



Instituto Tecnológico de Tepic
Ing. Sistemas Computacionales



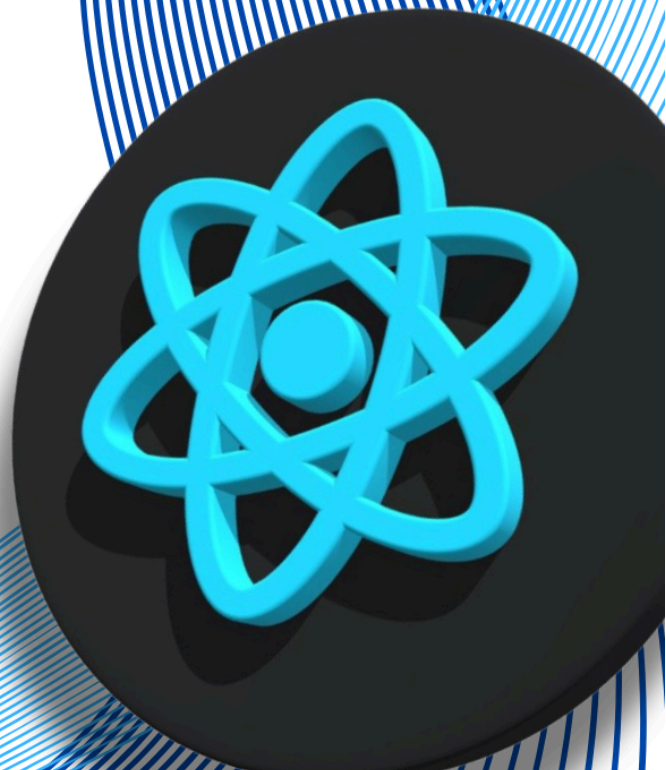
Interfaces Web

REACT: INVESTIGACIÓN

Por:

Acosta Carrillo Yvan Fernando
Ramírez Velázquez Lia Rebeca
Topete Arvizu Roman
Vargas Partida Jorge Luis

6 de Mayo 2025



ÍNDICE

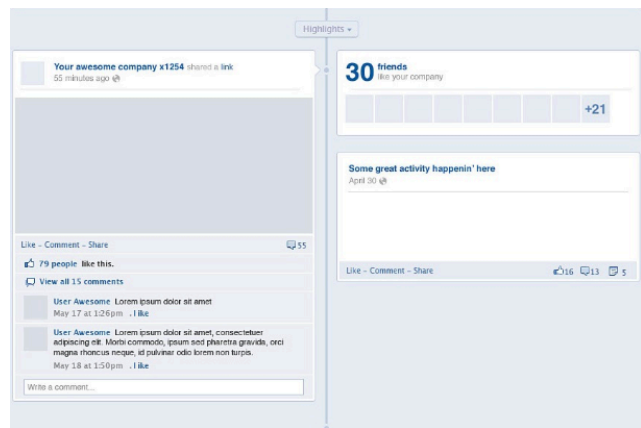
1. ¿Qué es React?	3
2. ¿Quién lo creó y por qué?	3
3. ¿Por qué es tan popular ?	3
4. Ventajas principales de React	3
5. ¿Cómo funciona React?	4
5.1. Componentes	5
5.2. JSX	5
5.3. Props y state	6
6. Hooks	7
6.1. ¿Qué son los hooks?	7
6.2. Hooks más comunes useState, useEffect y useContext	7
7. Estructura general de una app React	8
8. Estilizado	8
9. Routing ¿Qué es el enrutamiento en React?	10
¿Qué es React Router?	10
Componentes importantes de React Router.	10
Ejemplo de funcionamiento de React Router.	11
10. Consumo de APIs	13
¿Qué es una API Rest?	13
Como consumir APIs REST en React	13
10.1 Uso de fetch o axios con useEffect	13
FETCH API	13
AXIOS	15
11. Demo Sencilla de ejemplo	17
Que es Vite	18
Creación de proyecto demo con Vite y React.	18
Referencias	21

1. ¿Qué es React?

Es una librería Javascript para crear interfaces de usuario o para la vista Front de tu aplicación o página web. Es decir que cuando integras React JS con HTML y CSS, la carga de las páginas son más rápidas porque React JS usa componentes que los puedes escribir una sola vez y luego utilizarlos cuando sea necesario mostrar un elemento en el DOM como un botón, una imagen, video, textos, encabezados, etc.

2. ¿Quién lo creó y por qué?

Se creó en 2013 por el ingeniero de software de Facebook llamado Jordan Walke. React nació de una necesidad por ajustar con mayor facilidad y rapidez los timelines de Facebook. Y con esto, poco a poco fueron mejorando la librería hasta convertirse en lo que es hoy en día.



3. ¿Por qué es tan popular ?

React ha ganado popularidad por varias razones:

- Facilidad de uso: Su sintaxis basada en JSX permite escribir código más intuitivo.
- Componentes reutilizables: Facilita la creación de interfaces modulares.
- Alto rendimiento: Gracias al DOM virtual, React actualiza sólo los elementos necesarios en la página.
- Gran comunidad y soporte: Al ser ampliamente adoptado, cuenta con una comunidad activa y numerosos recursos de aprendizaje.

4. Ventajas principales de React

- Es fácil de aprender debido a su simplicidad en términos de sintaxis. No es necesario aprender TypeScript con Angular
- Alto nivel de flexibilidad y máxima capacidad de respuesta.
- DOM virtual (Document Object Model) que convierte los documentos en formato HTML, XHTML o XML en una estructura de árbol más manejable para los navegadores mientras analiza los diferentes elementos de la aplicación.

- Biblioteca JavaScript 100% de código abierto con frecuentes actualizaciones y mejoras de desarrolladores de todo el mundo.

5. ¿Cómo funciona React?

React es una biblioteca de JavaScript desarrollada por Facebook que permite construir interfaces de usuario de manera eficiente y modular. Su enfoque principal es facilitar la creación de aplicaciones web interactivas y dinámicas mediante la reutilización de componentes.

Una de las mayores ventajas de usar React es que puedes infundir código HTML con JavaScript. Los usuarios pueden crear una representación de un nodo DOM declarando la función Element en React. El siguiente código contiene una combinación de HTML y JavaScript:

```
React.createElement("div", { className: "rojo" }, "Texto Hijo");
React.createElement(MyCounter, { count: 3 + 5 });
```

Habrás notado que la sintaxis del código HTML anterior es similar a la de XML. Dicho esto, en lugar de utilizar la clase DOM tradicional, React utiliza `className`. Las etiquetas JSX tienen un nombre, hijos y atributos. Los valores numéricos y las expresiones deben escribirse entre llaves. Las comillas en los atributos JSX representan cadenas, de forma similar a JavaScript. En la mayoría de los casos, React se escribe utilizando JSX en lugar de JavaScript estándar para simplificar los componentes y mantener el código limpio.

Ejemplo de código React escrito usando JSX:

```
MyCounter count={3 + 5} />;
var GameScores = {jugador1: 2, jugador2: 5};
DashboardUnit data-index="2">
h1> Scores/h1> Scoreboard className="results" scores={GameScores} />
/DashboardUnit>;
```

A continuación, se desglosan las etiquetas HTML anteriores:

- **MyCounter**: representa una variable llamada “count” cuyo valor es una expresión numérica.
- **GameScores**: es un objeto literal que tiene dos pares de valores prop.
- **DashboardUnit**: es el bloque XML que se renderiza en la página.
- **scores={GameScores}**: es el atributo scores. Obtiene su valor del objeto literal GameScores definido anteriormente.

Una aplicación React suele tener un único nodo DOM raíz. Al renderizar un elemento en el DOM cambiará la interfaz de usuario de la página.

Siempre que un componente de React devuelve un elemento, el virtual DOM actualizará el DOM real para que coincida.

5.1. Componentes

En React, los componentes son las unidades fundamentales que componen la interfaz de usuario. Permiten dividir la aplicación en partes independientes y reutilizables, facilitando su desarrollo y mantenimiento.

Componentes Funcionales

Los componentes funcionales son funciones de JavaScript que reciben un objeto de propiedades (*props*) como argumento y retornan elementos de React que describen lo que se debe renderizar en la interfaz. Con la introducción de los Hooks en React 16.8, estos componentes pueden manejar estado y efectos secundarios, lo que los hace muy versátiles.

Por ejemplo:

```
import React from 'react';

function Saludo(props) {
  return <h1>Hola, {props.nombre}!</h1>;
}
```

Componentes de Clase

Los componentes de clase son clases de JavaScript que extienden de *React.Component*. Estos componentes pueden tener su propio estado interno y acceder a métodos del ciclo de vida, como *componentDidMount* o *componentWillUnmount*, que permiten ejecutar código en momentos específicos del ciclo de vida del componente.

Ejemplo:

```
import React from 'react';

class Saludo extends React.Component {
  render() {
    return <h1>Hola, {this.props.nombre}!</h1>;
  }
}
```

Actualmente, se recomienda utilizar componentes funcionales para nuevo código debido a su simplicidad y a las ventajas que ofrecen los Hooks.

5.2. JSX

JSX (JavaScript XML) es una extensión de la sintaxis de JavaScript que permite escribir código similar a HTML dentro de archivos JavaScript. Esto facilita la creación y visualización de la estructura de la interfaz de usuario. Aunque parece HTML, JSX es transformado por herramientas como Babel en llamadas a *React.createElement* para crear los elementos que React renderiza.

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

En este ejemplo, JSX permite escribir una etiqueta `<h1>` directamente en el código JavaScript, lo que mejora la legibilidad y la organización del código.

Es importante destacar que, en JSX, se deben seguir ciertas reglas, como cerrar todas las etiquetas y utilizar `className` en lugar de `class` para asignar clases CSS, ya que `class` es una palabra reservada en JavaScript.

5.3. Props y state

En React, los componentes pueden manejar datos mediante props y state.

Props

Las props (propiedades) son un mecanismo para pasar datos desde un componente padre a un componente hijo. Son de solo lectura y permiten que los componentes sean reutilizables y dinámicos al recibir diferentes valores.

```
function Saludo(props) {  
  return <h1>Hola, {props.nombre}!</h1>;  
}
```

En este ejemplo, el componente `Saludo` recibe una prop llamada `nombre` y la utiliza para mostrar un saludo personalizado.

State

El state (estado) es un objeto que representa los datos internos de un componente que pueden cambiar a lo largo del tiempo. A diferencia de las props, el state es mutable y se utiliza para manejar información que puede cambiar en respuesta a acciones del usuario o eventos del sistema.

```
import React, { useState } from 'react';  
  
function Contador() {  
  const [contador, setContador] = useState(0);  
  
  return (  
    <div>  
      <p>Has hecho clic {contador} veces</p>  
      <button onClick={() => setContador(contador + 1)}>  
        Haz clic  
      </button>  
    </div>  
  );  
}
```

```
}
```

En este ejemplo, el componente *Contador* utiliza el Hook *useState* para manejar el estado interno *contador*, que se incrementa cada vez que el usuario hace clic en el botón.

6. Hooks

6.1. ¿Qué son los hooks?

Los **hooks** son una de las características más importantes y modernas de React. Antes de que existieran, solo los componentes de clase podían manejar cosas como el estado o el ciclo de vida del componente. Sin embargo, desde React 16.8, los hooks permiten hacer todo esto **usando componentes funcionales**, lo que simplifica mucho el código y hace que sea más fácil de leer y mantener.

Uno de los principales beneficios de los hooks es que **organizan mejor la lógica**, permitiendo separar las responsabilidades de cada parte de un componente en funciones pequeñas y reutilizables. Todos los hooks en React comienzan con la palabra **use**, lo que ayuda a identificarlos fácilmente. Los tres hooks más comunes son **useState**, **useEffect** y **useContext**.

6.2. Hooks más comunes **useState**, **useEffect** y **useContext**

A. useState. Permite declarar y actualizar valores reactivos dentro de un componente.

Por ejemplo, si queremos mostrar un número que se incrementa cada vez que el usuario hace clic en un botón, usamos **useState** para guardar ese número y actualizarlo. Cada vez que el valor cambia, React vuelve a renderizar el componente mostrando la nueva información.

B. useEffect. es un hook que sirve para ejecutar efectos secundarios, como llamadas a APIs, temporizadores o interacciones con el DOM.

Es muy útil, por ejemplo, cuando queremos traer datos de un servidor en cuanto se cargue el componente por primera vez. Además, también nos permite limpiar esos efectos cuando el componente se desmonta.

C. useContext. Permite acceder a datos globales desde cualquier componente sin necesidad de pasar propiedades manualmente de un componente padre a sus hijos. Esto es especialmente útil para manejar temas de la aplicación (oscuro/claro), la información del usuario actual o configuraciones globales como el idioma. **useContext** se usa en combinación con el objeto **Context**, que actúa como un contenedor de valores compartidos.

7. Estructura general de una app React

Una aplicación hecha con React tiene una **estructura de carpetas y archivos** que ayuda a mantener el código organizado, sobre todo cuando el proyecto crece. Esta estructura puede adaptarse según las necesidades del desarrollador, pero generalmente sigue un patrón común.

El archivo **index.js** es el punto de entrada de la aplicación, donde React **renderiza el componente principal**, llamado **App.jsx**, y lo coloca en el DOM. A partir de ahí, se construyen los demás componentes dentro de la carpeta **/src**, como las vistas (**/pages**) o los componentes reutilizables (**/components**).

También se suelen crear carpetas para los hooks personalizados (**/hooks**), servicios para llamadas a APIs (**/services**) y archivos de estilos (**/styles**). Esta separación permite que cada parte del proyecto tenga su lugar y que el código sea más fácil de mantener y escalar.

Ejemplo

```
/mi-app
├── public/           # Archivos estáticos (favicon, index.html)
├── src/              # Código fuente principal
│   ├── index.js     # Punto de entrada
│   ├── App.jsx      # Componente principal
│   ├── /components/ # Componentes reutilizables (Botón, Card, etc.)
│   ├── /pages/      # Vistas de la app (Home, Login, Dashboard)
│   ├── /hooks/      # Hooks personalizados
│   ├── /context/    # Contextos globales
│   ├── /services/   # Lógica para llamadas a APIs
│   ├── /assets/     # Imágenes, íconos, fuentes
│   └── /styles/     # Archivos CSS o Tailwind config
├── package.json     # Dependencias y scripts
└── vite.config.js / config # Configuración de build
```

8. Estilizado

React no impone una única forma de trabajar con estilos, lo cual es una gran ventaja. Existen múltiples opciones para aplicar estilos a los componentes, y la elección depende del tipo de proyecto, la escala del equipo y las preferencias del desarrollador.

La forma más básica es usar archivos **CSS tradicionales**, importándolos en los componentes. Sin embargo, esto puede causar conflictos cuando muchos componentes usan clases con los mismos nombres.

```
// App.jsx
import './App.css';

function App() {
  return <div className="contenedor">Hola</div>;
}
```


Para evitar esto, se pueden usar **CSS Modules**, que permiten encapsular los estilos a nivel de componente, asegurando que las clases sean únicas y no interfieran con otras. Usan archivos **.module.css** y evitan que las clases se mezclen.

```
// Button.module.css
.boton {
  background-color: blue;
  color: white;
}

// Button.jsx
import styles from './Button.module.css';

<button className={styles.boton}>Click</button>
```

Otra opción es usar **estilos en línea**, definidos como objetos JavaScript. Aunque son rápidos para casos simples o dinámicos, pueden volverse difíciles de manejar en proyectos grandes.

```
const estilo = {
  backgroundColor: 'red',
  padding: '10px',
};

<div style={estilo}>Botón rojo</div>
```

También están los **frameworks de UI como Tailwind CSS**, que ofrece clases utilitarias para aplicar estilos directamente desde el HTML de React, o Material UI, que proporciona componentes estilizados listos para usar.

```
import { Button } from '@mui/material';

<Button variant="contained" color="primary">Guardar</Button>
```

Y finalmente, existe la opción de **styled-components**, una librería que permite escribir estilos usando sintaxis similar a CSS, pero directamente en JavaScript. Esto facilita la creación de componentes con estilos encapsulados y reutilizables.

```
import styled from 'styled-components';

const Boton = styled.button`
  background-color: green;
  color: white;
  padding: 10px;
`;

<Boton>Hola</Boton>
```

9. Routing ¿Qué es el enrutamiento en React?

Normalmente, en aplicaciones que tienen más de una página, cuando pulsas en algún enlace se envía una petición al servidor para que se muestre la nueva página HTML, haciendo que la carga tome más tiempo y la experiencia de usuario se vea un poco lenta.

En **React**, el contenido de la página es creado a partir de nuestros componentes. Así que lo que hace **React Router** es interceptar la petición que se envía al servidor y luego inyectar el contenido dinámicamente desde los componentes que hemos creado. Con este comportamiento podemos convertir nuestra aplicación en un *SPA* (siglas en inglés para *Single Page Application*), mejorando considerablemente la experiencia de usuario al mostrar el contenido más rápidamente.

¿Qué es React Router?

Se trata de una colección de componentes de navegación, con esta librería vamos a obtener un enrutamiento dinámico gracias a los componentes, en otras palabras tenemos unas rutas que renderizan un componente.

Si recordamos lo que se mencionó anteriormente sobre la estructura del proyecto con React, podemos ver que siempre existirá un archivo en la carpeta pública llamado 'index.html', aquí es donde se renderiza todo el contenido del código que se escriba en el archivo 'App.jsx', que es nuestro componente principal, de esta manera solo tendremos un archivo HTML donde se renderiza todo el contenido de nuestra aplicación.

Cuando queramos renderizar en alguna página diferente otro componente no se crea otro archivo HTML, sino que **React Router** nos ayuda a navegar y dirigirnos a dicho componente y cargarlo dentro del mismo archivo 'index.html'. Así que en un SPA, cuando se navega a un nuevo componente con **React Router**, el archivo 'index.html' será reescrito en función de la lógica del componente.

Componentes importantes de React Router.

- **BrowserRouter:**
Este componente es el encargado de envolver nuestra aplicación dándonos acceso al API historial de HTML5 (pushState, replaceState y el evento popstate) para mantener su UI sincronizada con la URL.
- **Switch:**
Este componente es el encargado de que solo se renderice el primer hijo Route o Redirect que coincide con la ubicación. Si no usa este componente todos los componentes Route o Redirect se van a renderizar mientras cumplan con la condición establecida.
- **Route:**
Con Route podemos definir las rutas de nuestra aplicación, quizás sea el componente más importante de React Router para llegar a comprender todo el manejo de esta librería. Cuando definimos una ruta con Route le indicamos qué componente debe renderizar.

Este componente cuenta con algunas propiedades:

- **Path:** la ruta donde debemos renderizar nuestro componente podemos pasar un string o un array de string.Exact: Solo vamos a mostrar nuestro componente cuando la ruta sea exacta. Ej: /home === /home.
- **Strict:** Solo vamos a mostrar nuestro componente si al final de la ruta tiene un slash. Ej: /home/ === /home/
- **Sensitive:** Si le pasamos true vamos a tener en cuenta las mayúsculas y las minúsculas de nuestras rutas. Ej: /Home === /Home
- **Component:** Le pasamos un componente para renderizar solo cuando la ubicación coincide. En este caso el componente se monta y se desmonta no se actualiza.
- **Render:** Le pasamos una función para montar el componente en línea.

Ejemplo de funcionamiento de React Router.

Para instalar la librería solo tenemos que ir a la terminal estar ubicados en la raíz de nuestro proyecto y ejecutar el siguiente comando.

```
npm install react-router-dom
```

Una vez instalada, lo primero que debemos hacer es que esto se encuentre disponible en cualquier punto de nuestra aplicación, para ello vamos al archivo 'index.js' y aquí adentro importamos 'BrowserRouter' desde react-router-dom, para luego rodear el componente raíz (Aplicacion) con él.

Después de estos cambios, el archivo debería verse algo como lo siguiente:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './indice.css';
import Aplicacion from './Aplicacion';
import { BrowserRouter } from "react-router-dom";

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

Ahora, para poder dirigir de manera dinámica, vamos a suponer que tenemos 3 componentes: Inicio, Sobre Nosotros y Contacto.

Creamos los componentes de la siguiente manera:

```
function Inicio() {
  return (
    <div>
      <h1>Esta es la página de inicio</h1>
    </div>
  );
}
```

```
    </div>
  );
}
export default Inicio;
```

```
function SobreNosotros() {
  return (
    <div>
      <h1>Esta es la página sobre nosotros</h1>
    </div>
  );
}

export default SobreNosotros;
```

```
function Contacto() {
  return (
    <div>
      <h1>Esta es la página de contacto</h1>
    </div>
  );
}

export default Contacto;
```

Ahora, como todo esto se carga en el archivo App.jsx como componente raíz, vamos a crear nuestras rutas en él.

Primero importamos las funcionalidades Routes y Route desde react-router-dom, con las cuales se declaran las rutas posteriormente, también importamos cada una de los componentes a los que navegará cada ruta.

```
import { Routes, Route } from "react-router-dom"
import Inicio from "./Inicio"
import SobreNosotros from "./SobreNosotros"
import Contacto from "./Contacto"

function Aplicacion() {
  return (
    <div className="Aplicacion">
      <Routes>
        <Route path="/" element={ <Inicio /> } />
        <Route path="sobre-nosotros" element={ <SobreNosotros /> } />
      </Routes>
    </div>
  );
}
```

```
export default Aplicacion;
```

Con esto hemos definido nuestras rutas y sus direcciones, así como asociarlas a sus respectivos componentes

10. Consumo de APIs

Como desarrolladores, si queremos construir aplicaciones robustas usando React, en algún momento van a requerir consumir algún tipo de API para agregar, mostrar, modificar o eliminar información dentro de su aplicación.

¿Qué es una **API Rest**?

Por sus siglas en inglés, **API** significa “*Interfaz de Programación de Aplicación*”. Se trata de un medio que permite la comunicación programática entre diferentes aplicaciones y que devuelve una respuesta en tiempo real.

En los 2000, Roy Fielding definió el concepto REST como un estilo de arquitectura y metodología usado frecuentemente en el desarrollo de servicios de internet. Es un acrónimo que significa, en inglés, Transferencia de Estado Representacional.

Cuando se hace una petición a través de una API REST, ésta envía a un endpoint, una representación del estado actual del recurso. Esta representación del estado puede tomar la forma de archivo JSON (JavaScript Object Notation), XML, o HTML.

Como consumir APIs REST en React

Es posible consumir APIs de distintas maneras en React, pero vamos a ver únicamente las dos formas más populares, por medio de Axios y Fetch API. Cada uno cuenta con características que pueden resultar positivas o negativas según el proyecto y las necesidades de nuestro proyecto.

10.1 Uso de fetch o axios con useEffect

A continuación, veremos a detalle las dos maneras más populares de consumir una API Rest.

FETCH API

La **Fetch API** es un método de JavaScript para obtener recursos de un servidor o de un endpoint de una API. Como ya viene incluida en el navegador, no es necesario instalar dependencias o paquetes.

El método **fetch()** requiere de un argumento obligatorio que es la URL o dirección al recurso que queremos obtener. Luego, este se convierte en una promesa para que podamos manejar su éxito o error a través de los métodos **then()** y **catch()**.

La respuesta por defecto es una respuesta HTTP estándar más que un JSON, pero podemos ver la información como un objeto JSON al usar el método `json()` incluida en la respuesta.

El siguiente es un ejemplo básico de cómo usar una Fetch API, mostrando los datos obtenidos en consola:

```
fetch('https://jsonplaceholder.typicode.com/posts?_limit=10')
  .then(response => response.json())
  .then(data => console.log(data));
```

Un ejemplo más completo de cómo podría usarse en una aplicación real es el siguiente:

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

const App = () => {
  const [posts, setPosts] = useState([]);
  useEffect(() => {

    fetch('https://jsonplaceholder.typicode.com/posts?
_limit=10')
      .then((response) => response.json())
      .then((data) => {
        console.log(data);
        setPosts(data);
      })
      .catch((err) => {
        console.log(err.message);
      });

  }, []);

  return (
    // ... consume here
  );
};
```

Creamos un **estado** para guardar la información que **obtenemos de la API** así la podemos usar después. También establecemos el valor predeterminado a un arreglo vacío.

Como estamos haciendo un get, el único parámetro en la función `fetch` es la URL

Esta promesa nos retorna un objeto, en el primer método **then()** lo convertimos en json, y en el segundo **then()** se obtiene la información

Manejamos el rechazo de la petición, por medio del **catch()**

El anterior ejemplo utiliza promesas en cadena con el método `then()`, lo que podría provocar cierta confusión en los programadores que no están tan familiarizados con esta sintaxis, podemos hacer que se vea un poco más sencillo de leer utilizando `async/await` con los métodos `Fetch`. A continuación se muestra el mismo ejemplo con GET pero usando `async/await`:

```
useEffect(() => {
  const fetchPost = async () => {
    const response = await fetch(
      'https://jsonplaceholder.typicode.com/posts?_limit=10'
    );
    const data = await response.json();
    console.log(data);
    setPosts(data);
  };
  fetchPost();
});
```

```
}, []);
```

A continuación se muestran ejemplos de los métodos POST y DELETE usando FETCH API

```
const addPosts = async (title, body) => {
  await fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify({
      title: title,
      body: body,
      userId: Math.random().toString(36).slice(2),
    }),
    headers: {
      'Content-type': 'application/json; charset=UTF-8',
    },
  })
  .then((response) => response.json())
  .then((data) => {
    setPosts((posts) => [data, ...posts]);
    setTitle('');
    setBody('');
  })
  .catch((err) => {
    console.log(err.message);
  });
};
```

```
const deletePost = async (id) => {
  await fetch(`https://jsonplaceholder.typicode.com/posts/${id}`, {
    method: 'DELETE',
  }).then((response) => {
    if (response.status === 200) {
      setPosts(
        posts.filter((post) => {
          return post.id !== id;
        })
      );
    } else {
      return;
    }
  });
};
```

AXIOS

Axios es un cliente HTTP basado en promesas que hace más simple el envío de peticiones HTTP asíncronas a endpoints REST.

A diferencia de Fetch, Axios no viene incluido en el navegador por lo que necesitamos instalarlo en nuestro proyecto para poder utilizarlo. Para instalar Axios en tu proyecto necesitas correr el siguiente comando:

```
npm install axios
```

Una vez que lo instalamos correctamente, podemos crear una instancia, esto no es necesario pero si recomendable ya que nos evita repeticiones innecesarias.

Para crear una instancia, usamos el método `.create()`, que podemos usarlo para especificar información como la **URL** y posibles encabezados:

```
import axios from "axios";

const client = axios.create({
  baseURL: "https://jsonplaceholder.typicode.com/posts"
});
```

Usaremos esta instancia para realizar la petición GET, aquí todo lo que necesitamos es establecer los parámetros si es que se requieren y obtener la respuesta como un JSON por defecto. A diferencia del método Fetch no se necesita una opción para declarar el método. Solamente agregamos el método a la instancia y hacemos la petición.

Ejemplo de GET con Axios, tomando como base una situación similar a la que se planteó en el mismo ejemplo con Fetch:

```
useEffect(() => {
  client.get('?_limit=10').then((response) => {
    setPosts(response.data);
  });
}, []);
```

Esto es con una estructura de promesa, por medio del método `then()`. De igual manera que con Fetch, podemos usar las estructuras `async/await` para facilitar la escritura y comprensión del código. Usando la sintaxis `await/async` el método nos queda de la siguiente manera:

```
useEffect(() => {
  const fetchPost = async () => {
    let response = await client.get('?_limit=10');
    setPosts(response.data);
  };
  fetchPost();
}, []);
```

Al hacer este tipo de peticiones también puede ocurrir algún tipo de error, para manejarlo, podemos hacer uso de un bloque **try-catch**, similar a lo que se hizo con el método Fetch en el ejemplo anterior.

```
const fetchPost = async () => {
  try {
    let response = await client.get('?_limit=10');
```



```
    setPosts(response.data);
  } catch (error) {
    console.log(error);
  }
};
```

A continuación se ponen algunas diferencias principales entre los métodos antes mencionados.

AXIOS	FETCH
Axios es un paquete de terceros independiente que es sencillo de instalar.	Fetch viene integrado en la mayoría de los navegadores modernos. No requiere instalación.
Axios utiliza la propiedad data .	Fetch utiliza la propiedad body .
La propiedad data de Axios contiene el objeto.	El body de Fetch debe ser convertido a cadena (stringificado).
La petición de Axios se considera aceptada cuando el status es 200 y el statusText es "OK".	La petición de Fetch se considera exitosa cuando el objeto de respuesta contiene la propiedad ok .
Axios realiza transformaciones automáticas de datos JSON.	El manejo de datos JSON con Fetch es un proceso de dos etapas: primero, realizar la petición; segundo, llamar al método .json() en la respuesta.
Axios permite cancelar la petición y establecer un tiempo de espera (timeout).	Fetch no lo permite.
Axios tiene soporte incorporado para mostrar el progreso de descarga.	Fetch no soporta el progreso de subida.
Axios tiene un amplio soporte en navegadores.	Fetch solo es compatible con Chrome 42+, Firefox 39+, Edge 14+ y Safari 10.1+ (esto se conoce como compatibilidad hacia atrás).

11. Demo Sencilla de ejemplo

Existen diferentes maneras de crear un proyecto de React, algunas por medio de otros frameworks o compiladores que facilitan y ayudan a crear los componentes básicos del proyecto. Sin embargo, para aprender de mejor manera y entender completamente cómo funciona React, considero que el mejor comenzar a crear el proyecto desde 0, para ello es necesario un “compilador” o entorno de desarrollo que favorezca la creación y desarrollo de nuestro proyecto con React.

Que es Vite

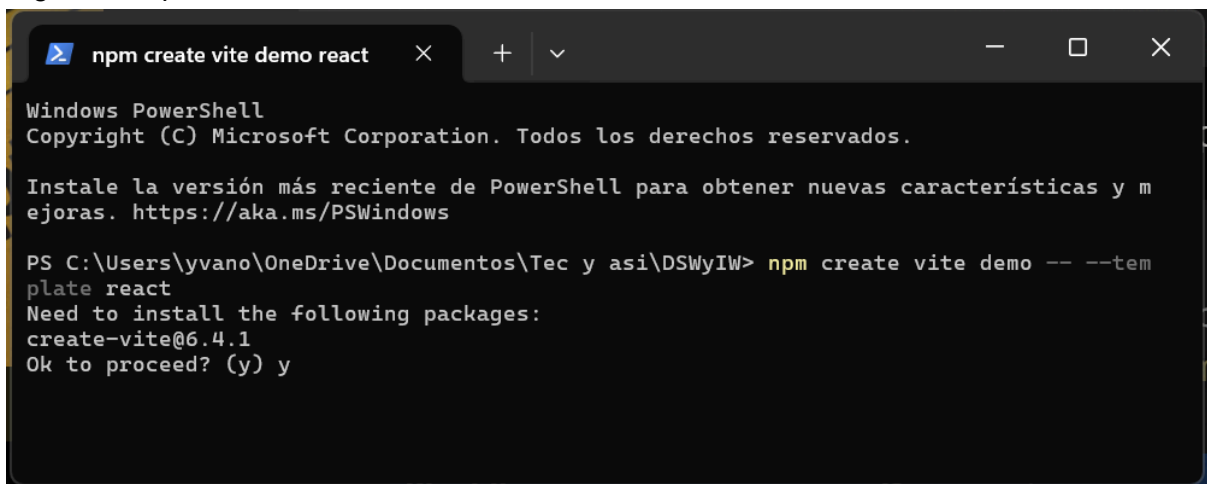
Vite se define como una herramienta de desarrollo frontend que nos permite crear y desarrollar proyectos web de manera sencilla y cómoda, Vite fue desarrollado por Evan You, el creador de Vue. Con Vite podemos crear tanto de proyectos sin frameworks, como proyectos utilizando Vue, **React**, Preact, Svelte o Lit (tanto usando Javascript como Typescript).

Creación de proyecto demo con Vite y React.

Comenzamos colocandonos en una carpeta donde queremos que se encuentre la carpeta de nuestro proyecto, y colocamos el siguiente comando:

```
npm create vite demo -- --template react
```

Esto nos creará la base del proyecto con la plantilla de React, seguimos las instrucciones según nos aparece en la terminal.

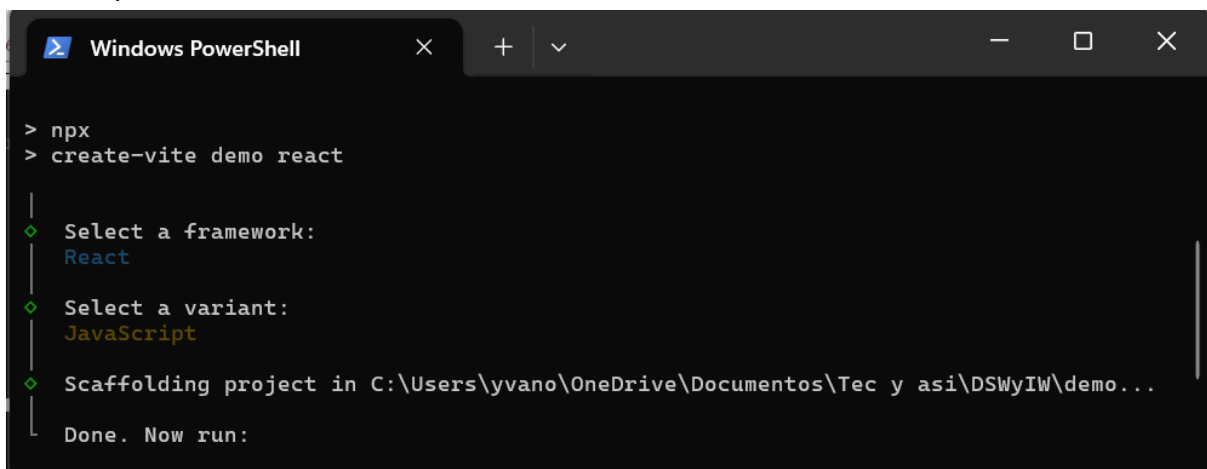


```
npm create vite demo react x + v - □ x
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Instale la versión más reciente de PowerShell para obtener nuevas características y m
ejoras. https://aka.ms/PSWindows

PS C:\Users\yvano\OneDrive\Documentos\Tec y asi\DSWyIW> npm create vite demo -- --tem
plate react
Need to install the following packages:
create-vite@6.4.1
Ok to proceed? (y) y
```

Después, veremos que nos solicita seleccionar el framework y el lenguaje que queremos trabajar, seleccionamos entonces el framework React y el lenguaje de programación JavaScript.



```
Windows PowerShell x + v - □ x

> npx
> create-vite demo react

|
| ♦ Select a framework:
|   React
|
| ♦ Select a variant:
|   JavaScript
|
| ♦ Scaffolding project in C:\Users\yvano\OneDrive\Documentos\Tec y asi\DSWyIW\demo...
|
| Done. Now run:
```

Después de esto podemos ver que la creación del proyecto ha concluido, por lo que podemos dirigirnos al directorio, instalar las dependencias que son necesarias con el comando `npm install`, y después correr el proyecto demo como viene la plantilla

```
L Done. Now run:

cd demo
npm install
npm run dev

PS C:\Users\yvano\OneDrive\Documentos\Tec y asi\DSWyIW> |
```

Nos adentramos en la carpeta principal del proyecto con `cd demo` y después ejecutamos el comando `npm install` para que las dependencias correspondientes sean instaladas.

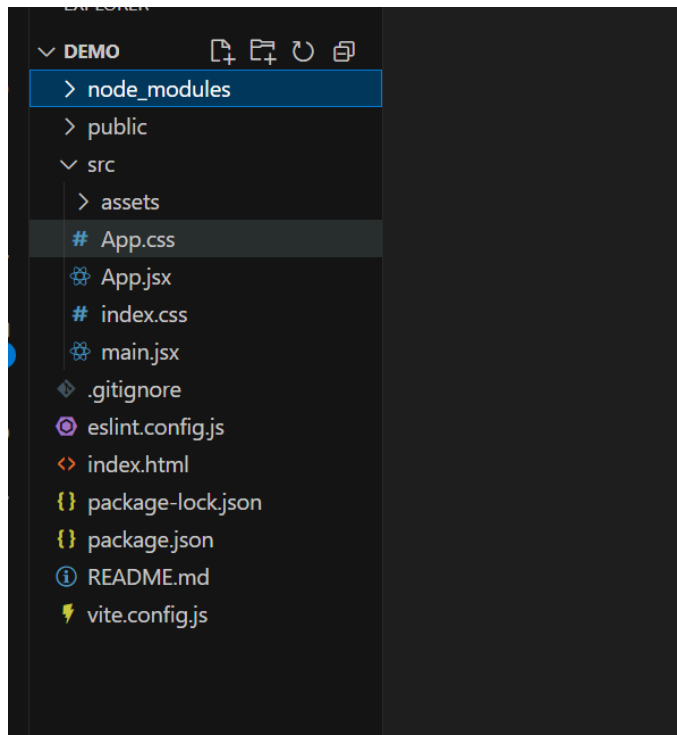
```
PS C:\Users\yvano\OneDrive\Documentos\Tec y asi\DSWyIW> cd demo
PS C:\Users\yvano\OneDrive\Documentos\Tec y asi\DSWyIW\demo> npm install

added 225 packages, and audited 226 packages in 37s

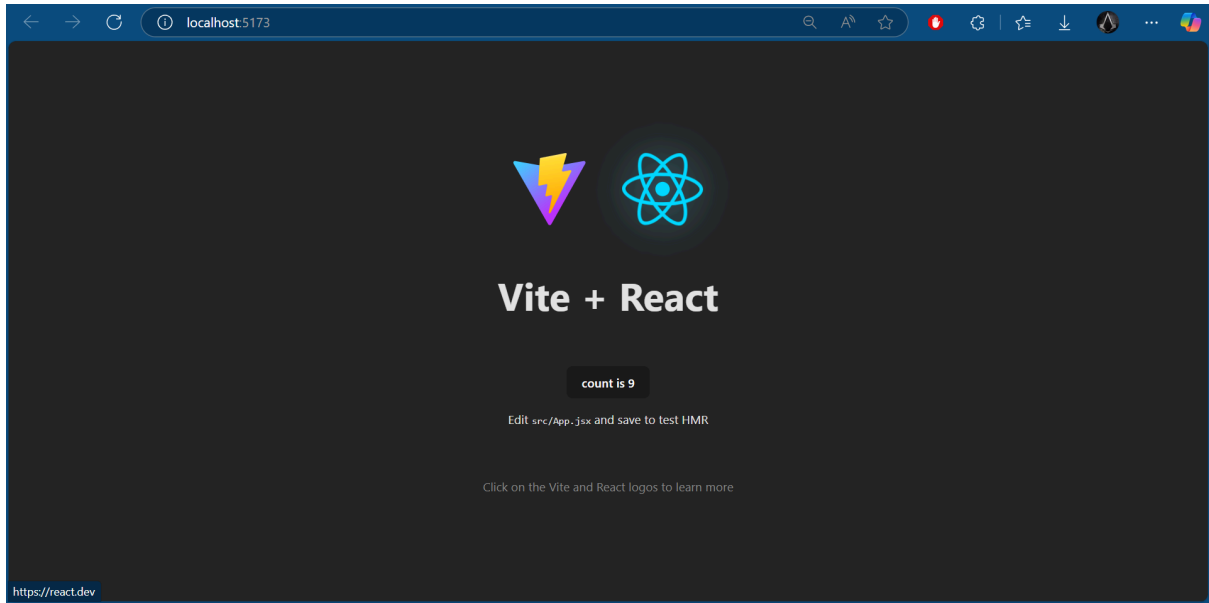
48 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS C:\Users\yvano\OneDrive\Documentos\Tec y asi\DSWyIW\demo> |
```

Podemos abrir el editor de texto o IDE dentro del proyecto por medio del comando `code .` con lo que podemos ver la estructura del proyecto, normalmente todo el contenido, páginas adicionales y demás recursos que se necesiten para el proyecto.



Finalmente ejecutamos el comando `npm run dev`, con lo cual podemos ver en el navegador el ejemplo base de el proyecto con React.



Referencias

- Meta Platforms, Inc. (2024). *React documentation*. React.dev. <https://react.dev>
- Remix Software. (2024). *React Router documentation*. React Router. <https://reactrouter.com>
- W3Schools. (2024). *React tutorial*. W3Schools. <https://www.w3schools.com/react>
- freeCodeCamp. (2020). *Learn React by building a simple app*. <https://www.freecodecamp.org/news/learn-react-by-building-a-simple-app-d7f348e79c5e/>
- Wieruch, R. (2022). *The Road to React: The one with Hooks*. Self-published. <https://www.roadtoreact.com>
- Scrimba. (2024). *Learn React for free*. <https://scrimba.com/learn/learnreact>
- Tailwind Labs, Inc. (2024). *Tailwind CSS documentation*. <https://tailwindcss.com/docs>
- MUI. (2024). *Material UI documentation*. <https://mui.com>
- Serrano, J. (2020, 14 junio). Aprende a crear rutas en React con React Router DOM | John Serrano. <https://johnserrano.co/blog/aprende-a-crear-rutas-con-react-router>
- Carlos. (2023, January 20). Tutorial de React Router versión 6 – Cómo navegar a otros componentes y configurar un enrutador. freeCodeCamp.org. <https://www.freecodecamp.org/espanol/news/tutorial-de-react-router-version-6-como-navegar-a-otros-componentes-y-configurar-un-enrutador/>
- Areal, C. (2024, August 20). Cómo consumir REST APIs en React: Guía para principiantes. freeCodeCamp.org. <https://www.freecodecamp.org/espanol/news/como-consumir-rest-apis-en-react-guia-para-principiantes/>
- Tutorial inicial de Vite - Javascript en español. (n.d.). <https://lenguajejs.com/javascript/automatizadores/vite/>
- React. (s.f.). *Componentes y propiedades*. <https://es.legacy.reactjs.org/docs/components-and-props.html>
- React. (s.f.). *Introducing JSX*. <https://legacy.reactjs.org/docs/introducing-jsx.html>
- GeeksforGeeks. (2025, febrero 11). *ReactJS State vs Props*. <https://www.geeksforgeeks.org/reactjs-state-vs-props/>
- React. (s.f.). *Writing Markup with JSX*. <https://react.dev/learn/writing-markup-with-jsx>