JavaScript

Databases and Web Applications Laboratory (LBAW)
Bachelor in Informatics Engineering and Computation (L.EIC)

Sérgio Nunes
Dept. Informatics Engineering
FEUP · U.Porto

Outline

- ➔ JavaScript fundamentals
- ➔ DOM manipulation
- ➔ Web APIs
- ➔ JavaScript Frameworks
- ➔ State management

Web APIs

Web APIs

- ➔ In addition to the language specification, HTML5 introduced several Web APIs that can be used with JavaScript. There is a large number of APIs in different stages of development.
  - ➔ Documents manipulation APIs (e.g. DOM, Drag and Drop)
  - ➔ Fetch remote data APIs (e.g. Fetch, Web Sockets)
  - ➔ Drawing and graphics manipulation APIs (e.g. Canvas, WebGL)
  - ➔ Audio and Video APIs (e.g. Web Audio, WebRTC)
  - ➔ Device APIs (e.g. Notification, Vibration, Fullscreen)
  - ➔ Client-side storage APIs (e.g. Web Storage, IndexedDB)

Storage APIs

Local Storage

- ➔ The Local Storage API provides a simple way to store key-value pairs persistently in the browser.
- ➔ Data stored using Local Storage remains available even after the browser is closed and reopened.
- ➔ The API offers a synchronous interface, making it easy to use but potentially blocking the main thread.
- ➔ Storage events allow different windows or tabs to communicate when data changes.

Local Storage Example

➔ Local Storage provides persistent key-value storage in the browser.

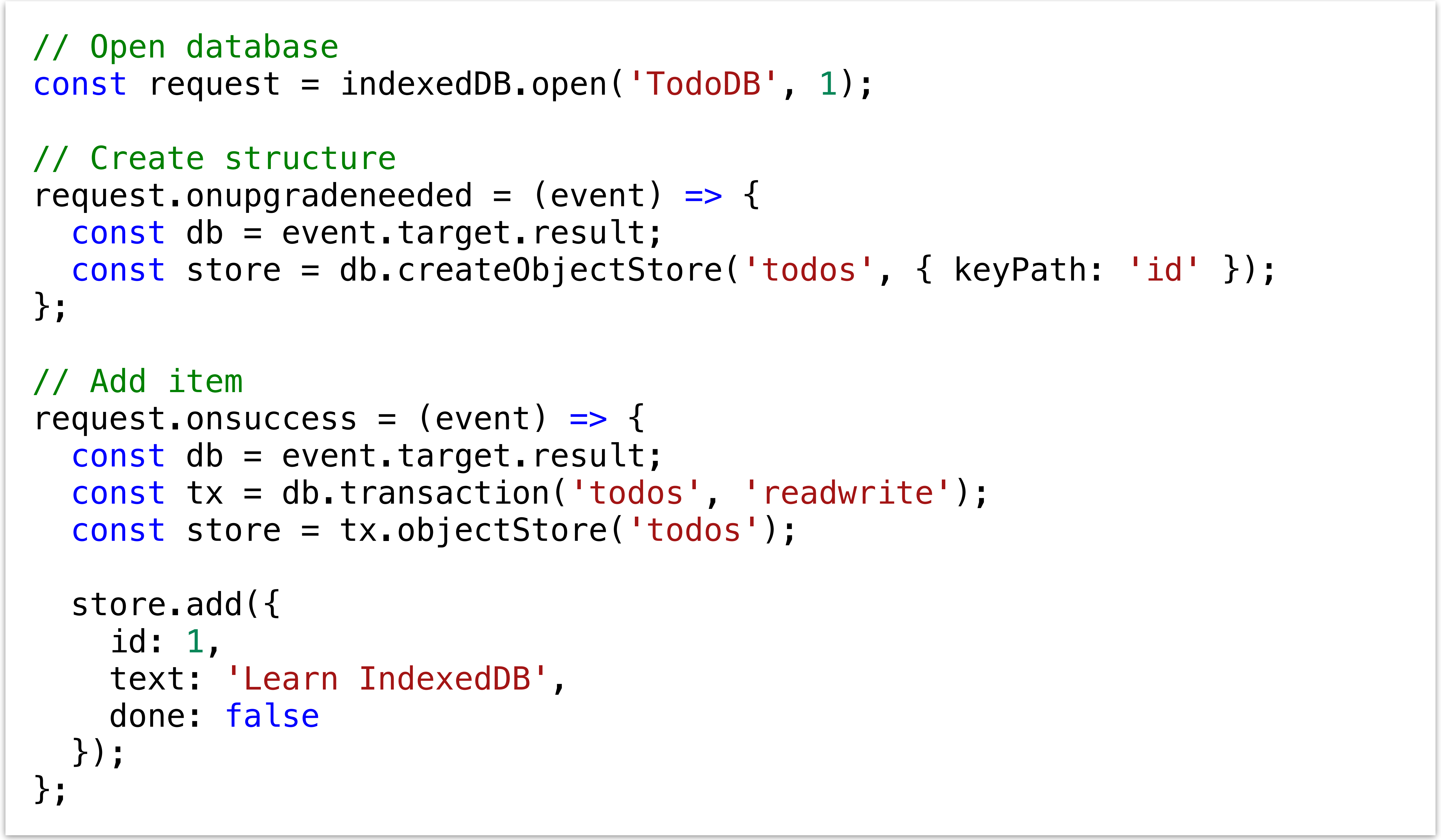
```javascript
// Store data
localStorage.setItem('username', 'john');

// Read data
const username = localStorage.getItem('username');
console.log('Stored username:', username);  // Outputs: "john"

// Remove data
localStorage.removeItem('username');

// Clear all data
localStorage.clear();
```

Local Storage

- ➔ IndexedDB serves as a low-level API for client-side storage of significant amounts of structured data.
- ➔ The database supports indexes to enable high-performance searches of your stored data.
- ➔ All operations in IndexedDB are executed within transactions, ensuring data integrity.
- ➔ The API is asynchronous by design, preventing blocking operations on the main thread.

IndexedDB Example

➔ IndexedDB enables structured data storage with indexes and queries.



```javascript
// Open database
const request = indexedDB.open('TodoDB', 1);

// Create structure
request.onupgradeneeded = (event) => {
  const db = event.target.result;
  const store = db.createObjectStore('todos', { keyPath: 'id' });
};

// Add item
request.onsuccess = (event) => {
  const db = event.target.result;
  const tx = db.transaction('todos', 'readwrite');
  const store = tx.objectStore('todos');

  store.add({
    id: 1,
    text: 'Learn IndexedDB',
    done: false
  });
};
```

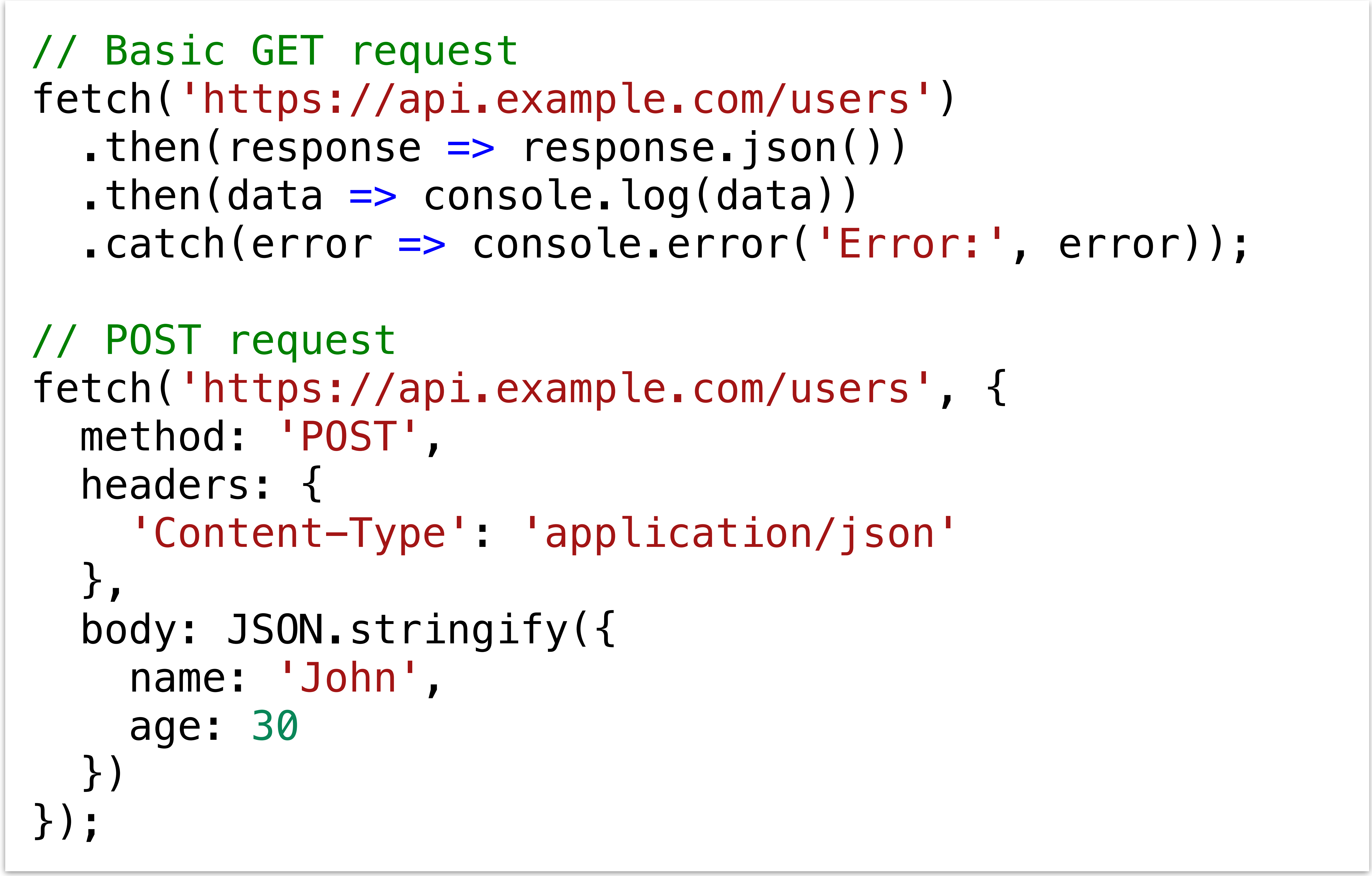
Network APIs

Fetch API

- ➔ The Fetch API provides a powerful and flexible replacement for XMLHttpRequest.
- ➔ It uses Promises to handle responses, making asynchronous code more manageable.
- ➔ The API supports various request types and allows fine-grained control over HTTP requests.
- ➔ Response parsing supports multiple formats including JSON, text, and binary data.

Fetch API Example

➔ Fetch provides a modern interface for making HTTP requests.



```javascript
// Basic GET request
fetch('https://api.example.com/users')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));

// POST request
fetch('https://api.example.com/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    name: 'John',
    age: 30
  })
});
```

WebSocket API

➔ WebSockets enable full-duplex communication channels over a single TCP connection.

➔ The protocol provides a way to exchange real-time data between clients and servers.

➔ WebSocket connections remain open, eliminating the need for repeated HTTP requests.

➔ The API includes built-in support for handling connection states and errors.

WebSockets Example

- WebSockets enable real-time bidirectional communication.

```javascript
// Create connection
const socket = new WebSocket('ws://example.com/socket');

// Send message
socket.onopen = () => {
  socket.send('Hello Server!');
};

// Receive message
socket.onmessage = (event) => {
  console.log('Message from server:', event.data);
};

// Handle errors
socket.onerror = (error) => {
  console.error('WebSocket error:', error);
};
```

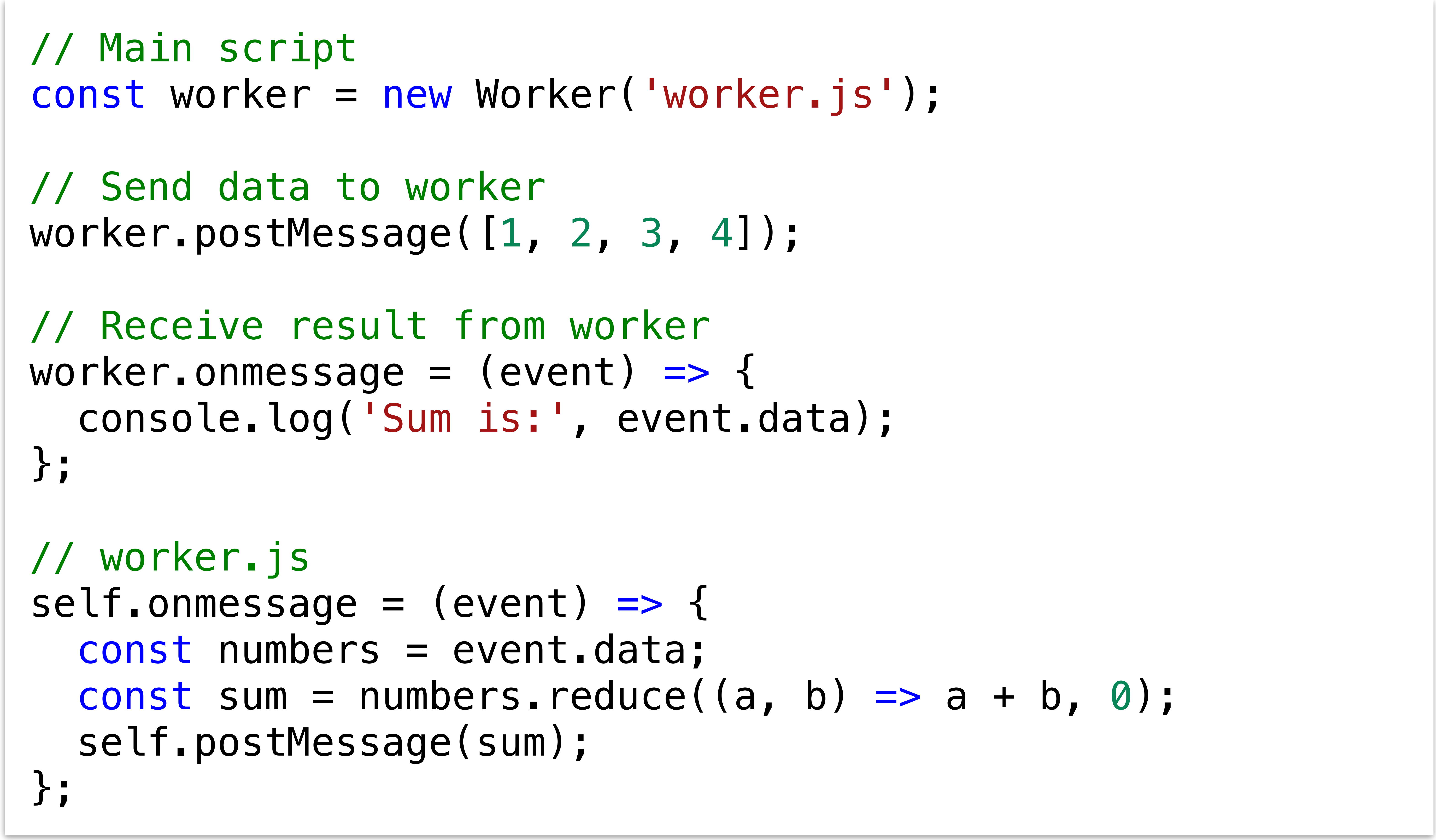
Background Processing

Web Workers

- ➔ Web Workers enable parallel execution of scripts in background threads.
- ➔ They prevent computationally intensive tasks from blocking the main UI thread.
- ➔ Workers communicate with the main thread through a messaging system.
- ➔ They cannot directly access the DOM but can handle complex calculations and data processing.

Web Worker Example

➔ ➔ Runs JavaScript in background threads.



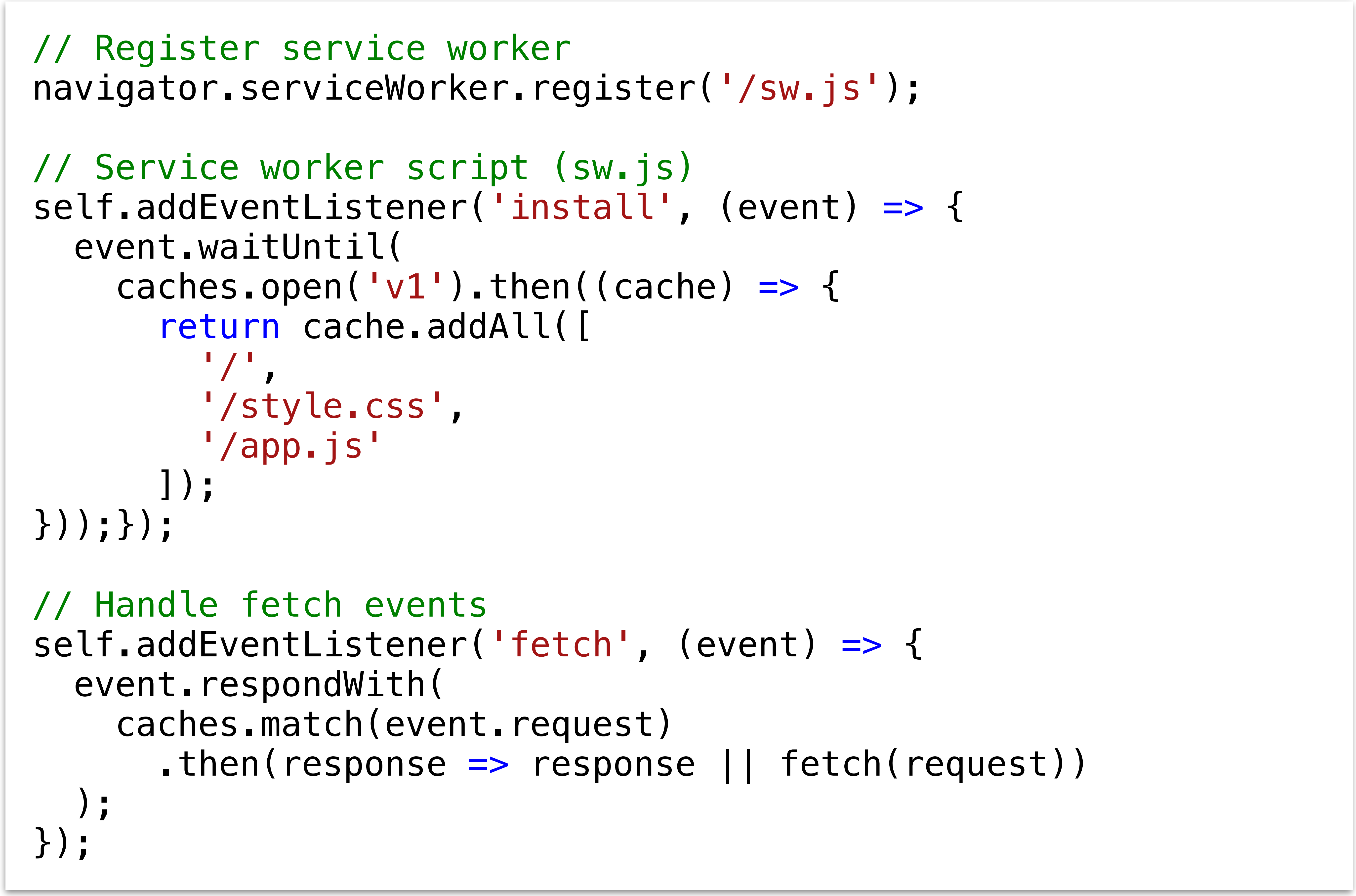
```
// Main script
const worker = new Worker('worker.js');

// Send data to worker
worker.postMessage([1, 2, 3, 4]);

// Receive result from worker
worker.onmessage = (event) => {
  console.log('Sum is:', event.data);
};

// worker.js
self.onmessage = (event) => {
  const numbers = event.data;
  const sum = numbers.reduce((a, b) => a + b, 0);
  self.postMessage(sum);
};
```

Service Workers

- ➔ Service Workers act as proxy servers sitting between web applications, browsers, and the network.
- ➔ They enable offline functionality by intercepting network requests and serving cached responses.
- ➔ The API supports push notifications and background sync capabilities.
- ➔ Service Workers follow a strict security model and require HTTPS except for local development.

Service Worker Example

➔ Acts as a proxy between web app, browser, and network.



```
// Register service worker
navigator.serviceWorker.register('/sw.js');

// Service worker script (sw.js)
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open('v1').then((cache) => {
      return cache.addAll([
        '/',
        '/style.css',
        '/app.js'
      ]);
}));});

// Handle fetch events
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
      .then(response => response || fetch(request))
  );
});
```

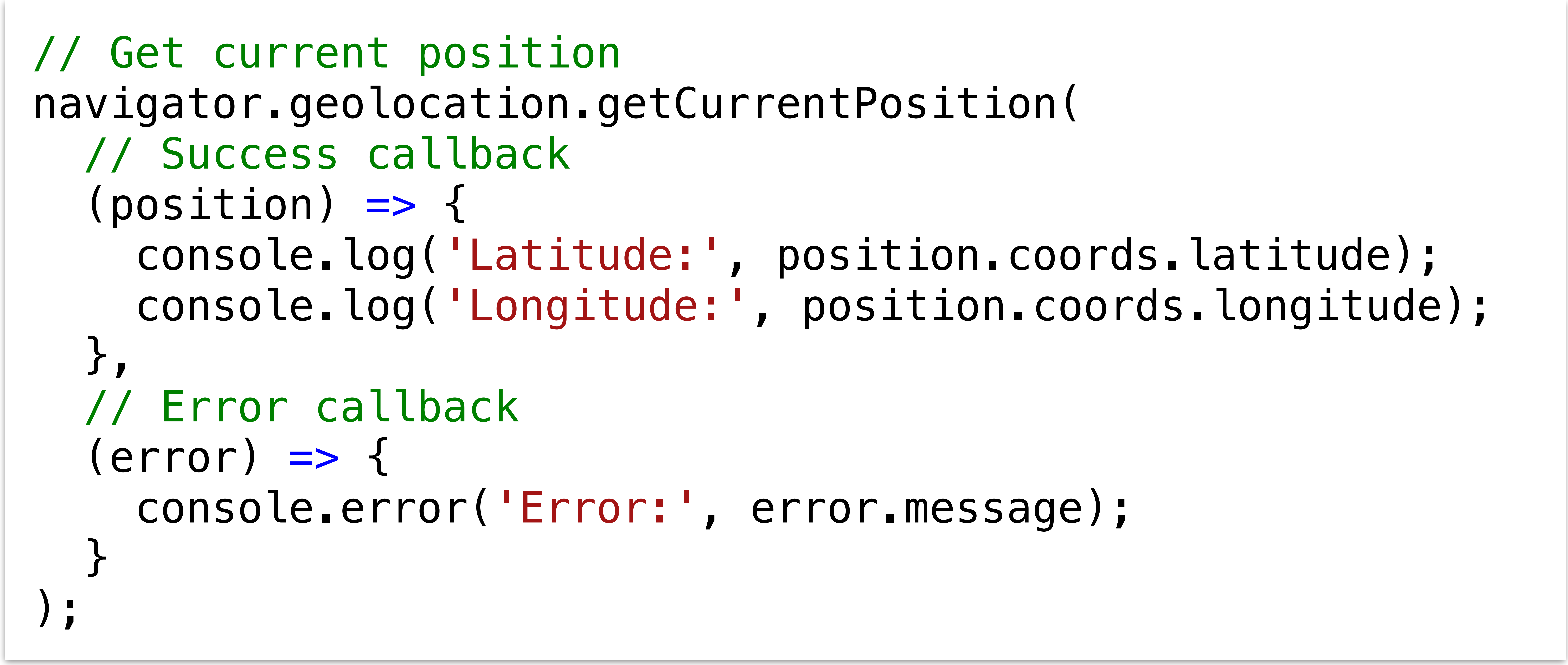
Device Integration

Geolocation API

- ➔ The Geolocation API enables web applications to access the user's geographical location.
- ➔ It provides both one-time location requests and continuous position updates.
- ➔ The API supports different levels of accuracy depending on application needs.
- ➔ Location access requires explicit user permission for privacy reasons.
- ➔ Common sources of location information include Global Positioning System (GPS) and location inferred from network signals such as IP address, RFID, WiFi and Bluetooth MAC addresses, and GSM/CDMA cell IDs, as well as user input.

Geolocation Example

- ➔ Gets the user's geographical location.



```
// Get current position
navigator.geolocation.getCurrentPosition(
  // Success callback
  (position) => {
    console.log('Latitude:', position.coords.latitude);
    console.log('Longitude:', position.coords.longitude);
  },
  // Error callback
  (error) => {
    console.error('Error:', error.message);
  }
);
```

Media Devices

- ➔ The Media Devices API provides access to connected media input devices like cameras and microphones.
- ➔ It enables web applications to capture and manipulate media streams in real-time.
- ➔ The API includes constraints to specify desired device capabilities and characteristics.
- ➔ Applications must handle permission requests and device availability gracefully.

Media Devices Example

➔ Accesses camera and microphone.

```
// Access camera
navigator.mediaDevices.getUserMedia({
  video: true
})
  .then(stream => {
    const video = document.querySelector('video');
    video.srcObject = stream;
  })
  .catch(error => {
    console.error('Camera access error:', error);
  });
```

Battery Status Example

➔ Gets device battery information.

```
navigator.getBattery().then(battery => {
  // Check battery level
  console.log('Battery level:', battery.level * 100 + '%');

  // Listen for changes
  battery.onlevelchange = () => {
    console.log('Battery level changed:', battery.level * 100 + '%');
  };
});
```

Clipboard Example

- ➔ Interacts with the system clipboard.

```
// Copy text
async function copyText(text) {
  try {
    await navigator.clipboard.writeText(text);
    console.log('Text copied!');
  } catch (error) {
    console.error('Failed to copy:', error);
  }
}

// Read text
async function pasteText() {
  try {
    const text = await navigator.clipboard.readText();
    console.log('Pasted text:', text);
  } catch (error) {
    console.error('Failed to paste:', error);
  }
}
```