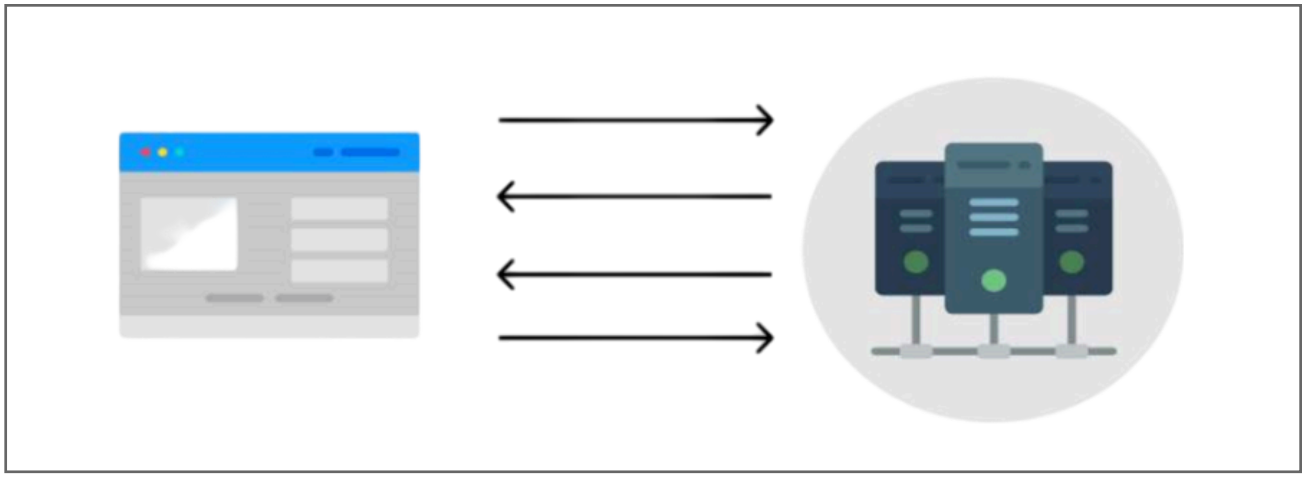
LES WEBSOCKETS



DÉFINITION

Technologie de communication bidirectionnelle en temps réel entre un navigateur web et un serveur.

DANS LE DÉTAIL

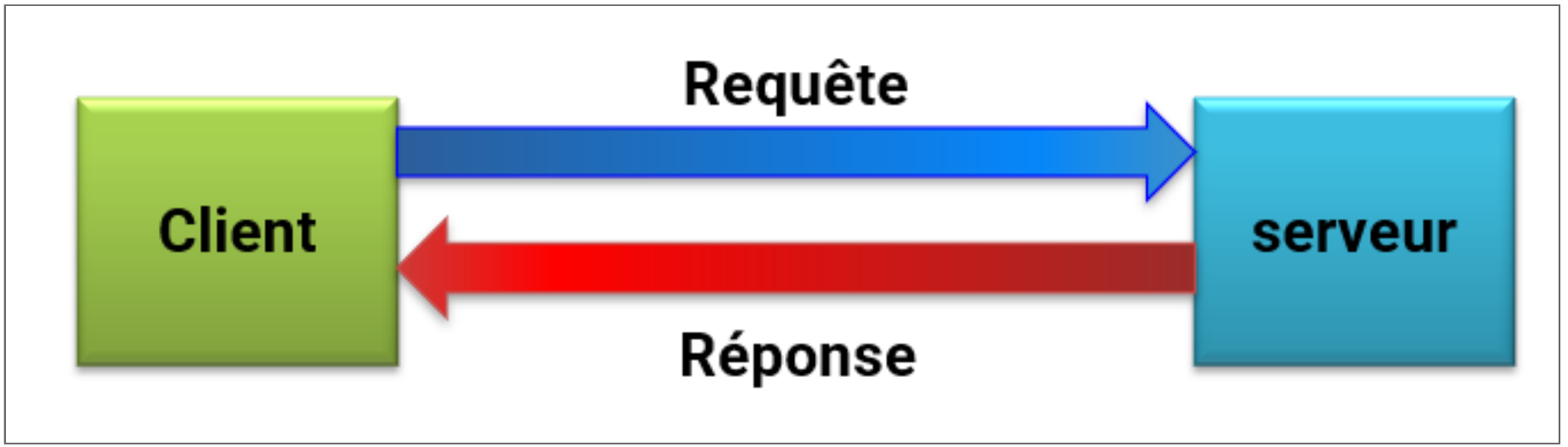
- A l'opposé des requêtes HTTP, qui sont unidirectionnelles (le client envoie une requête et le serveur répond)
- Les WebSockets permettent une communication continue et instantanée
- Le client et le serveur peuvent envoyer des données à tout moment sans avoir à attendre une nouvelle requête.

UTILITÉ

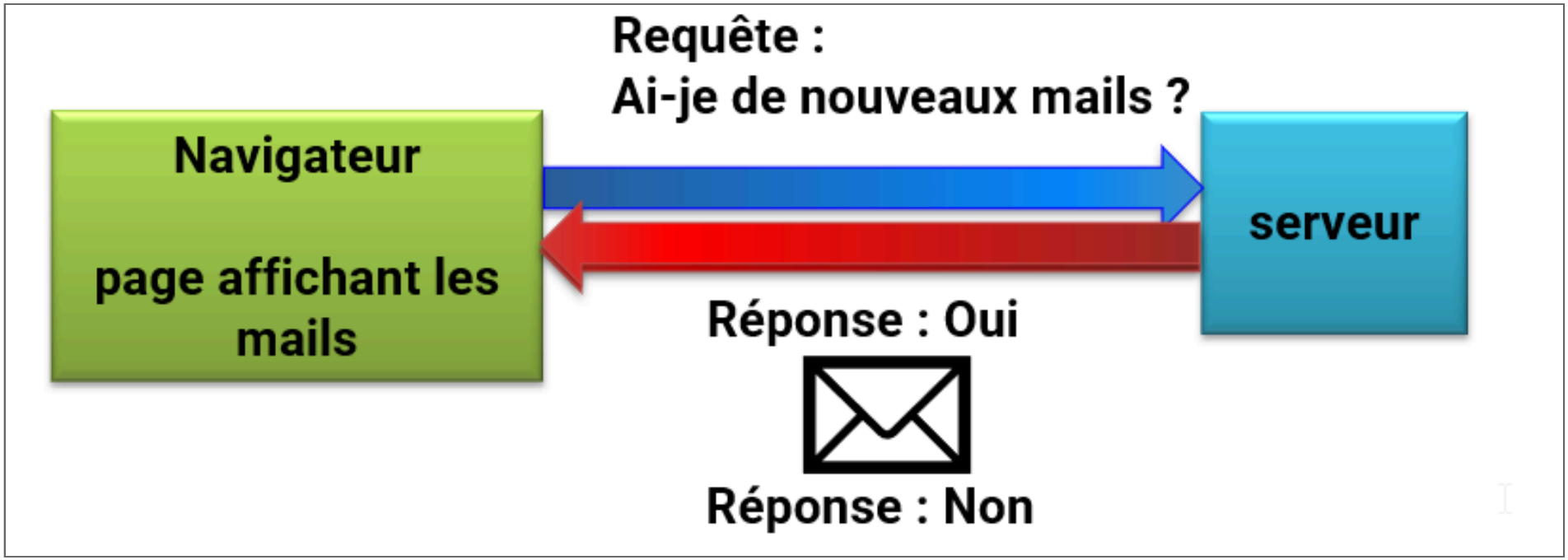
- Particulièrement adaptés pour les applications nécessitant des mises à jour en temps réel (chat en ligne, les jeux multijoueurs, dashboard de monitoring,...)

HTTP VS WEBSOCKETS

En HTTP, le serveur ne peut envoyer des donner que si le client les demande.



Le fonctionnement de l'AJAX force le client à faire une requête pour avoir des données.

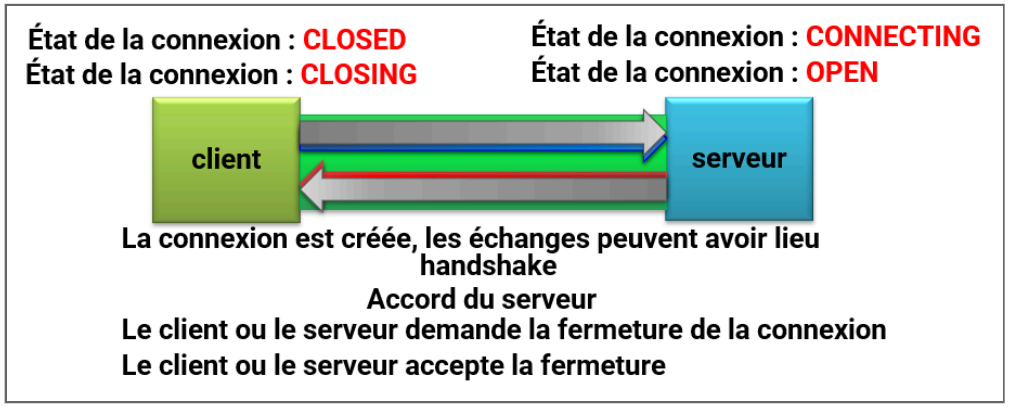


Il y a une forte consommation de ressources : bande passante, traitement serveur, etc.

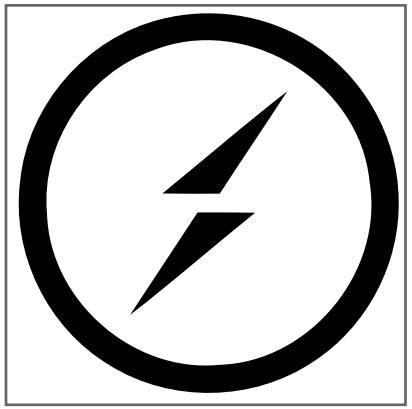
PAR WEB SOCKET :

COMMENT FONCTIONNE LES WEBSOCKETS?

- Le client initie une connexion vers le serveur par un handshake avec le protocole HTTP
- Le serveur accepte à la demande
- Le transfert peut commencer entre les deux parties



SOCKET.IO



- Bibliothèque JavaScript populaire pour les applications web en temps réel.
- Permet la communication bidirectionnelle en temps réel entre les navigateurs web (clients) et un serveur.

INSTALLATION

On installe la dépendences nécessaire

```
npm i socket.io
```

MISE EN PLACE (1/4)

Créons un fichier html: `index.html`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>Test Socket</title>
    <script src="./index.js"></script>
  </head>
  <body>
    <h1> test connexion Socket.io</h1>
  </body>
</html>
```

MISE EN PLACE (2/4)

Dans notre fichier `index.js` on ajoute :

```
console.log('Ok')
```

MISE EN PLACE (3/4)

Créons un fichier `serveur.js`

```
1   import express from 'express'
2   import path from 'path'
3   import { fileURLToPath } from 'url'
4
5   const app = express()
6
7   // Constante pour connaitre --dirname, et --filename
    en ES6 module
8   const __filename = fileURLToPath(import.meta.url);
9   const __dirname = path.dirname(__filename);
10
11  app.get("*", (req, res) => {
12      res.sendFile(path.join(__dirname, "index.html"))
13  })
14
15  // Notre serveur va écouter le port 8080
16  app.listen(8080, () => {
17      console.log("Votre serveur écoute sur le port
    8080");
18  })
```

```
18  });
```

MISE EN PLACE (4/4)

On Oublie pas de modifier notre `package.json`

```
"start": "nodemon serveur.js"
```

CRÉATION D'UN SERVEUR SOCKET.IO

- Importons `socket.io`, puis configurons le afin de définir sur quel serveur **HTTP** il doit se connecter.

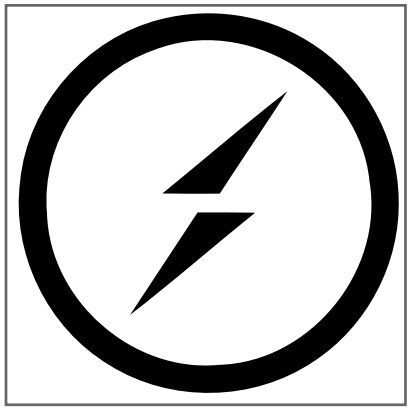
```
import { Server } from 'socket.io'

// On positionne notre écoute de port dans une variable
const server = app.listen(8080, () => {
    console.log("Votre serveur écoute sur le port 8080");
})

// On crée une instance d'un serveur
const io = new Server(server)
```

Par convention, on utilisera `io` pour nommer une instance de serveur

LES PARAMÈTRES LES PLUS UTILISÉS



CORS (CROSS-ORIGIN RESOURCE SHARING)

- Vous pouvez définir les politiques CORS pour votre serveur Socket.IO. Cela est particulièrement important si votre client Socket.IO est hébergé sur un domaine différent de votre serveur Socket.IO.
  Exemple :

```
const io = require('socket.io')(server, {
  cors: {
    origin: "http://exemple.com",
    methods: ["GET", "POST"]
```

```
    }
});
```

PATH

- Vous pouvez spécifier le chemin sur lequel le serveur Socket.IO va répondre. Par défaut, il utilise /socket.io.
  Exemple :

```
const io = require('socket.io')(server, {
  path: '/mon_socket'
});
```

SERVECLIENT

- Détermine si le serveur doit servir le client Socket.IO. Par défaut, c'est `true`. Si vous avez déjà le client Socket.IO sur votre front-end, vous pouvez le désactiver.
  Exemple :

```
const io = require('socket.io')(server, {
  serveClient: false
});
```

PINGTIMEOUT ET PINGINTERVAL

- Ces paramètres sont utilisés pour déterminer la santé de la connexion. `pingTimeout` est le temps en millisecondes après lequel un client sans réponse sera déconnecté, et `pingInterval` est le temps en millisecondes pour envoyer un nouveau ping au client.
  Exemple :

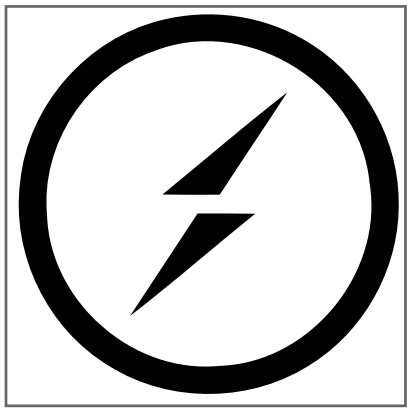
```
const io = require('socket.io')(server, {
  pingTimeout: 60000,
```

```
    pingInterval: 25000
});
```

COOKIE

- Vous pouvez configurer si Socket.IO doit envoyer un cookie avec l'ID de session. Par défaut, c'est activé.
  Exemple :

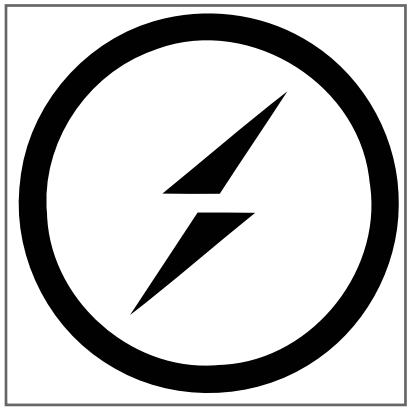
```
const io = require('socket.io')(server, {
  cookie: false
});
```



Liste complète **ici** !

SOCKET.IO

COTÉ CLIENT



VOICI DIFFÉRENTES MÉTHODES POUR INSTANCIER NOTRE SOCKET.IO COTÉ CLIENT

- Sur un site en html, on peut récupérer un `script` envoyé par le serveur avec cette syntaxe

```
<script src="/socket.io/socket.io.js"></script>
```

Si notre client est dans un autre `serveur`, ou si on utilise un `framework` :

- Nous utiliserons la libairie `socket.io-client` (lien **ici**)

```
npm i socket.io-client
```

Une fois fait, nous utilisons la méthode `io` au besoin

```
import { io } from 'socket.io-client';
```

CONNECTION

Dans l'exemple suivant, nous utiliserons notre la 1ere méthode

- Dans le fichier `index.html` on ajoute notre script

```
<head>
   <title>Document</title>
   <script src="/socket.io/socket.io.js"></script>
   <script src="./index.js"></script>
 </head>
```

Nous pouvons accéder a `io` dans notre `index.js`

```
// Crée et initialise une nouvelle connexion socket du
côté client.
const socket = io();

// Gestionnaire d'événements :
// Déclenché quand le client établit avec succès
// une connexion avec le serveur Socket.IO.
socket.on("connect", () => {
    console.log("Je me connecté !");
})
```

EXPLICATION

- `const socket = io()` : crée et initialise une nouvelle connexion socket du côté client.
- `io()` : Fonction fournie par Socket.IO. Établit automatiquement une connexion au serveur Socket.IO en cours d'exécution.

- Si aucun argument n'est fourni à la fonction `io()`, elle se connecte au serveur qui sert la page web actuelle.
- `socket.on("connect", () => {...});` est un gestionnaire d'événements. Il est déclenché lorsque le client établit avec succès une connexion avec le serveur Socket.IO.

- `connect` est un événement intégré de **socket.io** qui est automatiquement émis lorsqu'une connexion est établie.
- La fonction de rappel (callback) à l'intérieur de `socket.on("connect", ...)` est exécutée une fois que l'événement connect se produit.
- Dans notre exemple, il affiche : `Connecté !` dans la console du navigateur.

- Ce gestionnaire d'événements est souvent utilisé pour initialiser des fonctionnalités qui ne doivent être activées qu'après l'établissement d'une connexion réussie (demander des données au serveur, activer certaines interactions de l'interface utilisateur, ...)
- Une connection coté client, générera un id pour chaque nouvelle connexion

```
socket.on("connect", () => {
        console.log("Je me suis connecté !" + socket.id);
    });
// Je me suis connecté !akM3wjEb2yl-vTUgAAAD
```

SOCKET.USE()

- La méthode `socket.use()` dans **socket.io** est une fonction middleware qui vous permet d'exécuter du code pour chaque socket entrant avant que n'importe quel autre gestionnaire d'événements ne soit appelé.

1. **Fonctionnement**

- Cette méthode prend une fonction de rappel (callback) comme argument qui est exécutée chaque fois qu'un événement est reçu sur ce socket.
- La fonction de rappel reçoit 2 arguments : le 1er est l'objet socket, et le 2eme est une fonction pour passer au prochain middleware dans la pile.

2. Syntaxe

```
socket.use((socket, next) => {
  // Votre code ici
  next();
});
```

- `socket` est l'instance de connexion du client actuel. Vous pouvez l'utiliser pour accéder aux informations du client ou pour envoyer des messages au client.
- `next` est une fonction qui, lorsqu'elle est appelée, passe l'exécution au prochain middleware enregistré (ou au

gestionnaire d'événements si aucun autre middleware n'est enregistré).

3. **Utilisation**

- **Authentification** : Vérifie si le client est authentifié avant de lui permettre d'écouter ou d'émettre des événements supplémentaires.
- **Validation** : Vérifie la validité des données envoyées par le client avant de les traiter.
- **Journalisation** : Enregistre les activités ou les données de chaque socket pour le débogage ou la surveillance.

4. **Gestion d'erreurs**

- Si vous devez arrêter l'exécution dans le middleware (ex: si l'authentification échoue), vous pouvez passer une erreur à la fonction `next`.
  exemple : `next(new Error('Authentification échouée'));`.
- Cette erreur sera ensuite transmise au client et peut être traitée en conséquence.

5. Exemple

```
socket.use((socket, next) => {
  if (socket.handshake.query.token === "secret_token") {
    next();
  } else {
    next(new Error("Authentification échouée"));
  }
});
```

CONCLUSION

- En utilisant `socket.use()`, vous avez un contrôle précis sur le traitement et la validation des sockets entrants dans votre application **socket.io**, ce qui améliore la sécurité et la robustesse de votre application en temps réel.

SOCKET.EMIT()

- La méthode `socket.emit()` est une fonction fondamentale utilisée pour envoyer des messages d'un client à un serveur ou inversement, ou entre clients via un serveur.
- Essentielle pour la communication en temps réel dans les applications utilisant **socket.io**

1. Fonctionnement

- `socket.emit()` est utilisé pour envoyer des messages ou des données d'un point à un autre en utilisant une connexion **socket.io**
- Elle peut être utilisée côté serveur pour envoyer des données à un client spécifique, ou côté client pour envoyer des données au serveur.

```
socket.emit('nom_de_l_evenement', data);
```

- `nom_de_l_evenement` est une chaîne qui identifie le type de message ou d'événement que vous envoyez.
- Les destinataires utiliseront ce nom pour écouter et réagir à cet événement.
- `data` est l'information ou l'objet que vous souhaitez envoyer. Cela peut être une chaîne, un nombre, un objet, un tableau, etc.

1. Exemple

- **Coté serveur:** Envoyer un message à un client spécifique.

```
socket.emit('message', 'Ceci est un message du serveur');
```

- **Coté client**: Envoyer des données au serveur

```
socket.emit('submit_form', { nom: 'Alice', age: 30 });
```

3. Communication bidirectionnelle

- Les sockets peuvent écouter (`socket.on()`) et émettre (`socket.emit()`) des événements dans les deux sens, du client au serveur et du serveur au client.

4. Émission à Plusieurs Clients:

- Pour envoyer un message à tous les clients connectés, utilisez `io.emit()` côté serveur.

```
io.emit('nouvel_utilisateur', 'Un nouvel utilisateur s
est connecté');
```

- Pour envoyer à tous les clients sauf à celui qui émet, utilisez `socket.broadcast.emit()`

```
socket.broadcast.emit('notification', 'Un autre
utilisateur s est connecté')
```

5. Réception des données

- Pour recevoir ces données, les clients ou le serveur doivent écouter l'événement spécifié avec `socket.on('nom_de_l_evenement', callback)`.

  Ex : pour écouter l'événement **message** nous utiliserons :

```
socket.on('message', (data) => {
   console.log(data); // Affiche: 'Ceci est un message du serveur'
});
```

6. Fiabilité et Performance

- **socket.io** garantit que les messages sont reçus dans l'ordre et gère les détails bas niveau de la reconnexion et de la conservation de l'état des sockets.

CONCLUSION

- `socket.emit()` est donc un outil puissant pour la communication en temps réel, permettant une interaction flexible et dynamique entre clients et serveur dans une application web.

LES ÉVENEMENTS

- **socket.io** propose plusieurs événements intégrés qui permettent de gérer la communication et le cycle de vie des connexions socket.

CÔTÉ SERVEUR (NODE.JS)

- **connection / connect:** Se déclenche lorsqu'un client se connecte au serveur.

```
io.on('connection', (socket) => {
// Gérer la nouvelle connexion
});
```

- **disconnect:** Se produit lorsqu'un client se déconnecte.

```
socket.on('disconnect', (reason) => {
  // Gérer la déconnexion
});
```

- **error:** Se déclenche en cas d'erreur sur le socket.

```
socket.on('error', (error) => {
  // Gérer l'erreur
});
```

- **disconnecting:** Se déclenche lorsque le client commence le processus de déconnexion.

CÔTÉ CÔTÉ CLIENT (JAVASCRIPT DANS LE NAVIGATEUR):

- **connect:** Se déclenche lorsque le client se connecte avec succès au serveur.

```
socket.on('connect', () => {
  // Le client est connecté
});
```

- **disconnect:** Se produit lorsque le client est déconnecté du serveur.

```
socket.on('disconnect', (reason) => {
  // Gérer la déconnexion
});
```

- **connect_error:** Se déclenche lorsqu'une tentative de connexion échoue.

```
socket.on('connect_error', (error) => {
  // Gérer l'erreur de connexion
});
```

- **reconnect:** Se déclenche lorsqu'un client se reconnecte au serveur après une déconnexion.
- **reconnect_attempt:** Se déclenche à chaque tentative de reconnexion.
- **reconnecting:** Se produit lorsqu'une tentative de reconnexion est en cours.
- **reconnect_error:** Se déclenche lorsqu'une tentative de reconnexion échoue.
- **reconnect_failed:** Se produit lorsque toutes les tentatives de reconnexion ont échoué.

ÉVÉNEMENTS PERSONNALISÉS

- En plus des événements intégrés, vous pouvez définir et utiliser vos propres événements personnalisés pour envoyer et recevoir des données spécifiques à votre application.

```
  // Côté serveur
  socket.emit('mon_evenement', { data: 'quelques données'
});

  // Côté client
  socket.on('mon_evenement', (data) => {
    console.log(data);
    // { data: 'quelques données' }
  });
```

CONCLUSION

- Ces événements forment la base de la communication dans **socket.io** permettant aux développeurs de gérer la connexion, la déconnexion, la réception et l'envoi de données, et de traiter les erreurs.
- La capacité de définir des événements personnalisés offre une grande flexibilité pour créer des interactions spécifiques à l'application.

MERCI À TOUS-TES !

Speaker notes