CORFO
BecasCapital Humano

Adalid
aspasia
LA FORMACIÓN DE TU FUTURO

MÓDULO 5

DESARROLLO DE APLICACIONES CON FRONT - END CON REACT

Manual del Módulo 5: Desarrollo de aplicaciones Front-End con React

Introducción

Este módulo se centra en la implementación de aplicaciones web completas utilizando React, enfocándose en la interacción con APIs y el uso de librerías avanzadas. Los estudiantes aprenderán a conectar el frontend con servidores backend mediante el consumo de APIs, haciendo uso de tecnologías como `Fetch API` y `Axios` para realizar peticiones HTTP asíncronas. Además, se explorará el uso avanzado de Hooks como `useEffect` y `useState`, fundamentales para gestionar el ciclo de vida de las peticiones y el estado en componentes funcionales. Se tratarán también temas relacionados con el manejo de errores y la optimización del rendimiento en aplicaciones React. Al final del módulo, los estudiantes estarán capacitados para construir aplicaciones front-end robustas y escalables, integrando APIs y aplicando buenas prácticas de desarrollo.

1. Consumo de API

1.1 El rol del front en una aplicación Cliente/Servidor

En una arquitectura cliente/servidor, el frontend (cliente) y el backend (servidor) interactúan a través de APIs. El cliente realiza solicitudes a través de HTTP y el servidor responde con datos que el frontend utiliza para actualizar la interfaz de usuario. En React, esta interacción se facilita mediante el uso de bibliotecas como `Fetch API` y `Axios`, que permiten manejar las solicitudes asíncronas, trayendo datos y actualizando el estado de la aplicación en tiempo real sin necesidad de recargar la página.

Ejemplo de flujo de datos en una aplicación cliente/servidor:

- El usuario realiza una acción (clic en un botón).
- El frontend envía una solicitud a la API del backend.
-
-
-

-
- El backend procesa la solicitud y envía una respuesta con los datos.
- El frontend recibe los datos y actualiza la interfaz de usuario.

1.2 Interacción a través de APIs

En React, la interacción con APIs se realiza generalmente a través de métodos como `fetch` o la librería `Axios`. Ambas herramientas permiten hacer peticiones HTTP (GET, POST, PUT, DELETE) y manejar respuestas de manera eficiente.

1.3 Usando el Hook `useEffect`

El Hook `useEffect` es fundamental en React para gestionar los efectos secundarios, como las peticiones HTTP, dentro de componentes funcionales. Permite ejecutar código después de que el componente se renderiza, como hacer una solicitud a una API y actualizar el estado cuando los datos se reciben.

1.3.1 ¿Qué hace `useEffect`?

El Hook `useEffect` se utiliza para realizar acciones que ocurren fuera del ciclo de renderizado de React, como las peticiones HTTP. Se ejecuta después de que el componente se monta o se actualizan sus dependencias.

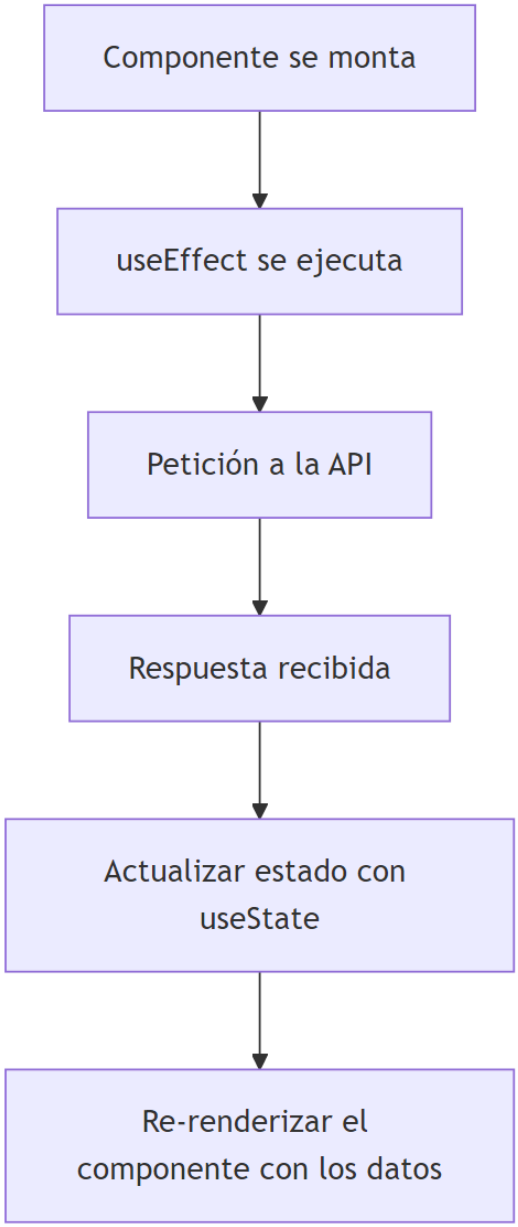
Ejemplo básico de `useEffect`: En este ejemplo, hacemos una petición a una API al montar el componente para obtener una lista de posts y mostramos estos datos en una lista.

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

function ListaDePosts() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {

    // Petición a una API para obtener posts
```

```
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then(response => response.json())
      .then(data => setPosts(data))
      .catch(error => console.error('Error:', error));

  }, []); // El array vacío significa que el efecto solo se ejecuta cuando el componente se monta

  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

export default ListaDePosts;
```

1.3.2 Omite efectos para optimizar el rendimiento

El segundo argumento de `useEffect` es el array de dependencias, que indica cuándo debe ejecutarse el efecto. Si el array está vacío (`[]`), el efecto solo se ejecutará una vez, cuando el componente se monte, lo que es útil para evitar repeticiones innecesarias de código.



1.3.3 Cómo realizar peticiones en React con `useEffect`

En aplicaciones React, las peticiones a APIs se realizan comúnmente dentro de `useEffect` para asegurarse de que ocurren después de que el componente ha sido montado.

Ejemplo:

```
useEffect(() => {

  const obtenerDatos = async () => {
    try {
      const response = await
fetch('https://jsonplaceholder.typicode.com/posts');
      const data = await response.json();
      setPosts(data);
    } catch (error) {
      console.error('Error:', error);
    }
  };

  obtenerDatos();

}, []); // Solo se ejecuta una vez
```

En este ejemplo, `fetch` se utiliza dentro de una función asíncrona que se ejecuta cuando el componente se monta.

1.3.4 Cómo realizar peticiones a partir de eventos del usuario

No todas las peticiones a APIs se hacen al montar el componente. También es común realizarlas en respuesta a eventos como clics de botones o cambios en inputs.

Ejemplo: Realizando una petición al hacer clic en un botón:

```
function BuscarUsuario() {

  const [usuario, setUsuario] = useState(null);
  const buscarUsuario = (id) => {

    fetch(`https://jsonplaceholder.typicode.com/users/${id}`)
      .then(response => response.json())
```

```
      .then(data => setUsuario(data))
      .catch(error => console.error('Error:', error));
  };

  return (
    <div>
      <button onClick={() => buscarUsuario(1)}>Buscar Usuario 1</button>
      {usuario && <p>Nombre: {usuario.name}</p>}
    </div>
  );
}
```

1.3.5 Manejando errores

Cuando trabajamos con APIs, es crucial manejar los errores de manera adecuada, tanto en peticiones exitosas como en fallidas. Esto mejora la experiencia del usuario y permite la depuración eficiente.

Ejemplo de manejo de errores con `fetch`:

```
useEffect(() => {

  const obtenerDatos = async () => {
    try {
      const response = await
fetch('https://jsonplaceholder.typicode.com/posts');

      if (!response.ok) {
        throw new Error('Error en la respuesta de la API');
      }

      const data = await response.json();
      setPosts(data);
    } catch (error) {
      console.error('Error:', error);
    }
```

```
  };
  obtenerDatos();

}, []);
```

Aquí estamos manejando los errores tanto si la petición falla (con `catch`), como si la API devuelve una respuesta con un código de error HTTP (mediante `response.ok`).

1.4 Usando el Hook `useState`

El Hook `useState` es crucial para gestionar el estado de los componentes funcionales en React. Cuando trabajamos con datos de una API, usamos `useState` para almacenar los datos obtenidos y actualizar la interfaz en consecuencia.

Ejemplo básico de `useState`:

```
const [datos, setDatos] = useState([]); // Inicializamos el estado con un array vacío
```

El valor de `datos` será actualizado cuando la API devuelva una respuesta exitosa, y la interfaz se volverá a renderizar para reflejar los nuevos datos.

1.5 Usando Fetch API

1.5.1 ¿Qué es `Fetch API`?

La `Fetch API` es una herramienta nativa de JavaScript para realizar peticiones HTTP. Es compatible con promesas, lo que permite escribir código asíncrono de manera más clara y manejable.

Ejemplo básico:

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(response => response.json()) // Convertimos la respuesta a JSON
  .then(data => console.log(data))   // Mostramos los datos en la consola
  .catch(error => console.error('Error:', error)); // Manejamos los errores
```

1.5.2 Cómo manejar errores con `Fetch API`

Es importante manejar los errores tanto en la solicitud como en la respuesta, verificando si la solicitud fue exitosa (código de estado HTTP 200-299) y gestionando excepciones cuando el servidor no responde.

Ejemplo con manejo de errores:

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(response => {
    if (!response.ok) {
      throw new Error('Error en la solicitud');
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

1.6 Usando Axios

1.6.1 Sintaxis

`Axios` es una librería de JavaScript que permite realizar peticiones HTTP de una manera más sencilla que `fetch`. Ofrece características adicionales como la configuración de tiempo de espera, interceptores de peticiones/respuestas, y maneja automáticamente la conversión de datos a JSON.

Ejemplo básico con `Axios`:

```
import axios from 'axios';

axios.get('https://json
placeholder.typicode.com/posts')
  .then(response => console.log(response.data)) // Los datos están en
response.data
  .catch(error => console.error('Error:', error));
```

1.6.2 Invocación

Con `Axios`, las peticiones pueden realizarse utilizando métodos como `get`, `post`, `put`, y `delete`. La sintaxis es sencilla y el código es más fácil de leer en comparación con `fetch`.

Ejemplo de invocación con `post`:

```
axios.post('https://jsonplaceholder.typicode.com/posts', {
  title: 'Nuevo Post',
  body: 'Este es el contenido del post',
  userId: 1
})
  .then(response => console.log(response.data))
  .catch(error => console.error('Error:', error));
```

1.6.3 Manejo de errores con `Axios`

Al igual que con `fetch`, los errores en `Axios` se manejan mediante `catch`.

Ejemplo con manejo de errores:

```
axios.get('https://jsonplaceholder.typicode.com/posts')

  .then(response => console.log(response.data))
  .catch(error => {
```

```
    if (error.response) {
      console.error('Error en la respuesta de la API:',
error.response.status);
    } else {
      console.error('Error en la solicitud:', error.message);
    }
  });
```

1.6.4 Usando `Async/Await` con `Axios`

La sintaxis de `async/await` hace que el código asíncrono sea más fácil de leer y escribir, especialmente cuando se usan múltiples promesas.

Ejemplo:

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

  try {
    const response = await
axios.get('https://jsonplaceholder.typicode.com/posts');
    console.log(response.data);
  } catch (error) {
    console.error('Error:', error);
  }
};

obtenerPosts();
```

1.7 Principales diferencias entre Fetch API y Axios

Característica	**Fetch API**	**Axios**
Soporte JSON	Necesita `response.json()`	Convierte automáticamente a JSON

Característica	Fetch API	Axios
Manejo de errores	Solo detecta errores de red	Maneja errores de red y de respuesta HTTP
Cancelación	No soporta cancelación nativa	Soporta cancelación de peticiones
Interceptors	No tiene	Sí soporta interceptores de petición y respuesta
Compatibilidad	Solo en navegadores modernos	Soporta más entornos, incluidos Node.js

2. Introducción a TypeScript

2.1 ¿Qué es TypeScript?

TypeScript es un lenguaje de programación desarrollado por Microsoft que extiende JavaScript añadiendo **tipado estático opcional**. Esto significa que, a diferencia de JavaScript, TypeScript permite declarar explícitamente los tipos de variables, funciones y objetos, lo que ayuda a prevenir errores en tiempo de compilación y mejora la calidad y mantenibilidad del código.

Ejemplo básico de TypeScript:

```
let mensaje: string = "Hola, TypeScript";

console.log(mensaje);
```

Aquí, `mensaje` es una variable de tipo `string`. Si intentamos asignar un valor de otro tipo (por ejemplo, un número), TypeScript nos advertirá sobre el error antes de ejecutar el código.

2.2 Para qué se utiliza TypeScript

TypeScript se utiliza principalmente para el **desarrollo de aplicaciones a gran escala**, donde la mantenibilidad del código es crucial. Al proporcionar un tipado fuerte y herramientas avanzadas para la refactorización, TypeScript reduce los errores comunes que pueden ocurrir en JavaScript debido a la falta de tipado. También facilita la colaboración en equipos grandes al hacer que las interfaces y estructuras de datos sean claras y comprensibles.

2.3 TypeScript en React.js

TypeScript es una herramienta poderosa cuando se combina con React, ya que proporciona un tipado fuerte que ayuda a mejorar la productividad y reducir los errores en aplicaciones React. En aplicaciones React, TypeScript se usa para:

1. **Tipar componentes**: Se pueden definir las props que espera un componente, garantizando que se pasan los tipos correctos.
2. **Definir estados**: Se puede tipar el estado interno de un componente para asegurar que solo se manipulen datos válidos.
3. **Mejorar la integración con IDEs**: TypeScript proporciona autocompletado, resaltado de errores y refactorización automática, lo que hace que el desarrollo en React sea más eficiente.

Ejemplo de componente funcional tipado en TypeScript:

```
import React from 'react';

interface Props {
  nombre: string;
  edad: number;
}

const TarjetaUsuario: React.FC<Props> = ({ nombre, edad }) => {
  return (
    <div>
      <h1>Nombre: {nombre}</h1>
      <p>Edad: {edad}</p>
    </div>
```

```
  );
};

export default TarjetaUsuario;
```

En este ejemplo, estamos usando una **interfaz** para definir las props del componente `TarjetaUsuario`. De esta manera, garantizamos que `nombre` siempre será un string y `edad` un número.

2.4 TypeScript vs. JavaScript

La principal diferencia entre TypeScript y JavaScript es el **tipado estático** que ofrece TypeScript, mientras que JavaScript tiene un tipado dinámico. TypeScript permite que los desarrolladores definen tipos para variables, parámetros y valores de retorno de funciones, lo que ayuda a evitar errores en tiempo de compilación.

Característica	**TypeScript**	**JavaScript**
Tipado	Estático, fuerte	Dinámico, débil
Verificación de tipos	En tiempo de compilación	En tiempo de ejecución
Soporte de IDE	Mejor autocompletado y refactorización	Autocompletado limitado
Compatibilidad	Compila a JavaScript	Nativo en navegadores

Ejemplo de diferencia entre TypeScript y JavaScript:

TypeScript:

```
function sumar(a: number, b: number): number {
  return a + b;
}
```

JavaScript:

```
function sumar(a, b) {
  return a + b;
}
```

En TypeScript, se declaran los tipos de los parámetros (`number`) y del valor de retorno de la función. En JavaScript, los tipos no se declaran, lo que puede llevar a errores si se pasan tipos incorrectos.

2.5 TypeScript y Webpack

Webpack es una herramienta que agrupa y compila archivos, incluidos archivos TypeScript, en JavaScript que el navegador puede ejecutar. Para usar TypeScript con Webpack, se necesita configurar el archivo `tsconfig.json` y añadir un **loader** para TypeScript, como `ts-loader`.

- **Pasos básicos para configurar TypeScript con Webpack:**

    1. Instalar las dependencias:

```
npm install typescript ts-loader webpack webpack-cli --save-dev
```

    2. Crear el archivo de configuración `webpack.config.js`:

```
const path = require('path');

module.exports = {
  entry: './src/index.tsx',

  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/,
      },
```

```
    ],
  },

  resolve: {
    extensions: ['.tsx', '.ts', '.js'],
  },

  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
};
```

3. Crear un archivo `tsconfig.json`:

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "jsx": "react",
    "strict": true
  }
}
```

Con esta configuración, Webpack compilará archivos TypeScript a JavaScript.

2.6 Definiendo tipos en TypeScript

En TypeScript, los **tipos** son una forma de restringir qué valores puede tomar una variable. Esto incluye tipos primitivos como `string`, `number`, `boolean`, así como tipos personalizados definidos por el desarrollador.

Ejemplo de tipos primitivos:

```
let nombre: string = "Carlos";
```

```
let edad: number = 30;

let esActivo: boolean = true;
```

Además, se pueden definir **tipos personalizados** usando interfaces o alias de tipos.

Ejemplo de un tipo personalizado usando `type`:

```
type Producto = {
  nombre: string;
  precio: number;

};

const nuevoProducto: Producto = {
  nombre: "Laptop",
  precio: 1200,
};
```

2.7 Tipos por inferencia

TypeScript también soporta **inferencia de tipos**, lo que significa que, en muchos casos, no es necesario declarar explícitamente el tipo de una variable, ya que TypeScript lo infiere automáticamente.

Ejemplo de inferencia de tipos:

```
let mensaje = "Hola, mundo"; // TypeScript infiere que el tipo es `string`
```

Si intentamos asignar un número a `mensaje`, TypeScript marcará un error.

2.8 Componiendo tipos

Los tipos en TypeScript pueden ser **compuestos** mediante operadores como & (intersección) y | (unión).

Ejemplo de unión de tipos (|):

```
let identificador: string | number;

identificador = "ABC123";

identificador = 123;
```

Ejemplo de intersección de tipos (&):

```
interface Persona {
  nombre: string;
}

interface Trabajador {
  empresa: string;
}

const empleado: Persona & Trabajador = {
  nombre: "Carlos",
  empresa: "TechCorp",
};
```

2.9 Sistema de tipo estructural

TypeScript utiliza un **sistema de tipos estructural**, lo que significa que un tipo es compatible con otro si sus estructuras coinciden, sin importar sus nombres.

Ejemplo de sistema de tipos estructural:

```
interface Persona {
  nombre: string;
  edad: number;
}

const persona: Persona = { nombre: "Carlos", edad: 30 };
const objeto = { nombre: "Ana", edad: 25, ciudad: "Santiago" };

// TypeScript permite asignar `objeto` a `persona` porque tiene las mismas propiedades.

const nuevaPersona: Persona = objeto;
```

Aunque `objeto` tiene propiedades adicionales, TypeScript permite la asignación porque `nombre` y `edad` coinciden.

2.10 Interfaces y Clases

En TypeScript, **interfaces** y **clases** son dos herramientas clave para definir y estructurar objetos.

- **Interfaces** definen la estructura que debe seguir un objeto, pero no implementan lógica.
    - **Ejemplo de una interfaz:**

```
interface Usuario {
  nombre: string;
  email: string;
}

const nuevoUsuario: Usuario = {
  nombre: "Carlos",
  email: "carlos@correo.com",
};
```

- **Clases** son una forma de definir objetos con propiedades y métodos. En TypeScript, se pueden tipar las propiedades y los parámetros de los métodos.

    - **Ejemplo de clase:**

```
class Persona {

  nombre: string;
  edad: number;

  constructor(nombre: string, edad: number) {
    this.nombre = nombre;
    this.edad = edad;
  }

  saludar() {
    return `Hola, mi nombre es ${this.nombre}`;
  }
}

const persona = new Persona("Carlos", 30);
console.log(persona.saludar());
```

2.11 Algunos Frameworks que soportan TypeScript (Next.js, Gatsby.js)

Algunos frameworks populares en el ecosistema React que soportan TypeScript son **Next.js** y **Gatsby.js**. Ambos permiten integrar TypeScript de manera sencilla, proporcionando plantillas y configuraciones para comenzar rápidamente.

- **Next.js**: Un framework de React que permite el **renderizado del lado del servidor (SSR)** y la **generación estática de sitios web (SSG)**. Con soporte nativo para TypeScript, facilita la creación de aplicaciones escalables y optimizadas para SEO.

    - **Documentación oficial**: https://nextjs.org/docs/basic-features/typescript

- **Gatsby.js**: Un framework que permite la **generación estática** de sitios utilizando React y GraphQL. Gatsby ofrece plantillas y configuración inicial para proyectos en TypeScript.

    - **Documentación oficial**: https://www.gatsbyjs.com/docs/how-to/custom-configuration/typescript/

3. Seguridad en un Aplicativo Front-End

3.1 Conceptos básicos de seguridad en aplicaciones Web

La seguridad en aplicaciones web es crucial para proteger los datos y evitar accesos no autorizados. A continuación, se describen algunos de los ataques más comunes que pueden afectar una aplicación web:

1. **Clickjacking**: Este ataque ocurre cuando un usuario es engañado para hacer clic en un elemento de la interfaz sin darse cuenta, como un botón o enlace oculto dentro de un iframe. Esto puede llevar al usuario a realizar acciones no deseadas, como enviar datos o iniciar una transacción.

    - **Prevención**: Se puede mitigar usando el encabezado `X-Frame-Options: DENY` para evitar que la página sea cargada en un iframe.

2. **Cross-Site Scripting (XSS)**: Ocurre cuando un atacante inyecta código malicioso en una página web que es ejecutado en el navegador de otros usuarios. Esto permite al atacante robar datos sensibles o realizar acciones no autorizadas en nombre del usuario.

    - **Prevención**: Sanear (limpiar) todas las entradas de los usuarios y escapar las salidas (output escaping) para prevenir la inyección de scripts.

3. **SQL Injection**: Un ataque donde el atacante inserta consultas SQL maliciosas en los campos de entrada de una aplicación. Esto puede permitirle al atacante acceder, modificar o eliminar datos en la base de datos.

    - **Prevención**: Utilizar consultas preparadas (parameterized queries) y evitar concatenar directamente datos del usuario en las consultas SQL.

4. **Denegación de Servicio (DoS)**: Este ataque busca hacer que una aplicación o un servidor no esté disponible sobrecargándolo con tráfico o solicitando muchos recursos.

    - **Prevención**: Implementar mecanismos de **rate limiting** para controlar el número de solicitudes permitidas por cliente, además de usar soluciones de mitigación como **firewalls** y servicios de protección contra DoS.

3.2 Recomendaciones de seguridad en una aplicación Web

Para asegurar una aplicación web, se deben seguir varias recomendaciones generales que ayudan a mitigar posibles vulnerabilidades:

- **Sanitización de entradas**: Todas las entradas de los usuarios deben ser limpiadas y verificadas antes de ser procesadas, evitando así la inyección de código malicioso.
- **Uso de HTTPS**: HTTPS cifra la comunicación entre el cliente y el servidor, protegiendo los datos sensibles como contraseñas e información personal.
- **Control de acceso**: Implementar una gestión de roles y permisos para asegurar que los usuarios solo accedan a los recursos que tienen autorizados.
- **Autenticación segura**: Utilizar autenticación multifactor (MFA) y tokens como JWT (JSON Web Tokens) para asegurar que solo los usuarios autenticados puedan acceder a ciertas áreas de la aplicación.

3.3 Recomendaciones de seguridad en una aplicación ReactJs

Las aplicaciones front-end creadas con React también son vulnerables a ataques comunes, por lo que es importante seguir buenas prácticas de seguridad específicas para este entorno:

- **Escapar las salidas en JSX**: React automáticamente escapa cualquier dato insertado en JSX para prevenir ataques XSS. Sin embargo, se debe evitar renderizar datos sin sanitizar en componentes React.

```
const data = "<script>alert('XSS');</script>";

return <div>{data}</div>;  // React escapa automáticamente
```

- **Evitar el uso de `dangerouslySetInnerHTML`**: Este atributo permite insertar HTML sin escapado, lo que puede facilitar ataques XSS si no se usa correctamente. Debe evitarse a menos que sea absolutamente necesario, y siempre se debe asegurar que el contenido esté bien sanitizado.

```
<div dangerouslySetInnerHTML={{ __html: content }} />
```

- **Uso de encabezados de seguridad**: Implementar encabezados de seguridad como `Content-Security-Policy (CSP)` para limitar la ejecución de scripts de fuentes no confiables.

3.4 Identificando vulnerabilidades en una aplicación ReactJs

La detección de vulnerabilidades en una aplicación React se puede hacer a través de herramientas automatizadas de análisis de seguridad y mediante auditorías manuales del código:

- **Herramientas de análisis estático**: Existen herramientas como **ESLint** y **SonarQube** que ayudan a detectar patrones de código inseguros y posibles vulnerabilidades en el código fuente.
- **SAST (Static Application Security Testing)**: Utilizar soluciones de pruebas de seguridad estática para identificar vulnerabilidades en el código JavaScript antes de que lleguen al entorno de producción.
- **OWASP Dependency-Check**: Herramienta para revisar las dependencias de tu aplicación en busca de vulnerabilidades conocidas en librerías de terceros.

3.5 Consumiendo servicios REST con Api Key y JWT

API Key y **JWT (JSON Web Token)** son mecanismos de autenticación utilizados para asegurar que solo usuarios o aplicaciones autorizadas puedan acceder a los servicios REST.

- **API Key**: Es un identificador único que se envía en las solicitudes HTTP para autenticar a la aplicación o al usuario.

```
fetch('https://api.example.com/data', {

  method: 'GET',
  headers: {
    'Authorization': 'Bearer your-api-key',
  }
})

.then(response => response.json())
.then(data => console.log(data));
```

- **JWT**: Un token JWT es un objeto firmado que contiene información sobre el usuario o la sesión. Se utiliza para autenticación sin necesidad de almacenar el estado en el servidor.

  Ejemplo de uso de JWT en React:

```
// Al autenticar al usuario, recibimos un JWT del backend
const token = "jwt-token-here";

// Al hacer peticiones protegidas, incluimos el token en los encabezados
fetch('https://api.example.com/private-data', {
  method: 'GET',

  headers: {
    'Authorization': `Bearer ${token}`,
  }
})

.then(response => response.json())
.then(data => console.log(data));
```

3.6 Cómo proteger rutas con React Router DOM

En aplicaciones React que utilizan **React Router DOM** para la navegación, es esencial proteger las rutas sensibles, permitiendo solo a los usuarios autenticados acceder a ellas. Esto se puede hacer implementando **Rutas Protegidas** (Private Routes).

Ejemplo de Rutas Protegidas con React Router DOM:

```
import { Route, Redirect } from 'react-router-dom';

const RutaProtegida = ({ component: Component, ...rest }) => (
  <Route {...rest} render={props =>
    localStorage.getItem('authToken') ? (
      <Component {...props} />
    ) : (
      <Redirect to="/login" />
    )
  } />
);
```

En este ejemplo, solo los usuarios que tienen un `authToken` almacenado en `localStorage` pueden acceder a la ruta protegida.

3.7 Implementando seguridad por Roles en React

El control de acceso basado en roles (RBAC) permite definir distintos niveles de acceso para diferentes tipos de usuarios en la aplicación.

Ejemplo de implementación de roles en React:

```
const userRoles = {
  ADMIN: 'admin',
  USER: 'user',
};

const RutaPorRol = ({ component: Component, roles, ...rest }) => (
```

```
  <Route {...rest} render={props =>
    localStorage.getItem('authToken') &&
roles.includes(localStorage.getItem('role')) ? (
      <Component {...props} />
    ) : (
      <Redirect to="/not-authorized" />
    )
  } />
);
```

Aquí se verifica si el usuario tiene un rol adecuado antes de permitirle acceder a ciertas rutas.

3.8 La seguridad en la autenticación de usuarios

La autenticación segura es crucial para proteger a los usuarios y sus datos. Algunas de las mejores prácticas incluyen:

1. **Usar siempre HTTPS**: Para garantizar que los datos de autenticación estén cifrados.
2. **Autenticación multifactor (MFA)**: Añadir una capa adicional de seguridad.
3. **Tokens cortos y expira**: Utilizar tokens JWT con una vida útil limitada y mecanismos de actualización para mejorar la seguridad.

3.9 Encriptación de datos en el front

Aunque el frontend no es un lugar ideal para encriptar datos sensibles, en algunos casos es útil aplicar **cifrado** antes de enviar datos al backend. Esto se puede hacer utilizando bibliotecas como **CryptoJS**.

Ejemplo básico de cifrado en el front con CryptoJS:

```
import CryptoJS from 'crypto-js';

const mensaje = "Este es un mensaje confidencial";
const clave = "clave-secreta";
```

```
// Cifrado

const mensajeCifrado = CryptoJS.AES.encrypt(mensaje, clave).toString();
console.log("Cifrado:", mensajeCifrado);

// Descifrado

const bytes = CryptoJS.AES.decrypt(mensajeCifrado, clave);
const mensajeDescifrado = bytes.toString(CryptoJS.enc.Utf8);

console.log("Descifrado:", mensajeDescifrado);
```

En este ejemplo, los datos se cifran antes de enviarse y luego se descifran en el backend, agregando una capa adicional de seguridad.

4. Hooks

Los hooks en React son una característica avanzada que permite usar estado y otras funcionalidades de React sin escribir una clase. A continuación, se desarrolla cada subtema de manera detallada:

4.1 ¿Qué son los Hooks?

Los hooks son funciones que permiten usar características de React, como el estado o el ciclo de vida, en componentes funcionales. Anteriormente, estas funcionalidades solo estaban disponibles en los componentes de clase. Los hooks eliminan la necesidad de clases y permiten una manera más sencilla de reutilizar lógica entre componentes.

Ejemplo básico de un hook (`useState`):

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

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

```
  return (
    <div>
      <p>Haz clic {contador} veces</p>
      <button onClick={() => setContador(contador + 1)}>
        Incrementar
      </button>
    </div>
  );
}
```

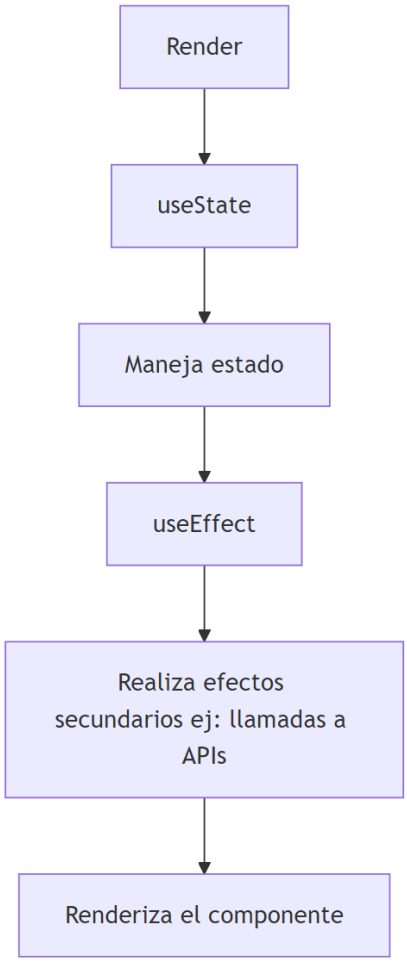
Este ejemplo utiliza el **hook `useState`** para manejar el estado de `contador`. Cada vez que el botón es clicado, el estado cambia.

4.2 Un vistazo en detalle a los Hooks

Existen varios hooks integrados en React, cada uno con una funcionalidad específica. Los más importantes son:

- **useState**: Permite añadir y gestionar estado local en componentes funcionales.
- **useEffect**: Gestiona efectos secundarios, como llamadas a APIs, eventos o subscripciones.
- **useContext**: Facilita el acceso al contexto en cualquier parte de la aplicación sin tener que pasar props.
- **useReducer**: Similar a `useState`, pero más adecuado para manejar lógica compleja de estado, como un `reducer` en Redux.

Diagrama básico de uso de hooks en un componente:



4.3 Usando el Hook de estado (`useState`)

El hook `useState` es el más común y permite gestionar el estado dentro de componentes funcionales.

Ejemplo:

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

function EjemploEstado() {
  // Inicializamos el estado con un valor de 0
```

```
  const [valor, setValor] = useState(0);

  return (
    <div>
      <p>El valor actual es {valor}</p>
      <button onClick={() => setValor(valor + 1)}>Incrementar</button>
      <button onClick={() => setValor(valor - 1)}>Decrementar</button>
    </div>
  );
}
```

- `useState(0)` crea un estado inicial de `0` y retorna un array con el estado actual y una función para actualizarlo (`setValor`).
- Al presionar los botones, el estado se actualiza y React vuelve a renderizar el componente con el nuevo valor.

4.4 Usando el Hook de efecto (`useEffect`)

`useEffect` permite ejecutar efectos secundarios, como la obtención de datos desde una API o la manipulación del DOM, en un componente funcional.

Ejemplo:

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

function FetchData() {
  const [data, setData] = useState(null);

  useEffect(() => {
    // Llamada a una API cuando el componente se monta
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data));
  }, []); // El array vacío asegura que solo se ejecute una vez (al montar
```

```
el componente)

  if (!data) {
    return <p>Cargando datos...</p>;
  }

  return (
    <div>
      <h1>Datos:</h1>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}
```

- `useEffect` se ejecuta después de que el componente se renderiza. En este caso, se realiza una llamada a una API externa.
- El array vacío `[]` hace que el efecto solo se ejecute una vez, al montar el componente. Si se ponen variables dentro del array, el efecto se ejecutará cuando esas variables cambien.

4.5 Reglas de los Hooks

Para evitar errores en el uso de hooks, React define algunas reglas básicas:

1. **Llamar a los hooks en el nivel superior**: No se deben llamar dentro de bucles, condicionales o funciones anidadas. Deben estar siempre en el cuerpo principal del componente.
2. **Solo en componentes funcionales o custom hooks**: No se pueden usar hooks en clases o funciones normales.

Estas reglas aseguran que React pueda seguir el orden en el que los hooks son invocados, lo cual es crucial para mantener la lógica de estado correcta.

4.6 Construyendo Hooks personalizados

Puedes crear tus propios hooks para reutilizar lógica que dependa de los hooks internos. Los custom hooks son simplemente funciones que pueden usar otros hooks.

Ejemplo de un hook personalizado:

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

// Hook personalizado que realiza una llamada a una API
function useFetch(url) {

  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch(url)
      .then(response => response.json())
      .then(data => {
        setData(data);
        setLoading(false);
      });
  }, [url]);
  return { data, loading };
}

// Componente que utiliza el hook personalizado
function MostrarDatos() {
  const { data, loading } = useFetch('https://api.example.com/data');
  if (loading) return <p>Cargando...</p>;
  return <div>{JSON.stringify(data, null, 2)}</div>;

}
```

- `useFetch` es un hook personalizado que encapsula la lógica para obtener datos de una API.
- Puede ser reutilizado en diferentes componentes que necesiten hacer la misma operación sin duplicar código.

5. Manejo y Límite de Errores

El manejo de errores es una parte esencial en el desarrollo de aplicaciones robustas con React. A medida que las aplicaciones crecen en complejidad, es fundamental implementar estrategias para capturar, manejar y prevenir errores, tanto en la lógica de negocio como en la interacción del usuario. Los hooks proporcionan herramientas y patrones específicos para abordar estos problemas de manera eficaz.

5.1 Describe el concepto de hook y sus utilidades de acuerdo al entorno React

Un **hook** es una función especial de React que permite acceder a funcionalidades clave como el estado o los efectos secundarios en componentes funcionales, sin la necesidad de usar clases. Los hooks son fundamentales en React, ya que permiten:

- **Manipular el estado** de un componente (por ejemplo, con `useState`).
- **Gestionar efectos secundarios** como llamadas a APIs o suscripciones (`useEffect`).
- **Acceder al contexto** de la aplicación globalmente (`useContext`).
- **Optimizar el rendimiento** con funciones como `useCallback` y `useMemo`.

Los hooks también son útiles para crear **hooks personalizados**, que encapsulan lógica repetitiva y permiten su reutilización en diferentes componentes.

5.2 Identifica los conceptos de Hook de Estado y de Hook de Efecto en un aplicativo React

Los hooks de estado y efecto son dos de los hooks más comunes y útiles en React.

- **Hook de Estado (`useState`)**: Permite manejar el estado local de un componente funcional. Cada vez que se actualiza el estado, el componente se vuelve a renderizar con los nuevos valores.

Ejemplo:

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

En este ejemplo, el estado `contador` se actualiza cada vez que el usuario hace clic en el botón.

- **Hook de Efecto (`useEffect`)**: Se utiliza para gestionar efectos secundarios como llamadas a APIs o manipulación del DOM después de que el componente haya sido renderizado.

Ejemplo:

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

function DatosUsuario({ userId }) {
  const [usuario, setUsuario] = useState(null);

  useEffect(() => {
    fetch(`https://jsonplaceholder.typicode.com/users/${userId}`)
      .then(response => response.json())
      .then(data => setUsuario(data));
  }, [userId]); // Solo se ejecuta cuando cambia userId
```

```
  return usuario ? <div>{usuario.name}</div> : <p>Cargando...</p>;
}
```

Aquí, `useEffect` obtiene datos de una API cuando el componente se monta o cuando `userId` cambia.

5.3 Reconoce los mecanismos para el manejo de errores en un aplicativo React

En React, es esencial capturar y manejar errores de manera adecuada, tanto a nivel de lógica de negocio como a nivel de la interfaz de usuario. Los principales mecanismos para el manejo de errores incluyen:

1. **Captura de errores en efectos (`useEffect`)**: Siempre que realices llamadas asíncronas dentro de `useEffect`, es crucial capturar los errores que puedan ocurrir.

   Ejemplo:

```
useEffect(() => {

  const fetchData = async () => {
    try {
      const response = await
fetch('https://jsonplaceholder.typicode.com/users');
      const data = await response.json();
      setData(data);
    } catch (error) {
      setError('Error al cargar los datos');
    }
  };
  fetchData();

}, []);
```

2. **Error Boundaries (Fronteras de error)**: Para manejar errores en componentes hijos, React utiliza componentes de clase conocidos como "Error Boundaries" que capturan errores de renderización, en métodos del ciclo de vida o en sus hijos.

   Ejemplo de Error Boundary:

```
class ErrorBoundary extends React.Component {

  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    console.error('Error capturado:', error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Algo salió mal.</h1>;
    }

    return this.props.children;
  }
}
```

5.4 Utiliza hooks en un aplicativo React para resolver un problema

A continuación, se muestra cómo utilizar `useState` y `useEffect` para manejar un problema común en una aplicación: obtener y gestionar datos desde una API externa.

Ejemplo de aplicación:

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

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

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

        if (!response.ok) {
          throw new Error('Error en la respuesta de la API');
        }

        const data = await response.json();
        setUsuarios(data);
      } catch (err) {
        setError(err.message);
      } finally {
        setCargando(false);
      }
    };

    obtenerUsuarios();
  }, []);

  if (cargando) {
    return <p>Cargando...</p>;
  }

  if (error) {
    return <p>Error: {error}</p>;
```

```
  }

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

- **useState** gestiona los estados de la lista de usuarios, los errores y el estado de carga.
- **useEffect** realiza una llamada a la API para obtener los datos, maneja posibles errores con un `try/catch` y asegura que el estado de "cargando" se actualice correctamente.

5.5 Utiliza mecanismos disponibles en el entorno React para el manejo y control de los errores y excepciones

Para mejorar el manejo de errores en React, podemos combinar **Error Boundaries** con hooks como `useEffect` y utilizar herramientas de depuración o monitoreo para obtener una visión completa de los errores en la aplicación.

1. **Error Boundaries**: Capturan errores que ocurren durante la renderización, en los métodos del ciclo de vida o en los componentes hijos. Esto es útil para evitar que errores en una parte de la aplicación afecten a toda la interfaz.

2. **Captura de errores en hooks personalizados**: Si creamos hooks personalizados que dependen de efectos secundarios o llamadas a APIs, es crucial manejar los errores dentro de estos hooks.

   Ejemplo de hook personalizado con manejo de errores:

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

function useFetch(url) {
```

```
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch(url);

        if (!response.ok) {
          throw new Error('Error en la respuesta de la API');
        }

        const result = await response.json();
        setData(result);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };
    fetchData();
  }, [url]);

  return { data, error, loading };
}
```

En este hook personalizado, capturamos errores dentro de la lógica de obtención de datos y los retornamos al componente que utilice el hook.

Aquí tienes una lista detallada de **material de referencia** para los temas que mencionaste, enfocados en React, hooks, consumo de APIs, TypeScript, y seguridad en aplicaciones front-end. Este material incluye libros, recursos online, y videos que pueden complementar tu aprendizaje.

Material de Referencia

Libros

- **"Pro React 16"** de Adam Freeman
  Este libro ofrece una guía completa sobre React, cubriendo tanto los fundamentos como temas avanzados como hooks, el manejo de estado, y la integración con APIs. Excelente recurso para desarrolladores que buscan dominar React desde cero.

- **"Learning React: Modern Patterns for Developing React Apps"** de Alex Banks y Eve Porcello
  Este libro profundiza en el uso de **hooks** como `useState` y `useEffect`, la integración con APIs mediante **Fetch API** y **Axios**, y la construcción de aplicaciones con **TypeScript**. Es perfecto para aprender cómo interactuar con APIs y manejar el estado de manera eficiente en aplicaciones React.

- **"TypeScript Quickly"** de Yakov Fain y Anton Moiseev
  Cubre todos los aspectos fundamentales de TypeScript, desde la definición de tipos hasta la integración con **React.js**. Es ideal para entender las ventajas de utilizar TypeScript en el desarrollo de aplicaciones React.

- **"Securing Node Applications"** de Liatrio
  Aunque enfocado en Node.js, este libro contiene secciones clave sobre **seguridad en aplicaciones web** que también son aplicables a las aplicaciones front-end desarrolladas con React. Cubre temas como la protección contra ataques XSS, CSRF y la autenticación segura con **JWT**.

Enlaces a Recursos Online

- **Documentación oficial de React**: El recurso más confiable y actualizado sobre React, que cubre todos los aspectos desde los componentes y JSX hasta el uso de hooks y la integración con APIs. Tiene secciones específicas sobre hooks (`useState`, `useEffect`), manejo de errores, y optimización de rendimiento.

- **MDN Web Docs: Fetch API**: Documentación detallada de **Fetch API**, que incluye cómo hacer peticiones HTTP, manejar errores y manejar respuestas en aplicaciones front-end. Ideal para entender cómo trabajar con APIs en React.

- **Axios GitHub Repository**: Página oficial de Axios, una biblioteca ampliamente usada para realizar peticiones HTTP en lugar de `fetch`. Incluye ejemplos de cómo manejar errores, hacer peticiones asíncronas y trabajar con promesas en React.

- **TypeScript Handbook**: Documentación oficial de **TypeScript**, que explica cómo integrar TypeScript con React, crear componentes tipados, definir interfaces y manejar tipos de datos complejos.

- **OWASP Top Ten**: Un excelente recurso para aprender sobre **seguridad web**. Proporciona detalles sobre los ataques más comunes como **XSS**, **SQL Injection**, y estrategias para mitigarlos en aplicaciones front-end, incluyendo recomendaciones para React.

- **JWT.io**: Plataforma que explica el uso de **JWT** (JSON Web Tokens) para la autenticación en aplicaciones web. Incluye ejemplos prácticos de cómo implementar autenticación segura en aplicaciones React mediante JWT.

Videos Recomendados

- **ReactJS Crash Course - Traversy Media**: Un curso introductorio de React que cubre los conceptos básicos como componentes, hooks (`useState`, `useEffect`), y la integración con APIs. Es un gran recurso para comenzar a trabajar con React desde lo fundamental.

- **React Hooks y useEffect explicado - Academind**: Este video cubre en detalle cómo funciona **useEffect**, cómo realizar peticiones a APIs, y cómo optimizar los efectos secundarios para mejorar el rendimiento de las aplicaciones.

- **Introduction to TypeScript - Fireship**: Un tutorial rápido y efectivo que explica los fundamentos de TypeScript y cómo utilizarlo en proyectos de React. Ideal para aprender a definir tipos, interfaces, y manejar TypeScript junto con React.

- **Manejo de Seguridad en Aplicaciones Web - Traversy Media**: Un tutorial sobre las mejores prácticas de **seguridad web**, incluyendo cómo proteger rutas con React Router, cómo manejar autenticación con JWT, y cómo implementar roles de usuario en una aplicación React.

Otros Recursos Útiles

- **Awesome React**: Una lista en constante actualización con recursos, librerías, tutoriales, y artículos sobre **React**, cubriendo desde fundamentos hasta temas avanzados como hooks personalizados y manejo de estado.

- **React Patterns**: Una recopilación de patrones de diseño comunes en React, que muestra cómo estructurar componentes, optimizar rendimiento, y manejar la interacción con APIs mediante hooks.

CORFO
BecasCapital Humano

Adalid
aspasia
LA FORMACIÓN DE TU FUTURO

MÓDULO 5

DESARROLLO DE APLICACIONES CON FRONT - END CON REACT