Módulo 5
Desarrollo de aplicaciones Front-End con React

Consumo de API

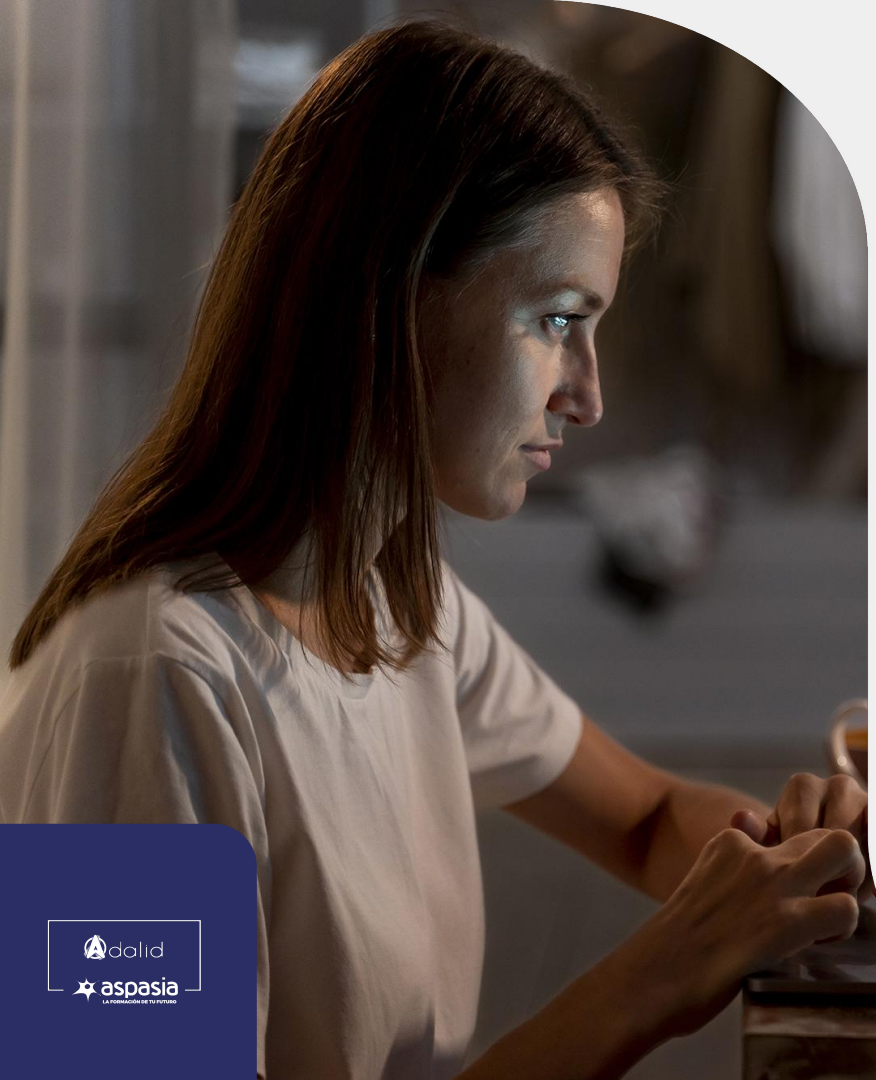
Módulo 5

AE 1.1

OBJETIVOS

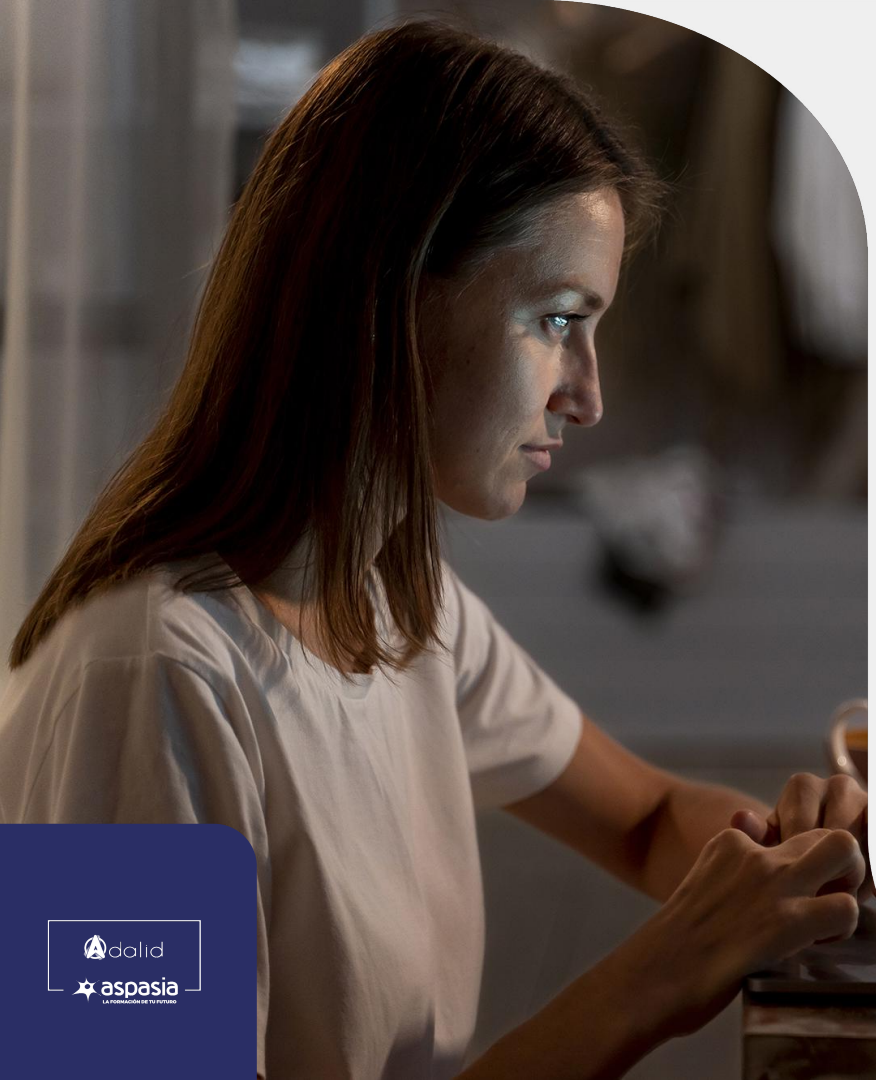
Aprender a consumir APIs en React usando Fetch y Axios, gestionando estado con hooks como useEffect y useState, para construir interfaces dinámicas y eficientes.

Chile
Adalid
aspasia
LA FORMACIÓN DE TU FUTURO



¿QUÉ VAMOS A VER?

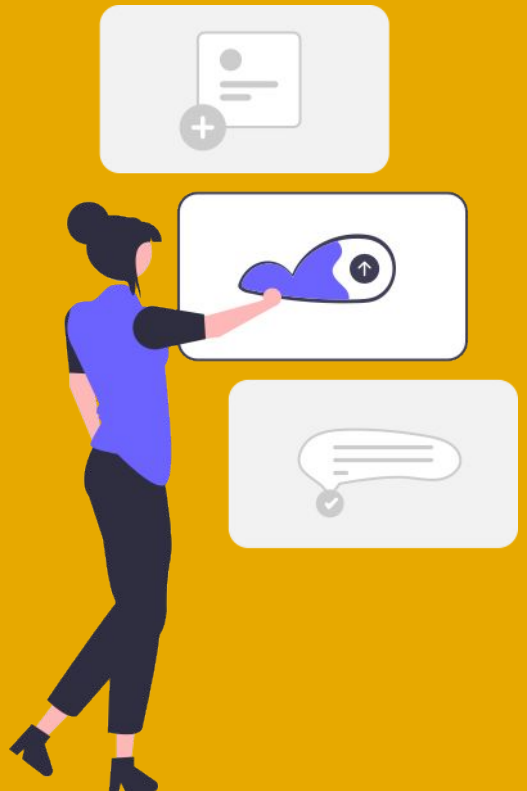
- Consumo de API.
- El rol del front en una aplicación Cliente/Servidor.
- Interacción a través de APIs.
- Usando el Hook useEffect.
  - Qué hace useEffect.
  - Omite efectos para optimizar el rendimiento.
  - Cómo realizar peticiones en React con useEffect.
  - Cómo realizar peticiones a partir de eventos del usuario.
  - Manejando errores.



¿QUÉ VAMOS A VER?

- Usando el Hook use Sate.
- Usando Fetch API.
  - Qué es Fetch API.
  - Cómo utilizar Fetch API.
  - Cómo manejar errores con Fetch API.
- Usando Axios.
  - Sintaxis.
  - Invocación.
  - Manejo de errores.
  - Usando Async/Await.
- Principales diferencias entre Fetch.
- API y Axios

¿Cuales son las solicitudes que hace una API?



Consumo de API

Adalid
aspasia
LA FORMACIÓN DE TU FUTURO

Consumo de API

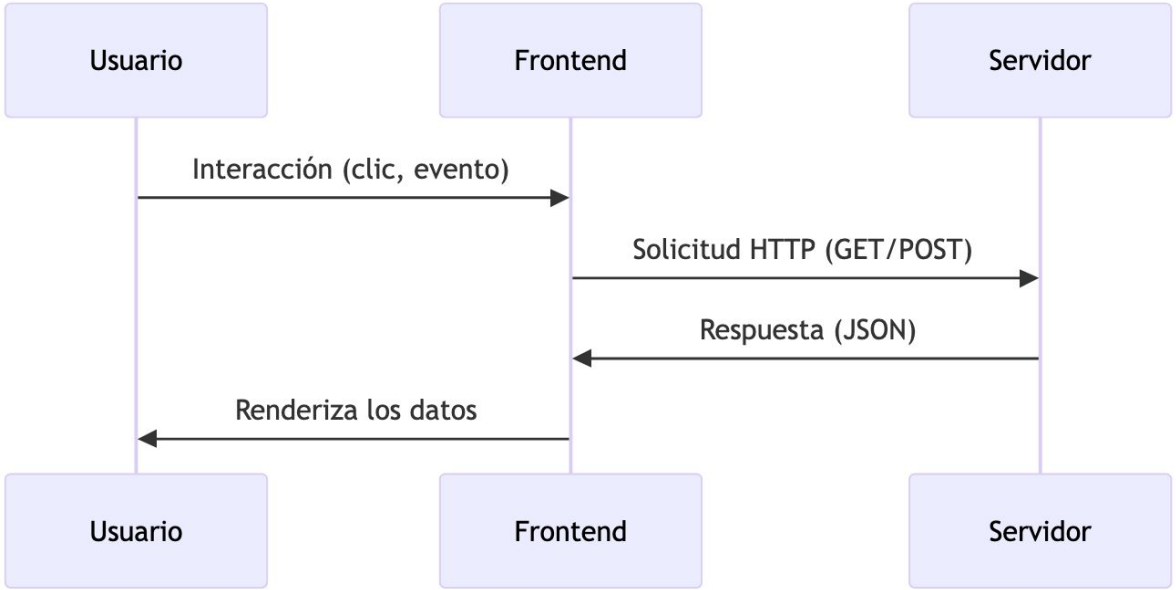
Consumir una API en React implica realizar **solicitudes HTTP** desde una aplicación cliente (frontend) para interactuar con un servidor. Esto puede incluir obtener, enviar, actualizar o eliminar datos.

- Permite integrar datos externos y construir aplicaciones que se actualizan en tiempo real.
- Permite usar APIs públicas como mapas, clima, o autenticación.
- Separa la lógica del frontend y el backend, lo que facilita el mantenimiento.

El rol del front en una aplicación Cliente/Servidor

En una arquitectura cliente/servidor, el frontend (cliente) interactúa directamente con el usuario y solicita recursos al backend. El servidor (backend) **procesa estas solicitudes y responde con datos** que el frontend utiliza para actualizar la interfaz.

El rol del front en una aplicación Cliente/Servidor



Interacción a través de APIs

React utiliza herramientas como **Fetch** o **Axios** para enviar solicitudes HTTP al servidor. Estas solicitudes pueden ser **síncronas o asíncronas** y se manejan dentro del ciclo de vida del componente usando hooks como useEffect.

Interacción a través de APIs

Pasos básicos para consumir una API en React:

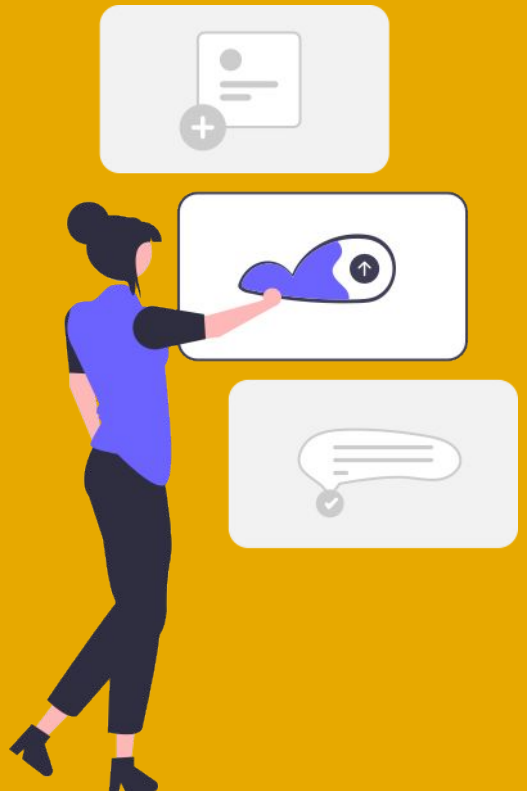
1. **Identificar el endpoint:** Definir la URL del recurso (ej.: /posts).
2. **Elegir el método HTTP:** GET (obtener), POST (crear), PUT (actualizar) o DELETE (eliminar).
3. **Enviar la solicitud:** Usar Fetch o Axios.
4. **Procesar la respuesta:** Convertir los datos JSON en estados manejables por React.
5. **Renderizar los datos:** Mostrar los datos en el DOM.

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

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

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

  return (
    <div>
      <h1>Posts:</h1>
      {posts.map((post) => (
        <div key={post.id}>
          <h2>{post.title}</h2>
          <p>{post.body}</p>
        </div>
      ))}
    </div>
  );
}
```



Usando el Hook useEffect

¿Qué hace useEffect?

useEffect es un hook que permite **manejar efectos secundarios** como:

- Cargar datos de una API.
- Configurar listeners o subscripciones.
- Actualizar el DOM.

¿Qué hace useEffect?

Ciclo de vida del componente:

- **Montaje:** Se ejecuta cuando el componente se renderiza por primera vez.
- **Actualización:** Puede ejecutarse al cambiar ciertas dependencias.
- **Desmontaje:** Se utiliza para limpiar efectos secundarios.

```
useEffect(() => {
  console.log('Componente montado');
  return () => console.log('Componente desmontado');
}, []);
```

Omite efectos para optimizar el rendimiento

Agregar un arreglo de dependencias permite que useEffect se ejecute **solo cuando esas dependencias cambian**, evitando renders innecesarios.

```
useEffect(() => {
  console.log('Ejecutado porque "count" cambió');
}, [count]);
```

Cómo realizar peticiones con useEffect

```
useEffect(() => {
  fetch('https://api.example.com/data')
    .then((response) => response.json())
    .then((data) => console.log(data));
}, []);
```

El arreglo vacío **[]** asegura que la solicitud se haga solo una vez al montar el componente.

Peticiones a partir de eventos del usuario

Puedes desencadenar solicitudes HTTP mediante **eventos como clics**.

```
function handleFetch() {
  fetch('https://api.example.com/data')
    .then((response) => response.json())
    .then((data) => console.log(data));
}
```

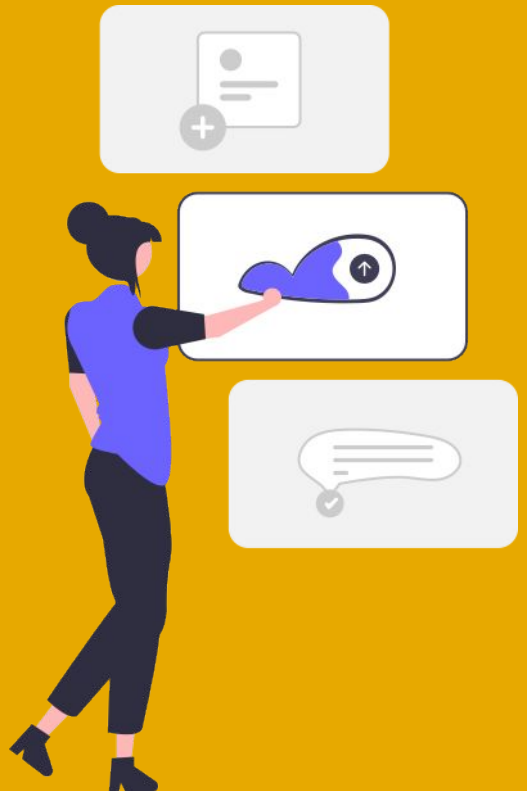
Manejando errores con useEffect

El manejo de errores asegura que la aplicación no falle a**nte respuestas inesperadas**.

```
useEffect(() => {
  fetch('https://api.example.com/data')
    .then((response) => {
      if (!response.ok) throw new Error('Error en la solicitud');
      return response.json();
    })
    .catch((error) => console.error('Error:', error));
}, []);
```

Explicación del código:

- **if (!response.ok):** Verifica si la respuesta HTTP es exitosa.
- **throw new Error:** Lanza un error si no es así.
- **catch:** Captura y maneja el error para evitar que la aplicación se detenga.



Usando el Hook useState

¿Qué es useState?

useState es un **hook en React** que permite manejar el **estado local de un componente funcional**. Cada vez que se actualiza el estado, React vuelve a renderizar el componente para reflejar los cambios.

```
const [state, setState] = useState(initialValue);
```

Explicación del código:

- **state:** El valor actual del estado.
- **setState:** Función que actualiza el estado.
- **initialValue:** Valor inicial del estado.

¿Qué es useState?

Ejemplo práctico:

Explicación del código:

- **useState(0):** Define el estado inicial (count) como 0.
- **increment y decrement:** Funciones que actualizan el estado sumando o restando 1.
- **onClick:** Llama a estas funciones al hacer clic en los botones, actualizando el contador dinámicamente.

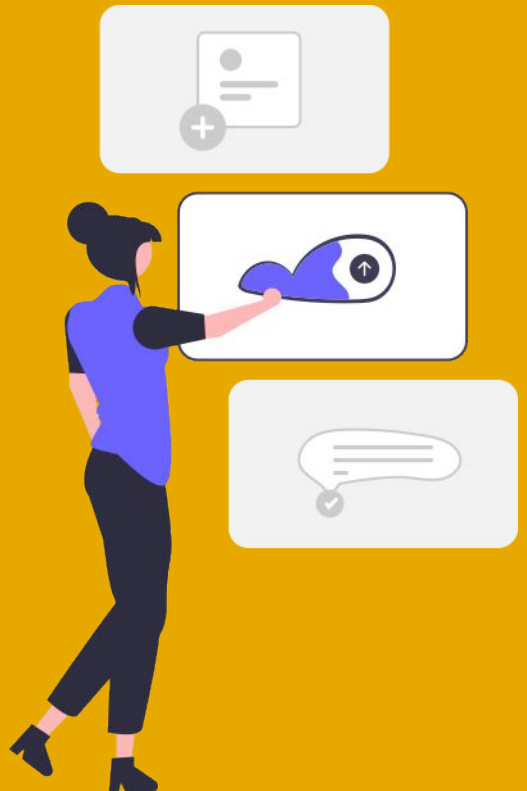
```jsx
import React, { useState } from 'react';

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

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);

  return (
    <div>
      <h1>Contador: {count}</h1>
      <button onClick={increment}>Incrementar</button>
      <button onClick={decrement}>Decrementar</button>
    </div>
  );
}

export default Counter;
```



Usando Fetch API

adalid
aspasia

¿Qué es Fetch API?

Fetch es una **API nativa de JavaScript** que permite realizar **solicitudes HTTP** de forma sencilla. Está **integrada en todos los navegadores modernos** y admite operaciones asíncronas.

Cómo utilizar Fetch API

useState es un **hook en React** que permite manejar el **estado local de un componente funcional**. Cada vez que se actualiza el estado, React vuelve a renderizar el componente para reflejar los cambios.

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

Explicación del código:

- **fetch(url):** Realiza una solicitud GET a la URL proporcionada.
- **.then(response => response.json()):** Convierte la respuesta en un objeto JSON.
- **.then(data => ...):** Procesa los datos obtenidos.
- **.catch(error => ...):** Captura y maneja cualquier error durante la solicitud.

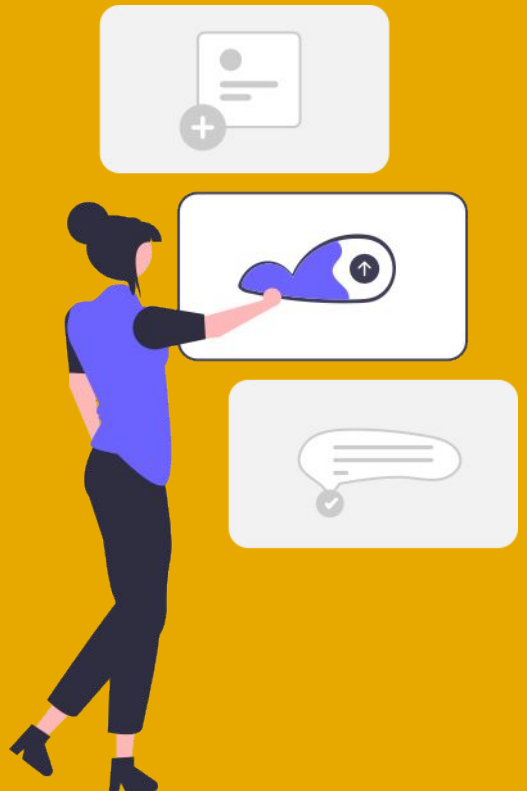
Cómo manejar errores con Fetch API

Manejar errores es crucial para asegurar que la aplicación no falle ante **problemas de conectividad o respuestas inesperadas**.

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

Explicación del código:

- **if (!response.ok):** Verifica si la respuesta tiene un código de estado exitoso (2xx).
- **throw new Error:** Lanza un error que será capturado en el bloque catch.



Usando Axios

Sintaxis básica Axios

Axios es una biblioteca basada en promesas que **simplifica las solicitudes HTTP en comparación con Fetch**.

```
import axios from 'axios';

axios.get('https://jsonplaceholder.typicode.com/posts')
  .then((response) => console.log(response.data))
  .catch((error) => console.error('Error:', error));
```

Explicación del código:

- **axios.get(url):** Realiza una solicitud GET a la URL especificada.
- **response.data:** Contiene los datos de la respuesta.
- **catch:** Maneja errores automáticamente, incluyendo detalles como el código de estado HTTP.

Invocación de métodos HTTP con Axios

Axios permite realizar solicitudes HTTP con métodos como **POST**, **PUT** y **DELETE**.

```
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));
```

Explicación del código:

- **Método POST:** Crea un nuevo recurso en el servidor.
- **Payload:** Los datos que se envían al servidor (en este caso, un objeto con title, body y userId).

Manejo de errores en Axios

Axios proporciona un manejo **más detallado de errores** en comparación con Fetch.

```
axios.get('https://api.example.com/data')
  .catch((error) => {
    if (error.response) {
      console.error('Error en el servidor:', error.response.data);
    } else {
      console.error('Error:', error.message);
    }
  });
```

Explicación del código:

- **error.response:** Contiene detalles de la respuesta del servidor, como el código de estado y los datos.
- **error.message:** Describe el error si no se recibió una respuesta.

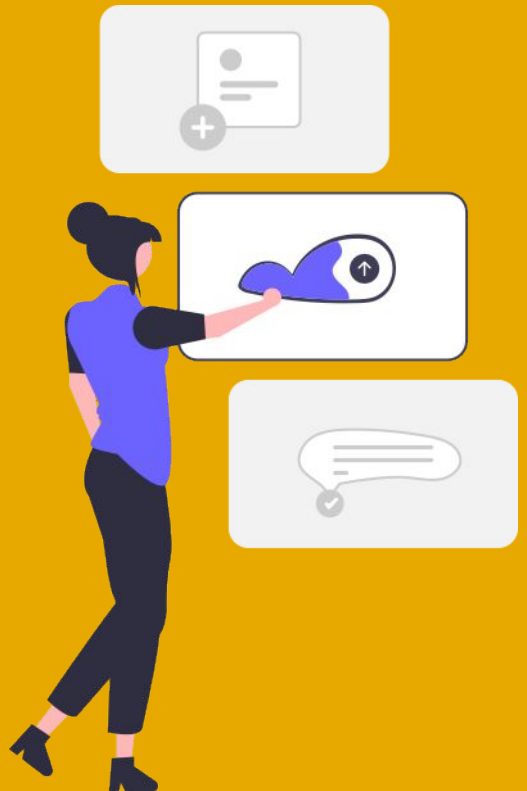
Usando async/await con Axios

async/await mejora la legibilidad del código al manejar **operaciones asíncronas** de forma más clara.

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

Explicación del código:

- **async function:** Declara una función asíncrona.
- **await axios.get(...):** Pausa la ejecución hasta que se complete la solicitud.
- **try/catch:** Maneja errores de forma estructurada.



Diferencias entre Fetch API y Axios

adalid
aspasia

Principales diferencias entre Fetch API y Axios

Fetch y Axios son dos herramientas populares para realizar **solicitudes HTTP en aplicaciones frontend**. Aunque tienen objetivos similares, presentan diferencias clave que pueden influir en su elección según las necesidades del proyecto.

Principales diferencias entre Fetch API y Axios

Característica	Fetch API	Axios
Nativo	Sí, integrado	No, requiere instalación
Manejo de errores	Manual	Automático
JSON automático	No	Sí
Cancelación de solicitudes	Compleja	Simple
Timeout	No soportado	Soportado
Configuración avanzada	Limitada	Amplia

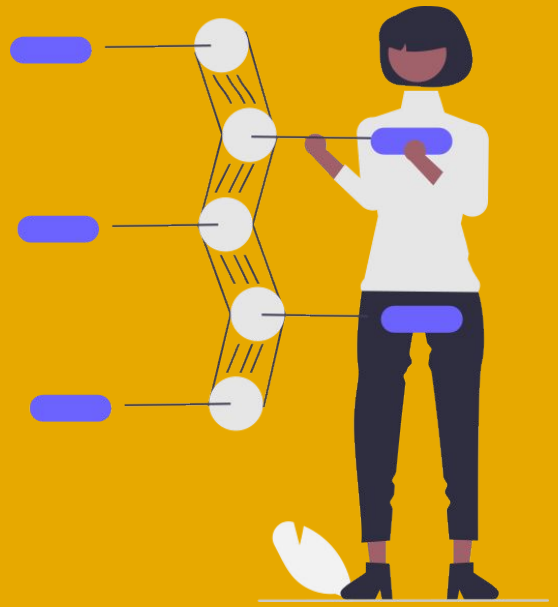
- **Fetch:** Ideal para proyectos simples con configuraciones básicas.
- **Axios:** Recomendado para aplicaciones complejas con necesidades avanzadas.

Acceso al Repositorio

En el siguiente enlace podrás acceder a un repositorio relacionado con la temática propuesta.

https://github.com/adalid-cl/ESPECIALIZACION_FRONTEND_M5_AE1

Resumen de lo aprendido 🧑‍🎓



Adalid aspasia

Resumen de lo aprendido

- Aprendiste cómo conectar tu aplicación React con APIs externas para obtener y enviar datos de manera eficiente.
- Entendiste el rol clave del frontend en la comunicación cliente/servidor y cómo gestionar esa interacción.
- Descubriste cómo usar herramientas como useEffect, Fetch y Axios para manejar peticiones HTTP y sus respuestas.
- Aprendiste a gestionar errores y optimizar el rendimiento al consumir datos en React.

GRACIAS POR TU ATENCIÓN

Nos vemos en la próxima clase

Chile
Adalid
aspasia
LA FORMACIÓN DE TU FUTURO