Introducción a React

Contenido

Introducción a React 1
- Introducción 3
  - Características 3
  - Comparativa React, Angular y Vue 3
  - Versiones y compatibilidad multiplataforma 4
  - Conceptos clave 4
- Como comenzar 5
  - Herramientas de depuración 9
  - Extensiones recomendadas 10
  - Estructura del proyecto 11
  - Descripción “package.json” 12
  - Código generado index.html, main.jsx y App.jsx 14
- Mi primer componente 16
  - Los estados – “useState” 17
  - Código JSX 18
  - Punto de entrada – librería React-dom 19
- Intercambio de parámetros entre componentes 20
  - Intercambio de información desde el Hijo al Padre 21
- Listas 23
  - Propiedad Children 23
  - Datos como propiedad 24
- Estilos 25
  - Propiedad “style” 26
  - Hoja de estilos global 26
  - Hoja de estilos propia de cada componente 27
  - CSS Modules 27
- Hook useEffect 28
- Cargar datos con fetch 31
- Formularios 32
- Estructura de un proyecto - Enrutado 34
  - Enrutado 35

Instalación módulo enrutado – react-router-dom 35
Usando react-router-dom 36
Lógica de una aplicación 40
Estructura de un proyecto – Contextos globales 47
Optimizaciones 52
Carga diferida con React.lazy y Suspense 52
Hook React.memo 54
Bibliografía 54

Introducción

React fue desarrollado originalmente por **Jordan Walke** en 2011 mientras trabajaba para Meta (anteriormente conocida como Facebook). Inicialmente, fue creado para gestionar dinámicamente la interfaz de usuario en su plataforma de anuncios. En 2013, **React** se lanzó como un proyecto de código abierto, lo que permitió que desarrolladores de todo el mundo adoptaran y contribuyeran al crecimiento de la herramienta. Más tarde, en 2015, **React** se expandió más allá del navegador con el lanzamiento de **React Native**, una solución para crear aplicaciones móviles nativas utilizando el mismo modelo basado en componentes.

Características

- **Declarativo**: Describimos cómo debería verse la interfaz de usuario en cualquier estado de la aplicación. Cuando los datos cambian **React** actualiza automáticamente el DOM, por esto último se dice que **React** es reactivo.
- **Basado en componentes**: La interfaz se construye como un conjunto de componentes reutilizables que encapsulan su lógica y diseño.
- **Agnóstico de la plataforma**: Aunque comenzó como una herramienta para aplicaciones web a día de hoy con **React Native** se puede desarrollar aplicaciones móviles y con **React Native para Windows y macOs** para aplicaciones de escritorio.
- **Pensado para la seguridad**: Su modelo basado en componentes y el aislamiento de las capas ayudan a mejorar la seguridad frente a vulnerabilidades como ataques XSS (Cross-Site Scripting).

IMPORTANTE: Dedícale un minuto a pensar que implica que **React** sea declarativo y no imperativo.

- En un lenguaje imperativo defino instrucciones que ejecutan pasos concretos para realizar una tarea.
- En un lenguaje descriptivo defino el resultado y el motor del lenguaje se encarga de generar ese resultado, esto ya lo has usado en SQL.

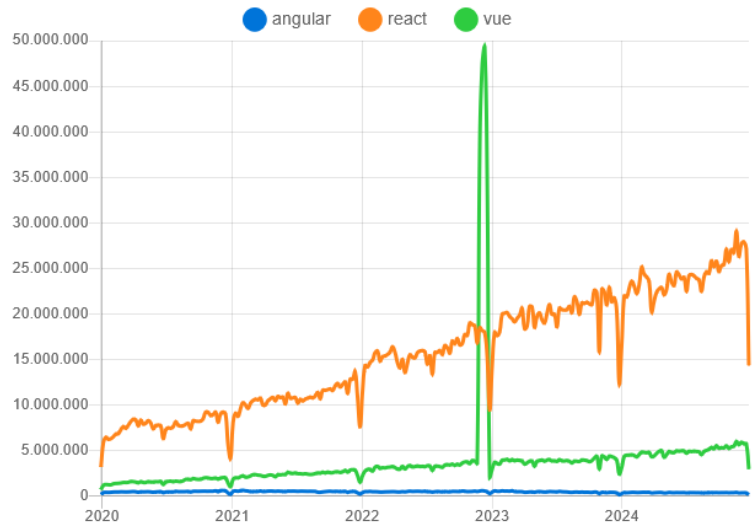
En un componente de **React** vamos a definir que queremos mostrar para un conjunto de datos de entrada, cuando alguno de estos datos de entrada se modifique el componentes se volverá a renderizar de manera íntegra. No pensamos en eventos.

Comparativa React, Angular y Vue

Aunque *React*, *Angular* y *Vue* suelen ser comparados, es importante destacar que ***React* es una librería**, mientras que *Angular* y *Vue* son **frameworks**. Esto significa que *React* se centra exclusivamente en la construcción de interfaces de usuario. Por el contrario, *Angular* y *Vue* suelen integrar estas funcionalidades de serie.

Sin embargo, con la ayuda de módulos como ***React Router*** o los contextos globales, ***React*** puede extenderse para ofrecer funcionalidades propias de un framework completo.

Puedes consultar el histórico de descargas de cada opción en *NPMTrends*
https://npmtrends.com/angular-vs-react-vs-vue



Versiones y compatibilidad multiplataforma

Es importante comprender la separación entre **React** y **React-DOM**. Mientras **React** define la estructura y lógica de los componentes de manera agnóstica a la plataforma, **React-DOM** (u otros renderizadores como React Native) se encargan de renderizar esos componentes en la plataforma específica.

Renderizadores:

- Para web -> librería "*react-dom*". Mas concretamente:
  - "***react-dom/client***"
  - "*react-dom/server*"
- Para móviles -> librería "*react-native*".
- Para Windows -> librería "*react-native-windows*".
- Para macOS -> librería "*react-native-macos*".

Conceptos clave

A continuación, se incluye una lista de conceptos que debes tener claro antes de comenzar con React.

- **Node.js**: Entorno de ejecución para JavaScript que permite ejecutar aplicaciones fuera del navegador.
- **npm y npx**: Herramientas de gestión de paquetes y ejecución de comandos para instalar y usar bibliotecas en el entorno de Node.js.
- **JSX y TSX**: Extensiones que indican el tipo de código del archivo. JSX para JavaScript y XML, y TSX para TypeScript y XML.

NOTA: el código **JSX** y **TSX** es código transpilado. Es decir, no puede ser procesador directamente por los navegadores y debe convertirse a JavaScript paso previo a su ejecución. Puedes probarlo en https://swc.rs/playground

- **Componente**: Unidad básica de construcción en *React*, con su propio estado, propiedades y ciclo de vida. Mezclan código JavaScript y código XML por ello la

extensión JSX. Se pueden definir mediante clases o mediante funciones, el código actual emplea funciones.

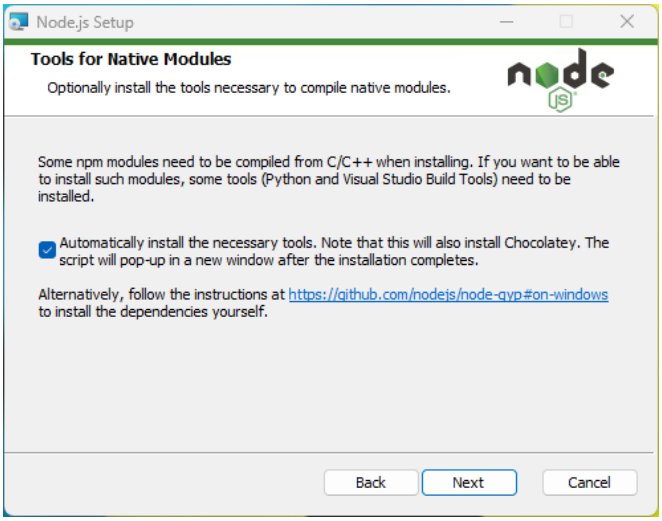
- **Estado de un componente**: Es el conjunto de datos internos que puede cambiar con el tiempo y que afecta directamente cómo se renderiza el componente.
- **Estados globales**: Métodos para gestionar estados compartidos entre componentes.
- **Hooks y customHooks**: Herramientas para gestionar el estado y los efectos en componentes funcionales. Los *hooks* más comunes son ***useState***, ***useEffect*** y ***useContext***, mientras que los ***customHooks*** son *hooks* personalizados que encapsulan lógica reutilizable.
- **Enrutado**: No forma parte de la configuración básica. Permite gestiona la navegación en la aplicación mediante librerías como *React Router* (***react-router-dom***).
- **Relación con CSS**: React permite hojas de estilo, hojas de estilo por componentes, CSS Modules y estilos en línea (envueltos en objetos JSON). Además, se pueden integrar frameworks CSS como **BootStrap**, **Taildwind CSS** o **Chakra UI** como paquetes npm. Dicho lo anterior la manera recomendada para trabajar es mediante el uso de clases CSS en la propiedad **className**.

Como comenzar

Descargamos **Nodejs**, versión LTS (Long Time Support).

https://nodejs.org/es

Durante la instalación he marcado la opción para instalar automáticamente las herramientas adicionales.



Al terminar la instalación asegúrate de que "**npm**" funciona correctamente en "cmd" y en "powershell". Es posible que en powershell aparezca un problema relacionado con los permisos de seguridad.

```
PS C:\Users\Alex> npm
npm : No se puede cargar el archivo C:\Program Files\nodejs\npm.ps1 porque la ejecución
de scripts está deshabilitada en este sistema. Para obtener más información, consulta el
tema about_Execution_Policies en https:/go.microsoft.com/fwlink/?LinkID=135170.
En línea: 1 Carácter: 1
+ npm
+ ~~~
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
PS C:\Users\Alex>
```

Podemos solucionarlo con los siguientes comandos.

- Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
- Get-ExecutionPolicy

```
PS C:\Users\Alex> Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
PS C:\Users\Alex> Get-ExecutionPolicy
RemoteSigned
PS C:\Users\Alex>
```

Creamos un nuevo proyecto de React con alguno de los siguientes comandos:

- **npm create vite@latest**
- **npm create vite@latest** <*nombre-proyecto*>

NOTA: hasta la versión 18 de React existía un paquete oficial para iniciar las aplicaciones (npx **create-react-app** <nombre-proyecto>), a partir de este momento los desarrolladores recomiendan como alternativa principal la opción de Vite.

Por ejemplo:

npm create vite@latest 01-teoria

```
PS C:\Users\Alex\OneDrive - Educantabria\curso 24-25\DWEC\React> npm create vite@latest 01-teoria
Need to install the following packages:
create-vite@6.1.1
Ok to proceed? (y)
```

En este caso al ser un entorno limpio nos va a solicitar instalar el paquete “create-vite”.

Continua el asistente solicitándonos el tipo de proyecto, elegimos “React”.

```
PS C:\Users\Alex\OneDrive - Educantabria\curso 24-25\DWEC\React> npm create vite@latest 01-teoria
Need to install the following packages:
create-vite@6.1.1
Ok to proceed? (y) y


> npx
> create-vite 01-teoria

? Select a framework: » - Use arrow-keys. Return to submit.
    Vanilla
    Vue
>   React
    Preact
    Lit
    Svelte
    Solid
    Qwik
    Angular
    Others
```

Después elegimos “JavaScript” como lenguaje por defecto de nuestro proyecto.

1. **cd 01-teoria** -> Entramos en la carpeta del proyecto creado.
2. **npm install** -> Descargamos los módulo de Nodejs asociados al proyecto.
3. **npm run dev** -> Arrancamos el servidor para poder trabajar.

```
PS C:\Users\Alex\OneDrive - Educantabria\curso 24-25\DWEC\React> npm create vite@latest 01-teoria
Need to install the following packages:
create-vite@6.1.1
Ok to proceed? (y) y


> npx
> create-vite 01-teoria

√ Select a framework: » React
? Select a variant: » - Use arrow-keys. Return to submit.
    TypeScript
    TypeScript + SWC
>   JavaScript
    JavaScript + SWC
    React Router v7 ↗
```

Y concluye el asistente indicándonos los siguientes pasos.

```
PS C:\Users\Alex\OneDrive - Educantabria\curso 24-25\DWEC\React> npm create vite@latest 01-teoria
Need to install the following packages:
create-vite@6.1.1
Ok to proceed? (y) y


> npx
> create-vite 01-teoria

√ Select a framework: » React
√ Select a variant: » JavaScript

Scaffolding project in C:\Users\Alex\OneDrive - Educantabria\curso 24-25\DWEC\React\01-teoria...

Done. Now run:

  cd 01-teoria
  npm install
  npm run dev

PS C:\Users\Alex\OneDrive - Educantabria\curso 24-25\DWEC\React>
```

Ejecutamos los comandos "**cd 01-teoria**" y "**npm install**". Fíjate como aparece una nueva carpeta "node_modules".

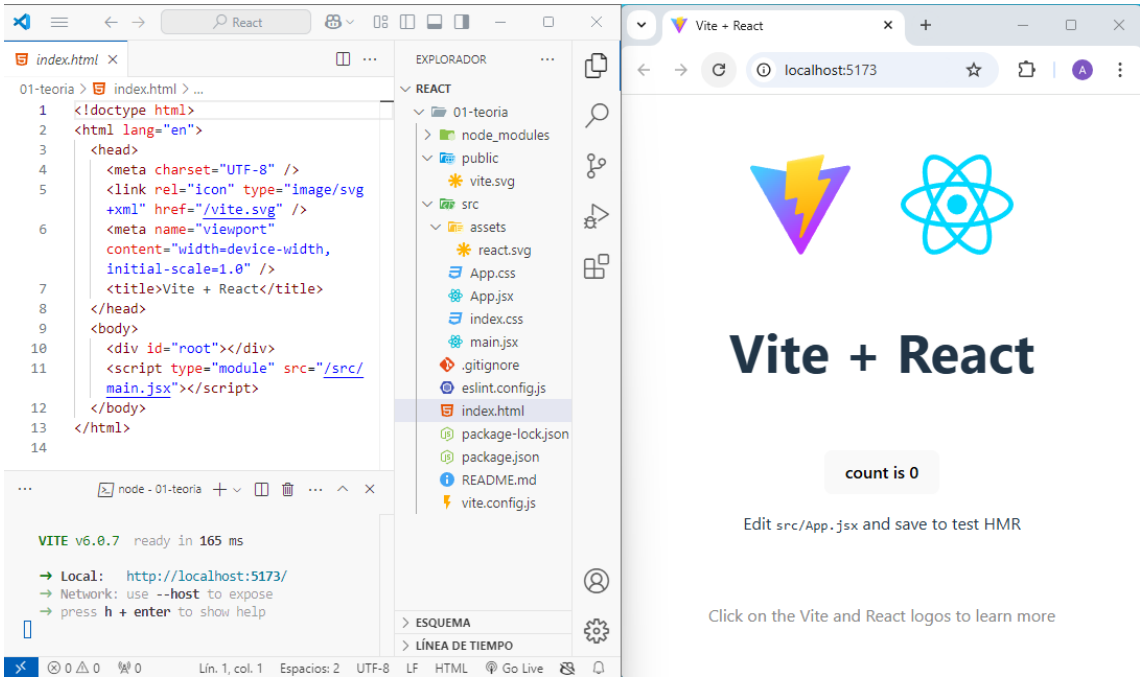
```
PS C:\Users\Alex\OneDrive - Educantabria\curso 24-25\DWEC\React\01-teoria> npm install

added 258 packages, and audited 259 packages in 9s

107 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Y arrancamos el proyecto con el comando "**npm run dev**". Ahora sólo queda abrir el enlace http://localhost:5173

```
VITE v6.0.7  ready in 165 ms

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

RECUERDA: "Ctrl + c" detiene el servidor web.

Ya tenemos nuestro entorno de desarrollo preparado.

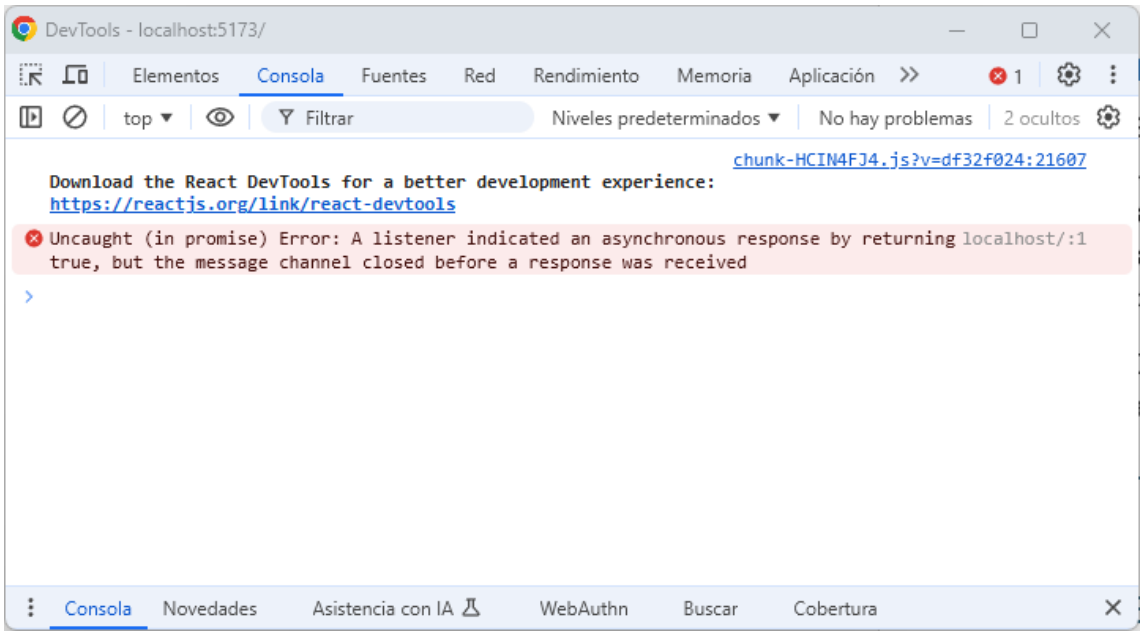


Si te fijas en el mensaje de la web "*Edit src/App.jsx and save to test HMR*" nos habla de una característica que incluye React llamada "***Hot Module Replacement***" o "***HMR***" la cual se encarga de llevar automáticamente cualquier cambio en el código al navegador, esto se conoce habitualmente como "***hot reload***" y es común en muchas tecnologías de desarrollo.

Otra ventaja de la recarga en caliente de "***react***" es que cuando cometamos un error automáticamente la web va a mostrar información del error.

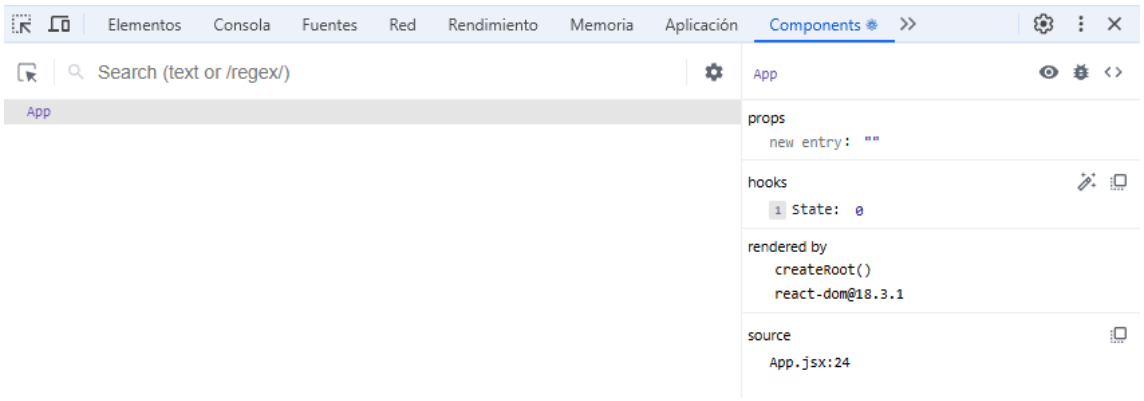
Herramientas de depuración

La primera vez que abras las herramientas de desarrollador del navegador (F12) vas a encontrarte con un mensaje informativo sugiriéndote la instalación de la extensión "***React DevTools***". Esta extensión no es obligatoria, pero si la instalas te facilitara la depuración de componentes de React en el navegador.



Una vez instalada se añaden a las herramientas de desarrollador las opciones: "***Components***" y "***Profiler***".

Desde "***Components***" podemos ver los distintos componentes renderizados en cada momentos, sus "***props***", sus "***hooks***", la pila de renderizado y su posición en el código.



Extensiones recomendadas

La documentación oficial de React recomienda dos extensiones para VS Code.

ESLint

Es el linter de Microsoft, se encarga de buscar errores comunes y malas prácticas de codificación y de autocorregirlos.

https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint

Prettier – Code formatter

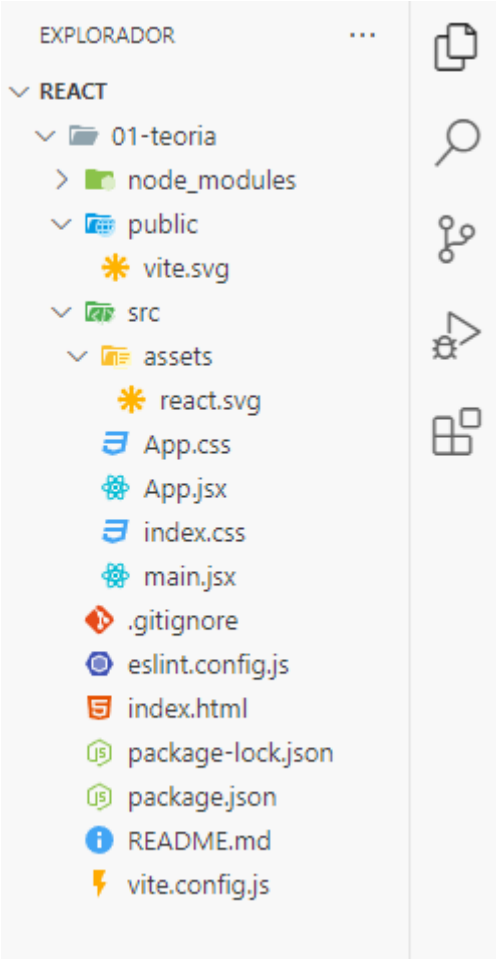
Aplica reglas de estilo pre establecidas, indentación, espaciado, comillas... Salvo que se indique lo contrario aplica automáticamente al guardar el fichero.

https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode

NOTA: la primera vez que emplees el formateador de código te solicita que elijas la opción por defecto o Prettier.

Estructura del proyecto

Vamos a dedicarle un minuto a la estructura del proyecto con el fin de localizar que carpetas y ficheros nos van a ser útiles y cuales, en principio, no.



Elementos que inicialmente no debemos tocar.

- Carpeta "***node_modules***": contiene todas las librerías descargadas por Nodejs.
- Fichero **"*.gitignore*"**: contiene la lista de elementos a excluir de nuestro repositorio git.
- Fichero "***eslint.config.js***": contiene parámetros de configuración del linter de React.
    - Considera añadir las siguientes reglas: quitar advertencias para variables y funciones declaradas pero no empleadas, y quitar advertencias por no definir los tipos de las propiedades.

```
rules: {
  ...
```

```
            'no-unused-vars': 'off',
            'react/prop-types': 'off',
        },
```

- Fichero "***package-lock.json***": contiene metadatos de los paquetes referenciados por el proyecto.
- Fichero "***README.md***": contiene información del proyecto en formato markdown.
- Fichero "***vite.config.js***": inicializa la configuración de Vite en el proyecto.

Elementos que si debemos conocer.

- Carpeta "***public***": contiene todos los elementos estáticos de nuestra aplicación que no van a ser gestionados por React.
- Carpeta "***src/assets***": parecida a "*public*" pero muy distinta. Contiene todos los elementos estáticas de nuestra aplicación, pero gestionados por React. Cuando un elemento es gestionado por React se incluye en el bundle o paquete a desplegar, es este caso se minifica o se descartan elementos no referenciados automáticamente. TODOS los elementos estáticos que se referencien desde un componente deberían ir aquí.
- Fichero "***package.json***": contiene la configuración del proyecto.
- Fichero "***index.html***": página HTML de entrada de la aplicación, es única ya que react monta páginas de tipo SPA.
- Carpeta "***src***": lugar en el que vamos a trabajar habitualmente, contendrá el código de nuestras aplicación y su estructura variará según como organicemos nuestro proyecto.
    - Fichero "***main.jsx***": Componente de entrada asociado a index.html, carga los estilos globales (index.css). Aquí normalmente se define el "*routing*" y los contextos globales ya que es el nodo raíz de la aplicación del que cuelgan todos.
    - Fichero "***index.css***": Estilos globales.
    - Fichero "**App.jsx**": Primer componentes de la aplicación, lo más seguro es que lo remplaces por uno nuevo.
    - Fichero "***App.css***": Hoja de estilos propia del componente "*App.jsx*".

NOTA: hemos dicho que la filosofía de React es montar aplicaciones de tipo SPA, pero nada nos impide en proyectos existentes anclar componentes a cualquier etiqueta contenedor y usar React como necesitemos.

Descripción "package.json"

Conocer la estructura de este fichero es importante porque depende tiene elementos importantes para nuestro proyecto y su estructura puede variar en función de las herramientas que empleemos.

La siguiente imagen muestra la estructura de un proyecto recién creado con Vite.

```json
{
  "name": "01-teoria",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  ▷ Depurar
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint .",
    "preview": "vite preview"
  },
  "dependencies": {
    "react": "^18.3.1",
    "react-dom": "^18.3.1"
  },
  "devDependencies": {
    "@eslint/js": "^9.17.0",
    "@types/react": "^18.3.18",
    "@types/react-dom": "^18.3.5",
    "@vitejs/plugin-react": "^4.3.4",
    "eslint": "^9.17.0",
    "eslint-plugin-react": "^7.37.2",
    "eslint-plugin-react-hooks": "^5.0.0",
    "eslint-plugin-react-refresh": "^0.4.16",
    "globals": "^15.14.0",
    "vite": "^6.0.5"
  }
}
```

Contamos con 4 zonas diferenciadas dentro del documento.

- Zona con la descripción de la aplicación: “name”, “version”.
- Zona “**scripts**”: contiene los comandos que podemos ejecutar con npm. Por ejemplo, “npm run **dev**” ejecuta en este caso “vite”.

En este apartado pueden aparecer más entradas, por ejemplo “test” si incluimos pruebas a nuestro código. Por ahora debes saber que:

  - “**dev**”: arranca el proyecto en modo desarrollo.
  - “**build**”: empaqueta el proyecto para producción en la carpeta “**/dist**”

NOTA: Las entradas de este apartado pueden variar según la herramienta que utilicemos para crear el proyecto por lo que debes comprenderlo.

- Zona “**dependencies**”: contiene las librerías externas y sus números de versión a emplear en producción.
- Zona “**devDependencies**”: contiene las librerías externas y sus números de versión para cuando estemos en desarrollo.

Código generado index.html, main.jsx y App.jsx

Vamos a analizar el flujo de la aplicación desde el punto de entrada hasta el componente que vemos.

Index.html > main.jsx > App.jsx

Index.html

Debe incluir un contenedor en el que inyectar el nodo raíz, en este caso el contenedor está identificado con **id="root"**. Seguido aparece el script que selecciona el contenedor y renderiza el nodo raíz.

```
<!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>Vite + React</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

Main.jsx

Este fichero es un envoltorio y contiene muchos elementos importantes.

"***react-dom/client***" es el motor de renderizado de React para la web, su finalidad es arrancar el motor web y renderizar el primer nodo con la lógica de la aplicación (App.jsx).

Dado que estamos trabajando con web podemos emplear hojas de estilo CSS externas (index.css).

El componente "***StrictMode***" es un envoltorio que en tiempo de desarrollo envuelve el código React y nos facilita la depuración en caso de error mostrando mensajes detallados. Se puede borrar o cambiar por cualquier otra etiqueta.

```
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>,
)
```

NOTA: recuerda que React emplea una estructura de árbol y que todo lo que definimos en un nodo padre puede ser visto por los nodos hijo. Por este motivo este es el lugar ideal para incluir funcionalidades globales:

- Componente encargado del enrutado de la página.
- Contextos globales.
- Configuración de frameworks CSS como BootStrap, Tailwind o Chakra UI

App.jsx

Este componente es un "*hola mundo*" con los elementos más significativos de React. Vamos a describirlo:

Definimos un componente App mediante la sintaxis de función. Cuenta con dos partes diferenciadas: el bloque de código JavaScript y el bloque JSX en el return.

Define un estado "*count*" para almacenar el número de veces que se hace click en el botón. Y la actualiza con la función definida para ello "*setCount*".

En la parte en la que devolvemos maquetado JSX, aparece la etiqueta "**Fragment**" (**<></>**) envolviendo múltiples nodos en el mismo nivel. **YA QUE SÓLO PODEMOS DEVOLVER UN NODO EN EL RETURN** (el método React.render recibe un único ReactNode como parámetro).

Todos los bloques de código dentro de JSX se resuelven con **{}**.

La definición de estilos se realiza con clases CSS a través de la propiedad "**className**", piensa ¿por qué no usa "*class*"?.

Los manejadores de eventos se definen en línea, pero emplean notación camelCase, es decir en JSX tenemos "**onClick**" y en JavaScript tenemos "*onclick*".

```
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
```

Mi primer componente

Vamos a realizar un "hola mundo" a partir de un proyecto creado con Vite. Nuestra primera tarea va a ser eliminar las referencias a las hojas de estilos:

- En "*main.jsx*" elimina la línea "*import './index.css'*".
- En "*App.jsx*" elimina la línea "*import './App.css'*".

Seguido creamos la carpeta "***src\components***". Esta carpeta es especial, es el lugar en el que por convenio vamos a crear nuestros componentes jsx.

Creamos "*HolaMundo.jsx*" y le añadimos el siguiente código, no te preocupes que lo comentamos.

```
import React, { useState } from "react";

//Los componentes con UpperCamalCasel (o PascalCase)
function HolaMundo() {
  const [mensaje, setMensaje] = useState("Hola Mundo");
  const [mostrar, setMostrar] = useState(true);

  //Las funciones con lowerCamelCase
  const handlerClick = () => {
    setMostrar(!mostrar);
  };

  return (
    <div>
      {/* Comentario en JSX, fíjate como usamos {} para inyectar código
*/}
      <h1>{mostrar ? mensaje : "¿No me dices nada?"}</h1>
      <button onClick={handlerClick}>
        {mostrar ? "Ocultar mensaje" : "Mostrar mensaje"}
```

```
      </button>
    </div>
  );
}

export default HolaMundo;
```

Una vez que lo guardamos podemos emplearlo en "*main.jsx*" de la siguiente manera.

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import HolaMundo from "./components/HolaMundo.jsx";

createRoot(document.getElementById("root")).render(
  <StrictMode>
    {/* <App /> */}
    <HolaMundo />
  </StrictMode>
);
```

Fíjate en la estructura del componente:

- Zona de "imports"
- Definición del componente mediante "function"
    - Zona de "Hooks", en este caso "useState" pero pueden ser más.
    - Código JavaScript auxiliar. Manejadores de eventos, validaciones, negocio…
    - "return" código JSX.
- Zona de "exports"

Los estados – "useState"

El estado es uno de los elementos principales de React. Un estado permite almacenar un dato del componente gestionando además sus modificaciones. Ten en cuenta que un componente puede tener tantos estados como necesite.

A efectos prácticos debes ver los estados como variables de tipo "***getter/setter***".

Por ejemplo.

```
const [mostrar, setMostrar] = useState(true);
```

- **"mostrar"** es la variable con la información del estado, en este caso "*true*".
- **"setMostrar"** es la función que modifica el estado. Por convenio se usa "*set*" seguido del nombre del estado.
- **"useState(true)** es la función que crea e inicializa el estado. Recibe como parámetro el valor inicial del estado.

El estado puede contener cualquier tipo de dato, tipos básico, null, e incluso objetos JSON.

Que tienes que saber, si o si, de los estados.

- La modificación de un estado si emplear la función específica de actualización produce un error.
- Al modificar un estado se renderiza el componente automáticamente.
- La función de actualización del estado es asíncrona, cuidado con leer el valor del estado y actualizarlo dentro del mismo bloque de código porque tendrás el valor previo.

Código JSX

El código JSX es una mezcla de HTML con JavaScript y tiene ciertas peculiaridades debido a esta mezcla.

```
  return (
    <div>
      {/* Comentario en JSX, fíjate como usamos {} para inyectar código
*/}
      <h1>{mostrar ? mensaje : "¿No me dices nada?"}</h1>
      <button onClick={handlerClick}>
        {mostrar ? "Ocultar mensaje" : "Mostrar mensaje"}
      </button>
    </div>
  );
```

Que tienes que saber, si o si, del código jsx.

- Los "***return***" devuelven el código entre paréntesis "**()**". Si es una única línea se puede omitir.
- Inyectamos código envolviéndolo entre llaves "**{}**".
- Definimos el código condicional con el **operador ternario "*?*"** o **mediante "*truthy*"/"falsy" más operador &&**. Por ejemplo: **"{mostrar ? "Ocultar mensaje" : "Mostrar mensaje"}"**.
    - El **if-else** tradicional puedes emplearlo fuera del return almacenando el código jsx en una variable, esta variable se puede posteriormente emplear en el return envolviéndola entre llaves { miCodigoJSX }.
- Los componentes se definen con PascalCase o UpperCamelCase. Por ejemplo: **"function HolaMundo() {"**
- Los eventos se definen con lowerCamelCase. Por ejemplo: "**<button onClick={handlerClick}>"**
- No escribimos HTML aunque se parezca, en JSX. Las propiedades de las etiquetas HTML se escriben con su nombre JavaScript. Por ejemplo:
    - Propiedad HTML "***class***" en JSX "***className***".
    - Propiedad HTML (label) "***for***" en JSX "***htmlFor***".

- Los comentarios dentro de código JSX (no son comunes) se definen mediante **{/* Cuerpo del comentario */}**.

NOTA: En React el valor 0 es "***truthy***", tenlo en cuanta para cuando muestres listados vacíos.

Punto de entrada – librería React-dom

El archivo "***main.jsx***" se encarga de enganchar la aplicación React con el contenedor HTML. En este caso "***index.html***" contiene un div con el id "***root***", a partir de este punto "***React-dom***" renderiza toda la aplicación y la incrusta en el contenedor HTML. El contenido que pudiera tener el contenedor se pierde.

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import HolaMundo from "./components/HolaMundo.jsx";

createRoot(document.getElementById("root")).render(
  <StrictMode>
    {/* <App /> */}
    <HolaMundo />
  </StrictMode>
);
```

La etiqueta `<StrictMode>` nos ayuda a depurar durante el tiempo de desarrollo, envuelve la aplicación y se encarga de mostrar mensajes de depuración de una manera más comprensible por el desarrollador. Puedes eliminarla y la aplicación funciona igual.

NOTA: con `<StrictMode>` la aplicación se renderiza dos veces por lo que si muestras trazas por consola las veras duplicadas.

La etiqueta `<HolaMundo />` contiene el primer nodo de la aplicación, normalmente será "App".

IMPORTANTE: "render" sólo renderiza un nodo, es decir, no puedes varios nodos a la misma altura ya que produce un error, por ejemplo: *".render(Primer bloque Segundo bloque)"*. **De lo anterior se deduce que los return de nuestros componentes sólo pueden devolver un único nodo**. Si necesitamos devolver varios elementos en un mismo nivel habrá que envolverlos dentro de un contenedor.

Para terminar unos conceptos básicos de react-dom. **React-dom es una librería especifica para el entorno de ejecución web**. Si quisiéramos desplegar nuestra aplicación en un dispositivo móvil, en Windows o en Mac deberíamos cambiarla por una librería equivalente pero propia de cada plataforma, el resto de la aplicación podría ser mismo. En esencia es el encargo de crear el nodo raíz y renderizar/visualizar el árbol de nuestra aplicación dentro de un nodo DOM.

- const root = createRoot(**domNode**, options?): Crea el nodo raíz, admite opcionalmente callbacks para la gestión de errores.
- root.render(**reactNode**): Muestra el código JSX dentro del nodo raíz.

Intercambio de parámetros entre componentes

Antes de comenzar debemos saber que el parámetro que recibe un componente en React de denomina **"*props*"**. "***props***" es un objeto JSON que incluye todas las propiedades que se le pasan al componente cuando se le usa.

NOTA: Los nombres "***props***" y "***children***", que lo verás en los siguientes apartados, son parámetros de la función de React "***createElement(type, props, ...children)***".

IMPORTANTE: "***props***" es inmutable, esto significa que si lo cambias en ejecución se produce un error.

Vamos probarlo definiendo dos componentes, el componente "***Padre***" incluye un componente "***Hijo***" que recibe un "*mensaje*".

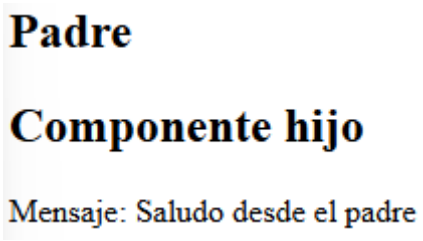
Código del componente "***Padre***".

```
import Hijo from "./Hijo";

function Padre() {
  return (
    <>
      <h1>Padre</h1>
      <Hijo mensaje="Saludo desde el padre" />
    </>
  );
}

export default Padre;
```

Código del componente "***Hijo***".

```
function Hijo(props) {
  return (
    <>
      <h2>Componente hijo</h2>
      <p>Mensaje: {props.mensaje}</p>
    </>
  );
}

export default Hijo;
```

Que genera la siguiente salida.



En código anterior funciona perfectamente pero plantea un problema, a React le duele no conocer la estructura de datos de "***props***" y se quejara constantemente. Que alternativas tenemos:

- Usar **TypeScript** que nos permite definir un tipo compuesto para las variables.
- Importar la librería de react "***prop-types***" y definir el tipo esperado.
- Cambiar "***props***" por un JSON con una estructura por defecto.

Lo correcto sería usar "***prop-types***", dicho esto nosotros vamos a reemplazar props por un objeto JSON con propiedades.

Con el nuevo formato de objeto JSON el componente "Hijo" queda así. Fíjate que le incluimos un valor por defecto (opcional).

```
function Hijo({ mensaje = "Mi padre no me dice nada" }) {
  return (
    <>
      <h2>Componente hijo</h2>
      <p>Mensaje: {mensaje}</p>
    </>
  );
}

export default Hijo;
```

Intercambio de información desde el Hijo al Padre

Como sabes, JavaScript nos permite almacenar una función en una variable. Este es el principio de uso de los "***callBacks***". El truco para que un componente hijo pueda comunicarse con el componente padre es que "***props***" defina una función de retorno.

Vamos a probarlo modificando los componentes "***Padre***" e "***Hijo***" del ejemplo anterior.

Código del componente "Padre".

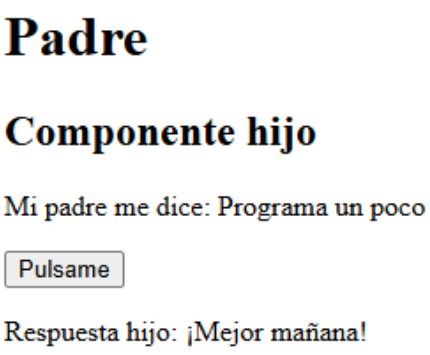
```
import { useState } from "react";
import Hijo from "./Hijo";

function Padre() {
  const [respuesta, setRespuesta] = useState("Esperando contestación");

  const responderAPadre = (contestacion) => {
    setRespuesta(contestacion);
```

```
  };

  return (
    <>
      <h1>Padre</h1>
      <Hijo mensaje="Programa un poco" comunicarPadre={responderAPadre}
/>

      <p>Respuesta hijo: {respuesta}</p>
    </>
  );
}

export default Padre;
```

Código del componente “Hijo”.

```
function Hijo({ mensaje, comunicarPadre }) {
  const handleClick = () => {
    comunicarPadre("¡Mejor mañana!");
  };

  return (
    <>
      <h2>Componente hijo</h2>
      <p>Mi padre me dice: {mensaje}</p>
      <button onClick={handleClick}>Pulsame</button>
    </>
  );
}

export default Hijo;
```

Que genera la siguiente salida.



Explicación

En el componente “Padre”.

- Se define un estado “***respuesta***” y las correspondiente función para actualizarlo “***setEstado***”, hacemos esto porque queremos que cuando el componente “hijo” nos diga algo se actualice el componente.
- Definimos la función “***responderAPadre***” con el objetivo de ser la función de callBack que le pasemos al componente “hijo”. En este caso su código es muy sencillo, recibe el “***mensaje***” del componente “***hijo***” y actualiza el estado llamando a “***setEstado***”.

En el componente “Hijo”.

- En “***props***” definimos un parámetro “***comunicarPadre***” para enganchar la función de callBack del componente “padre”, en este caso la hemos llamado “***responderPadre***”.
- Definimos una función manejador “***handleClick***” y la asociamos al evento “***click***” del botón. Esta función invoca el callback “***comunicarPadre***” pasándole el mensaje que queramos.

Listas

Tenemos dos maneras de pasarle contenido a una lista, y las dos se emplean. Vamos a ver las diferencias y cuando emplear cada una de ellas.

Propiedad Children

El componente recibe maquetado, componentes React o etiquetas HTML, y lo renderiza como nodos hijo directamente.

En este caso el maquetado del contenedor “padre” incluye entre las etiquetas de apertura y de cierre del componente los nodos hijos. Para que esto funcione debemos definir la propiedad “***children***” como propiedad del componente.

IMPORTANTE: La propiedad debe llamarse “***children***”, esta propiedad es una palabra reservada de React y este es su único cometido.

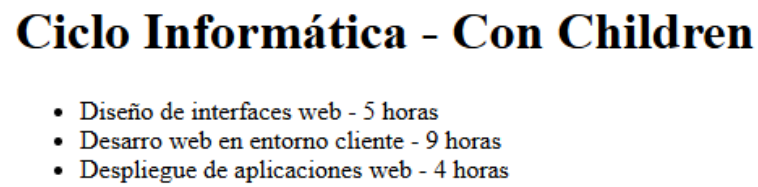
Por ejemplo, una lista no ordenada.

```
return (
<StrictMode>
<ListaChildren ciclo="Informática">
  <li>Diseño de interfaces web - 5 horas</li>
  <li>Desarrollo web en entorno cliente - 9 horas</li>
  <li>Despliegue de aplicaciones web - 4 horas</li>
</ListaChildren>
</StrictMode>
);
```

Y la definición del componente.

```
function ListaChildren({ ciclo, children }) {
  return (
    <div>
      <h1>Ciclo {ciclo} - Con Children</h1>
      <ul>{children}</ul>
    </div>
  );
}
```

Que genera la siguiente salida.



Datos como propiedad

El componente recibe una colección de datos mediante un array JSON u otro tipo, y dinámicamente construye los nodos hijos.

Por ejemplo, una lista no ordenada cuyos datos provienen de una array de objetos JSON.

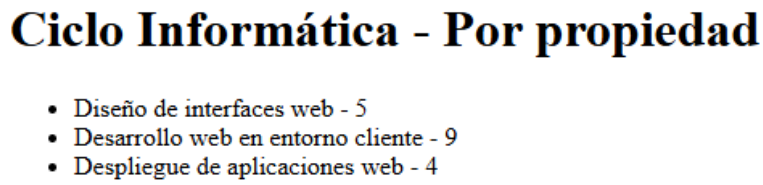
```
const datos = [
  { nombre: "Diseño de interfaces web", horas: 5 },
  { nombre: "Desarrollo web en entorno cliente", horas: 9 },
  { nombre: "Despliegue de aplicaciones web", horas: 4 },
];
...

return (
<StrictMode>
  <ListaArray ciclo="Informática" modulos={datos}></ListaArray>
</StrictMode>
);
```

Y la definición del componente.

```
function ListaArray({ ciclo, modulos }) {
  const componenteId = useId();
  return (
    <div>
      <h1>Ciclo {ciclo} - Por propiedad</h1>
      <ul>
        {modulos.map((modulo, indice) => {
          return (
            <li key={`${componenteId}-${indice}`}>
              {`${modulo.nombre} - ${modulo.horas}`}
```

```
            </li>
          );
        })}
      </ul>
    </div>
  );
}
```

Que genera la siguiente salida.



Explicación

A partir del array "*módulos*" devolvemos el JSX del componente que incluye un bucle para iterar cada módulo. Cosas que debes recordar:

- Tenemos dos "***return***", el del contenedor y el de cada elemento hijo.
- Empleamos "***.map***" y no "***.forEach***" porque el primero devuelve una copia del array y el segundo devuelve "***null***" con lo que no se renderiza nada.
- Debemos añadir la propiedad "***key***" con un valor único evitar problemas en ejecución.
- El método "***useId()***" genera un identificador único para el componente, su formato es una letra más un número envuelto entre "*:*". Por ejemplo "***:r1:***".

La propiedad "***key***" es de uso interno de React y lo emplea para identificar de manera univoca a cada componente renderizado. Estarás pensando en emplear la propiedad "***índice***" como identificador, el problema surge cuando renderizamos una página que tenga 2 identificadores idénticos. En este caso si se modifica el dato asociado al componente React no sabrá que componente debe redibujar produciéndose errores de lógica. Criterios con la "***key***":

- Si el objeto trae su propio identificador único úsalo (entidad+id).
- Si tienes que generarlo emplea "***useId***" a nivel de componente padre concatenando el índice del "***map***".
- NUNCA uses "***useId***" dentro del bucle, según la documentación oficial le duele.

Estilos

En este apartado hay muchas opciones. Vamos a verlas por encima y a tratar de aclarar cuando debes emplearlas.

IMPORTANTE: Se recomienda aplicar estilos mediante clases, hay casos de uso en React que no soportan los selectores CSS.

No vamos a tratar el uso de frameworks CSS como BootStrap, Tailwind CSS o Chakra UI. Para el que quiera emplearlos es tan sencillo como:

1. Añadir el módulo al proyecto (*npm i bootstrap@5.3.3*).
2. Importar en el "*index.jsx*" el fichero la hoja de estilos.
3. Comenzar a maquetar según las guías de estilo.

Propiedad "style"

Permite asignar estilos en línea en el propio código JSX. Fíjate que empleamos llaves dobles "***{{}}***", la primera para indicar que va código y la segunda para indicar que los estilos son un objeto JSON.

No debemos emplear las propiedades individuales de estilo directamente, porque posiblemente no los reconozca.

NOTA: esta es la manera de trabajar con "***React Native***" porque no soporta las hojas CSS, hay que envolverlas en ficheros js.

```
function EstiloPropiedad() {
  return (
    <>
      <h2>Estilos en línea - propiedad style</h2>
      <p style={{ color: "blue", fontWeight: "bold" }}>Esto es una
prueba</p>
    </>
  );
}
```

Que genera la siguiente salida.

Estilos en línea - propiedad style

Esto es una prueba

Hoja de estilos global

Como ya sabemos, el fichero de entrada a la aplicación "***main.jsx***" importa la hoja de estilos "***index.css***". Esta hoja de estilos contiene los estilos globales de mi aplicación web y la ven todos los componentes por colgar del nodo raíz.

Lo comentado anteriormente, si empleamos un framework CSS habría que eliminarla y añadir en "*main.jsx*" las referencias al estilo del nuevo framework.

Hoja de estilos propia de cada componente

Podemos definir una hoja de estilos para cada componente de manera individual. Normalmente se llama igual que el componente y se incluye junto a este en la misma carpeta.

Esto es CSS estándar y puede haber conflictos con los estilos generales.

Por ejemplo.

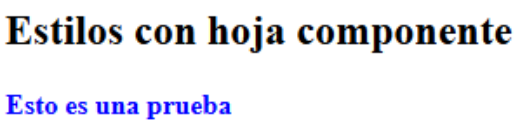
Hoja de estilos “EstiloHojaComponente.css”

```
.parrafo {
  color: blue;
  font-weight: bold;
}
```

Código del componente, importamos y asignamos clase.

```
import "./EstiloHojaComponente.css";
...
function EstiloHojaComponente() {
  return (
    <>
      <h2>Estilos con hoja componente</h2>
      <p className="parrafo">Esto es una prueba</p>
    </>
  );
}
```

Que genera la siguiente salida.



CSS Modules

Permite definir estilos a nivel de componente evitando los conflictos entre nombre de clases. Es decir, genera nombres únicos para cada clase CSS de forma automática.

Hoja de estilos

```
.parrafo {
  color: blue;
  font-weight: bold;
}

.parrafo:hover {
  background-color: yellow;
```

```
}
```

Código

```
import styles from "./05EstiloCSSModule.module.css";
...

function EstiloCSSModule() {
  return (
    <>
      <h2>Estilos con CSS Module</h2>
      <p className={styles.parrafo}>Esto es una prueba</p>
    </>
  );
}
```

Que genera la siguiente salida.

Estilos con CSS Module

Esto es una prueba

En caso fíjate como en el código HTML generado se han renombrado las clases de manera aleatoria evitando los conflictos con la hoja de estilos principal o con los estilos de otros componentes (_parrafo_8o9ew_3).

```
<h2>Estilos con CSS Module</h2>
<p class="_parrafo_8o9ew_3">Esto es una prueba</p> == $0
```

Hook useEffect

Este "***hook***" gestiona el ciclo de vida del componente y junto con el estado "***useState***" son los "*hooks*" principales de React. Su uso principal es el de cargar los datos en el componente mediante una petición a una fuente de datos, por ejemplo, mediante "***fetch***".

Los eventos del ciclo de vida que podemos gestionar son:

- Se ha montado el componente.
- Se ha actualizado el componente.
- Se va a desmontar el componente.

Y que uso le vamos a dar a cada momento:

- Al montar => carga de datos, definición de temporizadores.
- Al actualizar => recargar datos que puedan haber sido modificados.
- Al desmontar => detener temporizadores.

Veamos la estructura de "***useEffect***" y como gestionar cada estado.

```
useEffect(() => { /* código */ }, [/* dependencias */]);
```

Tenemos dos parámetros

- **Función de configuración**: Contiene el código de configuración o de limpieza del componente.
- **Dependencias (opcional)**: Lista de todos los valores reactivos de los que depende la función configuración. Admite estados, "*props*" o variables internas del componente.

Cosas que hay que saber.

- Cuando se indican las dependencias como array vacío "***[]***" sólo se ejecuta cuando se monta el componente.
- Cuando se indican las dependencias con un array con variables "***[estado, prop, var1, etc...]***", se ejecuta cada vez que alguna de ellas cambia de valor.
- Cuando no se especifican las dependencias (parámetro es null). Se vuelve a ejecutar después de cada renderizado.
- Cuando definimos un "***return***" dentro de la función de configuración se ejecuta cuando se va a desmontar el componente.
- No es obligatorio pero se puede combinar el montaje y desmontaje en la misma definición del useEffect.

Por ejemplo.

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

function ContenedorUseEffect() {
  const [activado, setActivado] = useState(false);
  const handleClick = () => {
    console.log("\n");
    setActivado(!activado);
  };

  return (
    <>
      <button onClick={handleClick}>{activado ? "Descargar" :
"Cargar"}</button>
      <p>{activado ? "Componente cargado" : "Componente no cargado"}</p>
      {activado && <Contador />}
    </>
  );
}

function Contador() {
  const [contador, setContador] = useState(0);

  //log de entrada
  useEffect(() => {
    console.log("Componente montado");
  }, []);
```

```
  //log de cambio
  useEffect(() => {
    console.log("Componente actualizado [contador] - " + contador);
  }, [contador]);
  //los de renderizado
  useEffect(() => {
    console.log("Componente actualizado - Renderizado");
  });

  //log de salida
  //NOTA: Quitar StrictMode, renderiza dos veces
  useEffect(() => {
    return () => {
      console.log("Desmontando componente");
    };
  }, []);

  const handleClick = () => {
    setContador(contador + 1);
  };

  return (
    <>
      <p>Contador: {contador}</p>
      <button onClick={handleClick}>Pulsame</button>
    </>
  );
}
```

Que genera la siguiente salida por consola.

```
Componente montado
Componente actualizado [contador] - 0
Componente actualizado - Renderizado
Componente actualizado [contador] - 1
Componente actualizado - Renderizado
Componente actualizado [contador] - 2
Componente actualizado - Renderizado
Componente actualizado [contador] - 3
Componente actualizado - Renderizado

Desmontando componente
```

Cargar datos con fetch

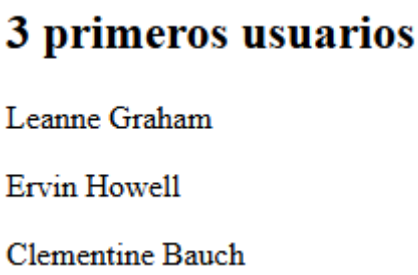
Este apartado es un ejemplo en el que ponemos en práctica lo visto anterior, el objetivo es tener un patrón básico de como pedir información a una API y generar un listado a partir de los datos. En este ejemplo vamos a cargar los 3 primeros usuarios de *jsonplaceholder*.

- En primer lugar, definimos un estado "***usuarios***", su valor inicial es array vacío. Cuando se actualicen los usuarios se volverá a renderizar la lista.
- En segundo lugar, definimos "***useEffect***" para al cargar el componente llame una única vez al método encargado de realizar la llamada a la API, en este caso "***getUsuarios***".
- En tercer lugar, definimos el método la función "***getUsuario***" que llama a la API con "***fetch***". Recuerda que este método es asíncrono por lo que podemos hacer que "***getUsuario***" también lo sea. Envolvemos el código en un try catch por si hubiera algún error.
- En el momento en el que hay datos la lista se rellena automáticamente, en este caso podemos rellenar la propiedad "***key***" con el id del usuario.

Por ejemplo.

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

function ListaFetch() {
  const [usuarios, setUsuarios] = useState([]);

  useEffect(() => {
    getUsuarios();
  }, []);

  async function getUsuarios() {
    try {
      const respuesta = await fetch(
        "https://jsonplaceholder.typicode.com/users?_limit=3"
      );
      const datos = await respuesta.json();
      setUsuarios(datos);
    } catch (e) {
      console.log(e);
    }
  }

  return (
    <>
      <h2>3 primeros usuarios</h2>
      {usuarios.map((usuario) => {
        return <p key={usuario.id}>{usuario.name}</p>;
      })}
    </>
  );
}
```

Que genera la siguiente salida.



Formularios

Para desarrollar un formulario en React debemos tener en cuenta que ejecutamos una aplicación SPA, que implica que vamos a tener gestionar el evento "***submit***" por código.

En nuestro ejemplo vamos a emplear la API de pruebas de *JSONPlaceHolder* teniendo en cuenta que cada vez que se le realiza una petición POST nos devuelve un objeto con los datos enviado, esto nos permite asegurarnos de que todo ha funcionado correctamente.

Debes fijarte en:

- El método "***handleSubmit***" recibe el evento para:
  - Detener el envío con "***e.preventDefault()***".
  - Acceder a los datos del envío mediante "***e.target***".
- El método "***enviarDatos***" es asíncrono porque lo es "***fetch***", y envuelve el código en un bloque "***try-catch***". Se podría emplear esta estructura para personalizar los estados de "enviando" y "error".
- El código JSX emplea en el "***form***" la propiedad "***onSubmit***" y los label la propiedad "***htmlFor***".

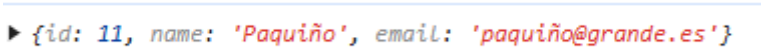
NOTA: React tiene un hook "***useRef()***" que permite referenciar una variable sin renderizar el componente, en este caso referencias a las etiquetas input. Empleo el formato tradicional "***e.target***" por preferencia personal (también creo que es más claro).

Por ejemplo.

```
function Formulario() {
  //https://jsonplaceholder.typicode.com/users/1/

  const handleSubmit = (e) => {
    e.preventDefault();
    const formData = new FormData(e.target);
    //Se puedo mandar el FormData directamente
    const datosJSON = {
      id: formData.get("id"),
      name: formData.get("name"),
      email: formData.get("email"),
    };
    //Validar
```

```
    enviarDatos(datosJSON);
  };

  async function enviarDatos(objUsuario) {
    try {
      const peticion = await fetch(
        "https://jsonplaceholder.typicode.com/users",
        {
          method: "POST",
          body: JSON.stringify(objUsuario),
          headers: {
            "Content-type": "application/json; charset=UTF-8",
          },
        }
      );

      const respuesta = await peticion.json();
      console.log(respuesta);
      //Redirigir con react "Navigate"
    } catch (e) {
      console.log(e);
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      <input type="hidden" name="id" id="id" />
      <label htmlFor="name">Nombre:</label>
      <input type="text" name="name" id="name"></input>
      <br />
      <label htmlFor="email">Email:</label>
      <input type="text" name="email" id="email"></input>
      <br />
      <button type="submit">Enviar</button>
    </form>
  );
}

export default Formulario;
```

Que genera la siguiente salida.



NOTA: No se ha desarrollado, pero puede darse el caso en el que sea conveniente introducir los datos del formulario en una variable de estado, por ejemplo, validaciones en

tiempo real o creación de campos dinámicos. Más adelante veremos un ejemplo completo que incluya un objeto estado para los datos.

Estructura de un proyecto - Enrutado

En el siguiente bloque vamos a ver como estructurar una aplicación típica de gestión, para ello vamos a necesitar las funciones de enrutado y navegación de React.

NOTA: algunos de los puntos que se trabajan en este apartado quedan incompletos hasta que veamos los estados globales.

Vamos a crear la estructura de proyecto.

Carpeta "***components***"

- AppMenu.jsx -> menú de navegación.
- AppLayout.jsx -> plantilla base para las páginas del sitio. Incluye el menú las páginas.
- ListaLinea.jsx -> plantilla para los elementos de la página "ListaPage".

Carpeta "***pages***"

- HomePage.jsx -> página inicial del sitio.
- ListaPage.jsx -> formulario de mantenimiento.
- DetallesPage.jsx -> formulario de propiedades parametrizado.
- AdministracionPage.jsx -> requiere permisos de acceso.
- NoPage.jsx -> página por defecto para rutas erróneas.

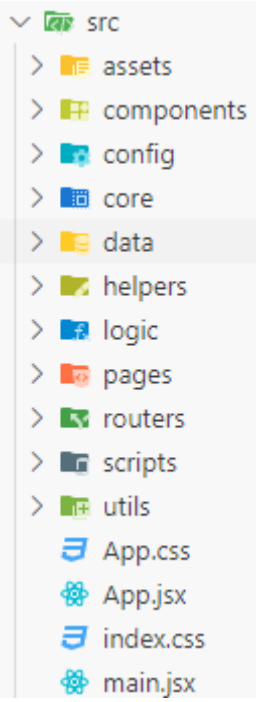
Carpeta "***routers***"

- AppEnrutador.jsx -> componente gestor de las rutas.

Carpeta "***core***"

- Negocio.js (probar js) -> auxiliar JavaScript con datos simulados.

Los nombres elegidos para las carpetas no son aleatorios, representan usos frecuentes al definir una arquitectura para el proyecto. Iremos describiendo cada una de las carpetas según sean necesarias. Fíjate en la siguiente imagen, incluye nombres que pueden aparecer en un proyecto, su uso esperado se remarca con un icono único.



Enrutado

El enrutado en una aplicación Web se encarga de redirigir las peticiones URL del navegador a las páginas encargadas de su procesamiento. Ten en cuenta que React es un SPA que va a permitir el uso de distintas rutas.

Recordar que una URL se compone en esencia de: dominio, recurso y parámetros. Por ejemplo:

www.midominio.com/recurso-clientes?parametro-nombre=pepe

Como sabemos por defecto React es una librería de frontend y no un framework. El enrutado es una funcionalidad común de frameworks y al ser una libreria React no lo incluye por defecto. Para React esto no es un problema porque a través de módulos se puede ampliar su funcionalidad.

Instalación módulo enrutado – react-router-dom

Lo primero instalar en nuestro proyecto el módulo de enrutado "***react-router-dom***". Nos posicionamos en la raíz de nuestro proyecto y ejecutamos el comando:

npm i react-router-dom

```
PS C:\Users\Alex\OneDrive - Educantabria\curso 24-25\DWEC\React\02-teoria-rutas> npm i react-router-dom

added 6 packages, and audited 265 packages in 3s

107 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Usando react-router-dom

Antes de comenzar a codificar vamos a resumir componentes y funciones que añade este módulo.

- ***BrowserRouter***: Proporciona el contexto necesario para habilitar la navegación y las rutas en una aplicación React.
- ***Routes***: Envuelve las diferentes rutas de la aplicación, actuando como un contenedor para los componentes "***Route***".
- ***Route***: Define una ruta específica que se renderiza cuando la URL coincide con su "*path*".
- ***Link***: Genera enlaces que permiten la navegación sin recargar la página.
- ***Outlet***: Representa el lugar donde se renderizan los componentes hijos de una ruta anidada.
- ***Navigate***: Redirige programáticamente al usuario a otra ruta.
- ***useNavigate***: Hook que proporciona una función para realizar redirecciones dentro del código.
- ***useParams***: Hook que devuelve un objeto con los parámetros dinámicos extraídos de la URL.

Comenzamos

1. Estructura de páginas

La carpeta "***pages***" contiene las páginas o vistas principales de la aplicación, una página se puede componer de varios componentes. Estos subcomponentes continúan definiéndose en la carpeta "***components***".

Para poder definir el enrutado debemos crear previamente las páginas, por ello definimos todas las páginas a enrutar en la aplicación con un contenido mínimo, a modo de ejemplo la página de inicio (consulta el índice y crea las 5 páginas).

```
function HomePage(){
    return(<>
        <h1>Página de inicio</h1>
    </>)
}

export default HomePage;
```

2. Definición del Layout de la aplicación

Como ya deberías saber, el "***layout***" (disposición) base es el diseño principal de una aplicación, es decir, todas las páginas o vistas de la aplicación se van a componer siguiendo la estructura definida en el layout.

NOTA: En aplicaciones complejas podemos definir múltiples layouts y emplearlos de manera condicional.

Una página web normalmente contiene una zona superior con un menú de navegación (*nav*) seguido de un contenedor (*main*) con el contenido especifico de cada página. Esta va a ser la estructura de nuestro "*layout*".

En primer lugar, creamos un componente para el menú de navegación y lo llamamos "***AppMenu.jsx***".

```
import { Link } from "react-router-dom";
import "./AppMenu.css";

function AppMenu(){
    return(<nav className="navegacion">
        <ul>
            <li><Link to="/">Inicio</Link></li>
            <li><Link to="/lista">Lista</Link></li>
            <li><Link to="/administracion">Administración</Link></li>
        </ul>
    </nav>);
}

export default AppMenu;
```

El componente "***Link***" define un enlace a una ruta interna de nuestra aplicación, la ruta se indica con la propiedad "***to***".

En este caso necesitamos un poco de CSS para que aparezca horizontal. Añadimos "***AppMenu.css***" con el código. Remarcar que maquetamos con clases.

```
.navegacion {
  ul {
    list-style: none;
    padding: 0;
    margin: 0;
  }

  ul li {
    display: inline;
    padding-right: 10px;
  }
}
```

Seguido creamos el componente con el diseño de la página y lo llamamos "***AppLayout.jsx***".

```
import { Outlet } from "react-router-dom";
import AppMenu from "./AppMenu";

function AppLayout(){
    return(<>
```

```
        <AppMenu />
        <main>
            <Outlet />
        </main>
    </>);
}

export default AppLayout;
```

Dedícale un minuto a analizar el código. A partir de un "*fragmento*" incluimos el componentes con la cabecera de navegación "*AppMenu*", y seguido dentro de una etiqueta "*main*" insertamos el componente de react "***Outlet***".

IMPORTANTE: En "***Outlet***" se van a inyectar las páginas gestionadas por el enrutador.

3. Definimos el enrutado de nuestra página

Llegado este momento disponemos de todos los elementos estructurales básicos de nuestra aplicación, ahora podemos definir las reglas de enrutado que nos permitan navegar entre las distintas páginas de la aplicación.

Creamos un nuevo componente que únicamente define las reglas de enrutado en nuestra aplicación. Lo llamamos "***AppEnrutador.jsx***".

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import AppLayout from "../components/AppLayout";
import HomePage from "../pages/HomePage";
import ListaPage from "../pages/ListaPage";
import DetallesPage from "../pages/DetallesPage";
import AdministracionPage from "../pages/AdministracionPage";
import NoPage from "../pages/NoPage";

function AppEnrutador() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<AppLayout />}>
          <Route index element={<HomePage />} />
          <Route path="lista" element={<ListaPage />} />
          <Route path="detalles/:id" element={<DetallesPage />} />
          <Route path="administracion" element={<AdministracionPage />}
/>
          <Route path="*" element={<NoPage />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}
```

```
export default AppEnrutador;
```

Explicación.

- **BrowserRouter**: gestiona las rutas en React, tiene que estar.
- **Routes**: es un contenedor con las definición de las distintas rutas.
- **Route**: define una ruta. Veamos sus propiedades.
    - **path**: URL que activa la ruta.
    - **element**: componente a renderizar.
    - **index**: define la ruta como ruta por defecto.

En la definición de rutas hemos definido un ruta raíz que apunta al componente con el layout de la aplicación, recuerda que en layout se emplea "***Outlet***" como punto de renderización de las páginas.

A partir de la ruta del layout definimos subrutas con las distintas páginas de nuestra aplicación.

La ruta con el ***path="*"*** es la ruta por defecto para cuando tengamos una entrada que no sepamos gestionar o redirigir.

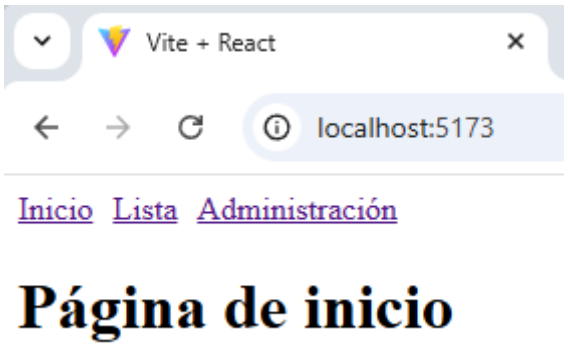
4. Activamos el enrutador en nuestra aplicación.

Como sabemos, React emplea una estructura lógica con forma de árbol por lo que debemos definir el enrutado en el nodo más alto posible "***main.jsx***".

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

createRoot(document.getElementById("root")).render(
  <StrictMode>
    {/*     <App /> */}
    <AppEnrutador />
  </StrictMode>
);
```

Fíjate que desaparece "***App***" y sólo definimos el "***AppEnrutador***", a partir de este momento el flujo de la aplicación varía según la ruta.

La página debería tener el siguiente aspecto.



Lógica de una aplicación

React es una librería de frontend y necesita algo por detrás para gestionar la información, no vamos a hablar de backend en este manual. Dicho lo anterior nos hace falta algo para unir el front con el back y poder explicar las operaciones básicas CRUD. Para ello vamos a crear un archivo JS que simule las peticiones a una API.

La carpeta "***core***" está pensada para incluir la lógica de la aplicación, por ello vamos a crear un archivo "***Negocio.js***", fíjate que es javacript, el cual va a envolver en un objeto las funciones de negocio que necesite mi aplicación.

En este caso vamos a mínimos y vamos a trabajar con la entidad "*Modulo*" compuesto por: id, nombre y horas. Las operaciones disponibles son:

- Obtener módulos, para poder crear una vista de tabla.
- Obtener módulo, para poder crear un formulario y trabajar con parámetros.
- Actualizar módulo, para poder modificar un dato de una tabla.

Todas las funciones de prueba son asíncronas, como lo son las llamadas a una API vía "***fetch***".

El código es el siguiente.

```
const negocio = (function () {
  console.log("Cargando negocio");
  const modulos = [
    { id: 1, nombre: "Diseño de interfaces web", horas: 5 },
    { id: 2, nombre: "Desarrollo web en entorno cliente", horas: 9 },
    { id: 3, nombre: "Despliegue de aplicaciones web", horas: 6 },
  ];

  async function obtenerModulos() {
    return modulos;
  }

  async function obtenerModulo(id) {
    for (let modulo of modulos) {
      // == para no tener que parsear
      if (modulo.id == id) {
```

```
        return modulo;
      }
    }
    return null;
  }

  async function actualizarModulo(modulo) {
    if ("id" in modulo) {
      let moduloBD = await obtenerModulo(modulo.id);
      if (moduloBD) {
        moduloBD.nombre = modulo.nombre;
        moduloBD.horas = modulo.horas;
      }
    }
    return;
  }

  return {
    obtenerModulos,
    obtenerModulo,
    actualizarModulo,
  };
})();

export default negocio;
```

Para poder utilizarlo únicamente debemos importarlo en los componentes que lo necesitemos. React lo cachea y sólo lo va a cargar una vez (fíjate en el console.log).

1. Página de mantenimiento - ListaPage.jsx

Una página de mantenimiento contine una colección de datos, normalmente una tabla o lista con los datos.

En este ejemplo vamos a una lista de módulos y cada modulo se mostrará en una ficha con su propio componente "***ListaLinea***".

```
import { useEffect, useState } from "react";
import negocio from "../core/Negocio.js";
import ListaLinea from "../components/ListaLinea.jsx";

function ListaPage() {
  const [modulos, setModulos] = useState([]);

  useEffect(() => {
    getModulos();
  }, []);
```

```
  const getModulos = async () => {
    try {
      const respuesta = await negocio.obtenerModulos();
      setModulos(respuesta);
    } catch (e) {
      console.log(e);
    }
  };

  return (
    <>
      <h1>Lista de módulos</h1>
      {modulos.map((cadaModulo) => {
        return <ListaLinea key={cadaModulo.id} modulo={cadaModulo} />;
      })}
    </>
  );

}

export default ListaPage;
```

Explicación.

Guardamos los datos de la lista en un estado "*modulos*", inicialmente el array está vacío "*[]*" para evitar que la llamada "*.map*" produzca un error.

Usamos "***useEffect***" para llamar una única vez a la función encargada de recuperar los datos de los módulos y de actualizar el estado de la variable "*modulos*".

Una vez actualizado el estado "*modulos*" el componente se renderiza, invocando para cada elemento al componente "***ListaLinea***". En este caso pasamos un objeto JSON como propiedad (también podríamos trocearlo).

2. Página de mantenimiento – subcomponente ListaLinea.jsx

Este componente genera un párrafo con la información modulo y un botón para navegar hasta el formulario de detalles (o de edición).

La función "***useNavigate***" devuelve otra función que nos permite navegar entre las distintas rutas de la aplicación.

Debes tener clara la diferencia con el componente "***Navigate***", los dos producen el mismo resultado, navegar entre rutas. La diferencia está en cuando debes usarlos:

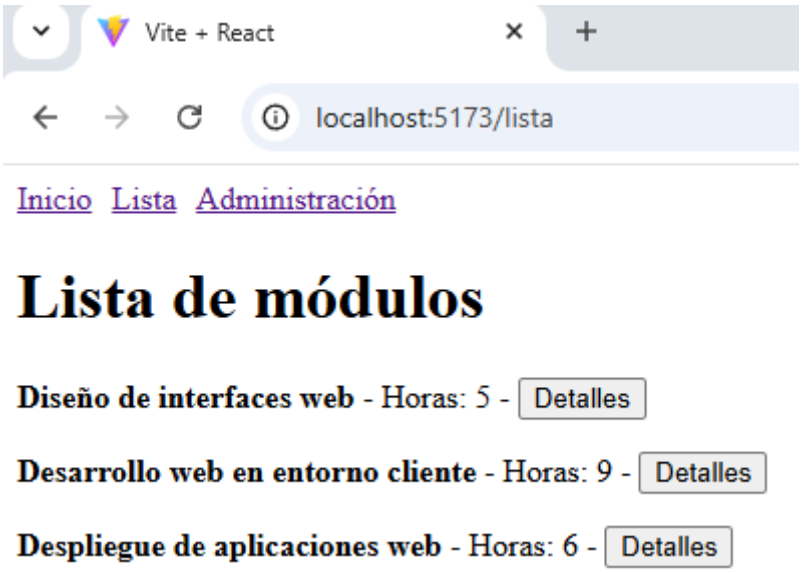
- **useNavigate**: en JavaScript, cuando gestiones eventos.
- **Navigate**: al renderizar el componente dentro de código jsx.

Ten en cuenta los casos de uso ya que puedes originar un buble infinito al renderizar el componente.

NOTA: Fíjate en como componemos la ruta añadiéndole el identificador del módulo.

```
import { useNavigate } from "react-router-dom";

function ListaLinea({ modulo }) {
  const { id = 0, nombre = "", horas = 0 } = modulo;
  const navegar = useNavigate();

  const handleClick = () => {
    navegar(`/detalles/${id}`);
  };

  return (
    <p>
      <strong>{nombre}</strong> - Horas: {horas} -{" "}
      <button onClick={handleClick}>Detalles</button>
    </p>
  );
}

export default ListaLinea;
```

La página debería tener el siguiente aspecto.



3. Formulario de propiedades – DetallesPage.jsx

En el apartado previo sobre formularios vimos un ejemplo muy básico en el que no se gestionaba el estado de los datos del formulario ni la navegación tras el evento *submit*.

En este caso aparecen varios conceptos nuevos, veámoslos uno a uno.

- Leemos los parámetros con la función "***useParams()***" de React Router.

- Definimos un estado "*modulo*" inicialmente sin valor y lo cargamos a través de "***useEffect***", en este caso asociado al valor de del identificador pasado por parámetro **[id]**.
- En el código JSX asignamos el valor a los distintos inputs a través de la propiedad "*value*". Además, definimos el evento "***onChange***" para que se actualice el estado "*modulo*". Te recuerdo que los estados sólo se pueden modificar a través el método set asociado. Si no empleas "***onChange***" no te dejara cambiar el valor del input.
- Al hacer el submit cancelamos el evento con "***e.preventDefault()***" y enviamos los datos de manera asíncrona. En este caso el estado "modulo" tiene la última versión de los datos por lo que no es necesario extraerlos del formulario. Una vez completada la petición volvemos al listado con "***useNavigate()***".
- El código JSX comprueba que estado "***modulo***" este definido antes de renderizar el formulario mediante un "*truthy*" (***{modulo && (<form>...}***)
- Fíjate como se actualiza un estado de tipo JSON, debemos crear un nuevo objeto cambiando la propiedad modificada. "***let actualizado = { ...modulo, [name]: value };***"

El código de la página.

```
import { useNavigate, useParams } from "react-router-dom";
import negocio from "../core/Negocio.js";
import { useEffect, useState } from "react";

function DetallesPage() {
  const { id } = useParams();
  const [modulo, setModulo] = useState();
  const navegar = useNavigate();

  useEffect(() => {
    cargarDatos(id);
  }, [id]);

  const cargarDatos = async (id) => {
    try {
      let respuesta = await negocio.obtenerModulo(id);
      setModulo(respuesta);
    } catch (e) {
      console.log(e);
    }
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    /* const formData = new FormData(e.target); */
    enviarDatos();
  };

  const enviarDatos = async () => {
```

```
    try {
      await negocio.actualizarModulo(modulo);
      navegar("/lista");
    } catch (e) {
      console.log(e);
    }
  };

  const handleChange = (e) => {
    /* Target es de tipo input, el nombre y el valor están disponibles */
    const { name, value } = e.target;
    let actualizado = { ...modulo, [name]: value };
    setModulo(actualizado);
  };

  return (
    <>
      <h1>Página de detalles</h1>
      {modulo && (
        <form onSubmit={handleSubmit}>
          <input type="hidden" name="id" id="id" value={modulo.id} />
          <label htmlFor="nombre">Nombre:</label>
          <input
            type="text"
            name="nombre"
            id="nombre"
            value={modulo.nombre}
            onChange={handleChange}
          />
          <br />
          <label htmlFor="horas">Horas:</label>
          <input
            type="number"
            name="horas"
            id="horas"
            value={modulo.horas}
            onChange={handleChange}
          />
          <br />
          <button type="submit">Enviar</button>
        </form>
      )}
    </>
  );
}

export default DetallesPage;
```

Que genera la siguiente salida.

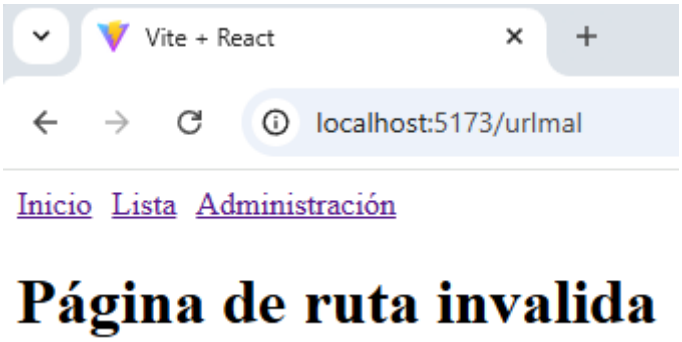
4. Gestión de rutas invalidad – NoPage.jsx

Esta página es sencilla, su contenido suele ser estático y muestra mensaje indicando que el recurso solicitado no existe (error 404).

El código de nuestro ejemplo.

```
function NoPage(){
    return(<>
        <h1>Página de ruta invalida</h1>
    </>)
}

export default NoPage;
```

Que genera la siguiente salida.



5. Gestión de la seguridad – AdministracionPage.jsx

Un elemento importante en una aplicación es poder controlar los permisos de acceso evitando que se muestre información que no se debe ver.

A continuación, vamos a plantear una posible solución para la gestión de la autorización. El componente define un estado, en este caso "***tienePermisos***", el valor del estado debe recuperarse de la sesión, del negocio, o de un estado global. Si el usuario no dispone de permisos de acceso le redirigimos a la página de inicio, en este caso renderizamos el componente "***Navigate***" asignándole la propiedad "***to='/'***".

El código de ejemplo mostrado no incluye la consulta de información del usuario porque necesitamos conocer previamente la definición de estados globales.

```
import { useState } from "react";
import { Navigate, useNavigate } from "react-router-dom";
//import negocio from "../core/Negocio.js";

function AdministracionPage() {
  const [tienePermisos, setTienePermisos] = useState(true);
  const navegar = useNavigate();

  if (!tienePermisos) {
    return <Navigate to="/" />;
  }

  const handleClick = () => {
    if (confirm("Ir a inicio")) {
      navegar("/");
    }
  };

  return (
    <>
      <h1>Página de administración</h1>
      <button onClick={handleClick}>Volver a inicio</button>
    </>
  );
}

export default AdministracionPage;
```

Estructura de un proyecto – Contextos globales

Hemos visto que podemos intercambiar información entre componentes de padre a hijo mediante propiedades, y en dirección contraria de hijo a padre mediante call backs. Si quisiéramos tener una información concreta disponible entre todos los componentes de la aplicación esta manera de intercambio sería muy poco eficiente y difícil de gestionar.

Existe una alternativa denominada contextos globales que nos permite definir uno o varios contextos visibles entre todos los nodos hijos. Si los definimos en el nodo raíz estarán disponibles en toda la aplicación.

Estos estados globales son ideales para:

- Almacenar los datos de autenticación del estado.
- Gestionar preferencias, por ejemplo, el color del tema, elementos por página….
- Guardar objetos de negocio.
- Almacenar datos de la sesión, por ejemplo, carritos de la compra.

Para poder trabajar con contextos vamos a necesitar 3 conceptos.

- nombreContexto**=createContext()**: Crea un contexto.
- nombreContexto**.Provider**: Define un componente que proporciona el valor del contexto a los componentes hijos.
- **useContext(**nombreContexto**)**: Es un hook que permite a un componente acceder al valor del contexto.

Veamos como definir un contexto global para gestionar la autenticación y autorización de nuestra aplicación.

1. Definición del contexto global.

Creamos una nueva carpeta "***contexts***" y un nuevo componente llamado "***SeguridadProvider.jsx***". Este componente va a incluir dos elementos, el contexto y el proveedor asociado, los definimos en un único fichero porque están relacionados.

Definimos un nuevo contexto "***SeguridadContext***" como constante.

Creamos "***SeguridadProvider***", este componente define la funcionalidad del contexto. Tiene la siguiente estructura:

- Al menos debe recibir como parámetro "***children***". Recuerda que los nodos hijos son los que pueden acceder al contexto.
- El estado del contexto. En este caso "*datos*" con la configuración de seguridad por defecto.
- Las funciones que necesitemos compartir. En este caso "*logIn*" y "*logOut*".
- Código JSX con la definición del "***Provider***" específico del contexto. Que debe cumplir:
    - Define un objeto "***value***" con los elementos públicos del contexto. En este caso: datos (usuario y tienePermisos), y los métodos logIn y logOut.
    - Incluir como elemento hijo del "***provider***" los elementos "***children***".

```
import { useState, createContext } from "react";
```

```
/* Creamos el contexto */
const SeguridadContext = createContext();

function SeguridadProvider({ children }) {
  const [datos, setDatos] = useState({ usuario: "", tienePermisos: false
});

  const logIn = async (nombre) => {
    /* Validar contra un servicio real */
    let nuevoDatos = { ...datos, usuario: nombre, tienePermisos: true };
    setDatos(nuevoDatos);
  };

  const logOut = async () => {
    let nuevoDatos = { ...datos, usuario: "", tienePermisos: false };
    setDatos(nuevoDatos);
  };

  /* Por convenio nombre.Provider */
  return (
    <SeguridadContext.Provider
      value={{
        datos,
        logIn,
        logOut,
      }}
    >
      {children}
    </SeguridadContext.Provider>
  );
}

export { SeguridadContext, SeguridadProvider };
```

2. Añadir opciones de registro para el contexto de seguridad - HomePage.jsx

Una vez definido el contexto de seguridad vamos a necesitar una página para poder autenticarnos. Lo correcto sería crear un nuevo formulario, nosotros en este ejemplo vamos a incluir las opciones de "*login*" y "*logout*" en la página de inicio.

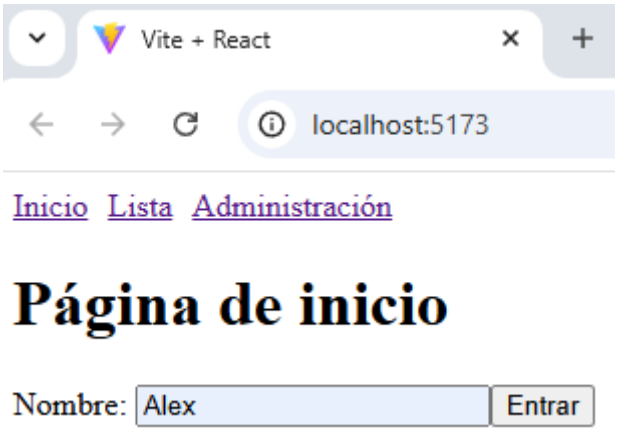
El código es muy sencillo, cargamos el contexto con "***useContext(SeguridadContext)***" asignándole a variables las propiedades a emplear en el componentes. En este caso todas, datos, login y logout.

```
import { useContext, useState } from "react";
import { SeguridadContext } from "../contexts/SeguridadProvider";
```

```
function HomePage() {
  const { datos, logIn, logOut } = useContext(SeguridadContext);
  const [nombre, setNombre] = useState("");

  function handleClick() {
    if (datos.tienePermisos) {
      setNombre("");
      logOut();
    } else {
      if (nombre === "") {
        return;
      }
      /* No es recomendable buscar directamente */
      //const nombre =
document.querySelector('input[name="nombre"]').value;
      logIn(nombre);
    }
  }

  const handleChange = (e) => {
    setNombre(e.target.value);
  };

  return (
    <>
      <h1>Página de inicio</h1>
      <div>
        {datos.tienePermisos ? (
          <>
            <span>Hola {datos.usuario} </span>
            <button onClick={handleClick}>Salir</button>
          </>
        ) : (
          <>
            <span>Nombre: </span>
            <input
              type="text"
              name="nombre"
              value={nombre}
              onChange={handleChange}
            ></input>
            <button onClick={handleClick}>Entrar</button>
          </>
        )}
      </div>
    </>
  );
}
```

```
export default HomePage;
```

La página debería tener el siguiente aspecto.



3. Gestionar el autorización – AdministracionPage.jsx

Cuando creamos la página de administración vimos como redirigir al usuario a otra página cuando no tiene permisos de acceso. En ese ejemplo los permios están definidos a mano mediante el estado booleano "***tienePermisos***".

En el siguiente ejemplo modificamos la página para que recupere los permisos del contexto de seguridad. En este caso únicamente necesitamos la propiedad "***datos***" del contexto.

Revisa las modificaciones.

```
import { useContext, useState } from "react";

import { Navigate, useNavigate } from "react-router-dom";
import { SeguridadContext } from "../contexts/SeguridadProvider";
//import negocio from "../core/Negocio.js";

function AdministracionPage() {
  //const [tienePermisos, setTienePermisos] = useState(true);
  const { datos } = useContext(SeguridadContext);
  const navegar = useNavigate();

  //if (!tienePermisos) {
  if (!datos.tienePermisos) {
    return <Navigate to="/" />;
  }

  const handleClick = () => {
    if (confirm("Ir a inicio")) {
      navegar("/");
    }
```

```
  };

  return (
    <>
      <h1>Página de administración</h1>
      <button onClick={handleClick}>Volver a inicio</button>
    </>
  );
}

export default AdministracionPage;
```

Optimizaciones

Con lo que hemos visto hasta ahora puedes construir aplicaciones perfectamente. El problema de React radica en que por defecto se carga todo el código antes de renderizar las páginas.

En los siguientes apartados vamos a ver varias técnicas para mejorar el rendimiento, como son la carga "***lazy***" o "perezosa" y memorización o cacheo de componentes.

Carga diferida con React.lazy y Suspense

React incluye el método "***lazy***" que permite cargar componentes de manera diferida. Este método devuelve una promesa que se resuelve cuando el componente se desee cargar.

La sintaxis es:

const NombreComponente = lazy(() => import("ruta/NombreComponente"));

El componente "***Suspense***" es un envoltorio que nos permite mostrar un contenido alternativo mientras se carga un componente asíncrono. La propiedad "***fallback***" define el contenido alternativo.

Vamos a ver un ejemplo sencillo de uso de "***Suspense***" pero lo ideal sería emplearlo cerca de la carga de cada elemento asíncrono. Por ejemplo, en la ruta de "*HomePage*" envolviendo el contenido de la propiedad "***element***" de la siguiente manera:

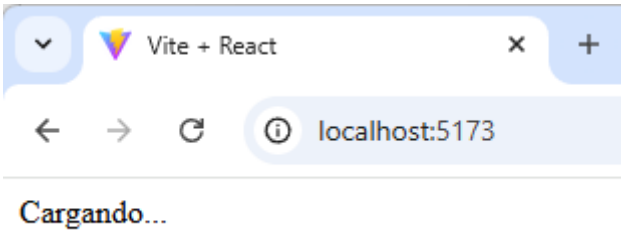
```
<Route index element={
    <Suspense fallback={<div>Cargando Home...</div>}>
      <HomePage />
    </Suspense>
}/>;
```

Revisa el código modificado del componente "***AppEnrutador.jsx***".

```
import { BrowserRouter, Routes, Route } from "react-router-dom";
import { Suspense, lazy } from "react";
```

```
//import AppLayout from "../components/AppLayout";
const AppLayout = lazy(() => import("../components/AppLayout"));

const HomePage = lazy(() => import("../pages/HomePage"));
const ListaPage = lazy(() => import("../pages/ListaPage"));
const DetallesPage = lazy(() => import("../pages/DetallesPage"));
const AdministracionPage = lazy(() =>
import("../pages/AdministracionPage"));
const NoPage = lazy(() => import("../pages/NoPage"));

function AppEnrutador() {
  return (
    <BrowserRouter>
      <Suspense fallback={<div>Cargando...</div>}>
        <Routes>
          <Route path="/" element={<AppLayout />}>
            <Route index element={<HomePage />} />
            <Route path="lista" element={<ListaPage />} />
            <Route path="detalles/:id" element={<DetallesPage />} />
           <Route path="administracion" element={<AdministracionPage/>}/>
            <Route path="*" element={<NoPage />} />
          </Route>
        </Routes>
      </Suspense>
    </BrowserRouter>
  );
}

export default AppEnrutador;
```

Si en herramientas de desarrollador, apartado "*Red*" habilitas la restricción "*3G*" podrás ver la siguiente salida.



NOTA: En teoría el componente "***AppLayout***" deberíamos importarlo directamente ya que va a estar en todas las páginas de la aplicación. Lo he puesto "***lazy***" porque debido a como hemos codificado nuestro ejemplo no veríamos el "***fallback***", lo correcto aquí sería un "***Suspense***" en cada ruta.

Hook React.memo

El hook "***memo***" de React se utiliza para optimizar componentes funcionales al evitar renders innecesarios. Si el componente recibe las mismas "***props***" y su lógica de renderizado no ha cambiado, "***React.memo***" puede prevenir que se vuelva a renderizar, mejorando el rendimiento de la aplicación.

Para usarlo únicamente debemos importar "***memo***" de react y envolver el componente con "***memo***" a la hora de exportarlo.

NOTA: si el componente cambia mucho, o es muy pequeño es posible que no mejoremos el rendimiento.

Revisa el código modificado del componente "***ListaLinea.jpx***"

```
import { useNavigate } from "react-router-dom";
import { memo } from "react";

function ListaLinea({ modulo }) {
  const { id = 0, nombre = "", horas = 0 } = modulo;
  const navegar = useNavigate();

  const handleClick = () => {
    navegar(`/detalles/${id}`);
  };

  return (
    <p>
      <strong>{nombre}</strong> - Horas: {horas} -{" "}
      <button onClick={handleClick}>Detalles</button>
    </p>
  );
}

export default memo(ListaLinea);
```

Bibliografía

Documentación oficial Vite

https://es.vite.dev/guide/

Documentación de React - Aprende

https://es.react.dev/learn

Documentación de React – Referencia

https://es.react.dev/reference/react

Documentación de React – Hook useState

https://es.react.dev/reference/react/useState

Documentación de React – Hook useEffect

https://es.react.dev/reference/react/useEffect

Documentación de React – Hook useContext

https://es.react.dev/reference/react/useContext

Documentación de React – Hook useId

https://es.react.dev/reference/react/useId

Documentación React Router

https://reactrouter.com/home

AMazaing – React – Curso completo desde 0

https://www.youtube.com/watch?v=vH1u6Xv6oXw