Implementation of Web Worker

CS-493: Enterprise Application Development

Supervisor
Mr. Atif Hussain

Project Developer

Amna Zafar 2021-CS-27

University of Engineering and Technology
Lahore, Pakistan

Contents

1 Introduction to WebWorker

1.1 What is WebWorker?

When executing scripts in an HTML page, the page becomes unresponsive until the script is finished. A web worker is a JavaScript that runs in the background, independently of other scripts, without affecting the performance of the page. You can continue to do whatever you want: clicking, selecting things, etc., while the web worker runs in the background. Web Workers are a simple means for web content to run scripts in background threads. The worker thread can perform tasks without interfering with the user interface. In addition, they can make network requests using the fetch() or XMLHttpRequest APIs. Once created, a worker can send messages to the JavaScript code that created it by posting messages to an event handler specified by that code (and vice versa).

1.2 How web workers can improve your website?

The browser uses a single thread (the main thread) to run all the JavaScript in a web page, as well as to perform tasks like rendering the page and performing garbage collection. Running excessive JavaScript code can block the main thread, delaying the browser from performing these tasks and leading to a poor user experience. On the web, JavaScript was designed around the concept of a single thread, and lacks capabilities needed to implement a multithreading model like the one apps have, like shared memory. Despite these limitations, a similar pattern can be achieved in the web by using workers to run scripts in background threads, allowing them to perform tasks without interfering with the main thread. Workers are an entire JavaScript scope running on a separate thread, without any shared memory.

2 How web worker work

2.1 Check Web Worker Support

Before creating a web worker, check whether the user's browser supports it:

```
    if (typeof(Worker) !== "undefined") {
  // Yes! Web worker support!
  // Some code.....
} else {
  // Sorry! No Web Worker support..
}
```

2.2 Create a Web Worker File

Now, let's create our web worker in an external JavaScript.Create any file name with extension .js

2.3 Create a Web Worker Object

Now that we have the web worker file, we need to call it from an HTML page. The following lines checks if the worker already exists, if not - it creates a new web worker object and runs the code in "$demo - workers.js$":

```
    if (typeof(w) == "undefined") {
  w = new Worker("demo_workers.js");
}
```

Then we can send and receive messages from the web worker. Add an ”onmessage” event listener to the web worker.

```
    w.onmessage = function(event){
  document.getElementById("result").innerHTML = event.data;
};
```

3 Advanced use cases of Web Workers

Web Workers are a valuable feature in web development that enable concurrent execution of scripts in the background, separate from the main thread. They are particularly useful for handling computationally intensive tasks without affecting the user interface responsiveness. Here are some advanced use cases of Web Workers:

- **Parallel Computing and Data Processing**
  Web Workers can be employed for parallel computing tasks, dividing a large computation into smaller chunks that can be processed simultaneously in separate threads. This is beneficial for tasks like complex mathematical calculations, data processing, and rendering.

- **Real-time Collaboration**
  In collaborative web applications, Web Workers can be used to facilitate real-time collaboration by handling background tasks such as synchronization, data processing, and conflict resolution. This ensures a smooth user experience while multiple users interact with the application simultaneously.

- **WebAssembly Integration**
  WebAssembly (Wasm) is a binary instruction format that enables near-native performance in web applications. Web Workers can be used in conjunction with WebAssembly to execute complex algorithms written in languages like C or Rust, providing high-performance capabilities for tasks such as cryptography, simulations, and other computationally intensive operations.

- **Machine Learning in the Browser**
  Web Workers can be utilized to run machine learning models in the browser efficiently. Libraries like TensorFlow.js leverage Web Workers to perform model inference, enabling developers to build AI-powered applications without causing significant delays in the user interface.

- **WebSocket Handling**
  Web Workers can be used to manage WebSocket connections, handling incoming data and events in the background. This is beneficial for real-time applications, chat systems, or any scenario where constant communication with a server is required without blocking the main thread.

- **Logging and Analytics**
  Web Workers can be used to collect and process analytics data in the background without affecting the main thread. This helps in maintaining a responsive user interface while still gathering important insights into user behavior and application performance.

4 Benefits of using web worker

Web Workers are a valuable feature in web development that enable parallel processing in the browser, allowing certain tasks to run in the background without affecting the main thread. Here are some benefits of using Web Workers:

- Multithreading
- Improved Performance
- Responsive User Interface
- Parallel Processing
- Distributed Computing
- Isolation and Security
- Consistent Performance Across Devices

5 Challanges of using web workers

While Web Workers offer various advantages, there are also challenges associated with their use. Here are some common challenges and strategies to overcome them:

- **Communication Overhead**
  Challenge: Communicating between the main thread and Web Workers involves message passing, which can introduce some overhead.
  Solution: Minimize the frequency of communication and send only essential data. Consider using transferable objects to avoid unnecessary data copying.

- **Limited Access to DOM**
  Challenge: Web Workers do not have direct access to the DOM, which can be limiting for certain tasks.
  Solution: Structure your application in a way that separates DOM-related tasks in the main thread and offload only non-DOM tasks to Web Workers. Use message passing to update the DOM based on the results obtained in the worker.

- **No Shared Memory**
  Challenge: Web Workers do not share memory with the main thread, making it challenging to share data efficiently.
  Solution: Use structured cloning or transferable objects to pass data between the main thread and workers. SharedArrayBuffer, which allows shared memory, is available but may require careful handling due to security concerns.

- **Dependency Management**
  Challenge: Web Workers run in a separate context and may not have access to the same dependencies as the main thread.
  Solution: Ensure that all necessary dependencies are explicitly loaded within the worker context. Some bundlers and build tools provide options to include dependencies in worker scripts.