

MÉTHODE FETCH

PRÉSENTATION DE LA FONCTION FETCH

La fonction **Fetch** est une méthode native de **JavaScript** pour récupérer des informations à partir d'**APIs**.
Elle simplifie les **requêtes HTTP** et les opérations d'accès aux données.

SYNTAXE ET UTILISATION

La syntaxe de base pour utiliser la méthode **Fetch** est la suivante :

```
fetch(url)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

REQUÊTES GET

La méthode **Fetch** utilise les requêtes **GET** par défaut, sans avoir besoin de spécifier l'option "method".

EXEMPLES D'UTILISATION

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

REQUÊTES POST

Pour effectuer des **requêtes POST**, vous devez fournir un **objet d'options** avec la méthode "**POST**" spécifiée.

EXEMPLES D'UTILISATION

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

REQUÊTES PUT

Pour effectuer des requêtes **PUT**, vous devez fournir un objet d'options avec la méthode "**PUT**" spécifiée.

```
fetch('https://api.exemple.com/posts/1', {
  method: 'PUT',
  headers: {
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    title: 'Nouveau titre',
    body: 'Nouveau corps',
  }),
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error(error));
```

EXEMPLES D'UTILISATION

```
fetch("https://api.example.com/data/1", {
  method: "PUT",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({ key: "updatedValue" })
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error(error));
```

REQUÊTES DELETE

Pour effectuer des **requêtes DELETE**, vous devez fournir un **objet d'options** avec la méthode "**DELETE**" spécifiée.

```
fetch("https://api.example.com/items/1", {  
  method: "DELETE",  
})  
.then((response) => response.json())  
.then((data) => console.log(data))  
.catch((error) => console.error(error))
```

EXEMPLES D'UTILISATION

```
fetch("https://api.example.com/data/1", {
  method: "DELETE"
})
  .then(response => response.json())
  .then(data => console.log("Item deleted successfully"))
  .catch(error => console.error(error));
```

GESTION DES RÉPONSES DE L'API

PROMESSES

Les **promesses** sont des objets JavaScript qui représentent l'éventuelle réalisation ou l'échec d'une **opération asynchrone**.

Exemple de création d'une Promesse :

```
const monPromesse = new Promise((resolve, reject) => {
    // code effectuant une opération asynchrone
});
```

Les méthodes principales de gestion des promesses :

- **.then()** : s'exécute lorsque la promesse est résolue
- **.catch()** : s'exécute lorsque la promesse est rejetée
- **.finally()** : s'exécute quelle que soit l'issue de la promesse

PROMESSES - CONCEPT

Une **promesse** est un objet qui représente la réalisation (ou l'échec) d'une **opération asynchrone** et sa valeur résultante.

- **En attente (pending)** : L'opération n'a pas encore été exécutée.
- **Réalisée (fulfilled)** : L'opération a été exécutée avec succès.
- **Rejetée (rejected)** : L'opération a échoué.

PROMESSES - SYNTAXE ET UTILISATION

```
const promise = new Promise((resolve, reject) => {
  // code asynchrone
});

promise.then((value) => {
  // code exécuté si la promesse est réalisée
}).catch((error) => {
  // code exécuté si la promesse est rejetée
});
```

ASYNC / AWAIT

Le mot-clé `async` permet de déclarer une fonction **asynchrone**, qui renvoie une **promesse**. Le mot-clé `await` permet d'**attendre** la résolution d'une promesse.

```
async function maFonctionAsync() {  
    // Utiliser await pour attendre la résolution d'une promesse  
    const resultat = await fonctionQuiRenvoieUnePromesse();  
    return resultat;  
}  
  
maFonctionAsync().then(resultat => console.log(resultat));
```

ASYNC / AWAIT - CONCEPT

- `async`: Utilisé pour déclarer une **fonction asynchrone**. Une fonction asynchrone retourne une **promesse**.
- `await`: Utilisé **uniquement** dans une fonction asynchrone, permet d'attendre la **RÉSOLUTION** d'une promesse.

ASYNC / AWAIT - SYNTAXE ET UTILISATION

```
async function myFunction() {  
  try {  
    const result = await asyncOperation();  
    // code exécuté après la résolution de la **promesse**  
  } catch (error) {  
    // code exécuté en cas d'**erreur**  
  }  
}
```

MANIPULATION DES DONNÉES JSON

PRÉSENTATION DU FORMAT JSON

JSON (**JavaScript Object Notation**) est un format **léger** d'échange de données. Il est facile pour les humains de le lire et l'écrire. Il est facile pour les machines de l'**analyser** et le **générer**.

JSONPARSE()

La méthode `JSON.parse()` permet de convertir une chaîne de caractères **JSON** en un objet **JavaScript**.

SYNTAXE

```
JSON.parse(text, reviver)
```

EXEMPLES D'UTILISATION

```
let jsonString = '{"name": "John", "age": 30, "city": "New York"}';  
  
let jsonObj = JSON.parse(jsonString);  
  
console.log(jsonObj.name); // Résultat: "John"
```

JSON.stringify()

La méthode `JSON.stringify()` permet de convertir un objet ou une valeur JavaScript en une chaîne de caractères **JSON**.

Exemple d'utilisation :

```
const person = {  
    name: "John",  
    age: 30,  
};  
  
const jsonString = JSON.stringify(person);  
console.log(jsonString);  
// résultat : '{"name": "John", "age": 30}'
```

SYNTAXE

```
JSON.stringify(value, replacer, space)
```

JSON.stringify() permet de convertir une **valeur** en chaîne JSON.

- **value** : la valeur à convertir en chaîne JSON
- **replacer** (optionnel) : une fonction ou un tableau pour modifier les valeurs avant de les convertir
- **space** (optionnel) : le nombre d'espaces à utiliser pour l'indentation ou une chaîne de caractères pour utiliser comme séparateur

EXEMPLES D'UTILISATION

```
let jsonObj = {  
    name: "John",  
    age: 30,  
    city: "New York"  
};  
  
let jsonString = JSON.stringify(jsonObj);  
  
console.log(jsonString);  
// Résultat: '{"name": "John", "age": 30, "city": "New York"}'
```

GESTION DES ERREURS

TRY / CATCH

Pour gérer les erreurs lors des appels d'API en **JavaScript**, on utilise la structure **try / catch** qui permet de "capturer" les erreurs et d'agir en conséquence.

```
try {
  // Code susceptible de générer une erreur
} catch (erreur) {
  // Code exécuté en cas d'erreur
}
```

SYNTAXE ET UTILISATION

```
try {
    // Bloc de code à exécuter
} catch (erreur) {
    // Bloc de code exécuté en cas d'erreur
}
```

LES ERREURS COURANTES AVEC FETCH

ERREURS DE CONNEXION

Les erreurs de connexion se produisent lorsque le **client** est incapable d'établir une connexion avec le **serveur**.

```
fetch(url)
  .then((response) => {
    // Gérer la réponse
  })
  .catch((error) => {
    console.error("Erreur de connexion :", error);
  });
}
```

ERREURS CÔTÉ SERVEUR/API (CODES D'ERREUR HTTP)

Les erreurs côté serveur sont signalées par des **codes d'erreur HTTP** (ex : 404 Not Found, 500 Internal Server Error).

```
fetch(url)
  .then((response) => {
    if (!response.ok) {
      throw new Error(`Erreur HTTP : ${response.status}`);
    }
    return response.json();
  })
  .then((data) => {
    // Gérer les données
  })
  .catch((error) => {
    console.error("Erreur côté serveur/API :", error);
  });
}
```

CAS PRATIQUES

EXEMPLE DE MANIPULATION D'UNE API SIMPLE

Cet exemple montre comment utiliser `fetch()` pour récupérer des données à partir d'une **API simple** sans **authentification**.

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

EXEMPLE DE MANIPULATION D'UNE API AVEC AUTHENTIFICATION

Pour accéder à une API sécurisée, vous devez inclure un **jeton d'authentification** dans les en-têtes de votre requête. Voici un exemple d'utilisation d'authentification avec `fetch()`.

```
fetch('https://api.example.com/secure-data', {
  headers: {
    'Authorization': 'Bearer ' + token
  }
})
  .then(response => response.json())
  .then(data => console.log(data))
```

EXEMPLE DE CRÉATION D'UNE MINI-APPLICATION UTILISANT UNE API REST

Créons une mini-application qui affiche des informations sur les utilisateurs à partir d'une **API REST**.

1. Créez un fichier HTML avec un élément `div` pour afficher les données :

```
<!DOCTYPE html>
<html>
<head>
  <script src="app.js"></script>
</head>
<body>
  <div id="user-info"></div>
</body>
</html>
```

2. Dans le fichier app.js, utilisez fetch() pour récupérer des données depuis l'API et les afficher :

```
function displayUserInfo(user) {
  document.getElementById('user-info').innerHTML = `
    <h2>${user.name}</h2>
    <p>Email: ${user.email}</p>
  `;
}

fetch('https://api.example.com/users/1')
  .then(response => response.json())
  .then(data => displayUserInfo(data))
```

