

Introducción al Desarrollo web

Curso: Desarrollo Frontend con React

Consumo de APIs con Fetch en JavaScript



¿Qué aprenderemos hoy?

1

Fundamentos de APIs

Qué son, tipos y su importancia en el desarrollo web moderno

2

JSON y Fetch API

Formato de intercambio de datos y método nativo para consumir APIs

3

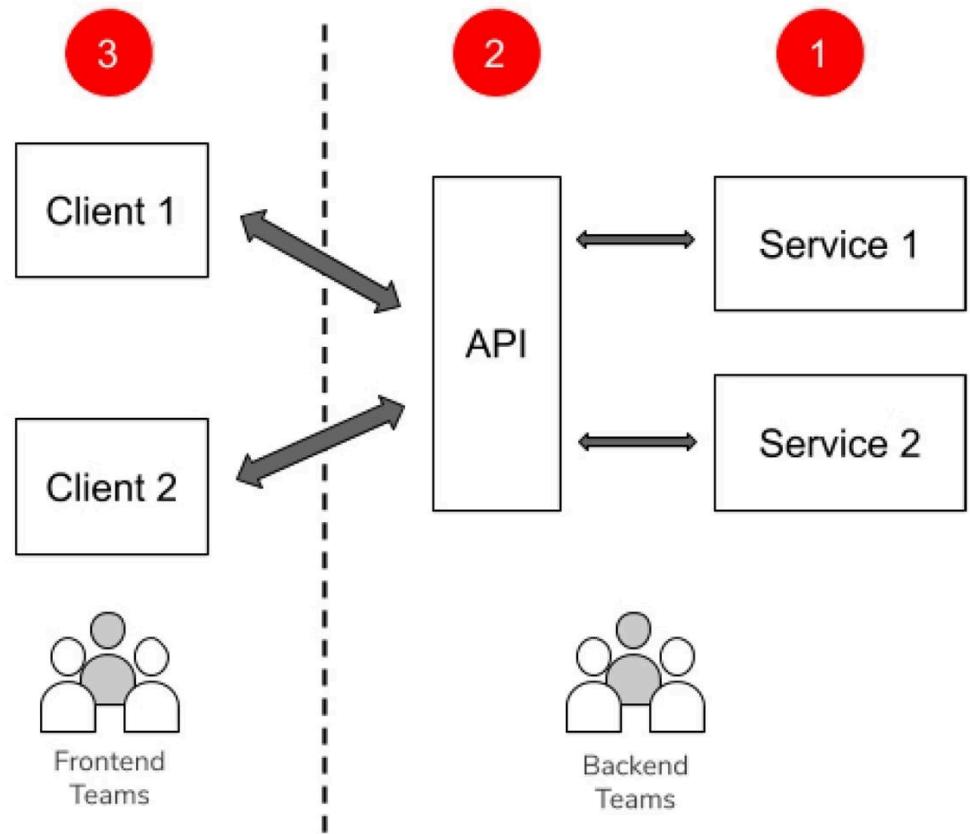
Implementación práctica

Ejemplos reales, manejo de errores y buenas prácticas

¿Qué es una API?

Una **Interfaz de Programación de Aplicaciones** (API) es un conjunto de reglas y protocolos que permite a diferentes aplicaciones comunicarse entre sí.

Actúa como un [mensajero](#) que toma tu solicitud, la lleva al sistema y devuelve la respuesta.



Las APIs en el desarrollo web

Acceso a datos externos

Permite a nuestra aplicación obtener datos de servicios como clima, mapas, redes sociales o cualquier base de datos remota.

Arquitectura desacoplada

Facilita la separación entre el frontend y el backend, permitiendo un desarrollo más modular y escalable.

Reutilización

Un mismo backend puede servir a múltiples clientes: web, móvil, desktop, etc.

Tipos de APIs

APIs REST

- Utilizan métodos HTTP (GET, POST, PUT, DELETE)
- Trabajan con formato JSON o XML
- Siguen principios arquitectónicos específicos
- Las más comunes en desarrollo web actual

Otras APIs

- SOAP: Más complejas, usan XML, comunes en sistemas empresariales
- GraphQL: Consultas flexibles, obtén solo lo que necesitas
- WebSockets: Comunicación bidireccional en tiempo real

APIs públicas populares

Información

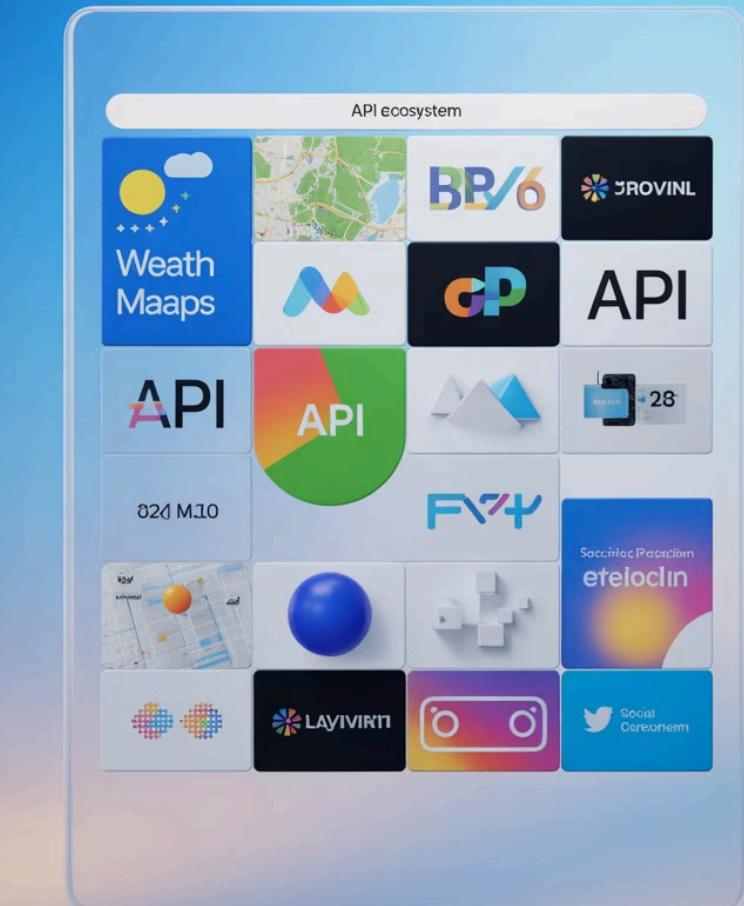
- OpenWeatherMap
- NASA
- Wikipedia

Servicios

- Google Maps
- Stripe (pagos)
- SendGrid (emails)

Redes Sociales

- Twitter API
- Instagram Graph API
- YouTube Data API



Formato JSON

JavaScript Object Notation

Formato ligero de intercambio de datos basado en la sintaxis de objetos de JavaScript.

Ventajas:

- Fácil de leer para humanos
- Fácil de procesar para máquinas
- Independiente del lenguaje

```
{  
  "nombre": "Ana García",  
  "edad": 28,  
  "activo": true,  
  "habilidades": ["HTML", "CSS", "JavaScript"],  
  "dirección": {  
    "calle": "Gran Vía 123",  
    "ciudad": "Madrid"  
  }  
}
```

Trabajando con JSON en JavaScript

JSON.parse()

Convierte una cadena JSON en un objeto JavaScript

```
const usuario = JSON.parse('{"nombre":"Ana","edad":28}');
console.log(usuario.nombre); // Ana
```

JSON.stringify()

Convierte un objeto JavaScript en una cadena JSON

```
const datos = {nombre: "Carlos", edad: 32};
const jsonStr = JSON.stringify(datos);
console.log(jsonStr); // '{"nombre":"Carlos","edad":32}'
```

Fetch API

La [Fetch API](#) proporciona una interfaz para recuperar recursos a través de la red de manera asíncrona.

Características:

- Nativa en navegadores modernos
- Basada en **Promesas**
- Más simple y flexible que XMLHttpRequest
- Permite trabajar con la caché y CORS



Sintaxis básica de Fetch

```
fetch('https://api.ejemplo.com/datos')
  .then(response => {
    // Manejar la respuesta inicial
    if (!response.ok) {
      throw new Error('Error en la petición: ' + response.status);
    }
    return response.json(); // Convertir a JSON
  })
  .then(data => {
    // Trabajar con los datos
    console.log(data);
  })
  .catch(error => {
    // Manejar errores
    console.error('Problema con la petición:', error);
  });
});
```

Observación: `fetch()` devuelve una Promesa que se resuelve en un objeto Response, que debemos procesar.

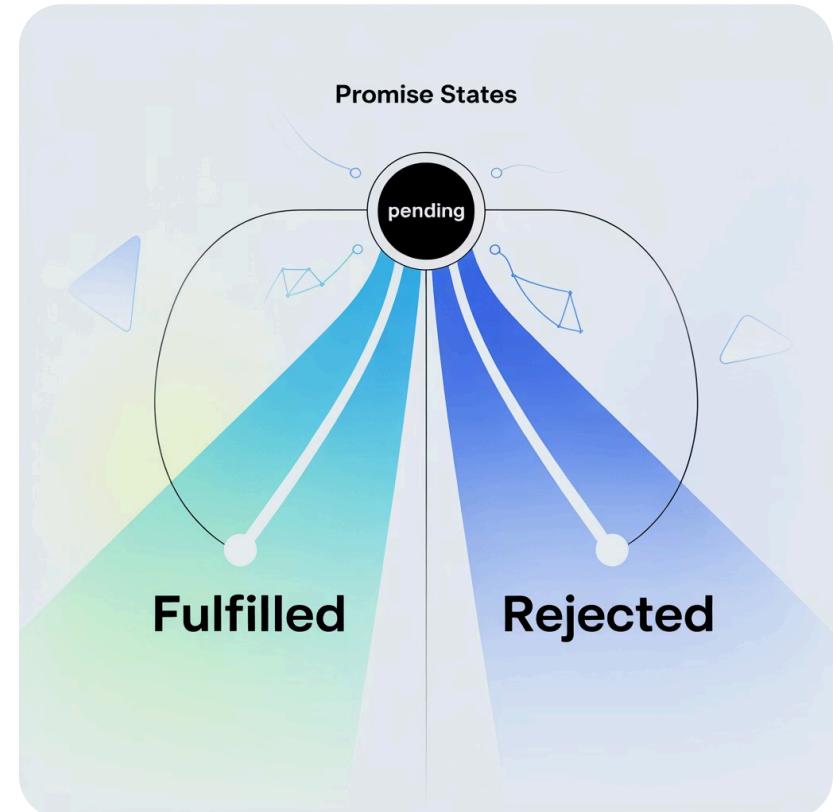
¿Qué son las Promesas?

Una [Promesa](#) representa un valor que puede estar disponible ahora, en el futuro, o nunca.

Estados de una promesa:

- **Pendiente (pending):** Estado inicial
- **Cumplida (fulfilled):** Operación completada con éxito
- **Rechazada (rejected):** Operación fallida

Las promesas nos permiten trabajar con operaciones asíncronas de forma más estructurada.



Métodos .then() y .catch()

.then()

Se ejecuta cuando la promesa se resuelve con éxito.

```
fetch('https://api.ejemplo.com/datos')
  .then(response => response.json())
  .then(data => console.log(data));
```

.catch()

Captura cualquier error que ocurra en la cadena de promesas.

```
fetch('https://api.ejemplo.com/datos')
  .then(response => response.json())
  .catch(error => console.error(error));
```

Métodos HTTP comunes



GET

Solicita un recurso. No modifica datos.

Ejemplo: Obtener lista de usuarios



POST

Envía datos para crear un nuevo recurso.

Ejemplo: Registrar un nuevo usuario



PUT/PATCH

Actualiza un recurso existente.

Ejemplo: Modificar datos de perfil



DELETE

Elimina un recurso específico.

Ejemplo: Borrar una cuenta

Configurando peticiones Fetch

```
fetch('https://api.ejemplo.com/usuarios', {  
  method: 'POST', // Método HTTP  
  headers: { // Cabeceras  
    'Content-Type': 'application/json',  
    'Authorization': 'Bearer token123'  
  },  
  body: JSON.stringify({ // Cuerpo de la petición  
    nombre: 'María',  
    email: 'maria@example.com'  
  })  
})  
.then(response => response.json())  
.then(data => console.log(data))  
.catch(error => console.error('Error:', error));
```

Ejemplo práctico: API de Pokémon



Vamos a crear una aplicación que muestre información de Pokémon usando la PokéAPI (que es gratuita).

Pasos:

1. No necesitamos API key (es gratuita)
2. Crear la estructura HTML básica
3. Realizar petición fetch a la PokéAPI
4. Mostrar los datos del Pokémon en nuestra web

Estructura HTML para nuestra app

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>Pokédex Digital</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
<div class="container">
<h1>Pokédex Digital</h1>

<div class="busqueda">
<input type="text" id="pokemon" placeholder="Introduce el nombre de un Pokémon">
<button id="buscar">Buscar Pokémon</button>
</div>

<div id="pokemon-info"></div>
</div>

<script src="app.js"></script>
</body>
</html>
```

JavaScript: Consumiendo la API

```
// Elementos del DOM
const pokemonInput = document.getElementById('pokemon'); // El ID 'ciudad' se mantiene del HTML previo para el input
const buscarBtn = document.getElementById('buscar');
const pokemonInfoDiv = document.getElementById('pokemon-info');

// Evento click en el botón
buscarBtn.addEventListener('click', () => {
  const pokemonName = pokemonInput.value.trim();
  if (pokemonName) {
    obtenerDatosPokemon(pokemonName);
  } else {
    mostrarError('Por favor, introduce un nombre de Pokémon');
  }
});

// Nota: La función 'obtenerDatosPokemon' y 'mostrarError' se definirían más adelante en 'app.js'
```

Función para obtener datos de Pokémon

```
function obtenerDatosPokemon(pokemonName) {  
    // Construir URL de la API  
    const url = `https://pokeapi.co/api/v2/pokemon/${pokemonName.toLowerCase()}`;  
  
    // Mostrar indicador de carga  
    resultadoDiv.innerHTML = '<p>Buscando Pokémon...</p>';  
  
    // Realizar petición  
    fetch(url)  
        .then(response => {  
            if (!response.ok) {  
                // Manejar el caso de Pokémon no encontrado (código 404)  
                if (response.status === 404) {  
                    throw new Error('Pokémon no encontrado.');                }  
                throw new Error(`Error: ${response.status}`);  
            }  
            return response.json();  
        })  
        .then(data => mostrarDatosPokemon(data))  
        .catch(error => mostrarError(error.message));  
}
```

Función para mostrar los datos de Pokémon

```
function mostrarDatosPokemon(data) {  
    // Extraer los datos relevantes del Pokémon  
    const nombre = data.name.charAt(0).toUpperCase() + data.name.slice(1);  
    const pokedexId = data.id;  
    const altura = (data.height / 10).toFixed(1); // Convertir de decímetros a metros  
    const peso = (data.weight / 10).toFixed(1); // Convertir de hectogramos a kilogramos  
    const tipos = data.types.map(typeInfo => typeInfo.type.name.charAt(0).toUpperCase() + typeInfo.type.name.slice(1));  
    const imagenSprite = data.sprites.front_default;  
    const habilidades = data.abilities.map(abilityInfo => abilityInfo.ability.name.replace('-', ' ').split(' ').map(word =>  
        word.charAt(0).toUpperCase() + word.slice(1)).join(' '));  
  
    // Insertar HTML con los datos de Pokémon de forma atractiva  
    resultadoDiv.innerHTML = ` ${nombre} #${pokedexId}  
        Altura: ${altura} m  
        Peso: ${peso} kg  
        Tipo(s): ${tipos.join(', ')}  
        Habilidades: ${habilidades.join(', ')}`;  
}
```

Función para mostrar errores

```
function mostrarError(mensaje) {  
    resultadoDiv.innerHTML = `  
        <div class="error">  
            <p>${mensaje}</p>  
        </div>  
    `;  
}
```

Es crucial manejar adecuadamente los errores para ofrecer feedback claro al usuario, especialmente cuando busca un Pokémon que no existe o cuando hay problemas de conexión con la PokéAPI.



Async/Await: Una forma más limpia

`async/await` es una sintaxis que hace que el código asíncrono parezca síncrono, facilitando su lectura y mantenimiento.

- **async:** Declara una función asíncrona
- **await:** Espera a que se resuelva una promesa

```
async function obtenerDatosPokemon(pokemonName) {  
  try {  
    const url =  
      `https://pokeapi.co/api/v2/pokemon/${pokemonName.toLowerCase()}`;  
    const response = await fetch(url);  
  
    if (!response.ok) {  
      // Manejo específico para Pokémon no encontrado (error  
      404)  
      if (response.status === 404) {  
        throw new Error(`Pokémon "${pokemonName}" no  
encontrado.`);  
      }  
      throw new Error(`Error de red: ${response.status}`);  
    }  
  
    const data = await response.json();  
    mostrarDatosPokemon(data); // Asume que esta función  
    muestra los datos del Pokémon  
  } catch (error) {  
    mostrarError(error.message);  
  }  
}
```

Manejo de errores en Fetch

Tipos de errores

- **Errores de red:** Sin conexión, timeout
- **Errores HTTP:** 404, 500, etc.
- **Errores en promesas:** Rechazo de promesas
- **Errores de parseo:** JSON inválido

Estrategias de manejo

1. Verificar `response.ok`
2. Usar bloques try/catch
3. Implementar reintentos automáticos
4. Mostrar mensajes amigables al usuario

Códigos de estado HTTP comunes



200-299: Éxito

200: OK

201: Created

204: No Content



300-399: Redirección

301: Moved Permanently

302: Found

304: Not Modified



400-499: Error del cliente

400: Bad Request

401: Unauthorized

404: Not Found



500-599: Error del servidor

500: Internal Server Error

502: Bad Gateway

503: Service Unavailable

Validación de datos recibidos

Nunca confíes en los datos que recibes de una API sin validarlos primero.

- Verifica que los datos existan
- Comprueba el formato y tipo de datos
- Establece valores por defecto
- Limpia y valida la información antes de mostrarla.

```
function mostrarPokemonData(pokemonData) {  
    // Validación de datos esenciales de Pokémon  
    if (!pokemonData || !pokemonData.name ||  
        !pokemonData.id || !pokemonData.types ||  
        !pokemonData.sprites) {  
        console.error('Error: Datos esenciales de Pokémon  
        incompletos.');//  
        return;  
    }  
  
    ...  
}
```

Buenas prácticas: Seguridad

Protege tus API keys

Nunca incluyas claves directamente en código del frontend. Usa variables de entorno y/o un backend como intermediario.

Valida los datos

Sanitiza la entrada del usuario y valida las respuestas de la API antes de procesarlas.

Verifica orígenes

Implementa CORS adecuadamente si desarrollas una API propia.



Buenas prácticas: Rendimiento



Implementa caché

Almacena temporalmente las respuestas para reducir peticiones repetidas.



Filtrá datos

Solicita solo la información que necesitas si la API lo permite.



Muestra estados de carga

Informa al usuario mientras espera que se completen las peticiones.

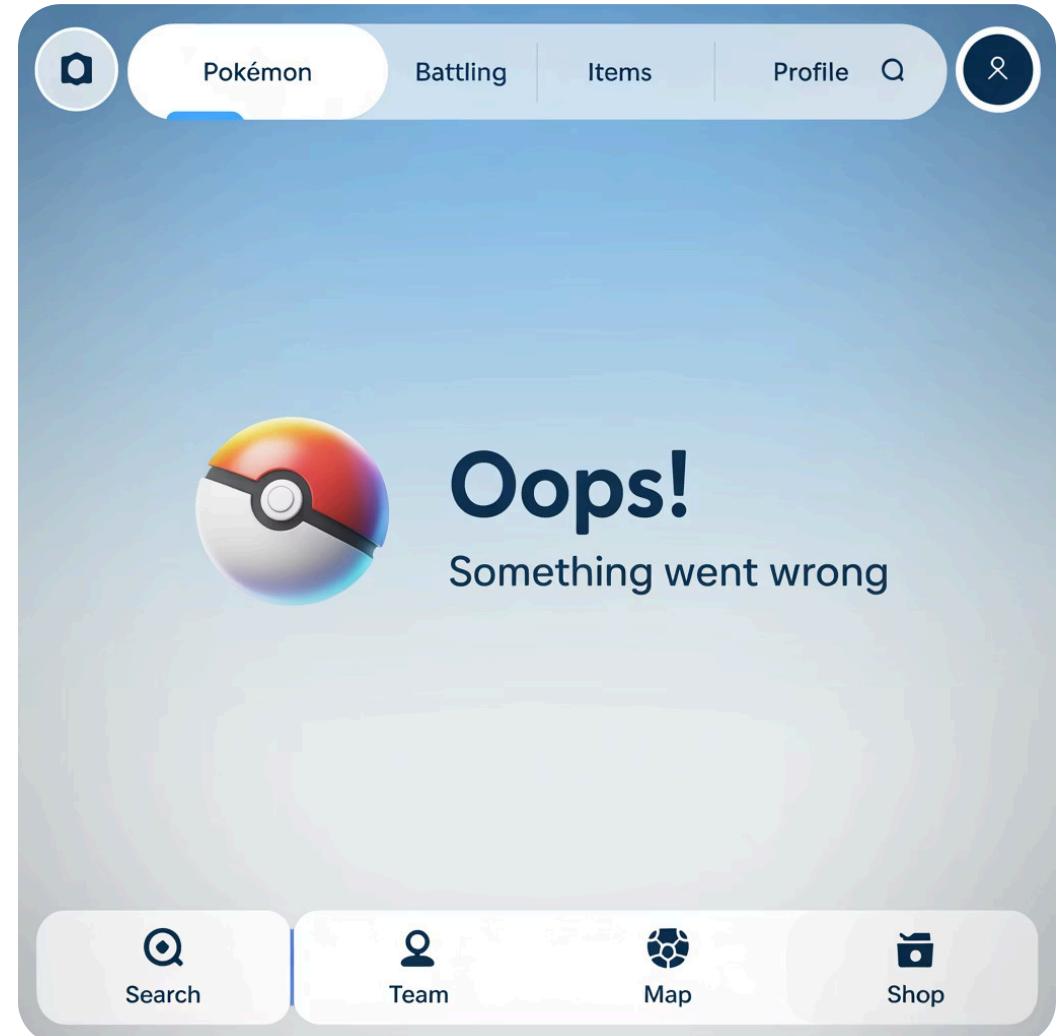
Buenas prácticas: Experiencia de usuario

Estado de carga

```
// Mostrar estado de carga  
resultadoDiv.innerHTML = '<p>Buscando Pokémon...</p>';  
  
fetch(url)  
.then(...);
```

Mensajes de error claros

```
// En vez de: "Error al cargar Pokémon"  
// Mejor: "¡Oh no! No pudimos encontrar a ese Pokémon.  
// ¿Estás seguro de que lo escribiste bien?"
```



Herramientas útiles para trabajar con APIs



Postman

Plataforma para probar APIs antes de implementarlas en tu código.



DevTools

La pestaña Network de las herramientas de desarrollo del navegador.



JSONView

Extensión para visualizar JSON formateado en el navegador.

Estas herramientas te ayudarán a comprender mejor cómo funcionan las APIs y a depurar problemas en tus implementaciones.

Reflexión: Importancia del manejo de errores

¿Por qué es crucial implementar un buen sistema de manejo de errores al consumir APIs?



Experiencia de usuario

Los usuarios merecen saber qué ha fallado y cómo pueden solucionarlo.



Seguridad

Previene la exposición de información sensible en mensajes de error genéricos.



Mantenimiento

Facilita la depuración y solución de problemas al proporcionar información detallada.

Resumen y próximos pasos

Lo que hemos aprendido

- Conceptos fundamentales de APIs
- Uso de Fetch para consumir datos
- Formato JSON para intercambio de información
- Manejo adecuado de errores y promesas

Práctica recomendada

Crea pequeños proyectos utilizando APIs públicas como:

- PokéAPI (Pokémon)
- JSONPlaceholder (posts y usuarios de prueba)
- REST Countries (información de países)
- The Cat API (imágenes de gatos)

#EDCOUNIANDES

<https://educacioncontinua.uniandes.edu.co/>

Contacto: educacion.continua@uniandes.edu.co

© - Derechos Reservados: La presente obra, y en general todos sus contenidos, se encuentran protegidos por las normas internacionales y nacionales vigentes sobre propiedad Intelectual, por lo tanto su utilización parcial o total, reproducción, comunicación pública, transformación, distribución, alquiler, préstamo público e importación, total o parcial, en todo o en parte, en formato impreso o digital y en cualquier formato conocido o por conocer, se encuentran prohibidos, y solo serán lícitos en la medida en que se cuente con la autorización previa y expresa por escrito de la Universidad de los Andes.