Armen Melkumyan • 1st
Technical / Solutions Architect
6mo •

Boost React App Performance with Web Workers: Handling Real-Time Updates with WebSockets

In high-pressure, client-facing applications like championship or tournament dashboards, the UI can easily get overwhelmed by the sheer volume of real-time data coming from multiple WebSocket connections. Whether it’s live results, score updates, or changing coefficients, continuously processing this data on the main thread can lead to performance bottlenecks, freezing the UI and impacting the user experience.

A practical solution is to offload WebSocket handling to a Web Worker, keeping the main thread free for rendering and other interactions. Here's how you can leverage Web Workers to handle WebSocket connections for real-time updates in a React app.

Use Case: Championship Board with Multiple WebSockets
Imagine you're building a championship dashboard where multiple tournaments are running simultaneously. Each tournament has its own WebSocket connection, continuously sending real-time updates. Instead of the main thread managing and processing all the incoming messages, we can use a Web Worker to handle WebSocket communication and only send processed updates to the React app.

Step-by-Step Implementation
1. Create the Web Worker for Handling WebSockets
In this Web Worker, multiple WebSocket connections can be created and managed in the background. Each WebSocket listens for updates and sends the relevant data back to the main thread via postMessage.

2. Using the Web Worker in Your React Component
Next, we integrate the worker into the React app to manage WebSockets for multiple tournaments.

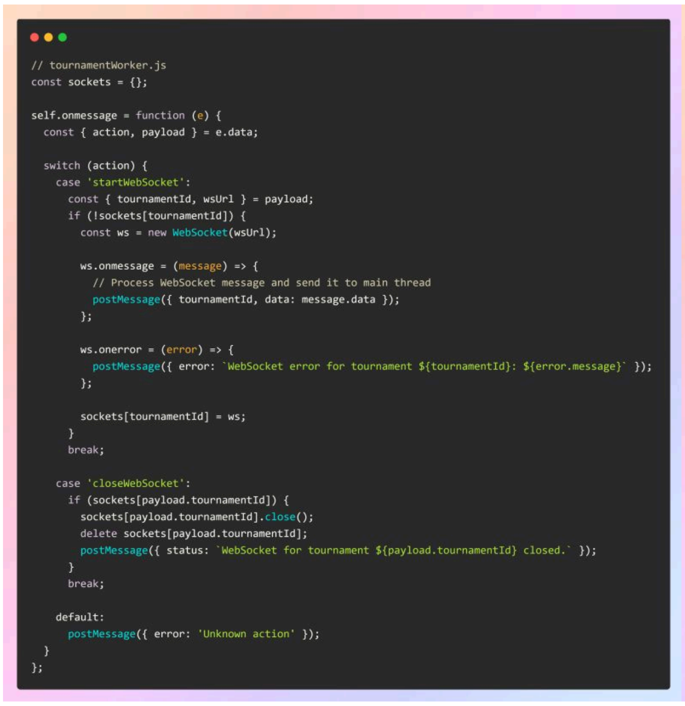
Why Use Web Workers in This Case?
Main Thread Offloading: The main benefit of using Web Workers here is offloading WebSocket handling

from the main thread. Without this, the main thread could get bogged down by frequent WebSocket messages, especially when dealing with multiple tournaments in real-time.

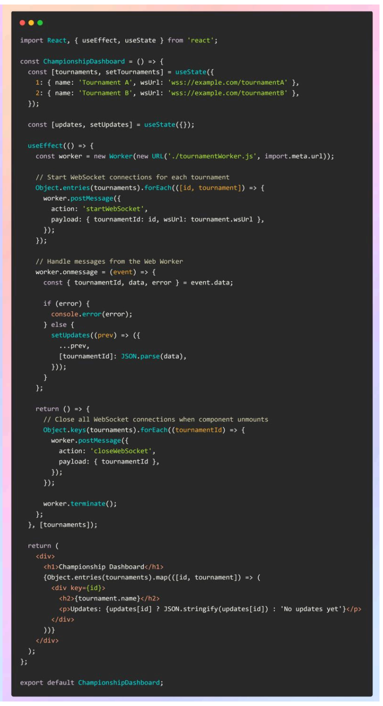
Smooth User Experience: Offloading WebSocket handling ensures that the React app’s UI remains responsive and smooth, even during high load situations with rapid updates.

Scalability: This approach scales well as new tournaments are added, with Web Workers handling the load, rather than overburdening the main thread.

#ReactPerformance #WebWorkers #RealTimeUpdates #WebSockets #ReactJS #FrontendPerformance #HighLoadHandling



```javascript
// tournamentWorker.js
const sockets = {};

self.onmessage = function (e) {
  const { action, payload } = e.data;

  switch (action) {
    case 'startWebSocket':
      const { tournamentId, wsUrl } = payload;
      if (!sockets[tournamentId]) {
        const ws = new WebSocket(wsUrl);

        ws.onmessage = (message) => {
          // Process WebSocket message and send it to main thread
          postMessage({ tournamentId, data: message.data });
        };

        ws.onerror = (error) => {
          postMessage({ error: `WebSocket error for tournament ${tournamentId}: ${error.message}` });
        };

        sockets[tournamentId] = ws;
      }
      break;

    case 'closeWebSocket':
      if (sockets[payload.tournamentId]) {
        sockets[payload.tournamentId].close();
        delete sockets[payload.tournamentId];
        postMessage({ status: `WebSocket for tournament ${payload.tournamentId} closed.` });
      }
      break;

    default:
      postMessage({ error: 'Unknown action' });
  }
};
```



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

const ChampionshipDashboard = () => {
  const [tournaments, setTournaments] = useState({
    1: { name: 'Tournament A', wsUrl: 'wss://example.com/tournamentA' },
    2: { name: 'Tournament B', wsUrl: 'wss://example.com/tournamentB' },
  });

  const [updates, setUpdates] = useState({});

  useEffect(() => {
    const worker = new Worker(new URL('./tournamentWorker.js', import.meta.url));

    // Start WebSocket connections for each tournament
    Object.entries(tournaments).forEach(([id, tournament]) => {
      worker.postMessage({
        action: 'startWebSocket',
        payload: { tournamentId: id, wsUrl: tournament.wsUrl },
      });
    });

    // Handle messages from the Web Worker
    worker.onmessage = (event) => {
      const { tournamentId, data, error } = event.data;

      if (error) {
        console.error(error);
      } else {
        setUpdates((prev) => ({
          ...prev,
          [tournamentId]: JSON.parse(data),
        }));
      }
    };

    return () => {
      // Close all WebSocket connections when component unmounts
      Object.keys(tournaments).forEach((tournamentId) => {
        worker.postMessage({
          action: 'closeWebSocket',
          payload: { tournamentId },
        });
      });

      worker.terminate();
    };
  }, [tournaments]);

  return (
    <div>
      <h1>Championship Dashboard</h1>
      {Object.entries(tournaments).map(([id, tournament]) => (
        <div key={id}>
          <h2>{tournament.name}</h2>
          <p>Updates: {updates[id] ? JSON.stringify(updates[id]) : 'No updates yet'}</p>
        </div>
      ))}
    </div>
  );
};

export default ChampionshipDashboard;
```

76 3 comments 4 reposts