

Tutorial Completo de React con Vite

Índice

1. [Introducción](#)
2. [Requisitos previos](#)
3. [¿Qué es React?](#)
4. [¿Qué es Vite?](#)
5. [Configuración del entorno](#)
6. [Creando tu primer proyecto](#)
7. [Estructura del proyecto](#)
8. [Componentes en React](#)
9. [JSX: JavaScript + XML](#)
10. [Props: Pasando datos entre componentes](#)
11. [Estado \(State\): Datos dinámicos](#)
12. [Eventos en React](#)
13. [Renderizado condicional](#)
14. [Listas y keys](#)
15. [Hooks básicos: useState y useEffect](#)
16. [Formularios en React](#)
17. [Estilizando componentes](#)
18. [React Router: Navegación](#)
19. [Consumiendo APIs](#)
20. [Construyendo un proyecto completo](#)
21. [Optimización y despliegue](#)
22. [Recursos adicionales](#)
23. [Conclusión](#)

Introducción

React es una de las bibliotecas de JavaScript más populares para construir interfaces de usuario, mientras que Vite es una herramienta de construcción moderna que proporciona una experiencia de desarrollo más rápida y eficiente. En este tutorial, aprenderás a utilizar estas tecnologías juntas para crear aplicaciones web modernas y dinámicas.

Requisitos previos

Antes de comenzar, asegúrate de tener instalado:

- **Node.js** (versión 14.18+ o 16+)
- **NPM** (generalmente viene con Node.js)
- **Editor de código** (recomiendo Visual Studio Code)
- **Conocimientos básicos de HTML, CSS y JavaScript**

¿Qué es React?

React es una biblioteca de JavaScript desarrollada por Facebook (ahora Meta) para construir interfaces de usuario. Sus principales características son:

- **Basada en componentes:** Todo en React son componentes reutilizables.
- **Virtual DOM:** Una representación en memoria del DOM real que optimiza las actualizaciones.
- **Flujo de datos unidireccional:** Los datos fluyen de componentes padres a hijos.
- **JSX:** Una extensión de sintaxis que permite escribir HTML en JavaScript.

React no es un framework completo como Angular; es una biblioteca específicamente diseñada para la capa de vista, lo que la hace más flexible y fácil de integrar con otras herramientas.

¿Qué es Vite?

Vite (pronunciado como "vit", significa "rápido" en francés) es una herramienta de construcción moderna que ofrece:

- **Servidor de desarrollo extremadamente rápido** gracias al uso de ESM (ECMAScript Modules) nativos.
- **Hot Module Replacement (HMR)** optimizado para React.
- **Construcción de producción optimizada** utilizando Rollup.
- **Configuración mínima** con soporte integrado para TypeScript, CSS, SASS, etc.
- **API de complementos** para extender funcionalidades.

Vite fue creado por Evan You (creador de Vue.js) y se ha convertido en una alternativa popular a Create React App debido a su velocidad y simplicidad.

Configuración del entorno

Para configurar tu entorno de desarrollo:

1. **Instala Node.js:** Visita nodejs.org y descarga la versión LTS.
2. **Verifica la instalación** abriendo una terminal/consola y ejecutando:

bash

 Copiar

```
node -v  
npm -v
```

Ambos comandos deberían mostrar los números de versión, confirmando que la instalación fue exitosa.

Creando tu primer proyecto

Para crear un nuevo proyecto de React con Vite:

1. **Abre una terminal** y navega hasta la carpeta donde quieras crear tu proyecto.
2. **Ejecuta el comando** para crear un nuevo proyecto:

bash  Copiar

```
npm create vite@latest mi-primer-app-react -- --template react
```

3. **Navega a la carpeta del proyecto:**

bash  Copiar

```
cd mi-primer-app-react
```

4. **Instala las dependencias:**

bash  Copiar

```
npm install
```

5. **Inicia el servidor de desarrollo:**

bash  Copiar

```
npm run dev
```

Verás un mensaje con la URL donde se está ejecutando tu aplicación (normalmente <http://localhost:5173/>). Al abrir esta URL en tu navegador, deberías ver la página de bienvenida de React + Vite.

Estructura del proyecto

Un proyecto de React con Vite tiene la siguiente estructura básica:

 Copiar

```
mi-primer-app-react/
├── node_modules/      # Módulos instalados (generado automáticamente)
├── public/            # Archivos estáticos accesibles públicamente
│   └── vite.svg        # Logo de Vite
├── src/               # Código fuente de la aplicación
│   ├── assets/          # Recursos como imágenes, fuentes, etc.
│   │   └── react.svg    # Logo de React
│   ├── App.css          # Estilos para el componente App
│   ├── App.jsx          # Componente principal App
│   ├── index.css        # Estilos globales
│   └── main.jsx         # Punto de entrada de la aplicación
├── .eslintrc.cjs       # Configuración de ESLint
├── .gitignore          # Archivos y carpetas ignorados por Git
├── index.html          # Archivo HTML principal
├── package.json         # Dependencias y scripts
├── package-lock.json    # Versiones exactas de dependencias
└── vite.config.js       # Configuración de Vite
```

Entendamos los archivos clave:

- **index.html**: El archivo HTML base donde se monta la aplicación React.
- **src/main.jsx**: El punto de entrada de JavaScript que inicializa React.
- **src/App.jsx**: El componente principal que contiene la estructura inicial.

Componentes en React

Los componentes son los bloques de construcción en React. Un componente es una función o clase que devuelve elementos de React (normalmente expresados como JSX).

Componentes funcionales

Este es el tipo de componente más común en React moderno:

jsx

 Copiar

```
// Componente simple que devuelve un elemento h1
function Saludo() {
  return <h1>¡Hola, mundo!</h1>;
}

export default Saludo;
```

Para usar este componente en otro lugar:

jsx

Copiar

```
import Saludo from './Saludo';

function App() {
  return (
    <div>
      <Saludo />
    </div>
  );
}
```

Componentes de clase

Aunque menos comunes en el desarrollo moderno de React, los componentes de clase todavía se usan:

jsx

Copiar

```
import React, { Component } from 'react';

class Saludo extends Component {
  render() {
    return <h1>¡Hola, mundo!</h1>;
  }
}

export default Saludo;
```

JSX: JavaScript + XML

JSX es una extensión de sintaxis para JavaScript que parece HTML pero te permite utilizar toda la potencia de JavaScript.

jsx

Copiar

```
function Bienvenida() {
  const nombre = "Estudiante";
  return (
    <div>
      <h1 className="titulo">Bienvenido, {nombre}</h1>
      <p>Este es un ejemplo de JSX que incluye JavaScript dentro de llaves {2 + 2}</p>
    </div>
  );
}
```

Observa que:

- Usamos `className` en lugar de `class` (ya que `class` es una palabra reservada en JavaScript).
- Las expresiones JavaScript van dentro de llaves `{}`.
- JSX debe tener un único elemento raíz (como el `<div>` en este caso).

Props: Pasando datos entre componentes

Las props (propiedades) son la forma de pasar datos de un componente padre a un componente hijo:

jsx

Copiar

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

// Componente padre que pasa props
function App() {
  return (
    <div>
      <Saludo nombre="María" />
      <Saludo nombre="Juan" />
    </div>
  );
}
```

También puedes usar la desestructuración para hacer el código más limpio:

jsx

Copiar

```
function Saludo({ nombre, edad }) {  
  return <h1>Hola, {nombre}! Tienes {edad} años.</h1>;  
}  
  
// Uso:  
<Saludo nombre="María" edad={25} />
```

Estado (State): Datos dinámicos

El estado permite a los componentes de React gestionar datos que cambian con el tiempo. En componentes funcionales, usamos el hook `useState`:

jsx

Copiar

```
import { useState } from 'react';  
  
function Contador() {  
  // useState devuelve un array con 2 elementos:  
  // 1. El valor actual del estado  
  // 2. Una función para actualizar ese estado  
  const [contador, setContador] = useState(0);  
  
  return (  
    <div>  
      <p>Has hecho clic {contador} veces</p>  
      <button onClick={() => setContador(contador + 1)}>  
        Incrementar  
      </button>  
    </div>  
  );  
}
```

Reglas importantes sobre el estado:

- Nunca modifiques el estado directamente; usa siempre la función actualizadora.
- Las actualizaciones de estado pueden ser asíncronas.
- El estado es local al componente a menos que lo pases explícitamente.

Eventos en React

React maneja eventos de forma similar a HTML, pero con algunas diferencias sintácticas:

jsx

Copiar

```
function BotonEvento() {
  const manejarClic = () => {
    alert('¡Botón clicado!');
  };

  return (
    <button onClick={manejarClic}>
      Haz clic en mí
    </button>
  );
}
```

Características de los eventos en React:

- Los nombres de eventos usan camelCase (`onClick`) en lugar de (`onclick`).
- Se pasa una función como manejador de eventos, no una cadena.
- Para prevenir el comportamiento predeterminado, debes llamar explícitamente a (`e.preventDefault()`).

jsx

Copiar

```
function FormularioSimple() {
  const manejarEnvio = (e) => {
    e.preventDefault(); // Previene el comportamiento predeterminado del formulario
    alert('Formulario enviado');
  };

  return (
    <form onSubmit={manejarEnvio}>
      <button type="submit">Enviar</button>
    </form>
  );
}
```

Renderizado condicional

Puedes renderizar diferentes elementos según ciertas condiciones:

Usando el operador ternario

jsx

Copiar

```
function SaludoCondicional({ estaLogueado }) {  
  return (  
    <div>  
      {estaLogueado  
        ? <h1>Bienvenido de nuevo!</h1>  
        : <h1>Por favor, inicia sesión</h1>  
      }  
    </div>  
  );  
}
```

Usando el operador lógico &&

jsx

Copiar

```
function ListaDeNotificaciones({ mensajes }) {  
  return (  
    <div>  
      <h1>Notificaciones</h1>  
      {mensajes.length > 0 &&  
        <p>Tienes {mensajes.length} mensajes sin leer.</p>  
      }  
    </div>  
  );  
}
```

Listas y keys

Para renderizar listas en React, usamos el método `map()` de JavaScript:

jsx

Copiar

```
function ListaDeElementos() {  
  const items = ['Manzana', 'Banana', 'Naranja', 'Uva'];  
  
  return (  
    <ul>  
      {items.map((item, index) => (  
        <li key={index}>{item}</li>  
      ))}  
    </ul>  
  );  
}
```

Las `keys` son muy importantes cuando renderizas listas en React:

- Ayudan a React a identificar qué elementos han cambiado.
- Deben ser únicas entre hermanos (pero no globalmente).
- Idealmente, usa IDs estables en lugar de índices del array cuando sea posible.

jsx

Copiar

```
function ListaDeUsuarios({ usuarios }) {  
  return (  
    <ul>  
      {usuarios.map(usuario => (  
        <li key={usuario.id}>{usuario.nombre}</li>  
      ))}  
    </ul>  
  );  
}
```

Hooks básicos: useState y useEffect

Los Hooks son funciones especiales que te permiten "enganchar" a características de React en componentes funcionales.

useState

Ya vimos `useState` para manejar el estado:

jsx

Copiar

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

function FormularioNombre() {
  const [nombre, setNombre] = useState('');

  return (
    <div>
      <input
        type="text"
        value={nombre}
        onChange={(e) => setNombre(e.target.value)}
        placeholder="Escribe tu nombre"
      />
      <p>Hola, {nombre || 'visitante'}!</p>
    </div>
  );
}
```

useEffect

useEffect te permite realizar efectos secundarios en componentes funcionales:

jsx

Copiar

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

function TemporizadorSimple() {
  const [segundos, setSegundos] = useState(0);

  useEffect(() => {
    // Este código se ejecuta después de cada renderizado
    const intervalo = setInterval(() => {
      setSegundos(segundos => segundos + 1);
    }, 1000);

    // Función de limpieza que se ejecuta antes del próximo efecto o cuando se desmonta
    return () => clearInterval(intervalo);
  }, []); // Array de dependencias vacío significa "ejecutar solo una vez"

  return <p>Han pasado {segundos} segundos</p>;
}
```

El array de dependencias controla cuándo se ejecuta el efecto:

- `[]`: Solo después del primer renderizado (montaje).
- `[valor1, valor2]`: Cuando cualquiera de estos valores cambia.
- Omitido: Después de cada renderizado.

Formularios en React

Los formularios en React normalmente se implementan como componentes "controlados":

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

function FormularioContacto() {
  const [formData, setFormData] = useState({
    nombre: '',
    email: '',
    mensaje: ''
  });

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData({
      ...formData, // Mantiene los valores existentes
      [name]: value // Actualiza solo el campo modificado
    });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Datos enviados:', formData);
    // Aquí irían las llamadas a APIs o acciones con los datos
  };
}

return (
  <form onSubmit={handleSubmit}>
    <div>
      <label htmlFor="nombre">Nombre:</label>
      <input
        type="text"
        id="nombre"
        name="nombre"
        value={formData.nombre}
        onChange={handleChange}
        required
      />
    </div>

    <div>
      <label htmlFor="email">Email:</label>
      <input
        type="email"
        id="email"
        name="email"
        value={formData.email}
      />
    </div>
  </form>
)
```

```

        onChange={handleChange}
        required
      />
    </div>

    <div>
      <label htmlFor="mensaje">Mensaje:</label>
      <textarea
        id="mensaje"
        name="mensaje"
        value={formData.mensaje}
        onChange={handleChange}
        required
      />
    </div>

    <button type="submit">Enviar</button>
  </form>
);
}

```

En este ejemplo:

- Cada campo de formulario tiene un `value` controlado por el estado.
- Un único controlador `handleChange` actualiza el estado adecuado basado en el atributo `name`.
- `[name]: value` es la sintaxis de propiedades computadas de ES6 para actualizar dinámicamente la clave.

Estilizando componentes

Hay varias formas de aplicar estilos en React:

1. CSS normal

jsx

 Copiar

```

// En App.jsx
import './App.css'; // Importa el archivo CSS

function App() {
  return <div className="container">Contenido con estilo</div>;
}

```

2. CSS en línea

jsx

Copiar

```
function EstiloEnLinea() {  
  const estilo = {  
    color: 'blue',  
    backgroundColor: 'lightgray',  
    padding: '10px',  
    borderRadius: '5px'  
  };  
  
  return <div style={estilo}>Estilo aplicado en línea</div>;  
}  

```

3. CSS Modules

Vite admite módulos CSS por defecto:

css

Copiar

```
/* Button.module.css */  
.button {  
  background-color: #4CAF50;  
  color: white;  
  padding: 10px 15px;  
  border: none;  
  border-radius: 4px;  
  cursor: pointer;  
}  
  
.button:hover {  
  background-color: #45a049;  
}
```

jsx

Copiar

```
// Button.jsx  
import styles from './Button.module.css';  
  
function Button({ children }) {  
  return <button className={styles.button}>{children}</button>;  
}  
  
export default Button;
```

4. Bibliotecas de CSS-in-JS

Puedes utilizar bibliotecas como Styled Components:

bash

 Copiar

```
npm install styled-components
```

jsx

 Copiar

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

// Crea un componente de título con estilos
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`;

function Header() {
  return <Title>¡Hola desde Styled Components!</Title>;
}
```

React Router: Navegación

React Router es la biblioteca estándar para manejar la navegación en aplicaciones React:

bash

 Copiar

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

Configuración básica:

```
// main.jsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import { BrowserRouter } from 'react-router-dom';
import App from './App';
import './index.css';

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>,
);
});
```

```
// App.jsx
import { Routes, Route, Link } from 'react-router-dom';
import Inicio from './pages/Inicio';
import SobreNosotros from './pages/SobreNosotros';
import Contacto from './pages/Contacto';
import NotFound from './pages/NotFound';

function App() {
  return (
    <div>
      <nav>
        <ul>
          <li><Link to="/">Inicio</Link></li>
          <li><Link to="/sobre-nosotros">Sobre Nosotros</Link></li>
          <li><Link to="/contacto">Contacto</Link></li>
        </ul>
      </nav>

      <Routes>
        <Route path="/" element={<Inicio />} />
        <Route path="/sobre-nosotros" element={<SobreNosotros />} />
        <Route path="/contacto" element={<Contacto />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </div>
  );
}

export default App;
```

Componentes principales de React Router:

- `BrowserRouter`: Proporciona el contexto de enrutamiento.
- `Routes`: Contenedor para las rutas.
- `Route`: Define una ruta y el componente a mostrar.
- `Link`: Crea enlaces de navegación sin recargar la página.

Consumiendo APIs

React permite consumir APIs fácilmente con métodos nativos como `fetch` o bibliotecas como Axios.

Usando `fetch`

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

function ListaDeUsuarios() {
  const [usuarios, setUsuarios] = useState([]);
  const [cargando, setCargando] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const obtenerUsuarios = async () => {
      try {
        const respuesta = await fetch('https://jsonplaceholder.typicode.com/users');

        if (!respuesta.ok) {
          throw new Error(`Error HTTP: ${respuesta.status}`);
        }

        const datos = await respuesta.json();
        setUsuarios(datos);
        setCargando(false);
      } catch (err) {
        setError(err.message);
        setCargando(false);
      }
    };
    obtenerUsuarios();
  }, []);

  if (cargando) return <p>Cargando usuarios...</p>;
  if (error) return <p>Error al cargar usuarios: {error}</p>;

  return (
    <div>
      <h2>Lista de Usuarios</h2>
      <ul>
        {usuarios.map(usuario => (
          <li key={usuario.id}>{usuario.name} - {usuario.email}</li>
        ))}
      </ul>
    </div>
  );
}
```

Usando Axios

Axios ofrece una API más conveniente para trabajar con solicitudes HTTP:

bash

 Copiar

```
npm install axios
```

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

function ListaDePublicaciones() {
  const [publicaciones, setPublicaciones] = useState([]);
  const [cargando, setCargando] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    // Cancelar solicitudes en curso si el componente se desmonta
    const source = axios.CancelToken.source();

    const obtenerPublicaciones = async () => {
      try {
        const respuesta = await axios.get('https://jsonplaceholder.typicode.com/posts', {
          cancelToken: source.token
        });

        setPublicaciones(respuesta.data);
        setCargando(false);
      } catch (err) {
        if (axios.isCancel(err)) {
          console.log('Solicitud cancelada:', err.message);
        } else {
          setError(err.message);
          setCargando(false);
        }
      }
    }

    obtenerPublicaciones();
  }, []);

  // Función de limpieza
  return () => {
    source.cancel('Componente desmontado');
  };
}, []);

if (cargando) return <p>Cargando publicaciones...</p>;
if (error) return <p>Error: {error}</p>;

return (
  <div>
    <h2>Publicaciones Recientes</h2>
```

```
{publicaciones.slice(0, 10).map(post => (
  <div key={post.id} className="post">
    <h3>{post.title}</h3>
    <p>{post.body}</p>
  </div>
))})
</div>
);
}
```

Construyendo un proyecto completo

Ahora, vamos a crear un pequeño proyecto completo: una aplicación de lista de tareas (Todo List) que combine todo lo que hemos aprendido.

Primero, crea una estructura de directorios:

 Copiar

```
src/
├── components/
│   ├── TodoForm.jsx
│   ├── TodoItem.jsx
│   └── TodoList.jsx
└── App.jsx
└── main.jsx
```

1. Componente TodoItem

```
// src/components/TodoItem.jsx
function TodoItem({ todo, toggleComplete, deleteTodo }) {
  return (
    <div className="todo-item">
      <input
        type="checkbox"
        checked={todo.completed}
        onChange={() => toggleComplete(todo.id)}
      />
      <span
        style={{
          textDecoration: todo.completed ? 'line-through' : 'none',
          color: todo.completed ? '#888' : '#000'
        }}
      >
        {todo.text}
      </span>
      <button onClick={() => deleteTodo(todo.id)}>Eliminar</button>
    </div>
  );
}

export default TodoItem;
```

2. Componente TodoForm

```
// src/components/TodoForm.jsx
import { useState } from 'react';

function TodoForm({ addTodo }) {
  const [text, setText] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    if (!text.trim()) return;

    addTodo(text);
    setText(''); // Limpiar el campo después de agregar
  };

  return (
    <form onSubmit={handleSubmit} className="todo-form">
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Nueva tarea..." />
      <button type="submit">Agregar</button>
    </form>
  );
}

export default TodoForm;
```

3. Componente TodoList

```
// src/components/TodoList.jsx
import TodoItem from './TodoItem';
import TodoForm from './TodoForm';
import { useState, useEffect } from 'react';

function TodoList() {
  // Intentar obtener tareas guardadas en localStorage
  const savedTodos = JSON.parse(localStorage.getItem('todos') || '[]');

  const [todos, setTodos] = useState(savedTodos);

  // Guardar todos en localStorage cuando cambian
  useEffect(() => {
    localStorage.setItem('todos', JSON.stringify(todos));
  }, [todos]);

  // Agregar una nueva tarea
  const addTodo = (text) => {
    const newTodo = {
      id: Date.now(),
      text,
      completed: false
    };

    setTodos([...todos, newTodo]);
  };

  // Cambiar estado de una tarea (completada/pendiente)
  const toggleComplete = (id) => {
    setTodos(
      todos.map(todo =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
      )
    );
  };

  // Eliminar una tarea
  const deleteTodo = (id) => {
    setTodos(todos.filter(todo => todo.id !== id));
  };

  return (
    <div className="todo-list">
      <h2>Lista de Tareas</h2>
```

```

<TodoForm addTodo={addTodo} />

<div className="todos">
  {todos.length === 0 ? (
    <p>No hay tareas pendientes.</p>
  ) : (
    todos.map(todo => (
      <TodoItem
        key={todo.id}
        todo={todo}
        toggleComplete={toggleComplete}
        deleteTodo={deleteTodo}
      />
    )))
  )}
</div>

<div className="todo-stats">
  <p>Total: {todos.length} tareas</p>
  <p>Completadas: {todos.filter(todo => todo.completed).length}</p>
  <p>Pendientes: {todos.filter(todo => !todo.completed).length}</p>
</div>
</div>
);

}

export default TodoList;

```

4. Componente App

```
// src/App.jsx
import TodoList from './components/TodoList';
import './App.css';

function App() {
  return (
    <div className="app">
      <header>
        <h1>Mi Lista de Tareas</h1>
      </header>
      <main>
        <TodoList />
      </main>
      <footer>
        <p>Aplicación creada con React y Vite</p>
      </footer>
    </div>
  );
}

export default App;
```

5. Estilos (App.css)

```
/* src/App.css */  
.app {  
  max-width: 600px;  
  margin: 0 auto;  
  padding: 20px;  
  font-family: Arial, sans-serif;  
}  
  
header {
```

```
  text-align: center;  
  margin-bottom: 20px;  
}  
  
footer {
```

```
  text-align: center;  
  margin-top: 40px;  
  color: #888;  
  font-size: 0.8rem;  
}
```

```
.todo-form {  
  display: flex;  
  margin-bottom: 20px;  
}
```

```
.todo-form input {  
  flex-grow: 1;  
  padding: 8px;  
  border: 1px solid #ddd;  
  border-radius: 4px 0 0 4px;  
}
```

```
.todo-form button {  
  padding: 8px 16px;  
  background-color: #4CAF50;  
  color: white;  
  border: none;  
  border-radius: 0 4px 4px 0;  
  cursor: pointer;  
}
```

```
.todo-item {  
  display: flex;  
  align-items: center;
```

```
padding: 10px;
border-bottom: 1px solid #eee;
}

.todo-item input[type="checkbox"] {
  margin-right: 10px;
}

.todo-item span {
  flex-grow: 1;
}

.todo-item button {
  padding: 5px 10px;
  background-color: #f44336;
  color: white;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

.todo-stats {
  margin-top: 20px;
  padding: 10px;
  background-color: #f5f5f5;
  border-radius: 4px;
}

.todo-stats p {
  margin: 5px 0;
}
```

Optimización y despliegue

Una vez que hayas completado tu aplicación, es hora de pensar en optimizarla y desplegarla en producción.

Optimización

Vite ya incluye muchas optimizaciones por defecto, pero aquí hay algunos consejos adicionales:

1. Lazy Loading (Carga perezosa)

Para aplicaciones más grandes, puedes dividir tu código en fragmentos más pequeños que se cargan solo cuando se necesitan:

jsx

Copiar

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

// Importaciones normales para las páginas principales
import Inicio from './pages/Inicio';

// Importaciones perezosas para páginas menos visitadas
const SobreNosotros = lazy(() => import('./pages/SobreNosotros'));
const Contacto = lazy(() => import('./pages/Contacto'));

function App() {
  return (
    <div>
      <nav>{/* ... */}</nav>

      <Suspense fallback={<div>Cargando...</div>}>
        <Routes>
          <Route path="/" element={<Inicio />} />
          <Route path="/sobre-nosotros" element={<SobreNosotros />} />
          <Route path="/contacto" element={<Contacto />} />
        </Routes>
      </Suspense>
    </div>
  );
}

}
```

2. Memoización

Para prevenir renderizados innecesarios:

jsx

Copiar

```
import { memo } from 'react';

const ComponenteCostoso = memo(function ComponenteCostoso({ valor }) {
  // Cálculos intensivos o renderizado complejo
  return <div>{/* ... */}</div>;
});

// Solo se volverá a renderizar si 'valor' cambia
```

También puedes usar `useMemo` y `useCallback` para memoizar valores y funciones:

jsx

Copiar

```
import { useMemo, useCallback } from 'react';

function MiComponente({ datos }) {
  // Valor memoizado - solo se recalcula si 'datos' cambia
  const datosProcesados = useMemo(() => {
    return datos.map(item => item * 2);
  }, [datos]);

  // Función memoizada - solo se recrea si 'datos' cambia
  const manejarClic = useCallback(() => {
    console.log('Procesando', datos);
  }, [datos]);

  return (
    <div>
      {/* ... */}
    </div>
  );
}
```

Despliegue

1. Construir para producción

Para crear una versión optimizada para producción:

bash

Copiar

```
npm run build
```

Esto generará una carpeta `dist/` con los archivos estáticos optimizados.

2. Previsualizar la versión de producción localmente

bash

Copiar

```
npm run preview
```

3. Opciones de despliegue

- **Netlify**: Integración sencilla con GitHub, despliegue automático.
- **Vercel**: Plataforma optimizada para aplicaciones React, excelente para proyectos personales.

- **GitHub Pages:** Opción gratuita para proyectos de código abierto.
- **Firebase Hosting:** Solución de Google con buena integración con otros servicios de Firebase.

Para Netlify o Vercel, simplemente conecta tu repositorio y estas plataformas reconocerán automáticamente tu proyecto de Vite.

Ejemplo para GitHub Pages

1. Instala gh-pages:

bash Copiar

```
npm install gh-pages --save-dev
```

2. Actualiza tu `vite.config.js`:

js Copiar

```
export default {  
  base: '/nombre-de-tu-repositorio/',  
  // resto de la configuración  
}
```

3. Agrega scripts a `package.json`:

json Copiar

```
"scripts": {  
  "predeploy": "npm run build",  
  "deploy": "gh-pages -d dist"  
}
```

4. Despliega:

bash Copiar

```
npm run deploy
```

Recursos adicionales

Para continuar tu aprendizaje de React y Vite:

Documentación oficial

- [Documentación de React](#)
- [Sitio oficial de Vite](#)
- [React Router](#)

Tutoriales y cursos

- [React - The Complete Guide \(Udemy\)](#)
- [Epic React de Kent C. Dodds](#)
- [Scrimba - Curso gratuito de React](#)

Herramientas y bibliotecas útiles

- [Redux Toolkit](#) - Gestión de estado
- [React Query](#) - Gestión de datos del servidor
- [Chakra UI](#) - Biblioteca de componentes
- [Framer Motion](#) - Animaciones
- [React Hook Form](#) - Manejo de formularios

Comunidad

- [Stack Overflow](#)
- [Subreddit de React](#)
- [Discord de Reactiflux](#)

Conclusión

¡Felicitaciones! Has completado un tutorial completo sobre React con Vite. Ahora deberías tener una comprensión sólida de:

- Los conceptos fundamentales de React
- Cómo usar Vite como herramienta de construcción
- Componentes, props y estado
- Hooks y gestión del ciclo de vida
- Formularios y manejo de eventos
- Enrutamiento con React Router
- Consumo de APIs
- Optimización y despliegue

El próximo paso es practicar construyendo tus propias aplicaciones. La mejor manera de aprender es haciendo. Intenta expandir la aplicación de lista de tareas que hemos creado con funcionalidades adicionales como:

- Filtrar tareas (completadas, pendientes, todas)
- Editar tareas existentes
- Categorizar tareas
- Añadir fechas límite
- Implementar un backend real con Firebase o una API propia

React es una biblioteca muy versátil y potente que tiene una curva de aprendizaje relativamente suave al principio, pero que ofrece muchísima profundidad a medida que avanzas. Continúa explorando, experimenta con diferentes patrones y, sobre todo, ¡diviértete construyendo interfaces de usuario increíbles!

¡Mucho éxito en tu viaje con React y Vite!