Web Workers for background processing

🧠 Web Workers in JavaScript: Background Processing Explained

Web Workers allow JavaScript code to run in **background threads**, separate from the **main thread**. This is useful for **intensive computations**, **data processing**, or any task that might otherwise freeze the UI.

🔍 Why Use Web Workers?

The **main thread** in a browser handles:

- Rendering the UI
- User interactions
- Running JavaScript

Long-running tasks (like image processing, large loops, JSON parsing, etc.) can block this thread, causing the page to **become unresponsive**.

Web Workers solve this by **offloading such tasks to another thread**.

🧱 Types of Web Workers

1. **Dedicated Worker** – One-to-one relationship with the main script
2. **Shared Worker** – Can be accessed by multiple scripts (not widely used)
3. **Service Worker** – Used for caching, offline capabilities, etc. (different use case)

This explanation focuses on **Dedicated Workers**.

⚙️ How Web Workers Work

1. Create a new file for the worker (e.g., `worker.js`)

```
// worker.js
self.onmessage = function (e) {
  const result = e.data * 2;
  self.postMessage(result);
};
```

2. Use the worker in your main script

```
// main.js
const worker = new Worker('worker.js');

worker.postMessage(5); // send data to worker

worker.onmessage = function (e) {
  console.log('Result from worker:', e.data);
};
```

> self in the worker file refers to the global scope of the worker (like window in the main thread).

⛔ What You Can’t Do in Web Workers

- No access to the DOM (`document`, `window`)
- Limited access to browser APIs (e.g., can't manipulate the page)
- Communication is **message-based** using `postMessage`

🛠️ Practical Use Cases

- Large data parsing (e.g., CSV or JSON)
- Image processing or filters
- Encryption/decryption
- Complex calculations (e.g., physics simulations)

- Real-time code analysis (e.g., linters)

📦 Transferring Data (Zero-Copy)

You can transfer large data without copying by using `Transferable Objects`:

```
const buffer = new ArrayBuffer(1024);
worker.postMessage(buffer, [buffer]); // Now buffer is moved, not copied
```

🧪 Terminating Workers

```
worker.terminate();
```

This stops the worker immediately and releases memory.

⚠️ Error Handling

```
worker.onerror = function (e) {
  console.error('Worker error:', e.message);
};
```

🌐 Modern Alternatives

For very complex use cases or easier development:

- **Comlink** – Simplifies worker communication using proxies
- **Workerize** – Automatically turns functions into workers
- **Threads.js** – A popular abstraction for multithreading in JS

📌 Summary

Feature	Description

Runs in background	Prevents blocking the UI
Communicates via	`postMessage` and `onmessage`
No DOM access	Can't manipulate HTML/CSS directly
Terminate manually	Use `terminate()` when done
Great for	Heavy computation, file parsing, crypto

Let me know if you'd like a working demo or to integrate a web worker into a real-world project!