Protocolos de Comunicación

HTTP, WebSockets, Hooks personalizados y Recharts

Javier Ribal del Río

2026-03-13

Table of contents

Contenido

- HTTP básico para desarrollo web
- Métodos GET, POST, PUT, DELETE, HEAD
- WebSockets

- Hook personalizado para conexión en tiempo real
- Gráficas con Recharts

1 HTTP

1.1 ¿Qué es HTTP?

HTTP (*HyperText Transfer Protocol*) es el protocolo fundamental de la web.

Define cómo:

- El cliente (navegador o app)
- Solicita recursos
- A un servidor

Modelo conceptual:

```
Cliente → Solicitud HTTP → Servidor
Cliente ← Respuesta HTTP ← Servidor
```

HTTP es **sin estado (stateless)**.

Cada petición es independiente.

1.2 Componentes de una petición

Una petición HTTP contiene:

- Método
- URL
- Cabeceras (*headers*)
- Cuerpo (*body*, opcional)

Ejemplo conceptual:

```
GET /users HTTP/1.1
Host: api.example.com
Authorization: Bearer token
```

1.3 Métodos HTTP principales

1.3.1 GET

- Solicita datos
- No modifica el servidor
- Debe ser idempotente

Ejemplo:

```
GET /users
```

Uso típico en React:

```
fetch("/api/users")
```

1.4 POST

- Envía datos al servidor
- Crea recursos nuevos
- Tiene cuerpo

Ejemplo:

```
fetch("/api/users", {
  method: "POST",
  body: JSON.stringify({ name: "Ana" }),
  headers: { "Content-Type": "application/json" }
});
```

1.5 PUT

- Actualiza completamente un recurso existente
- Idempotente

`PUT /users/3`

1.6 DELETE

- Elimina un recurso

`DELETE /users/3`

1.7 HEAD

- Igual que GET
- No devuelve cuerpo
- Solo cabeceras

Se utiliza para:

- Comprobar existencia
- Tamaño
- Metadatos

2 HTTP vs Comunicación en tiempo real

HTTP funciona bajo el modelo:

Petición → Respuesta → Fin

Si queremos datos continuamente:

- Debemos repetir peticiones
- Introduce latencia
- Ineficiente

Solución: **WebSockets**

3 WebSockets

WebSockets permiten comunicación bidireccional persistente.

Características:

- Conexión permanente
- Cliente y servidor pueden enviar datos
- Baja latencia
- Ideal para tiempo real

Ejemplos:

- Chats
- Juegos online
- Telemetría
- Trading
- Sistemas IoT

3.1 Flujo conceptual

```
Cliente   Servidor
     conexión abierta
```

No se crea una nueva conexión por mensaje.

3.2 Creación de un WebSocket en JavaScript

```
const socket = new WebSocket("ws://localhost:8080");
```

Eventos principales:

- `onopen`
- `onmessage`
- `onclose`
- `onerror`

3.3 Recepción de mensajes

```
socket.onmessage = (event) => {
  console.log("Mensaje:", event.data);
};
```

3.4 Envío de mensajes

```
socket.send("Hola servidor");
```

4 Hook Personalizado

Si usamos WebSockets directamente dentro del componente:

- Mezclamos lógica de conexión con renderizado
- Código difícil de reutilizar
- Problemas de ciclo de vida

Solución: **Hook personalizado**

4.1 Hook personalizado

Un hook personalizado es:

> Una función que utiliza hooks internos para encapsular lógica reutilizable

Convención:

`useNombre()`

4.2 Hook `useWebSocket`

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

export function useWebSocket(url) {

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

  useEffect(() => {

    const socket = new WebSocket(url);

    socket.onmessage = (event) => {
      setData(event.data);
    };

    return () => socket.close();

  }, [url]);

  return data;
}
```

4.3 Análisis conceptual

- Se crea la conexión al montar
- Se actualiza el estado al recibir datos
- Se cierra al desmontar
- El componente solo consume datos

4.4 Uso del hook

```
function Monitor() {

  const data = useWebSocket("ws://localhost:8080");

  return (
    <div>
      <h2>Datos en tiempo real</h2>
      <p>{data}</p>
    </div>
  );
}
```

Flujo:

1. Componente se renderiza
2. Hook abre conexión
3. Llegan datos
4. Estado cambia
5. Nuevo render

5 Visualización de datos

En aplicaciones reales no basta con mostrar texto.

Necesitamos:

- Gráficas
- Paneles de control
- Visualización clara

Una librería popular para React es **Recharts**.

6 Recharts

Recharts es una librería basada en:

- Componentes React
- SVG
- Composición declarativa

Instalación:

```
npm install recharts
```

6.1 Ejemplo básico

```
import {
  LineChart, Line, XAxis, YAxis,
  CartesianGrid, Tooltip
} from "recharts";

const data = [
  { t: 1, v: 10 },
  { t: 2, v: 15 },
  { t: 3, v: 8 }
];

function Grafica() {
  return (
    <LineChart width={400} height={300} data={data}>
      <CartesianGrid stroke="#ccc" />
      <XAxis dataKey="t" />
      <YAxis />
      <Tooltip />
      <Line type="monotone" dataKey="v" stroke="#8884d8" />
    </LineChart>
  );
}
```

6.2 Interpretación

Cada objeto representa un punto.

`{ t: 1, v: 10 }`

- `t` → eje X
- `v` → eje Y

7 Integración completa

WebSocket + Hook + Recharts

Objetivo:

> Visualizar datos en tiempo real

7.1 Ejemplo conceptual

```
function Dashboard() {

  const data = useWebSocket("ws://localhost:8080");

  const chartData = data
    ? JSON.parse(data)
    : [];

  return (
    <LineChart width={500} height={300} data={chartData}>
      <XAxis dataKey="time" />
      <YAxis />
      <Tooltip />
      <Line dataKey="value" stroke="#82ca9d" />
    </LineChart>
  );
}
```

7.2 Flujo completo de una app moderna

`Servidor → WebSocket → Hook → Estado → React → Gráfica`

Arquitectura típica de sistemas en tiempo real.

8 Material

Servidor de testing de *Python*: Enlace.

Instalar `websockets` con `pip install websockets`

Tester de prueba: Enlace (click derecho guardar enlace como)

https://recharts.github.io/