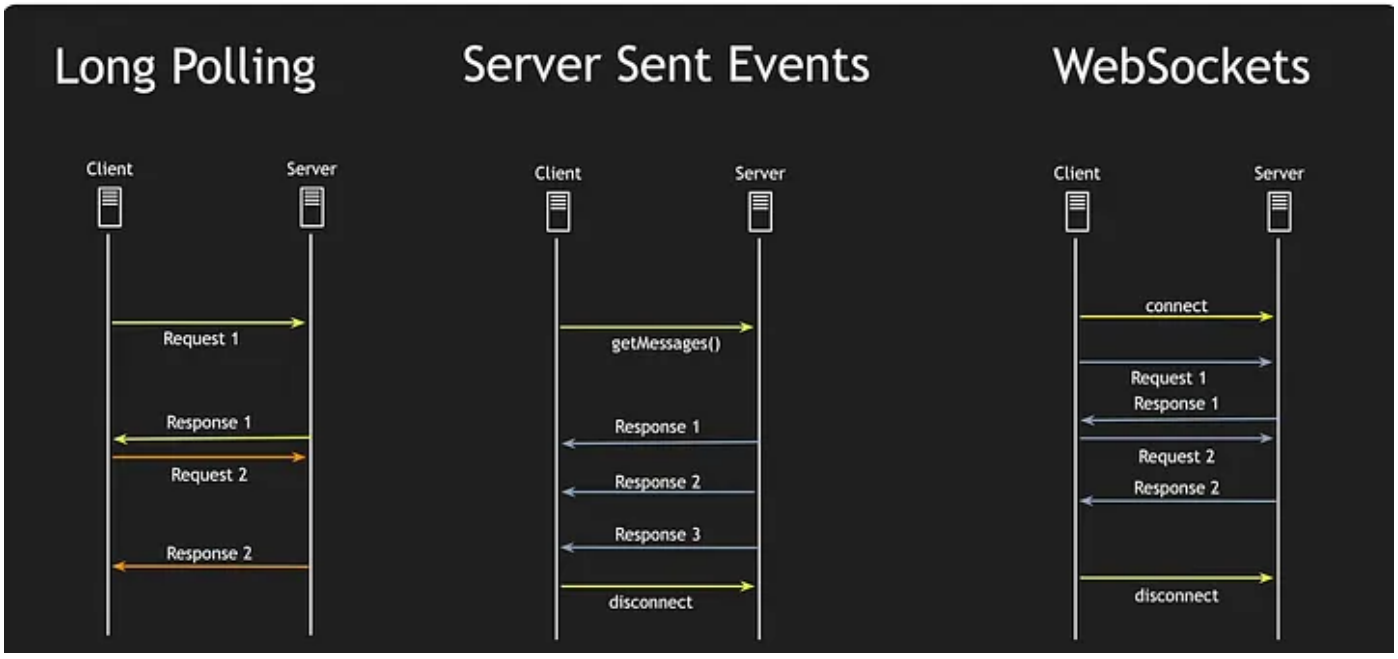
Long Polling vs Server-Sent Events vs WebSockets: A Comprehensive Guide

Asharsaleem Follow 5 min read · Jun 27, 2024



When developing web applications, real-time communication between the client and server can significantly enhance user experience. There are several methods to achieve this, each with its own advantages and use

cases. This guide will help into three popular techniques: Long Polling, Server-Sent Events (SSE), and WebSockets, explaining their mechanisms, benefits, drawbacks, and appropriate use cases.

Long Polling

Mechanism

Long polling is a technique where the client repeatedly requests information from the server at regular intervals. Instead of the server responding immediately, it holds the request open until new data is available or a timeout occurs. Once a response is received, the client typically sends a new request, restarting the process.

Advantages

- **Compatibility**: Works with older browsers and HTTP/1.1, requiring no special server or client support.
- **Simple Implementation**: Relatively straightforward to implement using standard HTTP requests.

Drawbacks

- **Latency**: Introduces latency since the client has to wait for the server's response.
- **Resource Intensive**: Can be resource-heavy on both the client and server due to the continuous open connections and frequent requests.
- **Scalability Issues**: High overhead can lead to scalability issues, particularly with a large number of clients.

Use Cases

- **Legacy Systems**: Useful in environments where WebSockets and SSE

are not supported.

- **Low-Frequency Updates:** Suitable for applications where real-time updates are not critical and data changes infrequently.

```
const express = require('express');
const app = express();
const port = 3000;

let messages = [];
app.use(express.json());

app.post('/messages', (req, res) => {
  messages.push(req.body.message);
  res.status(200).end();
});

app.get('/poll', (req, res) => {
  const interval = setInterval(() => {
    if (messages.length > 0) {
      clearInterval(interval);
      res.json({ message: messages.shift() });
    }
  }, 1000);
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

Client (HTML + JavaScript):

```
<!DOCTYPE html>
<html>
<head>
  <title>Long Polling Example</title>
</head>
<body>
  <div id="messages"></div>
  <script>
```

```
      function poll() {
        fetch('/poll')
          .then(response => response.json())
          .then(data => {
            const messagesDiv = document.getElementById('messages');
            const newMessage = document.createElement('div');
            newMessage.textContent = data.message;
            messagesDiv.appendChild(newMessage);
            poll(); // Poll again
          });
      }

      poll(); // Start polling
    </script>
  </body>
  </html>
```

Server-Sent Events (SSE)

Mechanism

SSE allows servers to push updates to the client over a single, long-lived HTTP connection. The server sends data to the client when new information is available without requiring the client to request it continuously.

Advantages

- **Efficient Data Transmission:** Reduces the need for repeated requests, saving bandwidth and reducing latency.
- **Simple API:** The EventSource API is straightforward to use and integrates well with existing web standards.
- **Automatic Reconnection:** Built-in support for automatic reconnection and event ID tracking.

Drawbacks

- **Uni-directional:** Only supports server-to-client communication, making it unsuitable for interactive applications where the client needs to send frequent data to the server.
- **Browser Support:** Limited support in some older browsers, although modern browsers widely support SSE.

Use Cases

- **Live Feeds:** Ideal for applications like news tickers, social media updates, or stock price updates.
- **Notification Systems:** Suitable for server-initiated notifications, such as chat applications or real-time analytics dashboards.

```
const express = require('express');
const app = express();
const port = 3000;

app.use(express.json());

app.get('/events', (req, res) => {
  res.setHeader('Content-Type', 'text/event-stream');
  res.setHeader('Cache-Control', 'no-cache');
  res.setHeader('Connection', 'keep-alive');

  setInterval(() => {
    res.write(`data: ${JSON.stringify({ message: 'Hello from server!' })}\n\
  }, 2000);
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

Client (HTML + JavaScript):

```
<!DOCTYPE html>
<html>
<head>
  <title>SSE Example</title>
</head>
<body>
  <div id="messages"></div>
  <script>
    const eventSource = new EventSource('/events');

    eventSource.onmessage = function(event) {
      const messagesDiv = document.getElementById('messages');
      const newMessage = document.createElement('div');
      newMessage.textContent = JSON.parse(event.data).message;
      messagesDiv.appendChild(newMessage);
    };
  </script>
</body>
</html>
```

WebSockets

Mechanism

WebSockets provide a full-duplex communication channel over a single, persistent TCP connection. Once established, either the client or server can send messages at any time, making it highly efficient for real-time, bidirectional communication.

Advantages

- **Low Latency**: Minimal overhead and latency, providing near-instantaneous communication.
- **Bidirectional Communication**: Supports real-time, two-way communication, suitable for interactive applications.
- **Scalability**: Efficiently handles large numbers of concurrent connections, making it scalable for high-traffic applications.

Drawbacks

- **Complexity:** Requires more complex server and client implementation compared to HTTP-based methods.
- **Compatibility:** May require additional configuration for firewalls and proxies that do not support WebSocket traffic.
- **Protocol Overhead:** Some overhead due to maintaining a persistent connection, although generally minimal compared to HTTP polling.

Use Cases

- **Real-Time Games:** Perfect for multiplayer games requiring rapid and frequent communication between clients and servers.
- **Chat Applications:** Ideal for chat and messaging applications where bidirectional communication is essential.
- **Collaborative Tools:** Suitable for applications like collaborative document editing, where multiple users need to see updates in real-time.

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', ws => {
  ws.on('message', message => {
    console.log('Received:', message);
    ws.send(`Hello, you sent -> ${message}`);
  });

  ws.send('Welcome to the WebSocket server!');
});
```

Client (HTML + JavaScript):

```
<!DOCTYPE html>
<html>
<head>
  <title>WebSockets Example</title>
</head>
<body>
  <div id="messages"></div>
  <script>
    const ws = new WebSocket('ws://localhost:8080');

    ws.onmessage = function(event) {
      const messagesDiv = document.getElementById('messages');
      const newMessage = document.createElement('div');
      newMessage.textContent = event.data;
      messagesDiv.appendChild(newMessage);
    };

    ws.onopen = function() {
      ws.send('Hello Server!');
    };
  </script>
</body>
</html>
```

Comparison and Choosing the Right Approach

When deciding which technique to use, consider the following factors:

1. **Frequency and Type of Data:**

- For low-frequency, one-way updates, SSE is often the best choice.
- For high-frequency, interactive communication, WebSockets are typically more suitable.
- For legacy systems or environments with limited support for modern

protocols, long polling may be necessary.

2. Complexity and Maintenance:

- SSE and long polling are simpler to implement but may not scale well for high-demand applications.
- WebSockets require more initial setup but offer superior performance and scalability.

3. Browser and Environment Support:

- Ensure that the chosen method is supported by the browsers and network environments in which your application will run.

Summary

- **Long Polling** is a fallback technique where the client repeatedly requests updates from the server, useful for environments without WebSocket or SSE support.
- **Server-Sent Events (SSE)** enable the server to push updates to the client over a single HTTP connection, ideal for applications requiring real-time, one-way updates.
- **WebSockets** provide full-duplex communication between the client and server over a single persistent connection, suitable for interactive applications needing real-time, bidirectional communication.

By choosing the right method for your specific use case, you can ensure efficient and effective real-time communication in your web applications.

JavaScript Programming Websocket Polling Coding

Written by Asharsaleem

20 followers · 8 following

Follow

Software Engineer https://www.linkedin.com/in/ashar-saleem-94976a114/

Responses (1)

I Ishu

What are your thoughts?

Thomas Hofmann he
Jul 17

Good one!

More from Asharsaleem

Asharsaleem

Introduction to NATS Streams in JavaScript

In the evolving landscape of distributed systems, real-time data streaming has...

Jun 21, 2024 2

Asharsaleem

LAMP STACK INSTALLATION USING BASH SCRIPT

This script prompts for a password for the PHPMyAdmin root user with the read...

Feb 26, 2023 7 1

Asharsaleem

Growing Up at 27: The Age of Responsibility

Turning 27 often feels like standing at a crossroads. It's the stage in life where...

Oct 23, 2024 52

Asharsaleem

Stripe ACH implementation in laravel

An ACH payment is a type of electronic bank-to-bank payment in the US. It's mad...

Apr 3, 2022 2

See all from Asharsaleem

Recommended from Medium

In ITNEXT by Animesh Gaitonde

Scaling Distributed Counters: Designing a View Count System...

Architecture, challenges and bottlenecks in counting views at scale

Jul 19 649 13

Arvind Kumar

Designing a Real-time Chat App (WhatsApp, Slack)

How would you build a real-time chat system that's scalable, consistent, and fast?

May 15 17 1

The Latency Gambler

I Interviewed 20+ Engineers. Here's Why Most Can't Code

Over the past year as a Senior Software Engineer at a B2B SaaS company, I've...

Sep 9 2.9K 95

Ankit Kumar Srivastava

Design Uber Backend

System requirements

May 25 2

Abhinav

Docker Is Dead—And It's About Time

Docker changed the game when it launched in 2013, making containers...

Jun 9 7.1K 200

Bhagya Rana

Building an Event-Driven Notification System with FastAP...

Triggering Outbound Events via API Actions Using Async HTTP and Webhook...

Jul 17 5

See more recommendations