Jesus Ortega Async Notes Part 4

Introducing Workers

Workers enable you to run some tasks in a separate thread of execution. WHen you have long-running synchronous tasks in your program - the whole window becomes unresponsive. Fundamentally, this is because the program is single-threaded. A **thread** is a sequence of instructions that a program follows. If a program consists of a single thread, it can only do one thing at a time.

Workers give you the ability to run some tasks in a different thread, so you can start the task, then continue with other processing (such as handling user actions).

CAUTION:

With multithreaded code, you never know when your thread will be suspended and the other thread will get a chance to run. So if both threads have access to the same variables, it's possible for a variable to change unexpectedly at any time, causing bugs that are difficult to find.

The best way to avoid these problems on the web is to prevent your main code and worker code from having direct access to each other's variables. Workers and the main code run in completely separate worlds, and only interact by sending each other messages. In particular, this means that workers can't access the DOM.

TYPES OF WORKERS

1. Dedicated workers
2. Shared workers
3. Service workers

USING WEB WORKERS

We're going to use workers to run a prime number calculation:

Start by making a local copy of the files at:

https://github.com/mdn/learning-area/blob/main/javascript/asynchronous/workers/start

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

// When the user clicks "Generate primes", send a message to the worker.
// The message command is "generate", and the message also contains
"quota",
// which is the number of primes to generate.
document.querySelector('#generate').addEventListener('click', () => {
  const quota = document.querySelector('#quota').value;
  worker.postMessage({
    command: 'generate',
    quota,
  });
});

// When the worker sends a message back to the main thread,
// update the output box with a message for the user, including the number
of
// primes that were generated, taken from the message data.
worker.addEventListener('message', (message) => {
  document.querySelector('#output').textContent = `Finished generating
${message.data} primes!`;
});

document.querySelector('#reload').addEventListener('click', () => {
  document.querySelector('#user-input').value = 'Try typing in here
immediately after pressing "Generate primes"';
  document.location.reload();
});
```

In the above code (main.js):

1. We create the worker using the Worker() constructor. We pass in a URL pointing to the worker script.
2. Next, we add a click event handler to the ‘Generate Primes' button. Instead of calling the generatePrimes() function, we send a message to the worker using **worker.postMessage()**. This message can take an argument, and in this case, we’re passing a JSON object containing two properties.
    - **command**: a string identifying the thing we want the worker to do (in case our worker could do more than one thing)
    - **quota**: the number of primes to generate.
3. Add a *message* event handler to the **worker**. This is so the worker can tell us when it has finished, and pass us any resulting data. Our handler takes the data from the *data* property of the message, and writes it to the output element (the data is exactly the same as *quota*, so this is a bit pointless, but it shows the principle).
4. Implement the click event handler for the ‘Reload’ button.

The code for the worker is shown below:

```
// Listen for messages from the main thread.
// If the message command is "generate", call `generatePrimes()`
addEventListener("message", (message) => {
    if (message.data.command === 'generate') {
      generatePrimes(message.data.quota);
    }
  });

  // Generate primes (very inefficiently)
  function generatePrimes(quota) {

    function isPrime(n) {
      for (let c = 2; c <= Math.sqrt(n); ++c) {
        if (n % c === 0) {
```

```
            return false;
        }
      }
      return true;
    }

    const primes = [];
    const maximum = 1000000;

    while (primes.length < quota) {
      const candidate = Math.floor(Math.random() * (maximum + 1));
      if (isPrime(candidate)) {
        primes.push(candidate);
      }
    }

    // When we have finished, send a message to the main thread,
    // including the number of primes we generated.
    postMessage(primes.length);
}
```

The **worker** begins to listen for messages from the main script. It does this using addEventListener(), which is a global function in a worker. In the *message* event handler, the *data* property of the event contains a copy of the argument passed from the main script. If the main script passed the generate command, we call generatePrimes(), passing in the *quota* value from the message event.

The generatePrimes() function is just like the synchronous version, except instead of returning a value, we send a **message** to the main script when we are done. We use the **postMessage()** function for this, which like addEventListener() is a global function in a **worker**. The main script is listening for this message and will update the DOM when the message is received.