos</h1>
    {this.state.rows.length >0 ?
    (
        <table>
            <tr>...</tr>
        </table>
    ) : (
        <div>No hay datos disponibles</div>
    )}
    </>
)
```

- switch-case. JSX no tiene una forma concreta de hacer estructuras switch/case aunque lo que sí puede es aplicar esa estructura para hacer distintos return:

```
const SwitchCase = (props) => {
    switch(props.valor){
        case 0:
            return(<>...</>)
        case 1:
            return(<>...</>)
        default:
            return(<>....</>)
    }
};
```

5.7 INTEGRACIÓN EN DISTINTOS NAVEGADORES

Como ya se ha visto anteriormente, en la actualidad prácticamente todos los navegadores soportan el estándar JavaScript tratándolo de una forma muy similar. También se ha comentado que el uso de bibliotecas y frameworks ayuda al personal de desarrollo a no tener que lidiar con las distintas particularidades de cada navegador.

En el caso de React, la existencia del Virtual DOM es el mayor activo a la hora de manejar la compatibilidad e integración en distintos navegadores y/o dispositivos, ya que no debe olvidarse que el cliente final puede ser desde un ordenador personal a un teléfono móvil pasando por una Smart TV o una Tablet.

¿Pero qué pasa si hay alguna funcionalidad no implementada en un navegador antiguo? Existe un concepto en JavaScript conocido como polyfill que sirve para traducir código más moderno en una versión más antigua y así poder ser soportada por software antiguo. Una de las herramientas más conocidas para crear polyfills es Babel. Además, webs como Can I Use son capaces de informar qué versiones de navegador soportan un tipo de código JavaScript.

Otro concepto interesante en este apartado es la integración progresiva. Se trata de una filosofía de diseño que proporciona un contenido y funcionalidad esenciales a la mayor cantidad posible de usuarios, al tiempo que ofrece la mejor experiencia posible solo a los usuarios de los navegadores más modernos que pueden ejecutar todo el código requerido.

Se trata de crear un diseño que logre una experiencia más simple, pero aún utilizable, para los usuarios de navegadores y dispositivos más antiguos con capacidades limitadas, al mismo tiempo que es un diseño que hace progresar la experiencia del usuario hasta una experiencia más atractiva y con todas las funciones para los usuarios de navegadores y dispositivos más nuevos con capacidades más ricas.

La detección de características se utiliza generalmente para determinar si los navegadores pueden manejar una funcionalidad más moderna, mientras que los polyfills se utilizan para agregar funciones faltantes con JavaScript. Modernizr es una herramienta muy conocida para la detección de características.

La mejora progresiva es una técnica útil que permite a los desarrolladores web centrarse en desarrollar los mejores sitios web posibles mientras hacen que esos sitios web funcionen en múltiples agentes de usuario desconocidos.

Volviendo a React, dentro del archivo **packages.json** existe un posible apartado llamado **browserlist** que está encaminado a manejar la compatibilidad con navegadores antiguos como Internet Explorer.

En un proyecto de ejemplo, browserlist tendría un contenido como el siguiente:

```
"browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
}
```

Como puede verse, distingue entre entornos de producción y desarrollo. En este caso, en producción se indica que la aplicación puede ser usada por navegadores utilizados por más del 0.2% a nivel mundial. Por otra parte, excluye los navegadores sin soporte oficial (not dead) y también excluye todas las versiones de Opera Mini ya que este navegador tiene muchas limitaciones técnicas que afectan al renderizado y soporte limitado para JavaScript, lo que puede afectar a la ejecución de una aplicación React. En desarrollo permite hasta una versión hacia atrás de los navegadores más conocidos: Chrome, Safari y Firefox.

5.8 PRUEBA Y DOCUMENTACIÓN DE CÓDIGO

5.8.1 PRUEBAS UNITARIAS

Las pruebas unitarias son esencialmente porciones de código que comprueban que una función o método hace lo que debe. Permiten comprobar que una parte muy concreta de un software hace lo que se espera de ella. Este tipo de test da un control muy preciso del código permitiendo detectar si cualquier modificación del mismo afecta a cualquier funcionalidad. Generalmente se prueban los métodos públicos ya que los privados se pueden testear de forma implícita a través de aquellos.

En una aplicación React, se debe comprobar al menos lo siguiente:

- Que los componentes muestran determinados contenidos: etiquetas HTML, subcomponentes, atributos, etc.
- Que ante determinados eventos el comportamiento es el esperado.

Lo principal en este caso es que el componente debe probarse de forma aislada, como si fuera independiente. Además, las pruebas unitarias por naturaleza han de ejecutarse de forma rápida. Dado que los componentes a veces dependen de otros programas como APIs o métodos pasados como propiedades, ¿cómo pueden probarse de forma aislada? Esto se hace utilizando clases o métodos en lo que se conoce como **mocks** u objetos falsos.

Cuando se crea una aplicación, se puede incluir lo necesario para poder hacer un testing unitario básico. Por ejemplo, en el caso de App.jsx se puede acompañar de un App.test.jsx que está precisamente para introducir pruebas sobre el componente. El siguiente ejemplo es similar al archivo de prueba que se genera cuando se usa create-react-app (obsoleto).

```
import { render, screen } from "@testing-library/react";
import App from "./App";
import "@testing-library/jest-dom";
test('muestra el título "Vite + React"', () => {
  render(<App />);
  expect(screen.getByText("Vite + React")).toBeInTheDocument();
});
```

Antes de lanzar las pruebas, hay que instalar un par de bibliotecas:

```
pnpm add -D vitest @testing-library/react @testing-library/jest-dom jsdom
```

En vite.config.js, dentro de defineConfig, hay que añadir este elemento (tras el plugins):

```
test: {
    globals: true,
    environment: "jsdom",
},
```

Y por último, añadir el script **test** en packages.json incluyendo las siguientes entradas:

```
"test": "vitest",
"test:ui": "vitest --ui",
"test:coverage": "vitest --coverage"
```

La realización de las pruebas es tan simple como lanzar el siguiente comando:

```
pnpm test
```

Si todo sale bien, el sistema mostrará si se han pasado todas las pruebas.

Antes de entrar en detalle en otras partes del código, se puede ver que la prueba se basa en encontrar una cadena que debe estar en el documento y esta es "Vite + React". Por tanto, si se ha modificado la aplicación por defecto y se ha omitido ese texto, la prueba fallará. En caso contrario, se pasará sin mayor problema.

Aunque históricamente se han usado otras bibliotecas, en las últimas versiones de React la que se utiliza por defecto es **react-testing-library** (lo que no quita para que se pueda usar cualquier otra). Principalmente se ha hecho porque esta biblioteca aboga por obviar los detalles de implementación, lo que se traduce en que no se comprobará si dentro de un componente habrá otro contenido, sino que se harán consultas o queries sobre elementos de la aplicación. A continuación, se muestran algunas de ellas:

- …ByLabelText: Selección por label o atributo aria-label
- …ByPlaceHolder: Selección por el atributo placeholder de un input
- …ByText: Selección por texto del elemento
- …ByDisplayValue: Selección por el valor actual del elemento.
- …ByAltText: Selección por atributo alt de una imagen.
- …ByTitle: Selección por atributo title
- …ByRole: Selección por el rol del elemento, heading, link, etc.
- …ByTestId: Selección a través de un atributo id específico para test. Debería ser la última opción a utilizar.

En el código de ejemplo se utiliza **getByText**. Como se puede ver, no hay un selector basado en clases o atributos id porque la biblioteca huye de detalles de implementación y así debe ser. Los roles de los elementos pueden ser muy útiles y se pueden consultar en el siguiente enlace: https://www.w3.org/TR/html-aria/#docconformance

Igualmente, a la hora de seleccionar se pueden usar funciones como getBy, findBy, getAllBy, etc. Cada prefijo tiene un efecto distinto según lo que interese: seleccionar un elemento o varios.

Dentro de los ficheros de test se usarán cláusulas **describe** para delimitar distintas partes de la prueba o los componentes a probar. Es una forma de organizar los test que facilita su lectura y comprensión. De hecho, una de las funciones de los test es servir de documentación práctica para el proyecto.

Por ejemplo, se podría coger el ejemplo anterior, introducirle una nueva prueba y agruparlas mediante describe.

```
import { render, screen } from "@testing-library/react";
import App from "./App";
import "@testing-library/jest-dom";
import { describe } from "vitest";
import { beforeEach } from "vitest";
import { expect } from "vitest";
describe("Pruebas en el componente principal", () => {
  beforeEach(() => {
    render(<App />);
  });

  test("muestra el título Vite + React", () => {
    const linkElement = screen.getByText("Vite + React");
    expect(linkElement).toBeInTheDocument();
  });
  test("nueva prueba"),
    () => {
      const linkElement = screen.getByAltText("logo");
      expect(linkElement).toBeInTheDocument();
    };
});
```

Como puede verse, primero se utiliza una descripción para ambas pruebas. A continuación, se usa **beforeEach** que permite ejecutar un código antes de cada prueba, en este caso, el que renderiza el componente. La primera prueba queda como está, pero sin el render. La segunda busca el texto alternativo logo en la imagen. Si ahora se prueba con **pnpm test**, el sistema debería pasar ambas pruebas en la página por defecto.

5.8.2 PRUEBAS END-TO-END

El hecho de que se pasen las pruebas unitarias no garantiza que funcione la aplicación en su conjunto. Los test de cliente, también llamados end-to-end o simplemente e2e se usan para ejecutar la aplicación como lo haría un usuario y para comprobar que los resultados son los esperados.

Por tanto, una prueba e2e no mira los resultados concretos que devuelve una función, sino que carga la aplicación, hace clic en los enlaces, interactúa, etc. y además verifica que lo que hay en pantalla es lo esperado.

Las pruebas e2e son de caja negra y con ellas no es necesario importar bibliotecas de la aplicación ni hacer llamadas directas como en las unitarias. Esto permite que las pruebas se puedan realizar en cualquier lenguaje y/o tecnología, aunque generalmente se suele optar por entornos y lenguajes similares.

Para aplicaciones web existen multitud de opciones. Una de las más utilizadas para JavaScript es el framework Cypress. Se trata de un framework particularmente poderoso, versátil y, lo mejor, fácil de usar. Además de test e2e también permite hacer test de componentes, lo que es muy útil para aplicaciones React.

Instalar Cypress es muy sencillo, basta con lanzar el siguiente comando npm en el proyecto que se quiera probar.

```
pnpm add -D cypress
pnpm cypress install
```

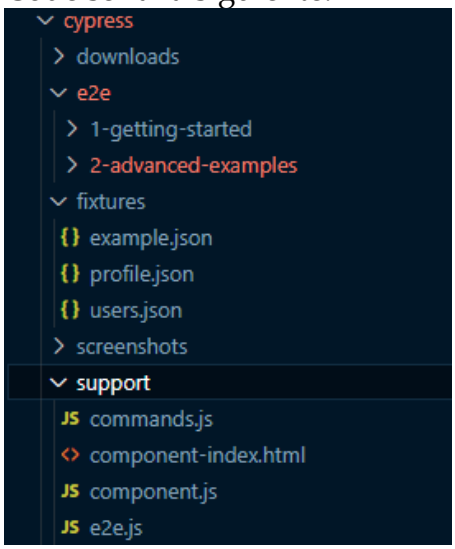
Ahora, en un terminal se lanza la aplicación React (con npm start) y en otro el siguiente código:

```
pnpm cypress open
```

Este comando hace lo siguiente:

- Crea un directorio llamado cypress en el directorio raíz del proyecto.
- Dentro del directorio cypress, genera una estructura predeterminada que incluye:
  - cypress/e2e: Para pruebas end-to-end.
  - cypress/fixtures: Para archivos de datos de prueba.
  - cypress/support: Para configuraciones y comandos personalizados.

La estructura en Visual Studio Code sería la siguiente:



14. Estructura de Cypress en proyecto React

A partir de ahí se abre una ventana para seleccionar el tipo de test (e2e o componente). A continuación, pedirá el navegador elegido. En cualquier momento se puede ejecutar todo el conjunto de test o solo uno de los ficheros. Para proyectos React, la opción más interesante es la de **componentes**, pero tanto esta como e2e deben ser configuradas.

Si se quieren ejecutar las pruebas sin interfaz gráfico hay que lanzar la siguiente orden:

```
pnpm cypress run
```

ÍNDICE DE FIGURAS

BIBLIOGRAFÍA – WEBGRAFÍA

Chaffer, J. et al. (2009) *Aprende jQuery 1.3*. Editorial Anaya.

Ornbo, G. (2013) *Node.JS.* Editorial Anaya.

Altadill Izura, P.X. (2023). *React Práctico.* Editorial Anaya

Mozilla. *Progressive Enhancement.*
https://developer.mozilla.org/en-US/docs/Glossary/Progressive_Enhancement

Manz. ¿Qué es Browserlist?
https://lenguajecss.com/herramientas-css/preprocesadores/browserslist/