# Web Workers

Introductie en Basisprincipes

Web Workers

## 1. What are Web Workers?

Definition:

Web Workers allow you to run JavaScript code in the background on a separate thread from the main UI thread. This prevents the main thread from being blocked during intensive tasks.

Benefits:

Improved performance and smoother user experience by handling heavy computations in parallel.

## 2. Basic Principles of Web Workers

Creating a Worker:

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

Sending Messages to the Worker:

```
worker.postMessage('data');
```

Receiving Messages from the Worker:

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

Terminating the Worker:

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

## 3. Worker Lifecycle

Creation:

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

Messaging:

worker.postMessage() and worker.onmessage

Termination:

worker.terminate()

## 4. Limitations and Considerations

No DOM Access:

Workers cannot directly modify the DOM.

Global Object Restrictions:

No access to window, document, or localStorage.

Security:

Workers cannot use eval() and have restricted access to certain JavaScript APIs.

## 5. Basic Functionality Example

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

# Web Workers

Introductie en Basisprincipes

Web Workers

## 1. Multiple Web Workers

Example with Multiple Workers:

```
const worker1 = new Worker('worker1.js');
const worker2 = new Worker('worker2.js');

worker1.postMessage('Data for worker1');
worker2.postMessage('Data for worker2');
```

## 2. Inter-Worker Communication

Using postMessage() and onmessage for Inter-Worker Communication:

```
// worker1.js

const worker2 = new Worker('worker2.js');

worker2.onmessage = function(event) {
console.log('Message from worker2:',event.data); };

worker2.postMessage('Hello from worker1');
```

## 3. Shared Workers

Definition: Shared Workers can be shared between multiple windows or tabs in the same browser

Creating a shared worker:

```
const sharedWorker = new
SharedWorker('sharedWorker.js');
```

Communicating with a Shared Worker:

```
sharedWorker.port.postMessage
('Hello Shared Worker');

sharedWorker.port.onmessage = function(event) {
console.log('Message from shared worker:', event.data);
};
```

## 4. Shared Worker Example

```
let connections = 0;
onconnect = function(event) {
  const port = event.ports[0];
  connections++;
  port.postMessage(`Number of connections: ${connections}`);

port.onmessage = function(event) {
  console.log('Message from main thread:', event.data);
  };
};
```

# Web Workers

## Service Workers and Practical Examples

Web Workers

1. Service Workers

Definition:

Service Workers are a type of Web Worker designed for tasks like caching and offline support.

Registering a Service Worker:

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/service-worker.js')
    .then(function(registration) {
      console.log('Service Worker registered with scope:', registration.scope);
    }).catch(function(error) {
      console.error('Service Worker registration failed:', error);
    });
}
```

Service Worker Example:

```
self.addEventListener('install', function(event) {
  console.log('Service Worker installing.');
});

self.addEventListener('fetch', function(event) {
  console.log('Fetching:', event.request.url);
  event.respondWith(
    fetch(event.request).catch(function() {
      return new Response('Offline');
    })
  );
});
```

Practical Use Cases:

Heavy Computations: Use Web Workers to perform intensive calculations without blocking the UI thread.

```
const worker = new Worker('worker.js');
worker.postMessage({action: 'compute', value: 100});
worker.onmessage = function(event) {
console.log('Computed result:', event.data);
};
```

Data Processing: Process large data sets using Web Workers

```
onmessage = function(event) {
  if (event.data.action === 'processData') {
    const processedData = processData(event.data.data);
    postMessage(processedData);
  }
};
```

# Web Workers

Best Practices and Resources

Web Workers

## 1. Best Practices

Error Handling in Workers:

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

Optimization: Minimize the number of messages between workers to improve performance.

Security: Be cautious with data and external scripts loaded in Workers.

## 2. Common Pitfalls

Insufficient Communication: This can lead to race conditions or incomplete data processing.

Improper Memory Management: Ensure Workers are terminated properly to avoid memory leaks.

## 3. Further Reading and Resources

- MDN Web Docs:
  - Web Workers
  - Shared Workers
  - Service Workers
- Google Developers: Service Workers

## 4. Tools and Extensions

- Web Worker Debugger: For debugging Web Workers.
- Performance Profilers: For analyzing the performance of your Web Workers, such as Chrome DevTools or Firefox Developer Tools.