WEBSOCKETS

INTRODUCTION AUX WEBSOCKETS

QU'EST-CE QU'UN WEBSOCKET ?

WebSocket est un **protocole de communication** permettant une interaction **bidirectionnelle** et en **temps réel** entre un client et un serveur sur une seule connexion **longue durée**.

COMPARAISON AVEC D'AUTRES PROTOCOLES (HTTP, AJAX)

Protocole	Communication	Temps réel	Connexion
HTTP	Unidirectionnelle	Non	A chaque requête
AJAX	Bidirectionnelle	Semi	A chaque requête
WebSocket	Bidirectionnelle	Oui	1 seule connexion

CRÉATION D'UN WEBSOCKET

SYNTAXE

```
const ws = new WebSocket("ws://exemple.com");
```

EXEMPLES D'UTILISATION

```
const ws = new WebSocket("ws://exemple.com");
ws.onopen = () => console.log("Connexion WebSocket ouverte");
```

ÉVÉNEMENTS WEBSOCKET

ONOPEN

SYNTAXE

```
ws.onopen = () => {
  // Code à exécuter lors de la connexion
};
```

EXEMPLES D'UTILISATION

```
ws.onopen = () => {
  console.log("WebSocket ouvert");
  ws.send("Message à envoyer au serveur");
};
```

ONMESSAGE

SYNTAXE

```
ws.onmessage = (event) => {
  // Code à exécuter lors de la réception d'un message
};
```

EXEMPLES D'UTILISATION

```
ws.onmessage = (event) => {
  console.log("Message reçu:", event.data);
};
```

ONERROR

SYNTAXE

```
ws.onerror = (error) => {
  // Code à exécuter en cas d'erreur
};
```

EXEMPLES D'UTILISATION

```
ws.onerror = (error) => {
  console.error("Erreur WebSocket:", error);
};
```

ONCLOSE

SYNTAXE

```
ws.onclose = (event) => {
  // Code à exécuter lors de la **fermeture de la connexion**
};
```

EXEMPLES D'UTILISATION

```
ws.onclose = (event) => {
  console.log("WebSocket fermé:", event.code, event.reason);
};
```

ENVOI ET RÉCEPTION DE DONNÉES

SYNTAXE

```
ws.send(data); // Envoyer des données au serveur
```

EXEMPLES D'UTILISATION

```
ws.send("texte");
ws.send(new Blob(["blob"]));
ws.send(new ArrayBuffer(16));
```

FORMATS DE DONNÉES

- Texte
- Blob
- ArrayBuffer

FERMETURE ET GESTION DES ERREURS

WEBWORKERS

INTRODUCTION AUX WEBWORKERS

QU'EST-CE QU'UN WEBWORKER ?

Un WebWorker est un mécanisme permettant d'exécuter des **scripts** en arrière-plan de manière **concurrente** par rapport au thread principal, sans bloquer la page web.

COMPARAISON AVEC D'AUTRES MÉTHODES D'EXÉCUTION CONCURRENTE

- **WebWorkers** vs **setInterval/setTimeout** : Les WebWorkers ne bloquent pas le thread principal, contrairement aux fonctions setInterval et setTimeout.
- **WebWorkers** vs **Promises/async-await** : Les deux permettent une exécution asynchrone, mais les WebWorkers sont plus adaptés aux tâches lourdes.

CRÉATION D'UN WEBWORKER

SYNTAXE

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

EXEMPLES D'UTILISATION

```
const worker = new Worker("imageProcessing.js");
```

INTERACTION AVEC LE FICHIER JAVASCRIPT DU WORKER

Le fichier JavaScript du Worker (ex: imageProcessing.js) est séparé du fichier principal et ne partage pas le même **contexte**.

- Le contexte d'un **fichier principal** inclut l'accès à l'objet **window, document** et le **DOM**.
- Le contexte d'un **fichier Worker** est limité à l'objet **self** et aux opérations spécifiques aux Web Workers.

COMMUNICATION AVEC UN WEBWORKER

POSTMESSAGE()

SYNTAXE

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

EXEMPLES D'UTILISATION

```
worker.postMessage({ url: "image.jpg", amount: 50 });
```

ONMESSAGE()

SYNTAXE

```
worker.onmessage = (event) => {
  console.log(event.data);
};
```

EXEMPLES D'UTILISATION

```
worker.onmessage = (event) => {
  const { processedImage } = event.data;
  displayImage(processedImage);
};
```

GESTION DES ERREURS

ONERROR()

SYNTAXE

```
worker.onerror = (error) => {
  console.error(error);
};
```

EXEMPLES D'UTILISATION

```
worker.onerror = (error) => {
  console.error("An error occurred: ", error);
};
```

TERMINAISON D'UN WEBWORKER

SYNTAXE

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

Note : La méthode `.terminate()` permet d'arrêter immédiatement un **Web Worker** en cours d'exécution, même s'il n'a pas terminé ses tâches.

EXEMPLES D'UTILISATION

```
worker.terminate();
console.log("WebWorker terminated.");
```

BEST PRACTICES

- Utilisez des **WebWorkers** pour les tâches lourdes et longues
- Préférez les **transférables** pour minimiser les coûts de transfert de données
- Évitez d'utiliser trop de WebWorkers simultanément
- Gérez les **erreurs** et l'état du worker

TYPES DE WEBWORKERS

DEDICATED WORKERS

Travaillent pour une **seule page web** et ne peuvent pas être **partagés**.

Avantages	Inconvénients
Isolation du code	Ressources additionnelles
Performances améliorées	Communication inter-script

SHARED WORKERS

Peuvent être **partagés** entre plusieurs pages web ou iframes ayant le même **domaine d'origine**.

SERVICE WORKERS

Agissent comme des **proxies** du réseau, interceptent les **requêtes** et gèrent le **cache**.

COMPARAISON ET EXEMPLES D'UTILISATION

- **Dedicated Workers** : traitement d'image, calculs lourds
- **Shared Workers** : communication entre pages web
- **Service Workers** : améliorer les performances, fonctionnalités hors ligne

INTERACTION ENTRE WEBSOCKETS ET WEBWORKERS

UTILISATION CONJOINTE

WebSockets et **WebWorkers** peuvent être combinés pour améliorer les performances des applications web en permettant la communication en temps réel et l'exécution simultanée de tâches complexes.

EXEMPLES D'UTILISATION

1. Exécution de **calculs intensifs** dans un **WebWorker** tout en recevant et envoyant des données via un **WebSocket**. ```javascript // main.js const worker = new Worker("worker.js"); const socket = new WebSocket("url");

worker.onmessage = (e) => { socket.send(e.data); };

// worker.js self.onmessage = (e) => { // Traitement intensif postMessage(result); };

```
2. Gérer les messages entrants du **WebSocket** dans un **WebWorker** pour décharger le thread principal.
```javascript
// main.js
const worker = new Worker("worker.js");

worker.onmessage = (e) => {
    // Gérer les données reçues du WebWorker
};

// worker.js
const socket = new WebSocket("url");

socket.onmessage = (e) => {
    // Traitement du message
```
```

COMMUNICATION ENTRE WEBSOCKETS ET WEBWORKERS

Les données peuvent être **envoyées** entre les **WebSockets** et les **WebWorkers** via des **messages**.

WebSockets	WebWorkers
Communication en temps réel	Exécution en arrière-plan

```
// Dans le WebWorker
self.addEventListener('message', function (event) {
  // Traiter les données du message
  console.log("Message reçu: ", event.data);
});

// Envoyer un message au WebWorker
worker.postMessage("message à envoyer");
```

EXEMPLES D'UTILISATION

1. Envoi des données du **WebSocket** au **WebWorker**. ```javascript // main.js const worker = new Worker("worker.js"); const socket = new WebSocket("url");

socket.onmessage = (e) => { worker.postMessage(e.data); };

// worker.js self.onmessage = (e) => { // Traiter les données reçues depuis le WebSocket };

```
2. Envoi des données du **WebWorker** au **WebSocket**.
```javascript
// main.js
const worker = new Worker("worker.js");
const socket = new WebSocket("url");

worker.onmessage = (e) => {
    socket.send(e.data);
};

// worker.js
self.onmessage = (e) => {
    // Traiter les données reçues depuis le WebSocket
    postMessage(result);
```
```

BEST PRACTICES

- Utilisez les **WebWorkers** pour l'exécution de tâches non DOM et les **WebSockets** pour la communication en temps réel.
- Privilégiez les transferts de données sous forme de **messages** plutôt que de partager des objets directement entre WebSockets et WebWorkers.
- N'utilisez pas les WebWorkers pour les tâches qui n'ont pas besoin de **parallélisme** et qui ne sont pas gourmandes en ressources.