

INTRODUCTION

RAPPEL SUR LES ARCHITECTURES CLIENT-SERVEUR

Les **architectures client-serveur** sont un modèle de communication où les clients envoient des requêtes aux serveurs qui leur répondent.

- **Client** : Application/frontend permettant l'interaction utilisateur
- **Serveur** : Fournit des services/ressources aux clients

RÔLE DE JAVASCRIPT DANS LA COMMUNICATION CLIENT-SERVEUR

JavaScript est un langage de programmation principalement utilisé sur le côté **client** (navigateur) pour :

- Manipuler la structure et le contenu du document **HTML**
- Communiquer avec le serveur en effectuant des **requêtes HTTP**
- Gérer les interactions et les animations côté utilisateur (**UX**)

Les méthodes et API JavaScript permettent d'envoyer des requêtes et de recevoir des réponses du serveur de manière **asynchrone**, sans interrompre l'interaction utilisateur.

CONNEXION SERVEUR EN JAVASCRIPT

INTRODUCTION

RAPPEL SUR LES ARCHITECTURES CLIENT-SERVEUR

- **Architecture client-serveur**
- Échanges de données entre **client** et **serveur**
- Client envoie des **requêtes**, serveur envoie des **réponses**

RÔLE DE JAVASCRIPT DANS LA COMMUNICATION CLIENT-SERVEUR

- **Manipulation des données** côté client
- **Envoi de requêtes** et traitement des réponses
- Communication **asynchrone** avec **Ajax**

CONNEXION SERVEUR EN JAVASCRIPT

FETCH API

La **Fetch API** est une méthode moderne et évolutive pour effectuer des **requêtes HTTP** en JavaScript, facilitant la communication avec les serveurs.

PRÉSENTATION DE L'API FETCH

L'API **Fetch** est basée sur les **Promesses**, offrant une méthode plus simple et lisible pour gérer les **requêtes HTTP** et les réponses par rapport à **XMLHttpRequest**.

SYNTAXE ET MÉTHODE POUR UTILISER FETCH

Pour utiliser **Fetch**, utilisez la fonction `fetch()` avec l'**URL** comme argument :

```
fetch('https://api.example.com/data')
```

GESTION DES PROMESSES

Fetch renvoie une **Promesse** qui résout en une réponse (`Response`) si la requête aboutit, et qui rejette une erreur sinon.

THEN()

Pour gérer le succès de la **requête** :

```
fetch('https://api.example.com/data')  
  .then(response => response.json()) // Convertit la réponse en JSON  
  .then(data => console.log(data)); // Traite les données reçues
```

CATCH()

Pour gérer les erreurs :

```
fetch('https://api.example.com/data')  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error('Erreur :', error)); // Gère les erreurs
```

EXEMPLES D'UTILISATION

Requête GET :

```
fetch('https://api.example.com/data')  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error('Erreur :', error));
```

Requête POST :

```
fetch('https://api.example.com/data', {  
  method: 'POST',  
  headers: { 'Content-Type': 'application/json' },  
  body: JSON.stringify({ key: 'value' })  
})  
  .then(response => response.json())  
  .then(result => console.log(result))  
  .catch(error => console.error('Erreur :', error));
```

CONNEXION WEBSOCKET

CONNEXION WEBSOCKET

Les **WebSockets** permettent une communication **bidirectionnelle** entre un client et un serveur sur une connexion **persistante**.

SYNTAXE

Pour créer une connexion **WebSocket** en JavaScript, utilisez la syntaxe suivante :

```
var socket = new WebSocket("ws://example.com/socket");
```

PROPRIÉTÉS ET MÉTHODES PRINCIPALES

send()

Méthode pour envoyer des **données** au serveur via la connexion **WebSocket**.

```
socket.send("Hello, server!");
```

close()

Méthode pour fermer la **connexion WebSocket**.

```
socket.close();
```

GESTION DES ÉVÉNEMENTS

onopen

Événement déclenché lorsque la **connexion WebSocket** est établie.

```
socket.onopen = function(event) {  
    console.log("Connexion établie");  
};
```

onmessage

Événement déclenché lors de la réception d'un **message** du serveur.

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


onerror

Événement déclenché en cas d'erreur sur la connexion **WebSocket**.

```
socket.onerror = function(event) {  
    console.log("Erreur:", event.message);  
};
```

onclose

Événement déclenché lorsque la **connexion WebSocket** est fermée.

```
socket.onclose = function(event) {  
  console.log("Connexion fermée:", event.reason);  
};
```

EXEMPLES D'UTILISATION

```
// Établir une connexion WebSocket
var socket = new WebSocket("ws://example.com/socket");

// Gérer les événements
socket.onopen = function(event) {
    console.log("Connexion établie");

    // Envoyer un message au serveur
    socket.send("Hello, server!");
};

socket.onmessage = function(event) {
    console.log("Message reçu:", event.data);
};

// Fermer la connexion
socket.close();
```

UTILISATION POUR LES APPLICATIONS WEB EN TEMPS RÉEL

- **Fetch** : Idéal pour des requêtes ponctuelles, mais pas pour les mises à jour en temps réel
- **WebSocket** : Parfait pour les applications en temps réel (chat, jeux en ligne, etc.)

UTILISATION POUR LES APPLICATIONS DE TRANSFERT DE DONNÉES ASYNCHRONE

- **Fetch** : Solution moderne et simple pour les transferts de données asynchrone
- **WebSocket** : Peut être utilisé pour les transferts de données asynchrone, mais généralement moins pratique que XMLHttpRequest et Fetch

BONNES PRATIQUES

GESTION DES ERREURS ET EXCEPTIONS

- Utiliser les événements **onerror**, **catch()** pour détecter les erreurs de communication
- Vérifier le **statut** de la réponse (**status**)
- Utiliser des blocs **try-catch** pour anticiper les erreurs JavaScript

SÉCURITÉ ET AUTHENTIFICATION

- Utiliser **HTTPS** pour les communications sécurisées
- Ne pas stocker d'informations sensibles dans le code **JavaScript** (tokens, clés)
- Utiliser des mécanismes d'**authentification** : tokens, JWT, OAuth

PERFORMANCES ET OPTIMISATION DES ÉCHANGES DE DONNÉES

- **Minimiser** la quantité de données échangées (compression, pagination)
- Utiliser des **en-têtes HTTP** pour la mise en cache (Cache-Control, ETag)
- Utiliser des **WebSockets** pour les applications en temps réel
- Préférer la **Fetch API** pour les requêtes simples et asynchrones

OUTILS ET RESSOURCES

BIBLIOTHÈQUES ET FRAMEWORKS POUR FACILITER LA CONNEXION AU SERVEUR

- **jQuery** : Bibliothèque populaire qui inclut des méthodes simplifiées pour les requêtes **AJAX**
- **Axios** : Bibliothèque spécialisée pour les requêtes **HTTP**
- **Socket.io** : Bibliothèque pour gérer les connexions **WebSockets** en temps réel