JavaScript API

QU'EST-CE QU'UNE API ?

Une API, ou Interface de Programmation d'Application (Application Programming Interface en anglais), est un ensemble de règles et de protocoles qui permettent à différentes applications de communiquer entre elles. Elle définit les moyens par lesquels les composants logiciels doivent interagir.

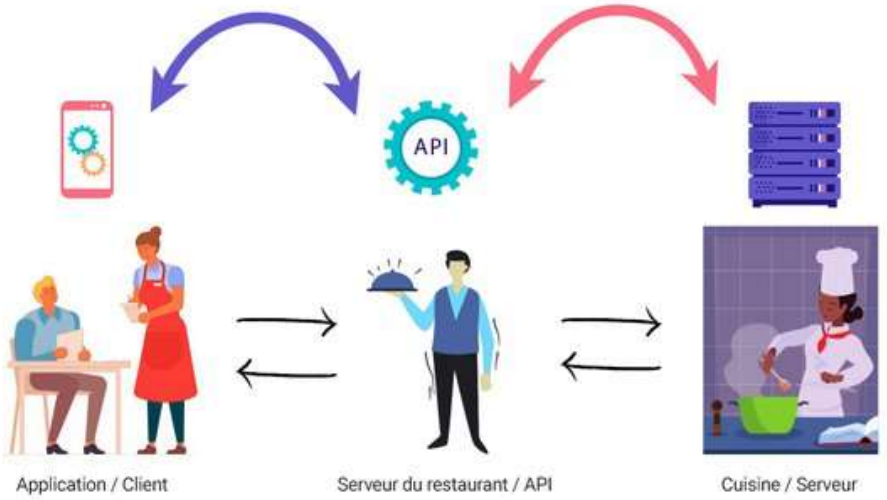
Principaux points:

1.Communication entre logiciels : Une API permet à des logiciels distincts de se parler. Elle offre un moyen structuré et normalisé pour que deux applications puissent échanger des données et des fonctionnalités.
2.Règles définies : Une API spécifie les règles et les conventions à suivre pour interagir avec un service ou une application. Cela inclut les formats de données acceptés, les méthodes d'accès, etc.
3.Abstraction : L'API agit comme une couche d'abstraction entre deux parties, masquant les détails internes de l'implémentation. Cela permet aux développeurs de travailler avec une interface simplifiée sans avoir à connaître tous les détails internes.
4.Réutilisation du code : Les API facilitent la réutilisation du code en permettant à différents logiciels de tirer parti des fonctionnalités existantes sans avoir à recréer l'ensemble du programme.

QU'EST-CE QU'UNE API ?

Exemple analogique :
Imaginez une API comme un menu dans un restaurant. Le menu représente les options disponibles (fonctionnalités ou données) que le restaurant (serveur ou application) offre à ses clients (autres applications ou services). Les clients n'ont pas besoin de connaître la cuisine (implémentation interne) pour commander et obtenir leur plat (résultat attendu).

Exemple pratique :
Dans le contexte du développement web, une API peut être utilisée pour permettre à un site web d'accéder aux fonctionnalités ou aux données d'un service tiers. Par exemple, une API de géolocalisation pourrait permettre à une application web de récupérer la position géographique d'un utilisateur.



HTTP (HYPERTEXT TRANSFER PROTOCOL)

HTTP est un protocole de communication utilisé sur le Web pour transférer des données entre un client et un serveur. C'est le fondement de toute communication sur le World Wide Web.

principaux aspects d'HTTP :

1. Protocole de communication :

HTTP est un protocole qui définit comment les messages sont structurés et transmis entre le client (généralement un navigateur web) et le serveur.

2. Basé sur le modèle requête-réponse :

L'interaction entre le client et le serveur se fait par l'envoi de requêtes depuis le client et de réponses depuis le serveur. Chaque requête du client est associée à une réponse du serveur.

HTTP (HYPERTEXT TRANSFER PROTOCOL)

3. Méthodes HTTP :

Les méthodes définissent le type d'action que le client souhaite effectuer sur une ressource donnée. Les méthodes couramment utilisées incluent :

GET : Récupérer des données depuis le serveur.
POST : Envoyer des données au serveur pour traitement.
PUT : Mettre à jour des données existantes sur le serveur.
DELETE : Supprimer des données sur le serveur.

4. URI (Uniform Resource Identifier) :

Chaque ressource accessible via HTTP est identifiée par une URI, souvent sous la forme d'une URL (Uniform Resource Locator).

HTTP (HYPERTEXT TRANSFER PROTOCOL)

5. En-têtes HTTP :

Les en-têtes sont des informations supplémentaires incluses dans les requêtes et les réponses pour fournir des métadonnées sur la requête ou la réponse. Ils peuvent inclure des informations telles que le type de contenu, la longueur du contenu, etc.

6. Corps de la requête et du message :

Les requêtes peuvent inclure un corps qui contient les données à envoyer au serveur. Les réponses peuvent également inclure un corps contenant les données renvoyées par le serveur.

HTTP (HYPERTEXT TRANSFER PROTOCOL)

7. Codes de statut HTTP :

Les codes de statut indiquent le résultat de la requête ou de la réponse :

1xx - Informations :

100 Continue : Le serveur a reçu la requête et l'attend pour continuer le processus.

2xx - Succès :

200 OK : La requête a été traitée avec succès.

201 Created : La requête a été traitée avec succès, et une nouvelle ressource a été créée.

204 No Content : La requête a été traitée avec succès, mais il n'y a pas de contenu à renvoyer.

3xx - Redirection :

301 Moved Permanently : La ressource demandée a été déplacée de manière permanente vers une nouvelle URL.

302 Found (ou 303 See Other) : La ressource demandée a été trouvée, mais elle est temporairement située à une autre URL.

304 Not Modified : La ressource n'a pas été modifiée depuis la dernière demande.

HTTP (HYPERTEXT TRANSFER PROTOCOL)

4xx - Erreur du client :

400 Bad Request : La requête du client est incorrecte ou mal formée.
401 Unauthorized : L'accès à la ressource est refusé en raison d'une authentification insuffisante.
403 Forbidden : L'accès à la ressource est interdit, même avec une authentification.
404 Not Found : La ressource demandée n'a pas été trouvée sur le serveur.

5xx - Erreur du serveur :

500 Internal Server Error : Une erreur interne du serveur s'est produite.
502 Bad Gateway : Le serveur agit comme une passerelle ou un proxy, et a reçu une réponse incorrecte du serveur en amont.
503 Service Unavailable : Le serveur n'est pas disponible. Il peut être temporairement surchargé ou en cours de maintenance.

8. Sécurité avec HTTPS :

HTTPS (HTTP Secure) est la version sécurisée de HTTP qui utilise une couche de chiffrement (SSL/TLS) pour sécuriser les données transitant entre le client et le serveur.

POSTMAN

Postman est une plateforme de collaboration pour le développement d'API. Il offre un ensemble d'outils permettant de simplifier le processus de création, de test et de documentation d'API. Postman est largement utilisé dans le développement d'applications web et mobiles pour faciliter le travail avec des API RESTful.

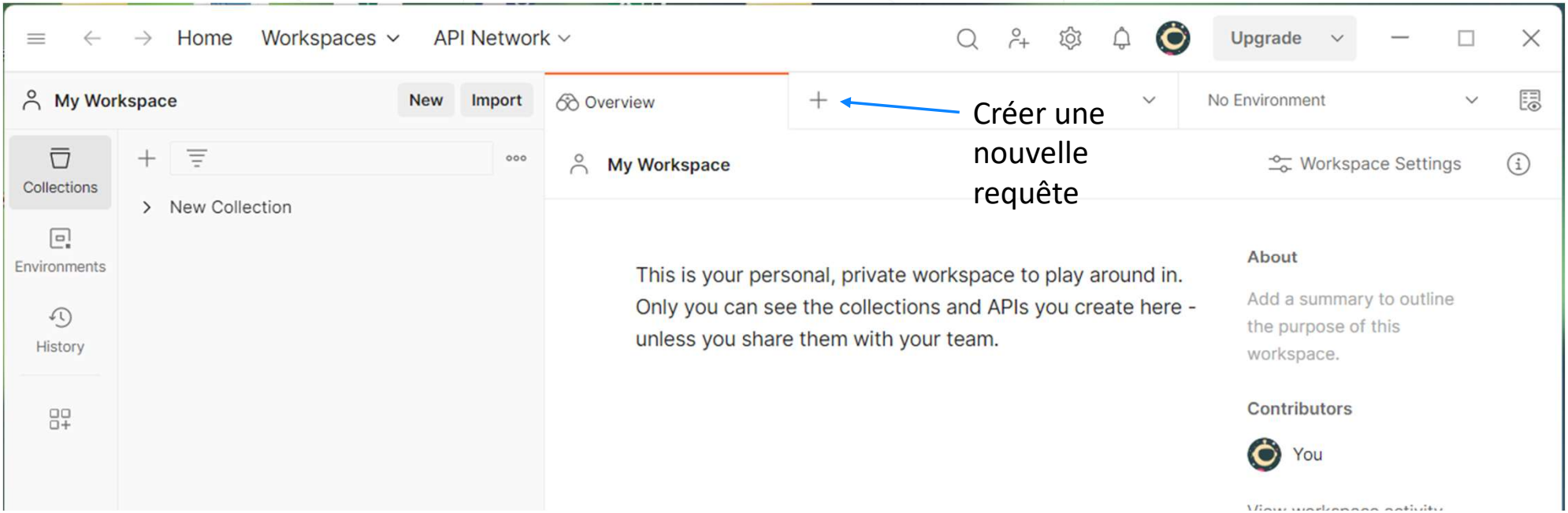
Postman permet de créer et d'envoyer facilement des requêtes HTTP (GET, POST, PUT, DELETE, etc.) à des endpoints spécifiques (URL) .

Postman est particulièrement utile lors du développement initial et du test d'API. Il permet de s'assurer que les endpoints fonctionnent correctement et de valider les réponses reçues.

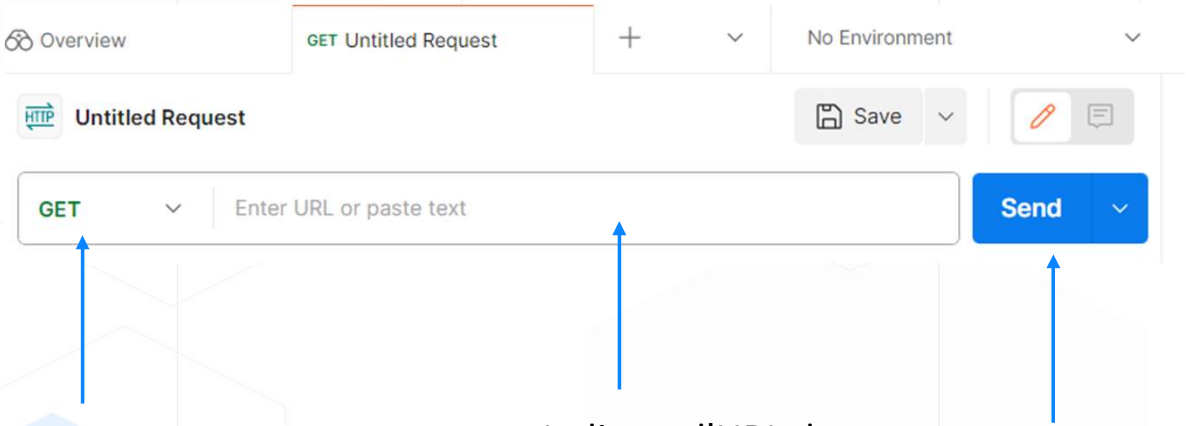
POSTMAN

Installation de Postman :

Postman est une application de bureau disponible pour Windows, macOS et Linux.
Vous pouvez le télécharger depuis le site officiel de Postman : https://www.postman.com/



POSTMAN



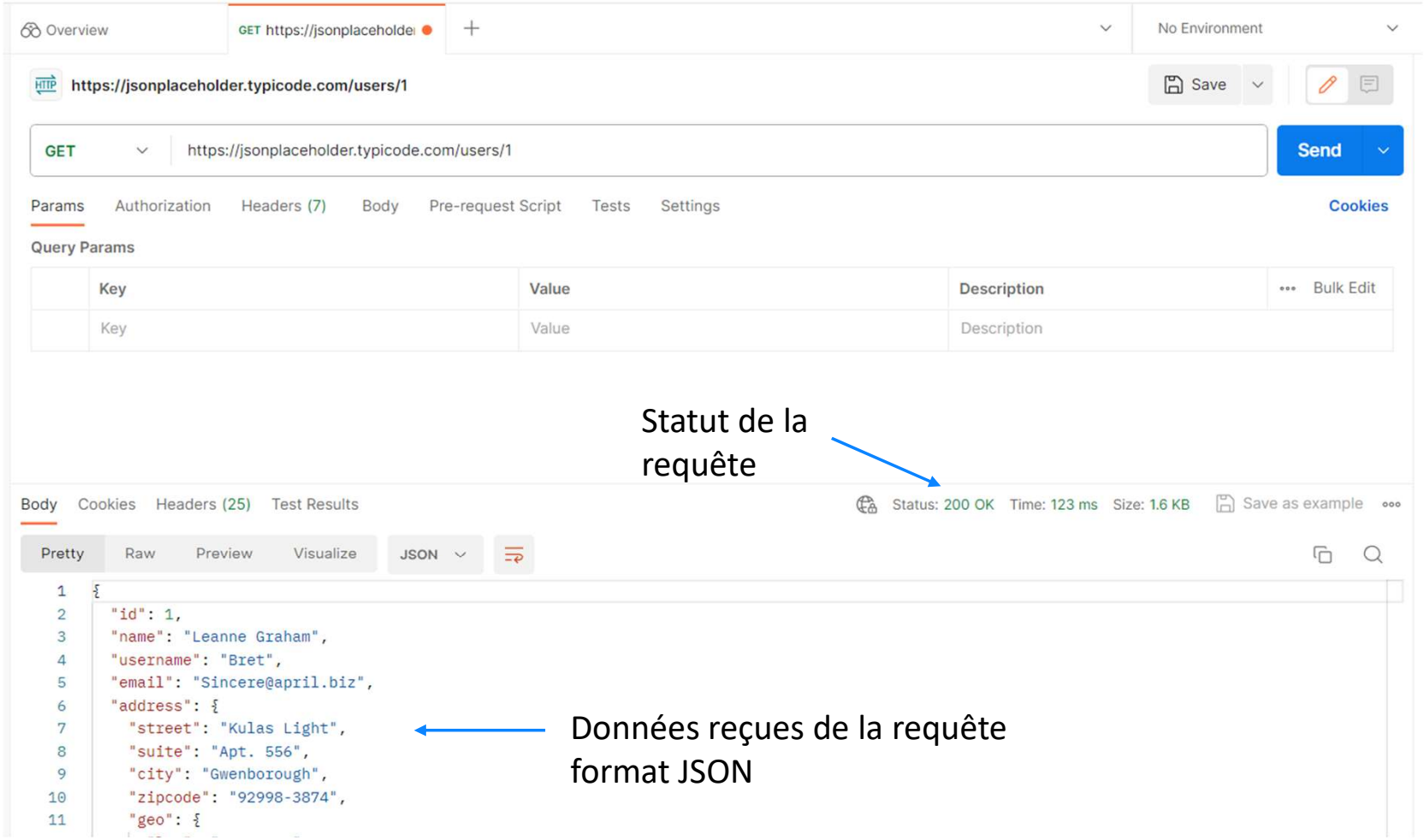
Définir la méthode :
GET, POST, PUT, DELETE…

Indiquer l'URL de ta requête

Envoyer la requête

POSTMAN

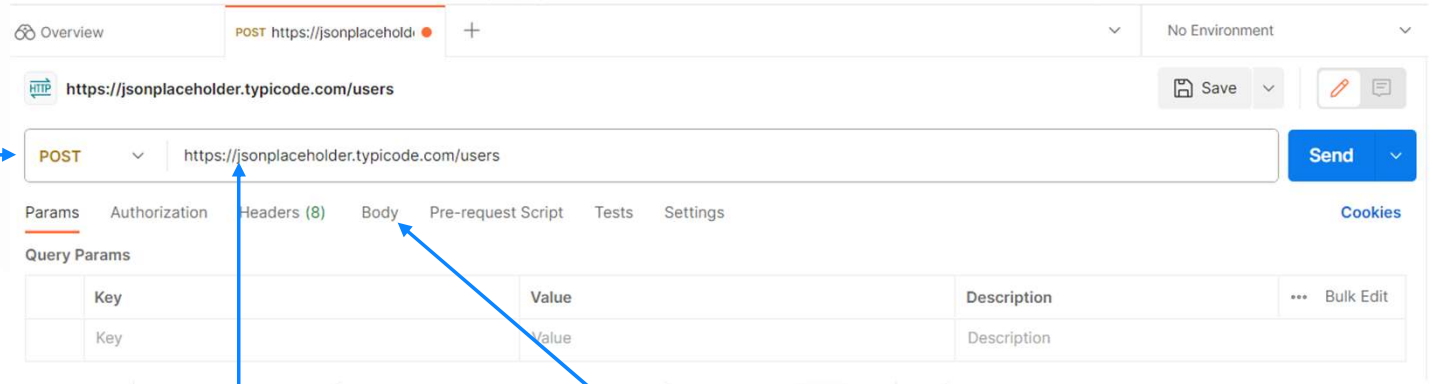
Méthode GET



POSTMAN

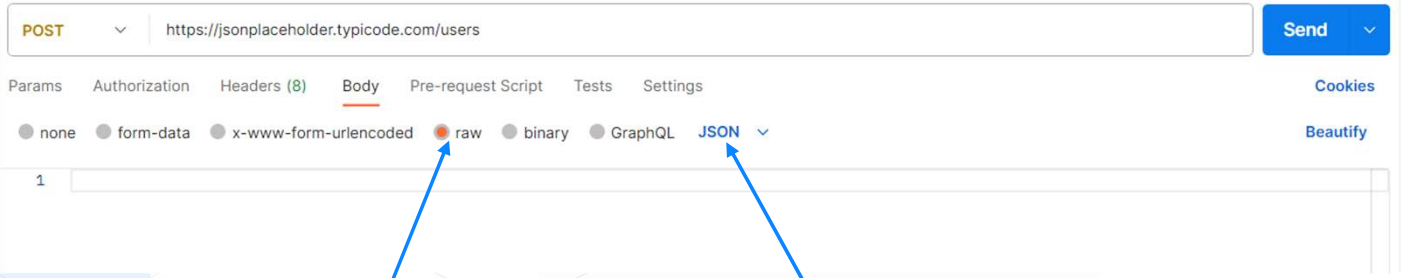
Méthode POST

1. Définir la méthode POST



2. Indiquer l'URL de ta requête

3. choisir l'onglet body



4. Choisir raw

5. Choisir JSON

POSTMAN

Méthode POST

7. Valider la requête

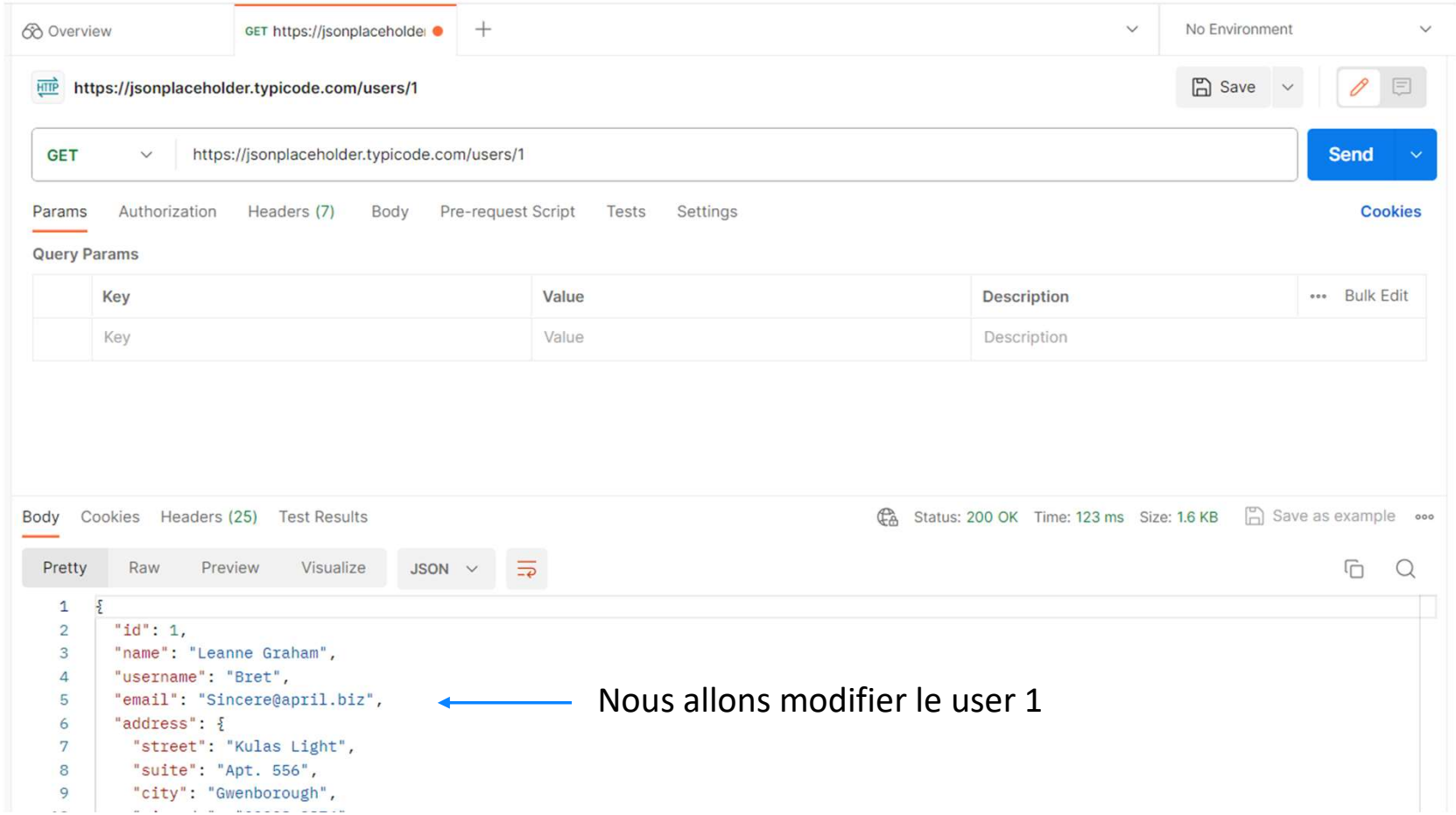
6. Ecrire la donnée sous forme pour de JSON

8. L'API vous envoie des données

9. le Statut confirme l'envoie

POSTMAN

Méthode PUT



Nous allons modifier le user 1

POSTMAN

Méthode PUT

1. Mettre la méthode PUT

2. Définir l'URL avec l'utilisateur à modifier (user 1)

3. Définir les données à modifier

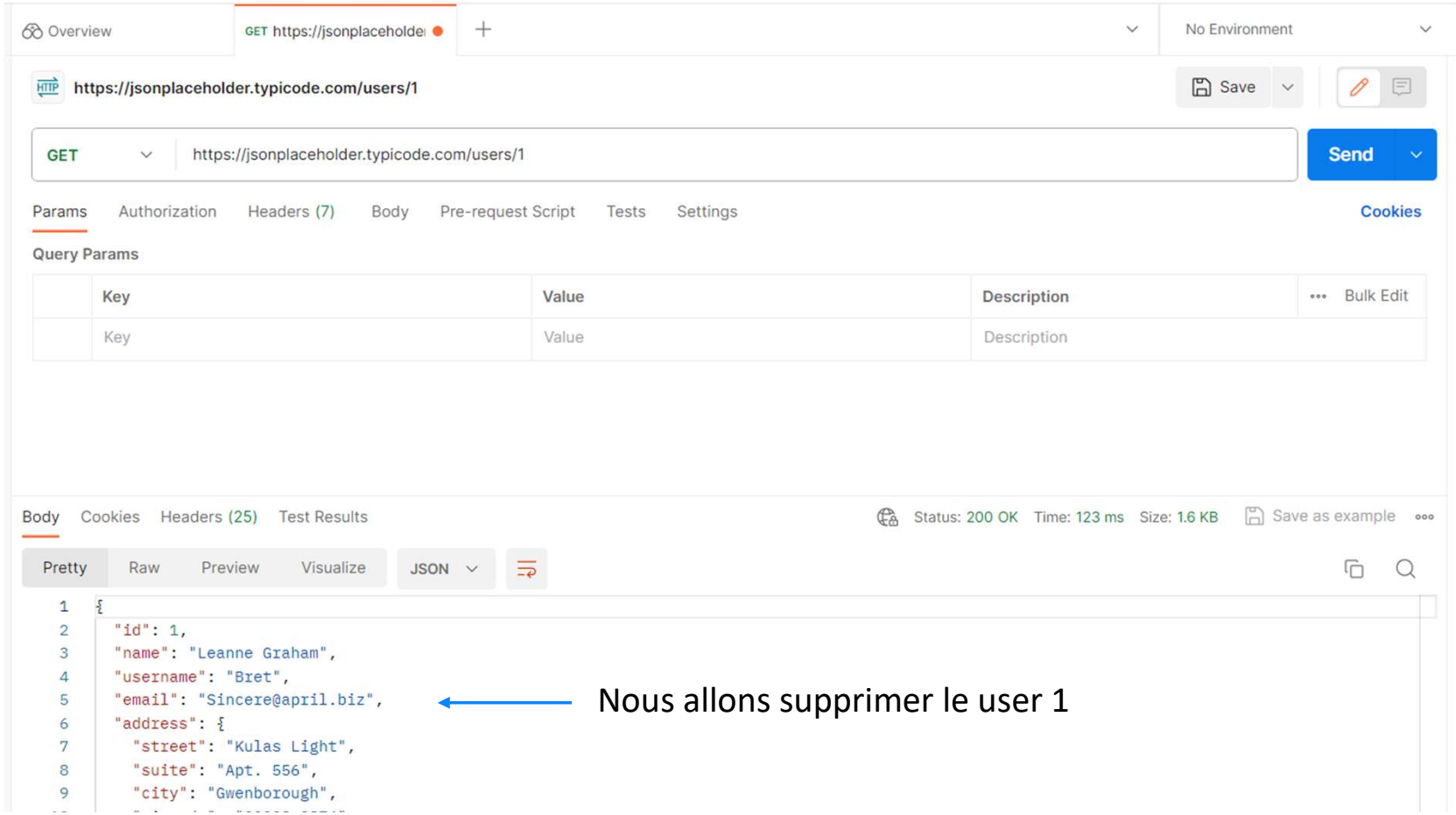
4. Envoyer la requête

5. Résultat des modifications

6. Statut de la requête

POSTMAN

Méthode DELETE



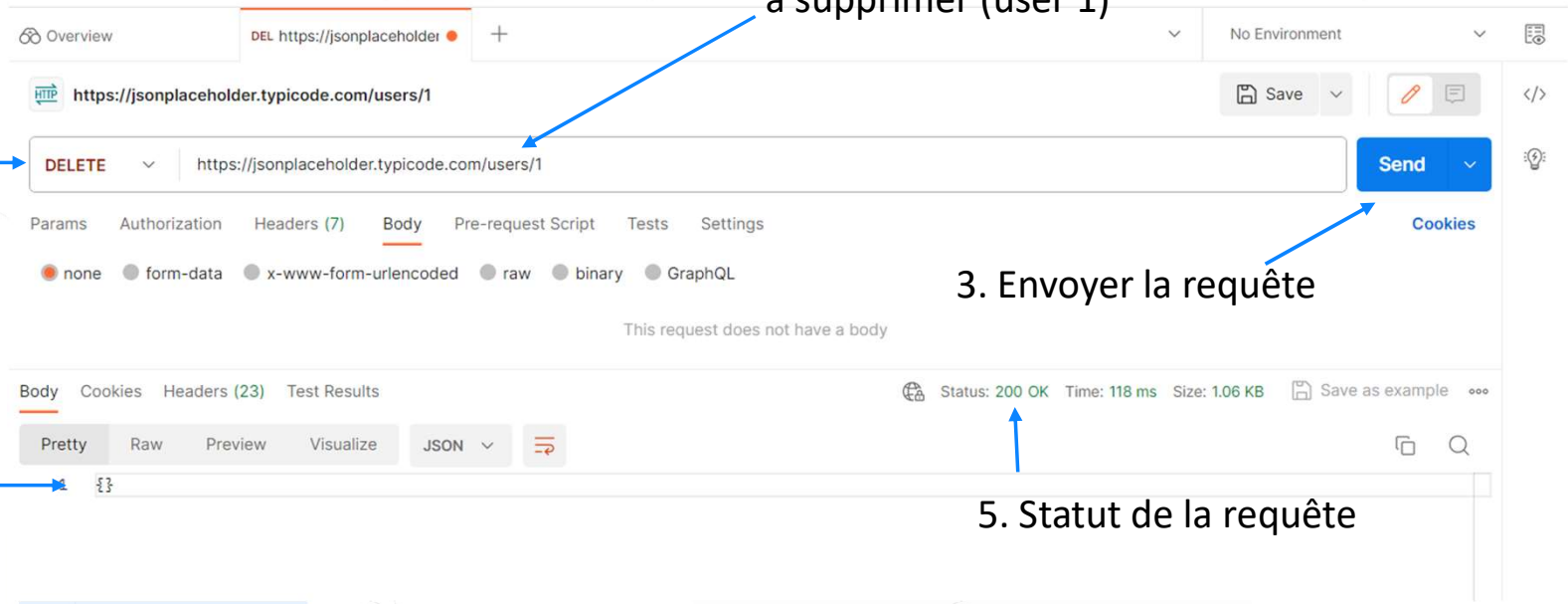
Nous allons supprimer le user 1

POSTMAN

Méthode DELETE

2. Définir l'URL avec l'utilisateur à supprimer (user 1)

1. Mettre la méthode DELETE



3. Envoyer la requête

4. Résultat à la suppression

5. Statut de la requête

AJAX

AJAX (Asynchronous JavaScript and XML) est une technique de développement web qui permet d'effectuer des requêtes HTTP asynchrones depuis le navigateur, sans recharger la page entière. Cette approche permet de créer des applications web plus réactives et dynamiques en mettant à jour des parties spécifiques de la page sans nécessiter un rechargement complet.

Principaux points :

Asynchronisme : AJAX permet d'effectuer des opérations de manière asynchrone, c'est-à-dire que le navigateur peut continuer à exécuter d'autres tâches pendant l'attente d'une réponse du serveur.

Réactivité : Les requêtes AJAX permettent de mettre à jour dynamiquement le contenu d'une page sans nécessiter un rechargement complet, offrant une expérience utilisateur plus fluide.

XMLHttpRequest (XHR) : C'est l'objet JavaScript traditionnellement utilisé pour effectuer des requêtes AJAX. Il permet d'envoyer des requêtes HTTP et de recevoir des réponses du serveur.

Fetch API : La Fetch API est une API plus moderne et flexible introduite dans les navigateurs récents, offrant une interface plus puissante et basée sur des promesses pour effectuer des requêtes HTTP.

XMLHTTPREQUEST (XHR)

L'objet XMLHttpRequest est une interface JavaScript souvent utilisée pour effectuer des requêtes HTTP asynchrones vers un serveur et mettre à jour dynamiquement le contenu d'une page web sans recharger l'ensemble de la page. Bien qu'il ait été largement utilisé dans le passé, il a été remplacé en grande partie par la Fetch API, qui offre une syntaxe plus moderne et basée sur les promesses. Cependant, il est toujours utile de comprendre le fonctionnement de base de XMLHttpRequest.

Création de l'objet XMLHttpRequest :

```
// Création d'un nouvel objet XMLHttpRequest
let xhr = new XMLHttpRequest();
```

Configuration de la requête :

```
// Configuration de la requête :
//   - 'GET' : Type de requête, dans ce cas, une requête pour récupérer des données.
//   - 'https://api.example.com/data' : URL de la ressource que nous souhaitons récupérer.
//   - true : Indique que la requête est asynchrone (true) ou synchrone (false).
xhr.open('GET', 'https://api.example.com/data', true);
```

XMLHTTPREQUEST (XHR)

Gestion des évenements:

```
// Gérer les événements liés à la requête
xhr.onload = function() {
  // L'événement onload est déclenché lorsque la requête est terminée avec succès
  if (xhr.status >= 200 && xhr.status < 300) {
    // La requête a réussi (statut de réponse entre 200 et 299 inclus)
    // Afficher la réponse du serveur dans la console
    // (pour la suite récupérer les données reçus)
    console.log(xhr.responseText);
  } else {
    // La requête a échoué
    // Afficher l'erreur de requête dans la console
    console.error('Erreur de requête : ' + xhr.status);
  }
};
```

XMLHTTPREQUEST (XHR)

Gestion des erreurs:

```
xhr.onerror = function() {
  // L'événement onerror est déclenché en cas d'échec de la requête
  console.error('Erreur réseau'); // Afficher l'erreur réseau dans la console
};
```

Envoie de la requête:

```
// Envoyer la requête
xhr.send();
```

XMLHTTPREQUEST (XHR)

Exemple de récupération de données pour l'affichage dans la page:

Code html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Exemple XHR</title>
</head>
<body>
    <div id="resultat"></div>

    <script src="script.js"></script>
</body>
</html>
```

Code js

```javascript
// Créer une instance de XMLHttpRequest
let xhr = new XMLHttpRequest();

// Configurer la requête
xhr.open('GET', 'https://api.example.com/data', true);

// Gérer les événements liés à la requête
xhr.onload = function() {
  if (xhr.status >= 200 && xhr.status < 300) {
    // La requête a réussi
    // Convertir la réponse JSON en objet JavaScript
    let resultat = JSON.parse(xhr.responseText);

    // Afficher les données dans la page HTML
    const resultatDiv = document.getElementById('resultat');
    resultatDiv.innerHTML = '<h2>Données récupérées :</h2>' +
                            '<p>Nom : ' + resultat.nom + '</p>' +
                            '<p>Email : ' + resultat.email + '</p>';
  } else {
    // La requête a échoué
    console.error('Erreur de requête : ' + xhr.status);
  }
};

xhr.onerror = function() {
  console.error('Erreur réseau');
};

// Envoyer la requête
xhr.send();
```

LES PROMESSES

Le concept de promesse est un modèle de programmation asynchrone qui facilite la gestion des opérations qui prennent du temps, comme les requêtes réseau, la lecture de fichiers, etc. Les promesses fournissent une syntaxe plus propre et plus lisible pour gérer les opérations asynchrones par rapport aux callbacks.

Une promesse peut être dans l'un des trois états suivants :

- Pendante (Pending) : L'état initial d'une promesse. La promesse est en cours d'exécution et n'a pas encore été résolue ou rejetée.
- Résolue (Fulfilled) : La promesse a été exécutée avec succès. La valeur résultante est disponible.
- Rejetée (Rejected) : La promesse a échoué pour une raison quelconque. Une raison d'échec est généralement associée à la promesse, indiquant la raison de l'échec.

LES PROMESSES

La syntaxe de base d'une promesse ressemble à ceci :

```
let maPromesse = new Promise(function(resolve, reject) {
  // Code asynchrone ici

  if (/* opération réussie */) {
    resolve(resultat); // La promesse est résolue avec le résultat
  } else {
    reject(erreur); // La promesse est rejetée avec une raison d'échec
  }
});
```

LES PROMESSES

Une fois qu'une promesse est créée, vous pouvez attacher des gestionnaires à son état à l'aide des méthodes .then(), .catch() et .finally()

.then() : Utilisé pour traiter le résultat réussi de la Promesse. Peut être enchaîné pour effectuer des opérations successives.

.catch() : Utilisé pour gérer les erreurs qui se produisent pendant l'exécution de la Promesse.

.finally() : Exécute du code indépendamment du résultat de la Promesse, que ce soit un succès ou un échec. Utile pour le nettoyage ou l'exécution de code après la résolution de la Promesse.

```
// Création d'une nouvelle promesse
var maPromesse = new Promise(function(resolve, reject) {
  // Code asynchrone ici (par exemple, une requête réseau, une lecture de fichier, etc.)

  if (/* opération réussie */) {
    // Si l'opération réussit, on appelle la fonction resolve
    resolve(resultat); // La promesse est résolue avec le résultat
  } else {
    // Si l'opération échoue, on appelle la fonction reject
    reject(erreur); // La promesse est rejetée avec une raison d'échec
  }
});

// Utilisation de la promesse
maPromesse
  .then(function(resultat) {
    // La promesse est résolue avec succès, le résultat est disponible ici
    console.log(resultat);
  })
  .catch(function(erreur) {
    // La promesse a été rejetée, la raison d'échec est disponible ici
    console.error(erreur);
  });
```

FETCH

fetch est une fonction JavaScript qui facilite les requêtes réseau asynchrones.
Elle renvoie une Promesse, ce qui simplifie la gestion des opérations asynchrones en utilisant les méthodes .then(), .catch(), et .finally().

```
  // Effectue une requête GET à l'URL spécifiée
fetch('https://api.example.com/data')
  .then(response => {
    // Vérifie si la requête a réussi en vérifiant le statut de la réponse
    if (!response.ok) {
      // Si le statut n'est pas dans la plage 200-299 (inclus),
      // lance une erreur avec un message personnalisé
      throw new Error('Erreur de réseau');
    }
    // Si la requête a réussi, convertit la réponse JSON en objet JavaScript
    return response.json();
  })
  .then(data => {
    // Utilise les données récupérées après la conversion JSON
    console.log(data);
  })
  .catch(error => {
    // Gère les erreurs survenues pendant le processus
    console.error('Erreur :', error);
  });
```

FETCH

Avantages de fetch par rapport à XMLHttpRequest

1. Syntaxe plus propre et moderne : La syntaxe de fetch utilise des Promesses, ce qui rend le code plus lisible et évite les problèmes liés aux callbacks en cascade (callback hell) de XMLHttpRequest.

2. Intégration native des Promesses : fetch intègre nativement les Promesses, ce qui simplifie la gestion des opérations asynchrones et facilite la lecture du code.

3. Options de requête plus flexibles : Les options de fetch (telles que la méthode, les en-têtes, le corps de la requête) peuvent être spécifiées de manière plus claire et directe, contrairement à XMLHttpRequest qui nécessite plusieurs appels de configuration.

4. Compatibilité avec les nouveaux standards : fetch est conforme aux nouveaux standards de développement web et est mieux intégré avec d'autres technologies modernes, telles que les Service Workers.

5. Gestion des erreurs améliorée : fetch rejette la Promesse pour les erreurs réseau et les erreurs de parsing de la réponse, ce qui simplifie la gestion des erreurs par rapport à XMLHttpRequest.

6. Pas de dépendance aux callbacks : fetch évite l'utilisation intensive des callbacks, rendant le code plus modulaire et facile à comprendre.

FETCH

Exemple de code

Prenons un exemple : récupérer des données à partir d'une API et les afficher dans une page HTML. L'API retourne des informations sur des utilisateurs au format JSON.

Code HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Exemple Fetch</title>
</head>
<body>
    <div id="utilisateurs-liste"></div>

    <script src="script.js"></script>
</body>
</html>
```

FETCH

Exemple de code

```
// Fonction pour effectuer la requête fetch et traiter les données
function chargerUtilisateurs() {
  // Utilisation de fetch pour récupérer des données depuis une API
  fetch('https://jsonplaceholder.typicode.com/users')
    .then(response => {
      // Vérifier si la requête a réussi (statut de réponse entre 200 et 299 inclus)
      if (!response.ok) {
        throw new Error('Erreur de réseau');
      }
      // Convertir la réponse JSON en objet JavaScript
      return response.json();
    })
    .then(utilisateurs => {
      // Utiliser les données récupérées (liste des utilisateurs)
      afficherUtilisateurs(utilisateurs);
    })
    .catch(error => {
      // Gérer les erreurs
      console.error('Erreur :', error);
    });
}
```

```
// Fonction pour afficher les utilisateurs dans la page HTML
function afficherUtilisateurs(utilisateurs) {
  // Sélectionner l'élément HTML où afficher les utilisateurs
  var utilisateursListe = document.getElementById('utilisateurs-liste');

  // Construire le contenu HTML avec les utilisateurs
  var html = '<h2>Liste des Utilisateurs</h2><ul>';
  utilisateurs.forEach(function(utilisateur) {
    html += '<li><strong>' + utilisateur.name + '</strong> - ' + utilisateur.email + '</li>';
  });
  html += '</ul>';

  // Injecter le contenu HTML dans la page
  utilisateursListe.innerHTML = html;
}

// Appeler la fonction pour charger et afficher les utilisateurs
chargerUtilisateurs();
```

Ici on récupère les utilisateurs et on les affiche sur la page, cet exemple utilise la méthode GET (appliquée par défaut si ce n'est pas renseigné), voyons ensuite d'autres verbes

FETCH

Exemple de code pour la méthode POST

Dans cet exemple, nous allons simuler l'envoi de données utilisateur à un serveur qui les enregistrera. Nous utiliserons l'API JSONPlaceholder pour simuler ce scénario.

Fichier html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Exemple Fetch POST</title>
</head>
<body>
    <form id="utilisateur-form">
        <label for="nom">Nom:</label>
        <input type="text" id="nom" name="nom" required>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>
        <button type="submit">Ajouter Utilisateur</button>
    </form>

    <div id="resultat"></div>

    <script src="script.js"></script>
</body>
</html>
```

On sélectionne les éléments nécessaires pour pouvoir récupérer les saisies de l'utilisation mais aussi l'écouter d'évènement. (fichier js)

```javascript
// Sélectionner le formulaire et l'élément où afficher les résultats
let formulaire = document.getElementById('utilisateur-form');
let resultatDiv = document.getElementById('resultat');

// Écouter l'événement de soumission du formulaire
formulaire.addEventListener('submit', function(event) {
  event.preventDefault(); // Empêcher le comportement par défaut du formulaire

  // Récupérer les valeurs du formulaire
  let nom = document.getElementById('nom').value;
  let email = document.getElementById('email').value;

  // Appeler la fonction pour effectuer la requête POST
  ajouterUtilisateur(nom, email);
});
```

FETCH

Exemple de code pour la méthode POST

Envoi des données à l'API

```
// Fonction pour effectuer la requête POST et traiter la réponse
function ajouterUtilisateur(nom, email) {
  // Configuration de la requête POST avec les données du formulaire
  fetch('https://jsonplaceholder.typicode.com/users', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      name: nom,
      email: email,
    }),
  })
    .then(response => {
      // Vérifier si la requête a réussi (statut de réponse entre 200 et 299 inclus)
      if (!response.ok) {
        throw new Error('Erreur de réseau');
      }
      // Convertir la réponse JSON en objet JavaScript
      return response.json();
    })
    .then(utilisateur => {
      // Utiliser les données de l'utilisateur ajouté
      afficherResultat('Utilisateur ajouté avec ID: ' + utilisateur.id);
    })
    .catch(error => {
      // Gérer les erreurs
      afficherResultat('Erreur : ' + error.message, true);
    });
}
```

Affichage du résultat

```
// Fonction pour afficher les résultats dans la page HTML
function afficherResultat(message, estErreur) {
  // Définir la classe CSS en fonction de si c'est une erreur ou non
  let classeCSS = estErreur ? 'erreur' : 'succes';

  // Construire le contenu HTML avec le message
  let html = '<p class="' + classeCSS + '">' + message + '</p>';

  // Injecter le contenu HTML dans la page
  resultatDiv.innerHTML = html;
}
```

FETCH

Exemple de code pour la méthode PUT

Fichier HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Exemple Fetch PUT</title>
</head>
<body>
    <form id="utilisateur-form">
        <label for="utilisateur-id">ID de l'utilisateur à mettre à jour:</label>
        <input type="number" id="utilisateur-id" name="utilisateur-id" required>
        <label for="nom">Nouveau Nom:</label>
        <input type="text" id="nom" name="nom" required>
        <label for="email">Nouvel Email:</label>
        <input type="email" id="email" name="email" required>
        <button type="submit">Mettre à Jour Utilisateur</button>
    </form>

    <div id="resultat"></div>

    <script src="script.js"></script>
</body>
</html>
```

On sélectionne les éléments nécessaires pour pouvoir récupérer les saisies de l'utilisation mais aussi l'écouter d'évènement. (fichier js)

```javascript
// Sélectionner le formulaire et l'élément où afficher les résultats
const formulaire = document.getElementById('utilisateur-form');
const resultatDiv = document.getElementById('resultat');

// Écouter l'événement de soumission du formulaire
formulaire.addEventListener('submit', function(event) {
  event.preventDefault(); // Empêcher le comportement par défaut du formulaire

  // Récupérer les valeurs du formulaire
  let userId = document.getElementById('utilisateur-id').value;
  let nom = document.getElementById('nom').value;
  let email = document.getElementById('email').value;

  // Appeler la fonction pour effectuer la requête PUT
  mettreAJourUtilisateur(userId, nom, email);
});
```

FETCH

Exemple de code pour la méthode PUT

Envoi des données à l'API

```
// Fonction pour effectuer la requête PUT et traiter la réponse
function mettreAJourUtilisateur(userId, nom, email) {
  // Construire l'URL avec l'ID de l'utilisateur à mettre à jour
  var url = 'https://jsonplaceholder.typicode.com/users/' + userId;

  // Configuration de la requête PUT avec les nouvelles données
  fetch(url, {
    method: 'PUT',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      name: nom,
      email: email,
    }),
  })
    .then(response => {
      // Vérifier si la requête a réussi (statut de réponse entre 200 et 299 inclus)
      if (!response.ok) {
        throw new Error('Erreur de réseau');
      }
      // Convertir la réponse JSON en objet JavaScript
      return response.json();
    })
    .then(utilisateur => {
      // Utiliser les données de l'utilisateur mis à jour
      afficherResultat('Utilisateur mis à jour avec ID: ' + utilisateur.id);
    })
    .catch(error => {
      // Gérer les erreurs
      afficherResultat('Erreur : ' + error.message, true);
    });
}
```

Affichage du résultat

```
// Fonction pour afficher les résultats dans la page HTML
function afficherResultat(message, estErreur) {
  // Définir la classe CSS en fonction de si c'est une erreur ou non
  var classeCSS = estErreur ? 'erreur' : 'succes';

  // Construire le contenu HTML avec le message
  var html = '<p class="' + classeCSS + '">' + message + '</p>';

  // Injecter le contenu HTML dans la page
  resultatDiv.innerHTML = html;
}
```

FETCH

Exemple de code pour la méthode DELETE

Fichier HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Exemple Fetch DELETE</title>
</head>
<body>
    <form id="utilisateur-form">
        <label for="utilisateur-id">ID de l'utilisateur à supprimer:</label>
        <input type="number" id="utilisateur-id" name="utilisateur-id" required>
        <button type="submit">Supprimer Utilisateur</button>
    </form>

    <div id="resultat"></div>

    <script src="script.js"></script>
</body>
</html>
```

On sélectionne les éléments nécessaires pour pouvoir récupérer les saisies de l'utilisation mais aussi l'écouter d'évènement. (fichier js)

```javascript
// Sélectionner le formulaire et l'élément où afficher les résultats
const formulaire = document.getElementById('utilisateur-form');
const resultatDiv = document.getElementById('resultat');

// Écouter l'événement de soumission du formulaire
formulaire.addEventListener('submit', function(event) {
  event.preventDefault(); // Empêcher le comportement par défaut du formulaire

  // Récupérer l'ID de l'utilisateur à supprimer
  let userId = document.getElementById('utilisateur-id').value;

  // Appeler la fonction pour effectuer la requête DELETE
  supprimerUtilisateur(userId);
});
```

FETCH

Exemple de code pour la méthode DELETE

Envoi des données à l'API

```
// Fonction pour effectuer la requête DELETE et traiter la réponse
function supprimerUtilisateur(userId) {
  // Construire l'URL avec l'ID de l'utilisateur à supprimer
  var url = 'https://jsonplaceholder.typicode.com/users/' + userId;

  // Configuration de la requête DELETE
  fetch(url, {
    method: 'DELETE',
  })
    .then(response => {
      // Vérifier si la requête a réussi (statut de réponse entre 200 et 299 inclus)
      if (!response.ok) {
        throw new Error('Erreur de réseau');
      }
      // Si la requête réussit, afficher un message de succès
      afficherResultat('Utilisateur avec ID ' + userId + ' supprimé avec succès');
    })
    .catch(error => {
      // Gérer les erreurs
      afficherResultat('Erreur : ' + error.message, true);
    });
}
```

Affichage du résultat

```
// Fonction pour afficher les résultats dans la page HTML
function afficherResultat(message, estErreur) {
  // Définir la classe CSS en fonction de si c'est une erreur ou non
  var classeCSS = estErreur ? 'erreur' : 'succes';

  // Construire le contenu HTML avec le message
  var html = '<p class="' + classeCSS + '">' + message + '</p>';

  // Injecter le contenu HTML dans la page
  resultatDiv.innerHTML = html;
}
```

ASYNC AWAIT

async et await sont des fonctionnalités introduites dans ECMAScript 2017 (ES8) pour simplifier la gestion des promesses en JavaScript. Ces mots-clés facilitent l'écriture du code asynchrone de manière plus lisible et plus similaire à la programmation synchrone.

Le mot-clé async est utilisé pour déclarer une fonction asynchrone. Une fonction asynchrone retourne toujours une promesse, même si aucune promesse explicite n'est retournée dans le code.

L'opérateur await est utilisé à l'intérieur d'une fonction asynchrone pour attendre la résolution d'une promesse. Il ne peut être utilisé que dans les fonctions déclarées avec async. L'utilisation de await rend le code plus lisible en évitant l'enchaînement complexe de .then().

ASYNC AWAIT

Utilisation d'Async/Await avec Fetch :

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');

    if (!response.ok) {
      throw new Error('Erreur de réseau');
    }

    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Erreur :', error);
  }
}

// Appeler la fonction asynchrone
fetchData();
```

Digital Academy by insy2s

ASYNC AWAIT

Exemple d'utilisation d'Async/Await avec le Fetch avec la méthode GET (vu plus haut dans les slides)

```
async function fetchUsers() {
  try {
    // Effectuer une requête GET à l'URL des utilisateurs sur JSONPlaceholder
    const response = await fetch('https://jsonplaceholder.typicode.com/users');

    // Vérifier si la requête a réussi (statut de réponse entre 200 et 299 inclus)
    if (!response.ok) {
      throw new Error('Erreur de réseau');
    }

    // Convertir la réponse JSON en objet JavaScript
    const users = await response.json();

    // Utiliser les données récupérées
    console.log(users);
  } catch (error) {
    // Gérer les erreurs
    console.error('Erreur :', error);
  }
}

// Appeler la fonction asynchrone
fetchUsers();
```