COMPTE RENDU TP1 DEVELOPPEMENT WEB

KITSOUKOU & DJEUNGA

3D_TD2_TP4

PROJET 1 - REST API

Guide pour Créer une Simple API REST avec Node.js

Une API REST (Representational State Transfer) est une manière standardisée de permettre la communication entre différents systèmes à travers le protocole HTTP. Dans ce guide, nous allons apprendre à créer une API REST basique à l'aide de Node.js et du framework Express. Nous partirons de zéro et ajouterons des routes pour effectuer des opérations CRUD (Create, Read, Update, Delete).

Prérequis

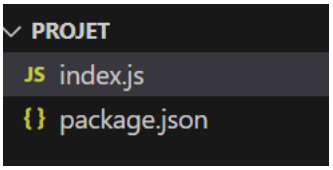
- Node.js et npm installés sur votre machine.
- Un éditeur de texte comme Visual Studio Code.

Étape 1 : Créer le Projet

-Ouvrons le terminal et accédons à un répertoire où nous avons crée le projet.

-Initialisons un nouveau projet Node.js : Cela créera un fichier package.json avec des réglages par défaut.

```
PS C:\Users\debor\OneDrive\Desktop\ENSEA\ENSEA_24-25\Developpement web\Projet> npm init -y
Wrote to C:\Users\debor\OneDrive\Desktop\ENSEA\ENSEA_24-25\Developpement web\Projet\package.json:

{
  "name": "projet",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```



Quelques tests

Créons un nouveau fichier appelé index.js et ajoutons cette ligne de code

```
JS index.js
1   console.log("test from node");
```

Pour exécuter notre code nous avons tapé sur votre terminal node index.js

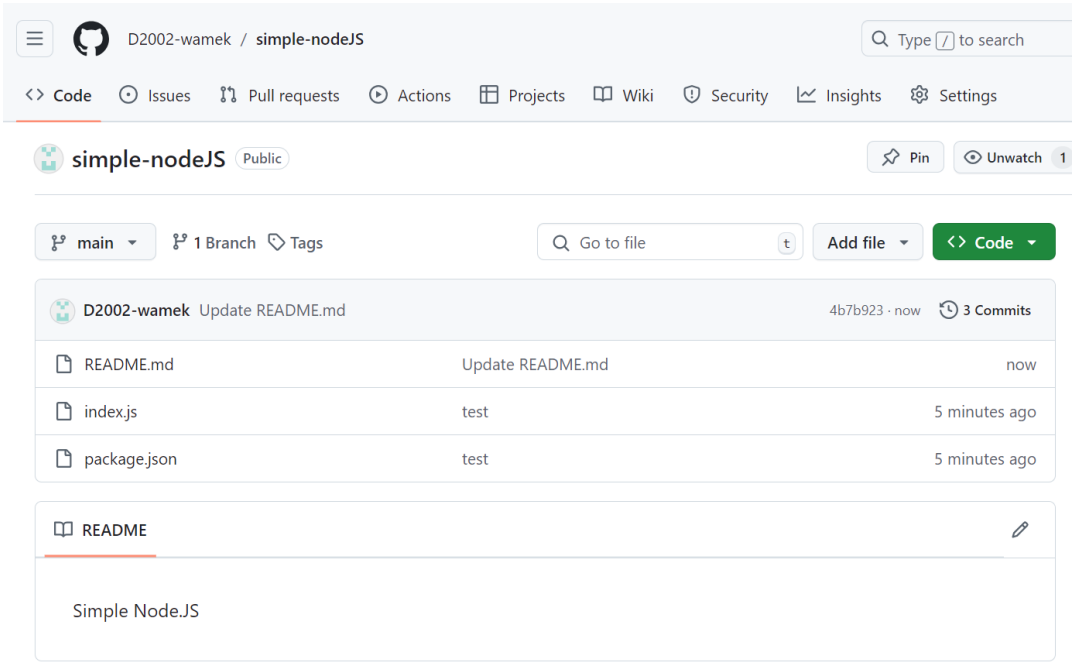
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\debor\OneDrive\Desktop\ENSEA\ENSEA_24-25\Developpement web\Projet> node index.js
test from node
PS C:\Users\debor\OneDrive\Desktop\ENSEA\ENSEA_24-25\Developpement web\Projet>
```

Pour le plaisir

```
2   console.log("test from node", Math.random(), new Date())
```

```
PS C:\Users\debor\OneDrive\Desktop\ENSEA\ENSEA_24-25\Developpement web\Projet> node index.js
test from node 0.9091618139971902 2024-10-24T11:59:30.911Z
PS C:\Users\debor\OneDrive\Desktop\ENSEA\ENSEA_24-25\Developpement web\Projet>
```

GITHUB



Étape 2 : Installer les Dépendances

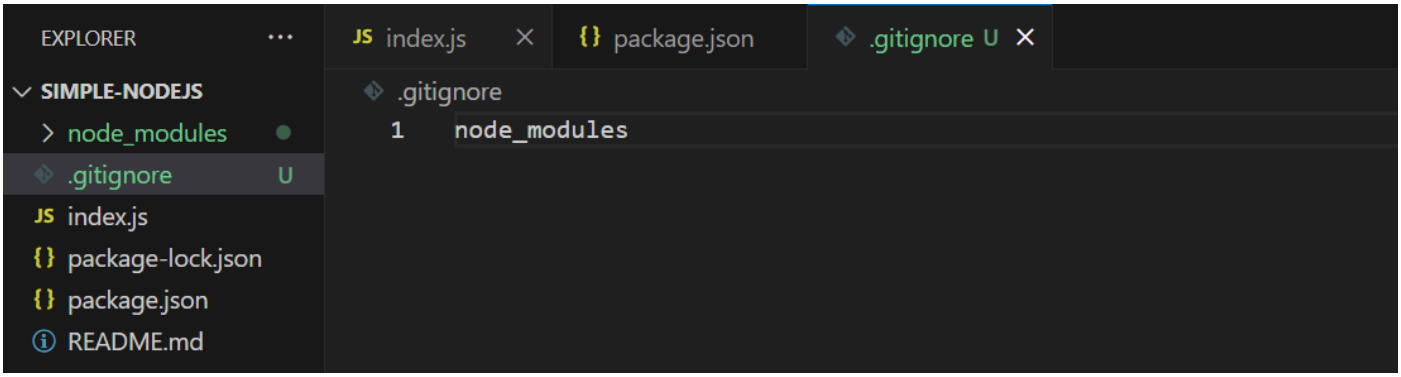
1-Installons Express (le framework web pour Node.js) et Nodemon (pour recharger automatiquement le serveur à chaque modification) :

Partie enlevée :

```
/* "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },*/
```

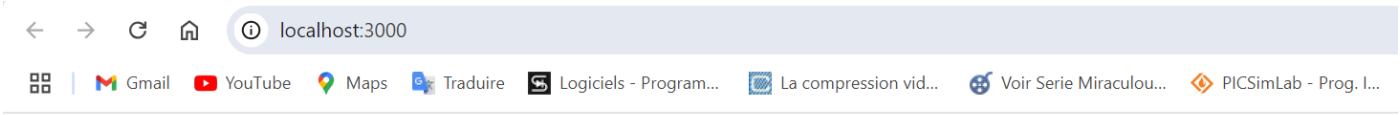
```
"keywords": [],
"author": "",
"license": "ISC",
"description": "",
"dependencies": {
  "express": "^4.21.1"
},
"devDependencies": {
  "nodemon": "^3.1.7"
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

added 29 packages, and audited 95 packages in 2s

17 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

C:\Users\debor\OneDrive\Desktop\ENSEA\ENSEA_24-25\Developpement web\Projet\simple-nodeJS>npm start

> projet@1.0.0 start
> nodemon index.js

[nodemon] 3.1.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
test from node 0.5479687760406668 2024-10-24T13:09:27.767Z
[nodemon] clean exit - waiting for changes before restart
```



Étape 3 : Créer le Serveur de Base

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

> nodemon index.js

[nodemon] 3.1.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
test from node 0.5479687760406668 2024-10-24T13:09:27.767Z
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Serveur en cours d'exécution sur http://localhost:3000
```



Bienvenue sur notre API REST simple!

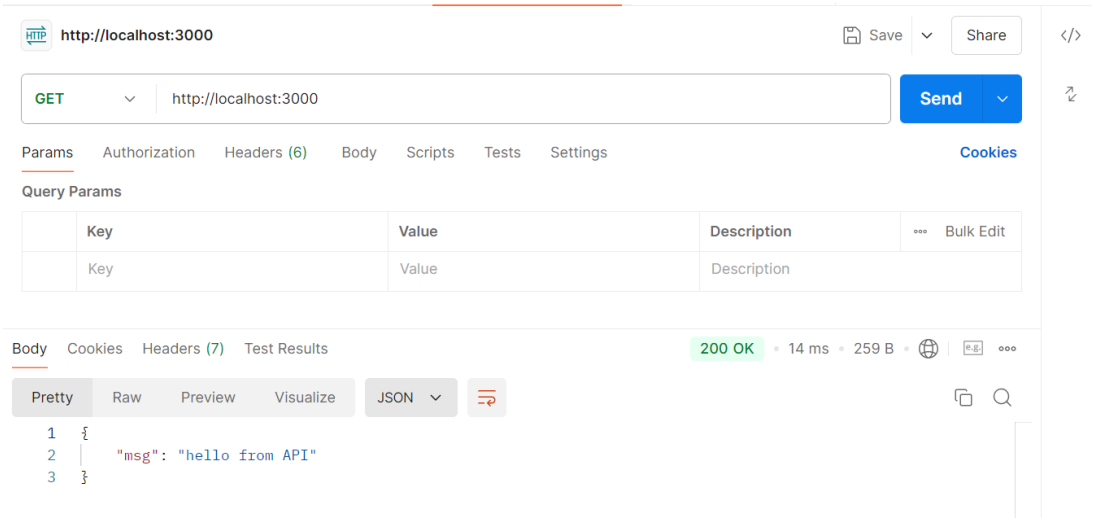
index.js M | package.json | .gitignore

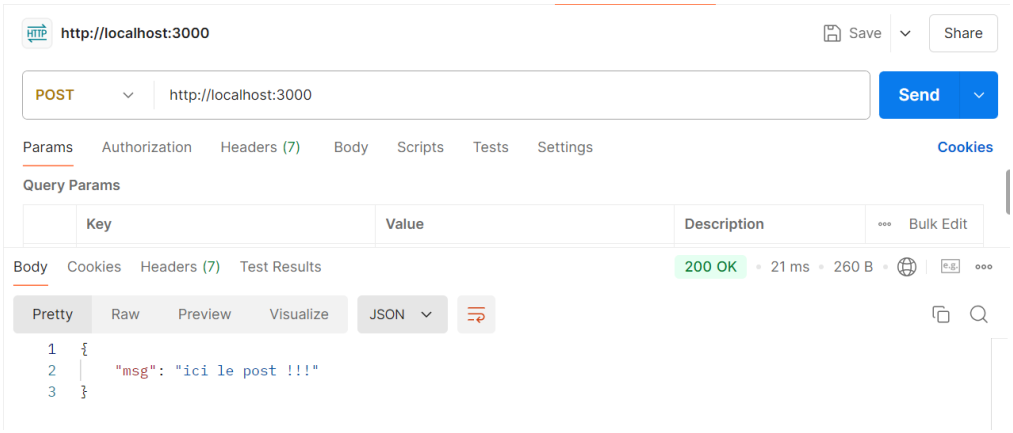
index.js > ...

```js
const express = require('express');
const app = express();
const port = 3000;
app.use(express.json());

app.get("/", (req, res) => {
    res.json({
        msg: "hello from API"
    })
})

app.listen(port, () => {
    console.log(`Serveur en cours d'exécution sur http://localhost:${port}`);
});
```

Impression élégante ☑


```
{
  "msg": "hello from API"
}
```





<u>Methods pour CRUD</u> (Create, Read, Update, Delete)

GET : Récupérer les éléments

POST : Ajouter un élément. Il a besoin d'un corps (body)

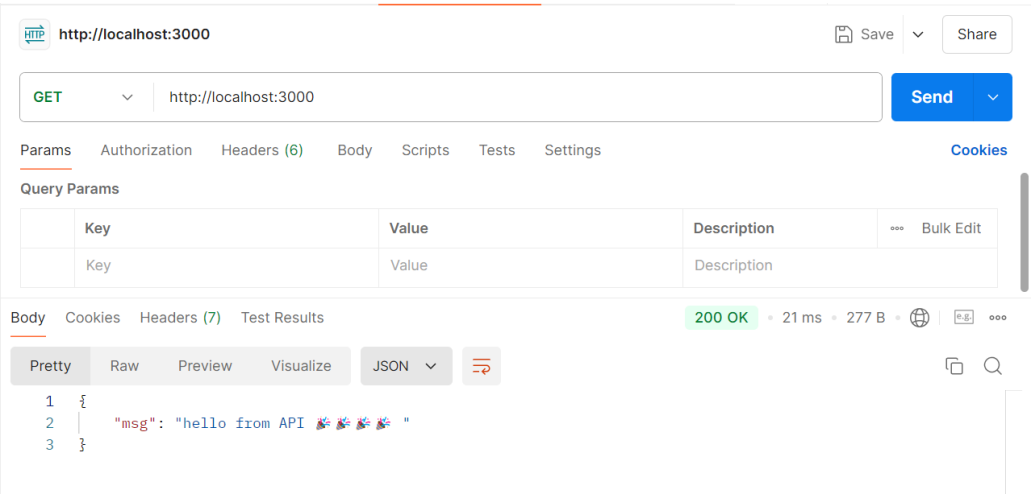
PUT : Mettre à jour un élément. Il a besoin d'un corps (body)

DELETE : Supprimer un élément. Basé sur les paramètres

Nous devons gérer 4 méthodes qui peuvent arriver sur notre serveur : GET, POST, PUT, DELETE.

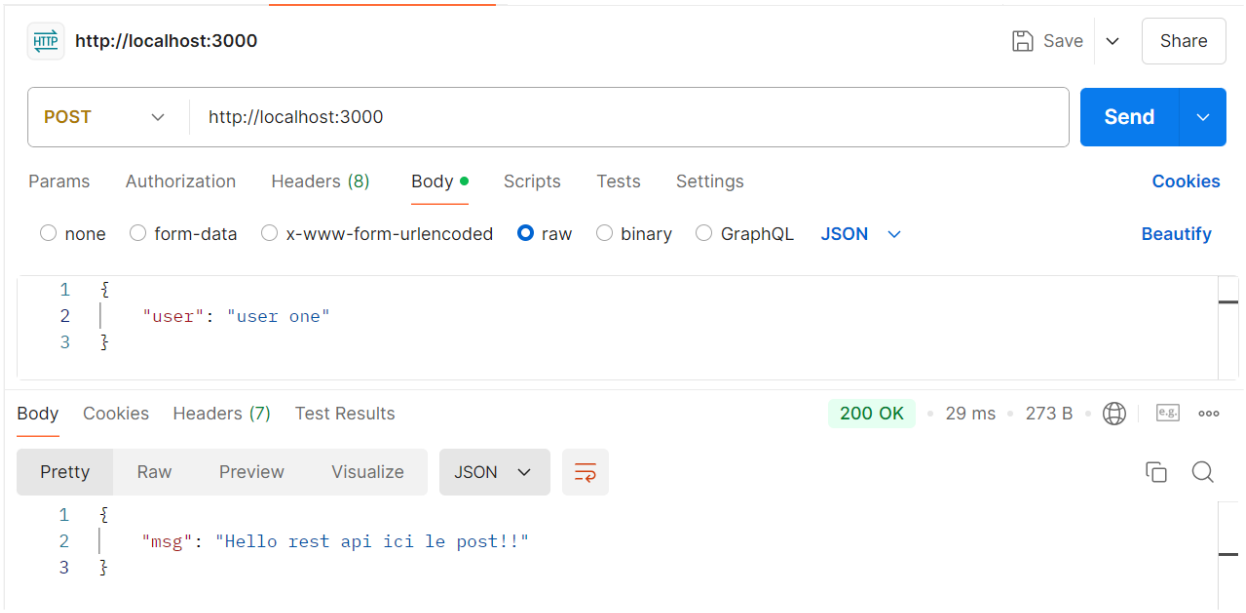
Pour cela, nous allons générer chaque méthode une à la fois.

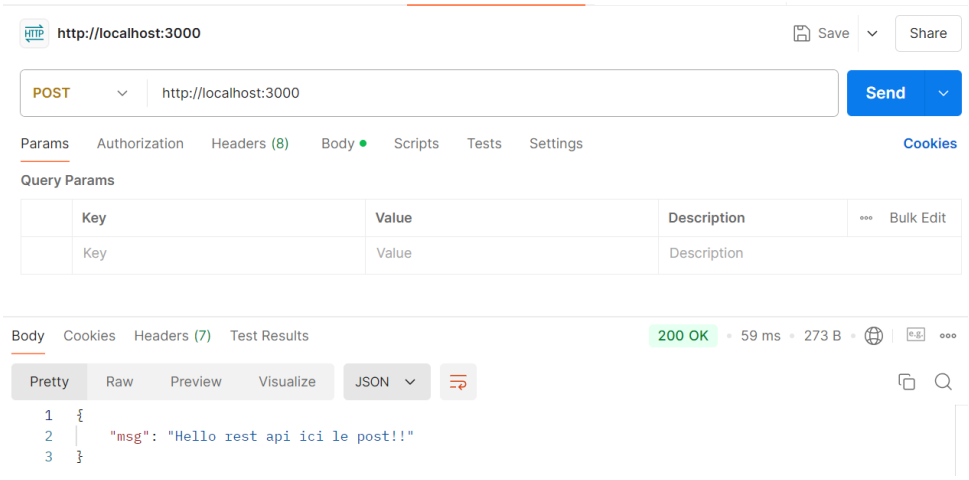
GET : ici, nous envoyons toutes les données requises

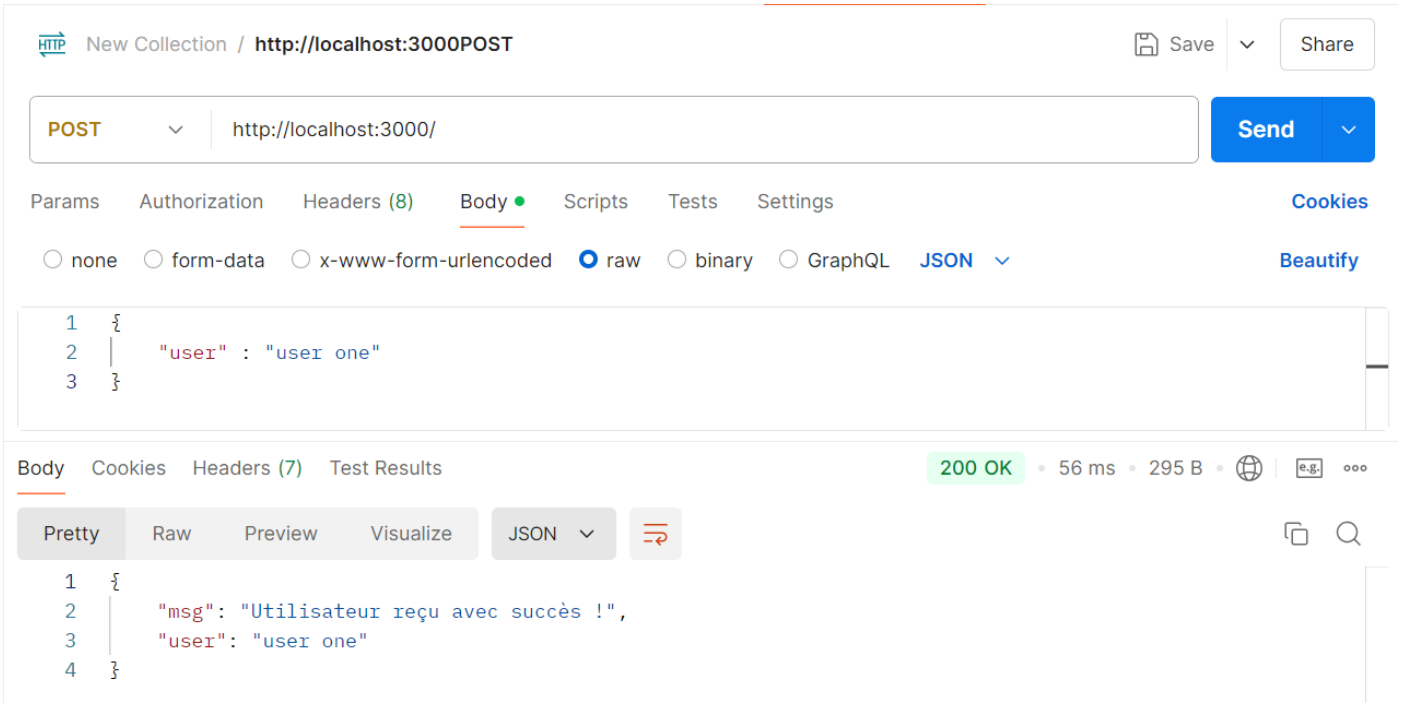


<u>POST</u> : ici, nous gérons les requêtes POST, et nous allons également avoir besoin des données dans le 'body'

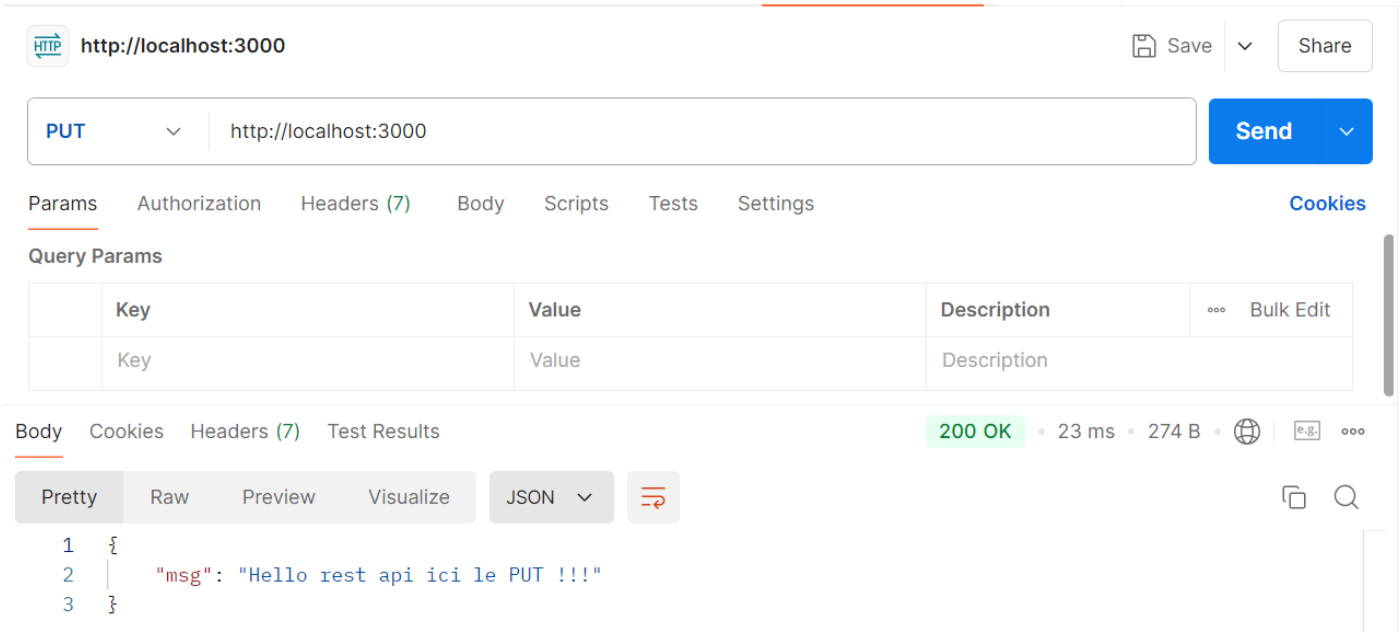
```
const express = require('express');
const app = express();
const port = 3000;

// Middleware pour permettre de traiter les données JSON
app.use(express.json());

// Route POST pour recevoir et traiter les données du body
app.post("/", (req, res) => {
    // Extraire l'information 'user' du body de la requête
    const { user } = req.body;

    // Vérification de base pour s'assurer que le champ 'user' est fourni
    if (!user) {
        return res.status(400).json({ message: "Le champ 'user' est requis." });
    }

    // Répondre avec un message de succès et les données reçues
    res.json({
        msg: "Utilisateur reçu avec succès !",
        user: user
    });
});

// Démarrer le serveur
app.listen(port, () => {
    console.log(`Serveur en cours d'exécution sur http://localhost:${port}`);
});
```


http://localhost:3000
Save
Share
POST
http://localhost:3000
Send
Params
Authorization
Headers (8)
Body
Scripts
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"user": "user one"
}
Body
Cookies
Headers (7)
Test Results
200 OK
29 ms
273 B
Pretty
Raw
Preview
Visualize
JSON
{
"msg": "Hello rest api ici le post!!"
}

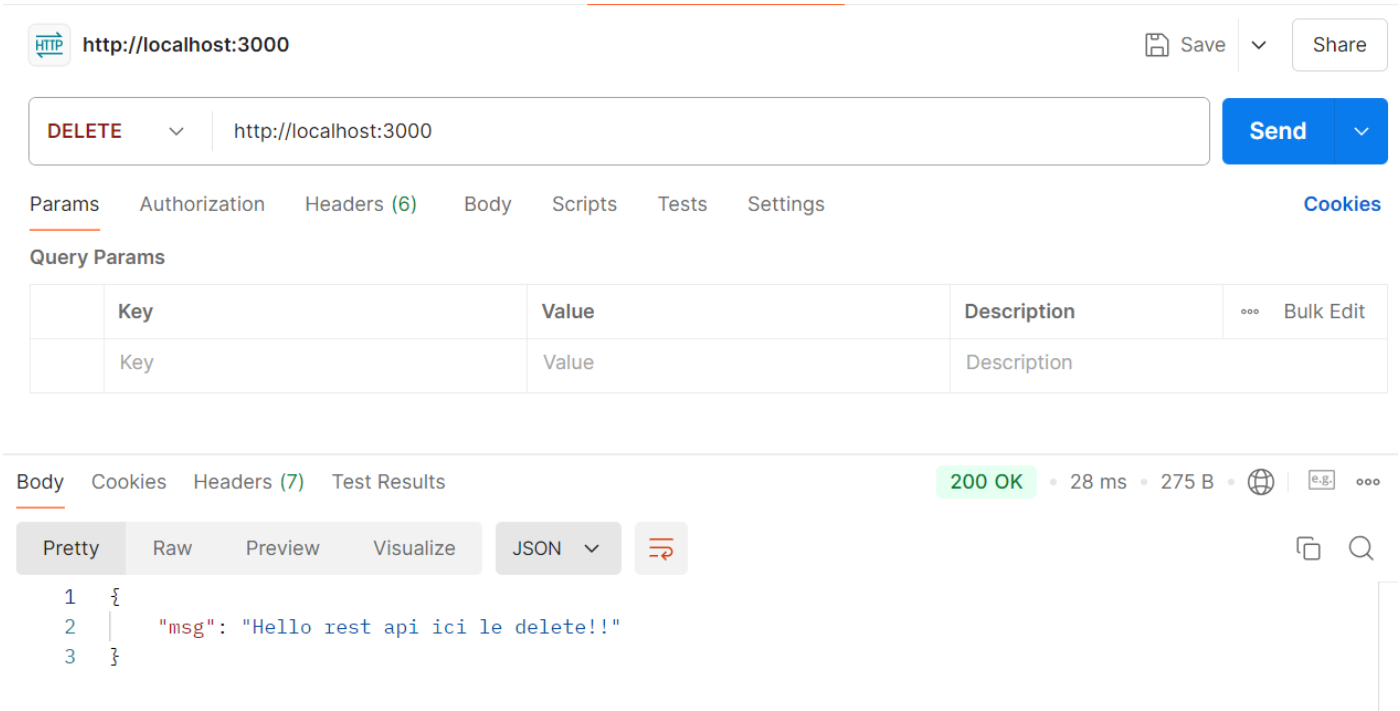

http://localhost:3000
Save
Share
POST
http://localhost:3000
Send
Params
Authorization
Headers (8)
Body
Scripts
Tests
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Key
Value
Description
Body
Cookies
Headers (7)
Test Results
200 OK
59 ms
273 B
Pretty
Raw
Preview
Visualize
JSON
{
"msg": "Hello rest api ici le post!!"
}


New Collection / http://localhost:3000POST
Save
Share
POST
http://localhost:3000/
Send
Params
Authorization
Headers (8)
Body
Scripts
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"user" : "user one"
}
Body
Cookies
Headers (7)
Test Results
200 OK
56 ms
295 B
Pretty
Raw
Preview
Visualize
JSON
{
"msg": "Utilisateur reçu avec succès !",
"user": "user one"
}

PUT : nous allons modifier un utilisateur en fonction de son ID passé en paramètre dans l'URL

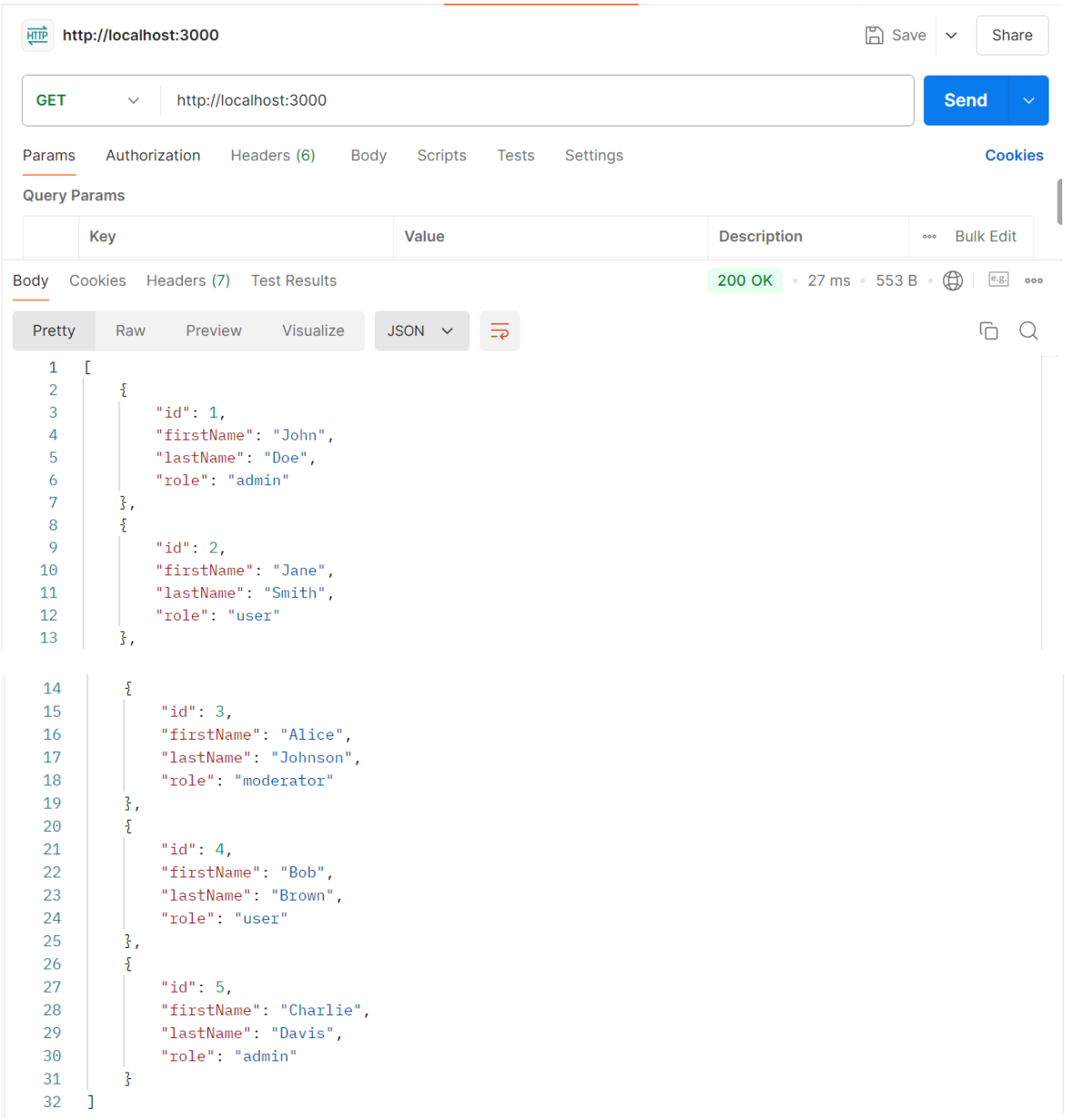


DELETE : nous supprimons un utilisateur en fonction de son ID



Ajout de données en MÉMOIRE (pas encore de base de données)

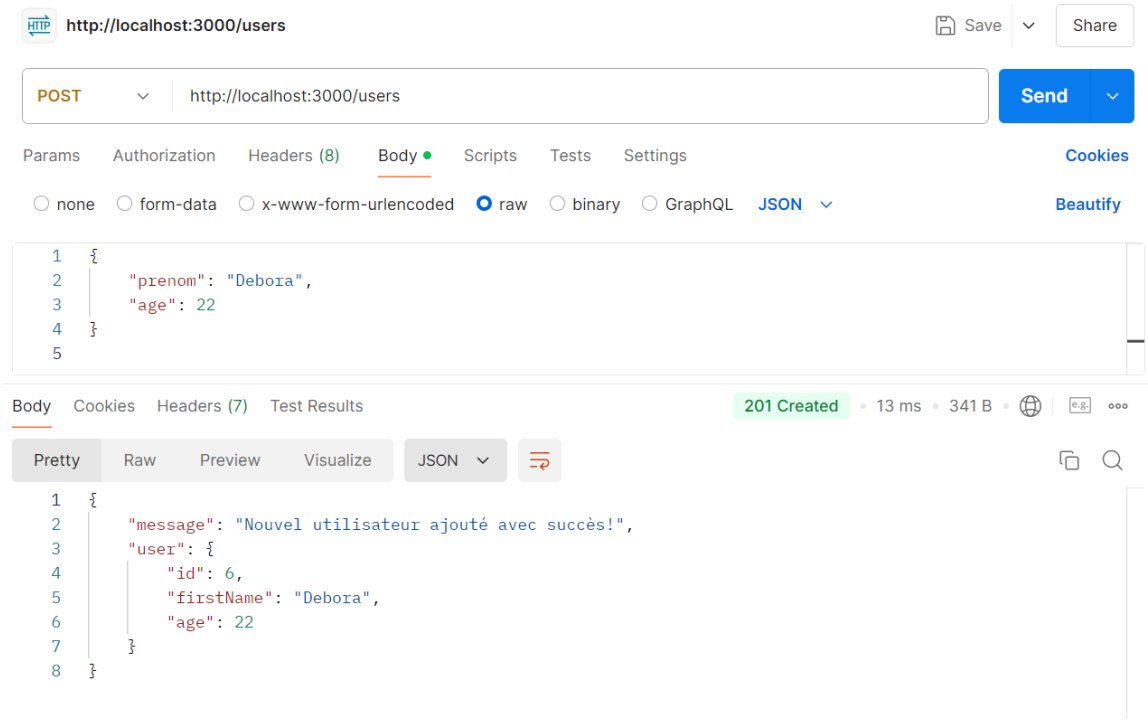
Créons une liste d'éléments (utilisateurs)

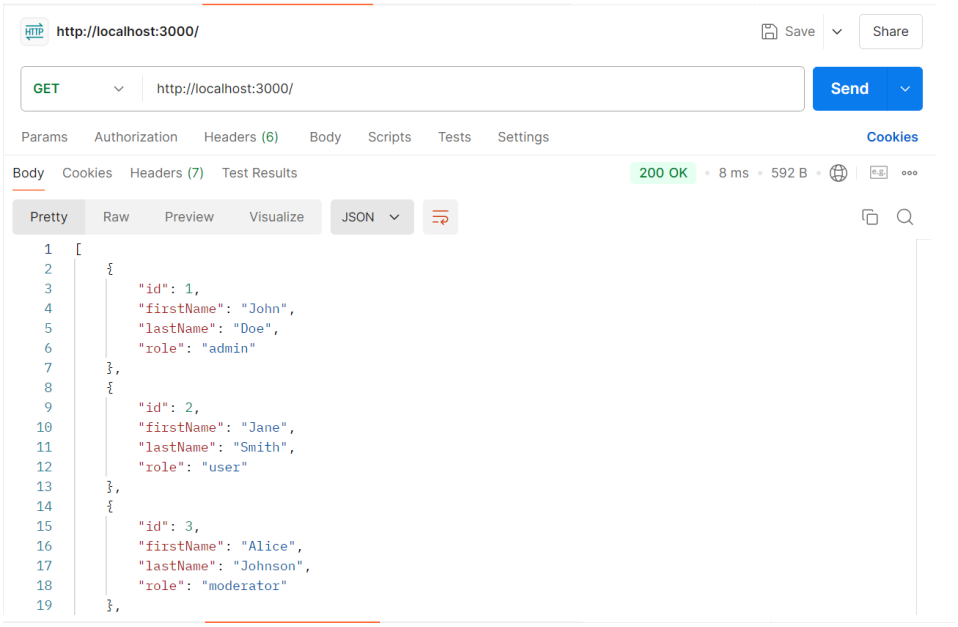


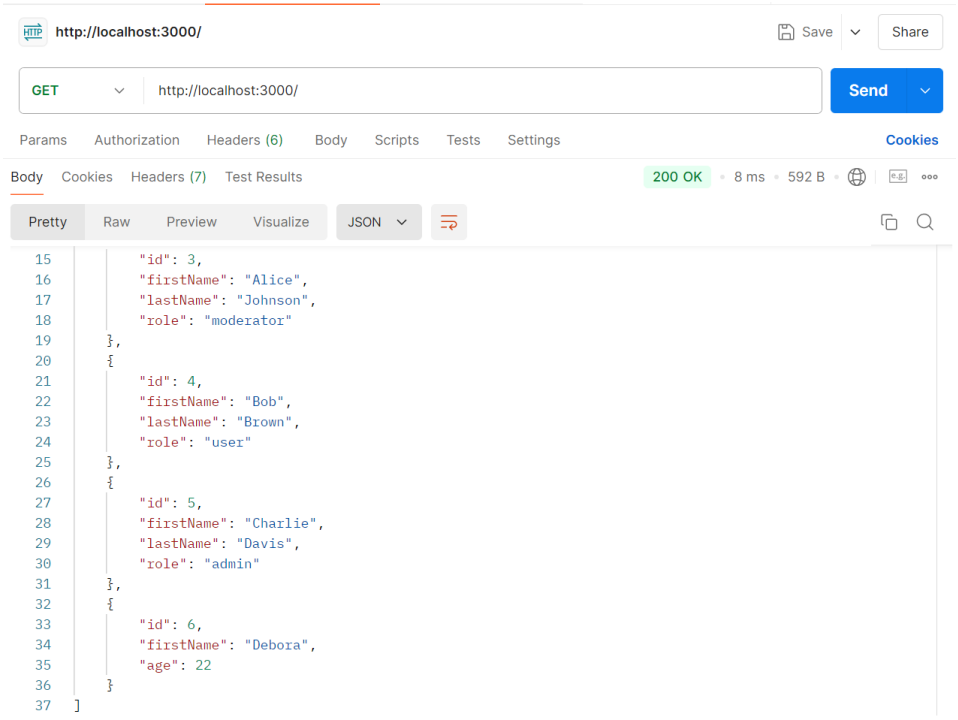
<u>WRITE</u> : ajout d'un nouvel utilisateur

Pour ajouter un nouvel utilisateur, nous allons utiliser la méthode POST et nous devons extraire les données du 'body' de la requête.

Sur POSTMAN on click sur “body”








```
const express = require('express');
const app = express();
const port = 3000;
app.use(express.json());

const users = [
    { id: 1, firstName: "John", lastName: "Doe", role: "admin" },
    { id: 2, firstName: "Jane", lastName: "Smith", role: "user" },
    { id: 3, firstName: "Alice", lastName: "Johnson", role: "moderator" },
    { id: 4, firstName: "Bob", lastName: "Brown", role: "user" },
    { id: 5, firstName: "Charlie", lastName: "Davis", role: "admin" },
];

// GET : Lire tous les utilisateurs
app.get("/", (req, res) => {
    res.json(users);
});
```

```
// POST : Ajouter un nouvel utilisateur
app.post("/users", (req, res) => {
    // Extraire les données du body
    const { prenom, age } = req.body;

    // Vérifier que les champs sont bien fournis
    if (!prenom || !age) {
        return res.status(400).json({ message: "Le champ 'prenom' et 'age' sont requis." });
    }

    // Créer un nouvel utilisateur avec un ID unique
    const newUser = {
        id: users.length + 1, // Utilisation de la longueur de la liste pour générer un ID simple
        firstName: prenom,     // Utilisation de 'prenom' pour le champ 'firstName'
        age                    // Ajout de l'âge
    };

    // Ajouter le nouvel utilisateur à la liste
    users.push(newUser);

    // Répondre avec un message de succès et les infos du nouvel utilisateur
    res.status(201).json({
        message: "Nouvel utilisateur ajouté avec succès!",
        user: newUser
    });
});

// Démarrer le serveur
app.listen(port, () => {
    console.log(`Serveur en cours d'exécution sur http://localhost:${port}`);
});
```

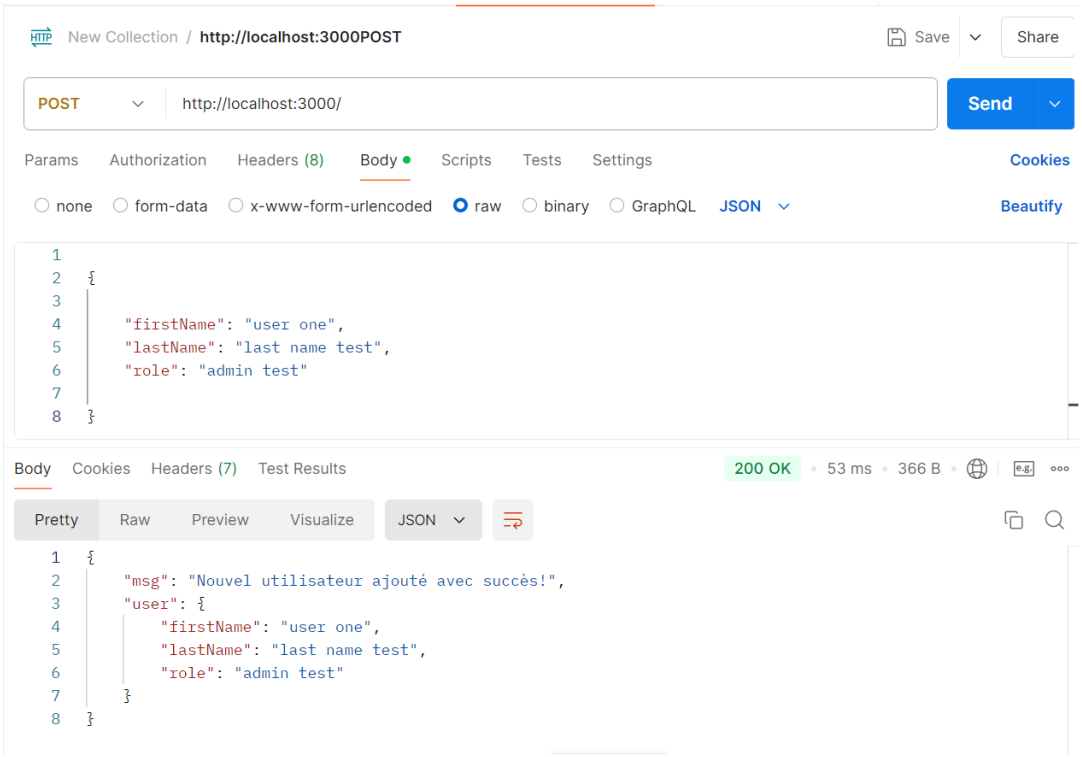
Récupérer les données du body

```
const express = require('express');
const app = express();
const port = 3000;

// Middleware pour permettre de traiter les données JSON
app.use(express.json());

app.post("/", (req, res) => {
    // Afficher le body de la requête dans le terminal
    console.log("Requête reçue:", req.body);

    res.json({
        msg: "ici le post !!!",
    });
});
```

```
// Démarrer le serveur
app.listen(port, () => {
    console.log(`Serveur en cours d'exécution sur http://localhost:${port}/`);
});
```

nous avons les données qui arrivent au serveur

nous envoyons plus de données nas le “body” sur postman

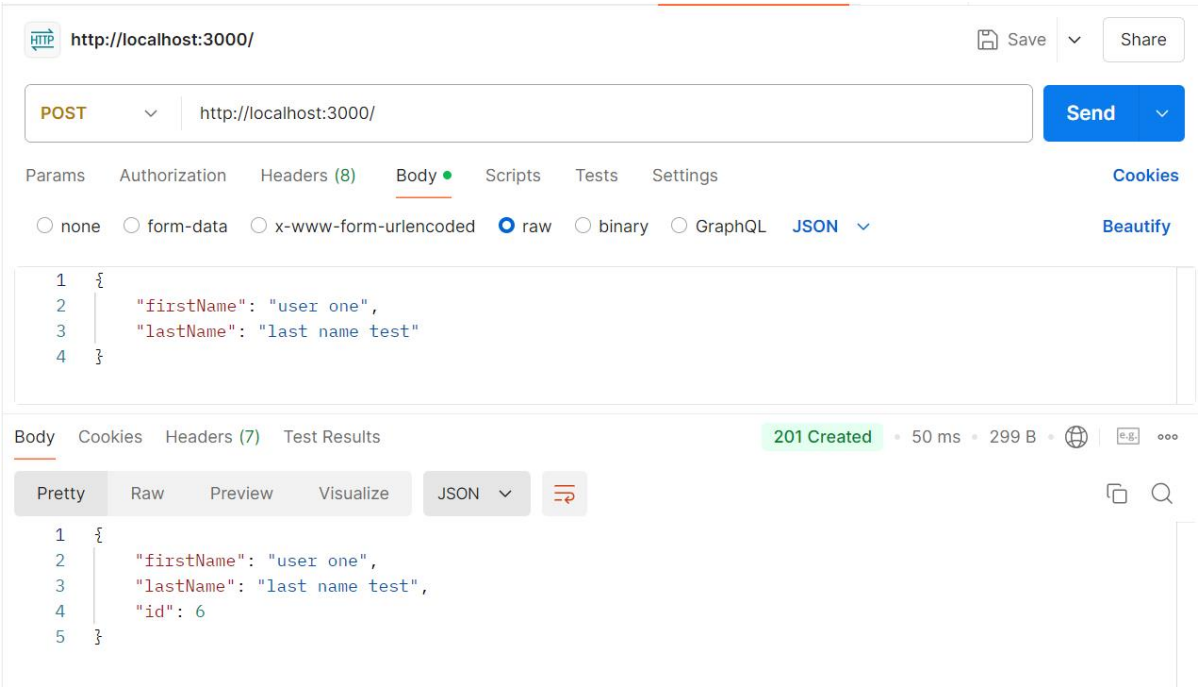



```
const express = require('express');
const app = express();
const port = 3000;

// Middleware pour traiter les données JSON envoyées dans le body
app.use(express.json());

app.post("/", (req, res) => {
    // Extraire les données du body
    const { firstName, lastName, role } = req.body;

    // Afficher les données reçues dans le terminal
    console.log("Requête reçue:", req.body);

    // Vérification de base pour s'assurer que tous les champs sont présents
    if (!firstName || !lastName || !role) {
        return res.status(400).json({ message: "Tous les champs (firstName, lastName, role) sont requis." });
    }

    // Répondre avec un message de confirmation et les données envoyées
    res.json({
        msg: "Nouvel utilisateur ajouté avec succès!",
        user: {
            firstName,
            lastName,
            role
        }
    });
});

// Démarrer le serveur
app.listen(port, () => {
    console.log(`Serveur en cours d'exécution sur http://localhost:${port}/`);
});
```

POST : Ajouter un Nouvel Utilisateur

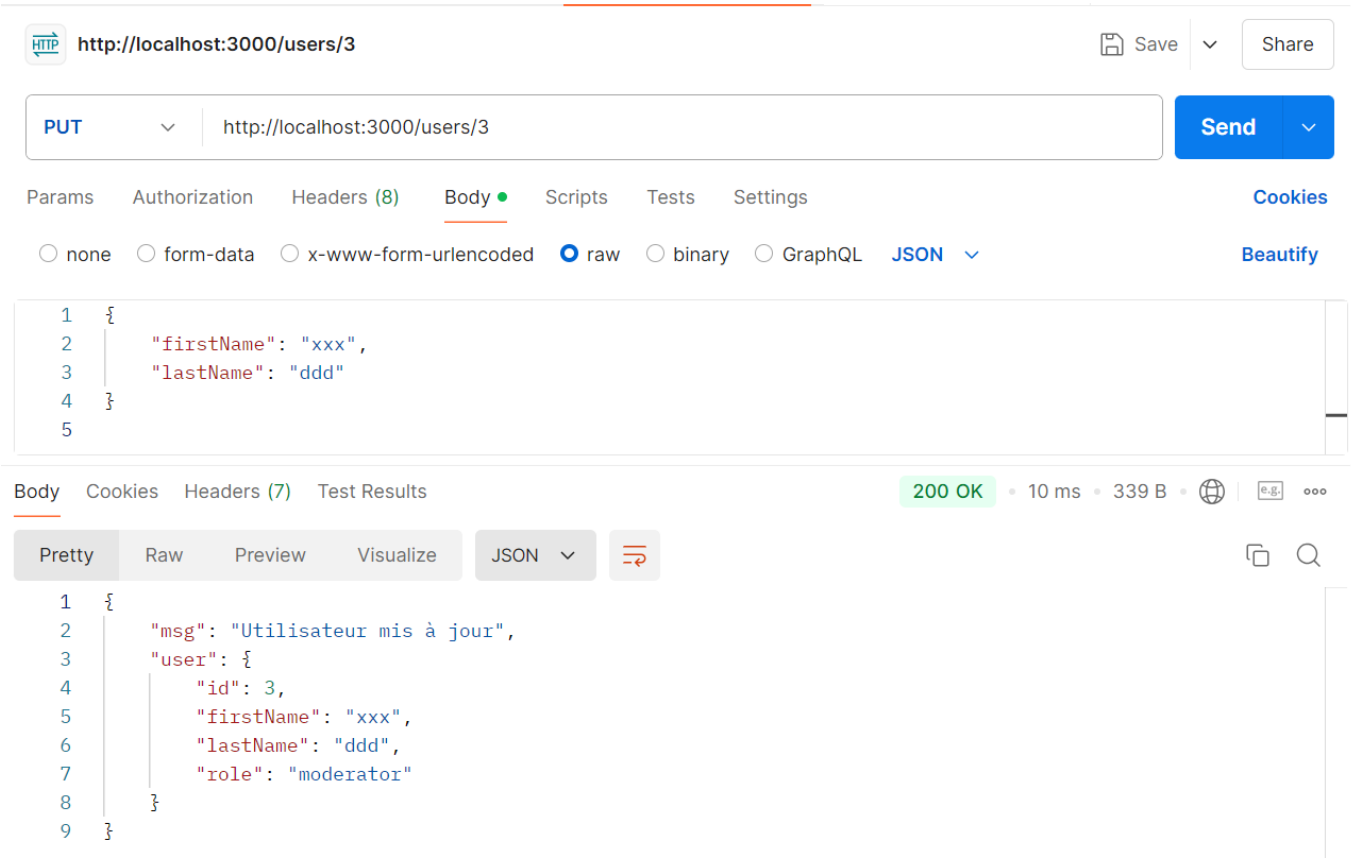
Ajouter un nouvel utilisateur est facile, nous devons utiliser la méthode "push". Mais avant cela, récupérons l'identifiant du dernier utilisateur obtenir l'identifiant du dernier utilisateur.

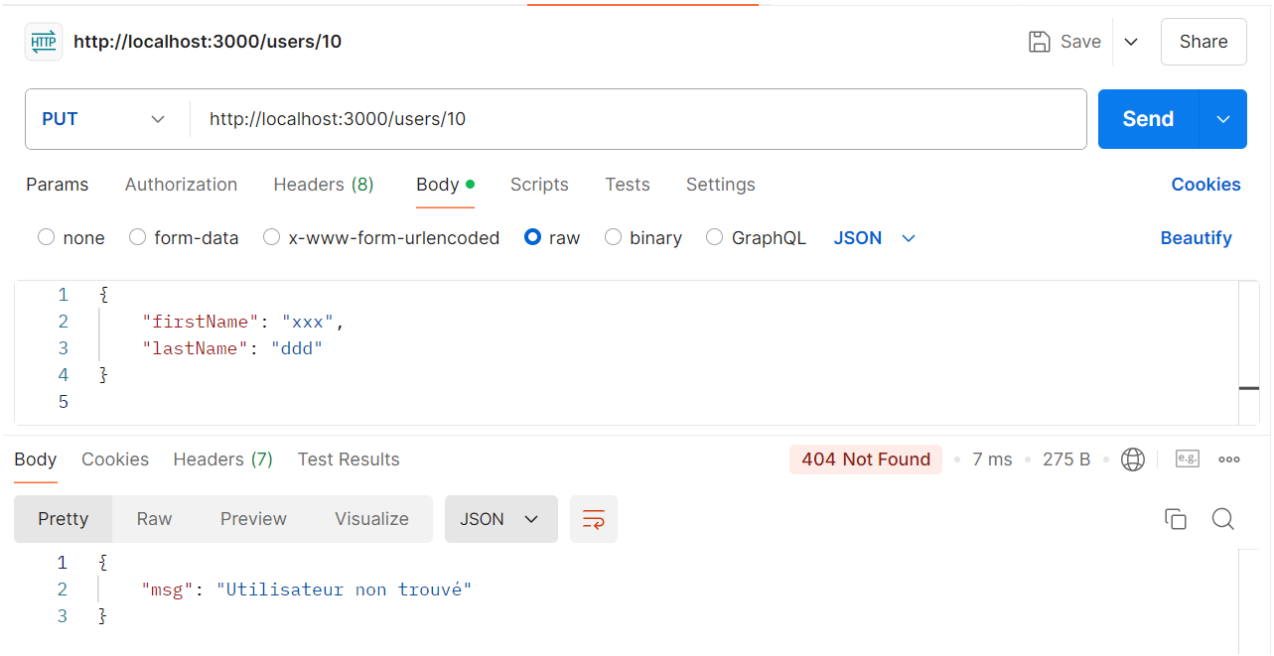
```
const express = require('express');
const app = express();
const port = 3000;

// Middleware pour traiter les données JSON envoyées dans le body
app.use(express.json());

const users = [
    { id: 1, firstName: "John", lastName: "Doe", role: "admin" },
    { id: 2, firstName: "Jane", lastName: "Smith", role: "user" },
    { id: 3, firstName: "Alice", lastName: "Johnson", role: "moderator" },
    { id: 4, firstName: "Bob", lastName: "Brown", role: "user" },
    { id: 5, firstName: "Charlie", lastName: "Davis", role: "admin" },
]


// POST : CRÉER un nouvel utilisateur, basé sur les données passées dans le corps(body) de la requête
app.post("/", (req, res) => {
    // récupérer toutes les données qui arrivent dans le corps de la requête (body)
    const { firstName, lastName } = req.body

    // récupérer l'ID du dernier utilisateur en fonction du nombre d'utilisateurs dans notre variable de tableau 'users'.
    const lastId = users[users.length - 1].id
    // ajouter un pour créer un utilisateur unique
    const newId = lastId + 1

    // créer le nouvel utilisateur avec les données du corps de la requête et l'ID calculé
    const newUser = {
        firstName,
        lastName,
        id: newId,
    }

    // ajouter le nouvel utilisateur à notre liste d'utilisateurs en utilisant la méthode 'push'
    users.push(newUser)
    // envoyer le code de statut 201 (créé) et les données du nouvel utilisateur afin de confirmer au client.
    res.status(201).json(newUser)
})

// Démarrer le serveur
app.listen(port, () => {
    console.log(`Serveur en cours d'exécution sur http://localhost:${port}/`);
});
```



PUT : Modifier un utilisateur en fonction de son ID

Nous allons récupérer toutes les données du corps de la requête. Ce sont les données que nous allons utiliser pour modifier notre utilisateur.



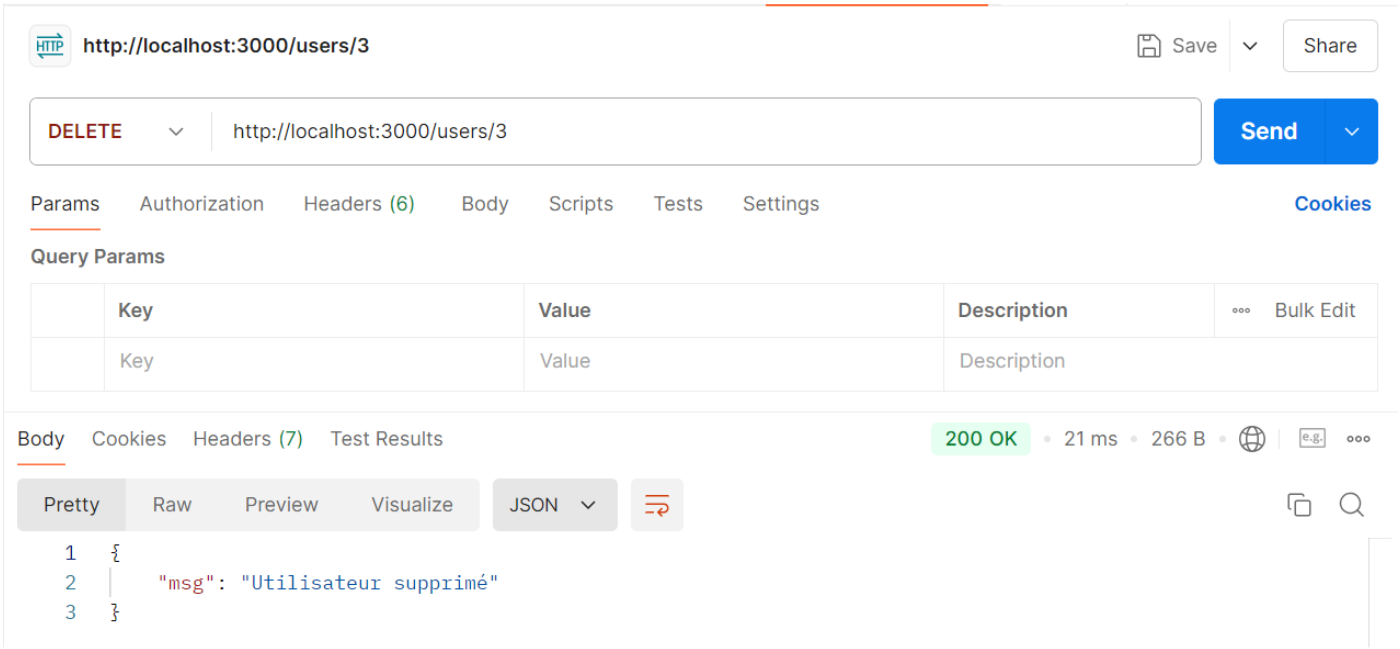


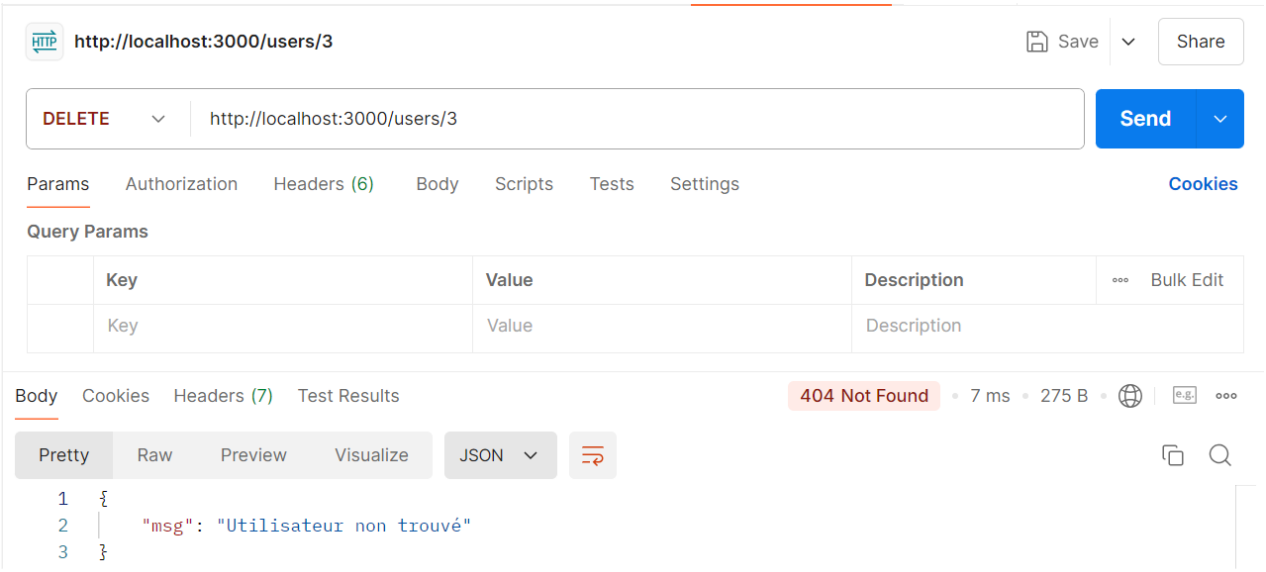

```
const express = require('express');
const app = express();
const port = 3000;

// Middleware pour traiter les données JSON envoyées dans le body
app.use(express.json());

const users = [
    { id: 1, firstName: "John", lastName: "Doe", role: "admin" },
    { id: 2, firstName: "Jane", lastName: "Smith", role: "user" },
    { id: 3, firstName: "Alice", lastName: "Johnson", role: "moderator" },
    { id: 4, firstName: "Bob", lastName: "Brown", role: "user" },
    { id: 5, firstName: "Charlie", lastName: "Davis", role: "admin" },
];

// POST : Ajouter un nouvel utilisateur
app.post("/", (req, res) => {
    const { firstName, lastName } = req.body;
    const lastId = users[users.length - 1].id;
    const newId = lastId + 1;

    const newUser = {
        id: newId,
        firstName,
        lastName
    };

    users.push(newUser);
    res.status(201).json(newUser);
});

// PUT : Modifier un utilisateur existant en fonction de son ID
app.put("/users/:id", (req, res) => {
    // Récupérer l'ID de l'utilisateur depuis les paramètres de l'URL et les données du body
    const id = parseInt(req.params.id);
    const { firstName, lastName } = req.body;
```

```
    // Trouver l'index de l'utilisateur avec l'ID donné
    const userIndex = users.findIndex((user) => user.id === id);

    // Vérifier si l'utilisateur existe
    if (userIndex < 0) {
        return res.status(404).json({ msg: "Utilisateur non trouvé" });
    }

    // Mettre à jour les informations de l'utilisateur si elles sont présentes dans le body
    if (firstName) users[userIndex].firstName = firstName;
    if (lastName) users[userIndex].lastName = lastName;

    // Retourner une réponse avec l'utilisateur mis à jour
    res.json({
        msg: "Utilisateur mis à jour",
        user: users[userIndex],
    });
});

// Démarrer le serveur
app.listen(port, () => {
    console.log(`Serveur en cours d'exécution sur http://localhost:${port}/`);
});
```

DELETE : Supprimer un utilisateur

Nous allons récupérer l'identifiant de l'utilisateur à partir des paramètres envoyés dans l'URL, tout comme nous l'avons fait avec la méthode POST.

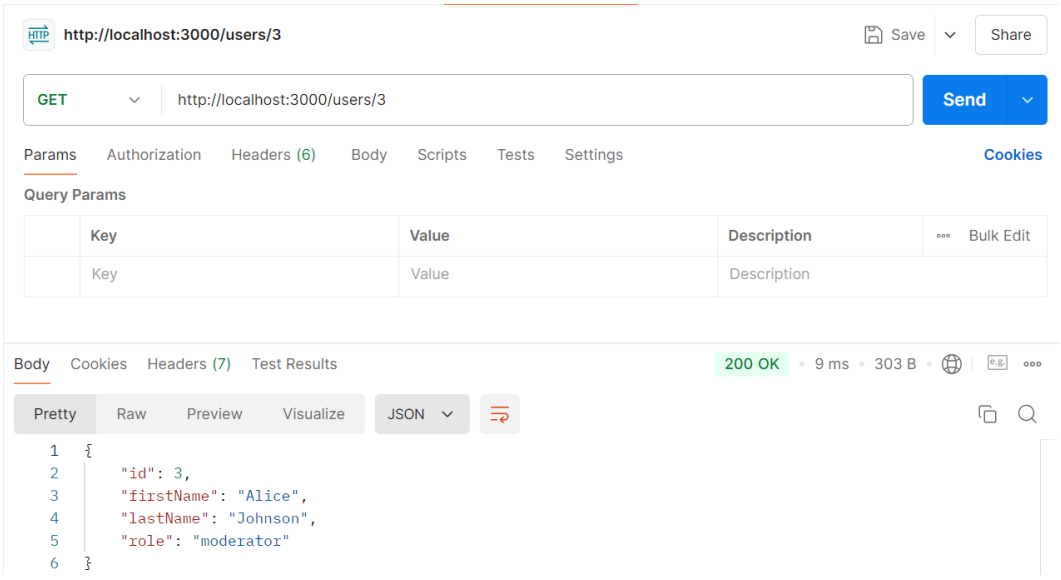


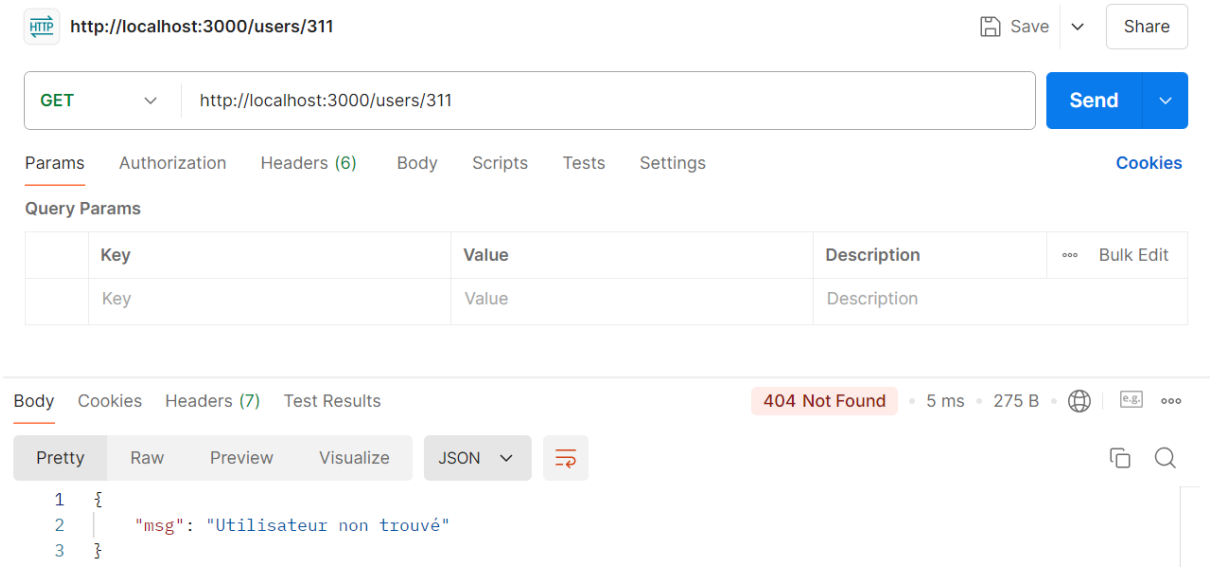



```
const express = require('express');
const app = express();
const port = 3000;

// Middleware pour traiter les données JSON envoyées dans le body
app.use(express.json());

const users = [
    { id: 1, firstName: "John", lastName: "Doe", role: "admin" },
    { id: 2, firstName: "Jane", lastName: "Smith", role: "user" },
    { id: 3, firstName: "Alice", lastName: "Johnson", role: "moderator" },
    { id: 4, firstName: "Bob", lastName: "Brown", role: "user" },
    { id: 5, firstName: "Charlie", lastName: "Davis", role: "admin" },
];

// POST : Ajouter un nouvel utilisateur
app.post("/", (req, res) => {
    const { firstName, lastName } = req.body;
    const lastId = users[users.length - 1].id;
    const newId = lastId + 1;

    const newUser = {
        id: newId,
        firstName,
        lastName
    };

    users.push(newUser);
    res.status(201).json(newUser);
});

// PUT : Modifier un utilisateur existant en fonction de son ID
app.put("/users/:id", (req, res) => {
    const id = parseInt(req.params.id);
    const { firstName, lastName } = req.body;
    const userIndex = users.findIndex((user) => user.id === id);

    if (userIndex < 0) {
```

```
        return res.status(404).json({ msg: "Utilisateur non trouvé" });
    }

    if (firstName) users[userIndex].firstName = firstName;
    if (lastName) users[userIndex].lastName = lastName;

    res.json({
        msg: "Utilisateur mis à jour",
        user: users[userIndex],
    });
});

// DELETE : Supprimer un utilisateur en fonction de son ID
app.delete("/users/:id", (req, res) => {
    // Récupérer l'ID de l'utilisateur depuis les paramètres de l'URL
    const id = parseInt(req.params.id);

    // Trouver l'index de l'utilisateur avec l'ID donné
    const userIndex = users.findIndex((user) => user.id === id);

    // Vérifier si l'utilisateur existe
    if (userIndex < 0) {
        return res.status(404).json({ msg: "Utilisateur non trouvé" });
    }

    // Supprimer l'utilisateur en utilisant la méthode splice
    users.splice(userIndex, 1);

    // Retourner une réponse confirmant la suppression
    res.json({
        msg: "Utilisateur supprimé",
    });
});

// Démarrer le serveur
app.listen(port, () => {
    console.log(`Serveur en cours d'exécution sur http://localhost:${port}/`);
});
```

GET : Un seul utilisateur






```
const express = require('express');
const app = express();
const port = 3000;

// Middleware pour traiter les données JSON envoyées dans le body
app.use(express.json());

const users = [
    { id: 1, firstName: "John", lastName: "Doe", role: "admin" },
    { id: 2, firstName: "Jane", lastName: "Smith", role: "user" },
    { id: 3, firstName: "Alice", lastName: "Johnson", role: "moderator" },
    { id: 4, firstName: "Bob", lastName: "Brown", role: "user" },
    { id: 5, firstName: "Charlie", lastName: "Davis", role: "admin" },
];

// GET : Récupérer tous les utilisateurs
app.get("/", (req, res) => {
    res.json(users);
});
```

```
// GET : Récupérer un utilisateur spécifique en fonction de son ID
app.get("/users/:id", (req, res) => {
    // Récupérer l'ID de l'utilisateur depuis les paramètres de l'URL
    const id = parseInt(req.params.id);

    // Trouver l'utilisateur correspondant à l'ID
    const userIndex = users.findIndex((user) => user.id === id);

    // Vérifier si l'utilisateur existe
    if (userIndex < 0) {
        return res.status(404).json({ msg: "Utilisateur non trouvé" });
    }

    // Si l'utilisateur est trouvé, retourner ses informations
    res.json(users[userIndex]);
});

// POST : Ajouter un nouvel utilisateur
app.post("/", (req, res) => {
    const { firstName, lastName } = req.body;
    const lastId = users[users.length - 1].id;
    const newId = lastId + 1;

    const newUser = {
        id: newId,
        firstName,
        lastName
    };

    users.push(newUser);
    res.status(201).json(newUser);
});

// PUT : Modifier un utilisateur existant en fonction de son ID
app.put("/users/:id", (req, res) => {
    const id = parseInt(req.params.id);
    const { firstName, lastName } = req.body;
    const userIndex = users.findIndex((user) => user.id === id);

    if (userIndex < 0) {
        return res.status(404).json({ msg: "Utilisateur non trouvé" });
    }

    if (firstName) users[userIndex].firstName = firstName;
    if (lastName) users[userIndex].lastName = lastName;

    res.json({
        msg: "Utilisateur mis à jour",
        user: users[userIndex],
    });
});

// DELETE : Supprimer un utilisateur en fonction de son ID
app.delete("/users/:id", (req, res) => {
    const id = parseInt(req.params.id);
```

```
    const userIndex = users.findIndex((user) => user.id === id);

    if (userIndex < 0) {
        return res.status(404).json({ msg: "Utilisateur non trouvé" });
    }

    users.splice(userIndex, 1);
    res.json({ msg: "Utilisateur supprimé" });
});

// Démarrer le serveur
app.listen(port, () => {
    console.log(`Serveur en cours d'exécution sur http://localhost:${port}/`);
});
```

Conclusion :

Ce TP a été une introduction approfondie au développement d'une API REST simple en utilisant Node.js et le framework Express. En partant de zéro, nous avons réalisé l'ensemble du processus pour créer une API capable de gérer des opérations CRUD (Create, Read, Update, Delete) pour une liste d'utilisateurs, ce qui constitue la base de nombreuses applications web modernes.

Nous avons commencé par l'initialisation du projet en configurant l'environnement avec **Node.js** et npm, puis en installant Express pour simplifier la gestion des routes et faciliter la manipulation des requêtes HTTP. Ensuite, un serveur de base a été mis en place dans le fichier `index.js` pour écouter les requêtes entrantes sur un port spécifié.

Dans le code, nous avons défini plusieurs routes pour chaque opération CRUD :

- Lecture de la liste des utilisateurs : La route GET `/users` retourne tous les utilisateurs en format JSON. Cela permet d'exposer la liste complète, simulant une requête pour récupérer les données d'un backend.

- Ajout d'un nouvel utilisateur : La route POST `/users` permet de créer un nouvel utilisateur à partir des informations envoyées dans le corps de la requête (body). Un utilisateur est créé avec un ID unique généré dynamiquement en fonction de la longueur du tableau d'utilisateurs. Avant l'ajout, des vérifications sont effectuées pour s'assurer que les champs obligatoires (firstName, lastName, role) sont fournis. Cette route a permis d'illustrer la gestion des requêtes POST et la validation des données.

- Mise à jour d'un utilisateur existant : La route PUT `/users/:id` permet de modifier les informations d'un utilisateur spécifique, identifié par son `id`. Cette route nous a permis d'explorer l'extraction de paramètres d'URL, une technique courante en gestion d'API. Des conditions vérifient l'existence de l'utilisateur et mettent à jour uniquement les champs présents dans la requête, pour s'assurer que les modifications sont ciblées.

- Suppression d'un utilisateur : La route DELETE `/users/ :id` est utilisée pour supprimer un utilisateur en fonction de son `id`. Cette route a permis d'implémenter la logique de suppression dans le tableau des utilisateurs et de renvoyer une confirmation de suppression au client. Ce type de route est essentiel pour assurer une gestion complète des données au sein de l'API.

Ces étapes ont permis d'explorer plusieurs concepts clés, notamment :

1. Les Middlewares : En utilisant `express.json() `, nous avons appris à gérer et à traiter des données JSON provenant du corps de la requête, ce qui est indispensable dans les API modernes.

2. Manipulation des données en mémoire : Bien que ces données soient stockées en mémoire pour les besoins du TP, le code pourrait être facilement adapté pour interagir avec une base de données dans un contexte de production.

3.Validation des données et gestion des erreurs : Chaque route comporte des vérifications pour s'assurer que les données fournies sont complètes et valides. Les messages d'erreur pertinents permettent d'informer le client de l'état de la requête, un aspect fondamental pour le débogage et la maintenance d'une API.

Grâce à ce TP, nous avons compris les concepts fondamentaux d'une API REST, notamment comment structurer et gérer les données en mémoire, même si, dans un projet plus avancé, ces données seraient stockées dans une base de données.