

Express.js

1) Introduction

Qu'est-ce qu'Express.js

Express est un **framework Web** rapide, affirmé, essentiel et modéré de **Node.js**. Vous pouvez supposer que express est une **couche construite au-dessus de Node.js** qui aide à **gérer** un **serveur** et des **itinéraires**. Il fournit un ensemble robuste de **fonctionnalités** pour développer des **applications Web** et **mobiles**.

Voyons quelques-unes des **principales fonctionnalités du framework Express** :

- Il peut être utilisé pour concevoir des applications Web **monopage**, **multipage** et **hybrides**.
- Il permet de configurer des **middlewares** pour répondre aux requêtes HTTP.
- Il définit une **table de routage** qui est utilisée pour effectuer différentes actions basées sur la méthode HTTP et l'URL.
- Il permet de restituer dynamiquement des pages HTML en passant des arguments aux modèles.

EXEMPLE

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Welcome to JavaTpoint');
})

var server = app.listen({
  host: 'localhost',
  port: 8084,
  exclusive: true
}, function () {
  console.log("Le serveur est en cours d'exécution sur http://%s:%s",
server.address().address, server.address().port);
})
```

2) Installation d'Express

Le code `npm install express --save` est une commande utilisée dans le terminal pour installer le module **Express de Node.js** en utilisant le gestionnaire de paquets **npm** (Node Package Manager).

- `npm` est l'acronyme de "Node Package Manager", c'est un gestionnaire de paquets pour les bibliothèques et les modules de Node.js.
- `install` est la commande qui indique à npm d'installer un module.
- `express` est le nom du module à installer. Dans ce cas, nous installons le module Express, qui est un framework web pour Node.js très populaire.
- `--save` est un indicateur optionnel qui enregistre le module dans le fichier `package.json` du projet et ajoute une dépendance à ce module dans le fichier `package-lock.json`.

En ajoutant l'option `--save`, le module Express sera installé localement dans le dossier du projet et enregistré dans les fichiers `package.json` et `package-lock.json`. Cela permet aux autres développeurs travaillant sur le même projet de facilement installer les mêmes dépendances en utilisant simplement la commande `npm install`.

Notez que depuis la version de **npm 5.0.0**, l'option `--save` est **incluse par défaut** lors de l'installation d'un module. Donc, si vous utilisez une version plus récente de npm, vous pouvez simplement utiliser la commande `npm install express` pour installer le module Express.

Donc La commande ci-dessus installe express dans le répertoire `node_module` et crée un répertoire nommé `express` dans le `node_module`. Vous devez installer d'autres modules importants avec express. Voici la liste : **body-parser**, **cookie-parser** et **multer** .

REMARQUE :

Il existe plusieurs logiciels que vous pouvez utiliser pour modifier votre application Node.js **sans arrêter le serveur**. Voici quelques-uns des **plus populaires** :

1. **Nodemon** : Nodemon est un outil de surveillance qui redémarre automatiquement votre application Node.js à chaque fois que vous modifiez un fichier. Il peut être installé en utilisant NPM (Node Package Manager) en tapant la commande suivante dans votre terminal :
`npm install -g nodemon`
`npm install nodemon --save-dev`

Si vous avez une ERREUR :

```
nodemon : Impossible de charger le fichier
C:\Users\lenovo\AppData\Roaming\npm\nodemon.ps1. Le fichier
C:\Users\lenovo\AppData\Roaming\npm\nodemon.ps1 n'est pas signé numériquement. Vous ne
pouvez pas exécuter ce script sur le
système actuel. Pour plus d'informations sur l'exécution de scripts et la définition de
stratégies d'exécution, voir la
rubrique about_Execution_Policies à l'adresse
https://go.microsoft.com/fwlink/?LinkID=135170.

Au caractère Ligne:1 : 1
+ nodemon e1
+ ~~~~~

+ CategoryInfo          : Erreur de sécurité : (:) [], PSSecurityException

+ FullyQualifiedErrorId : UnauthorizedAccess
```

Il suffit de supprimer le fichier : `C:\Users\lenovo\AppData\Roaming\npm\nodemon.ps1`

2. **PM2** : PM2 est un gestionnaire de processus pour Node.js qui peut être utilisé pour gérer plusieurs processus Node.js en même temps. Il offre également des fonctionnalités de surveillance et de redémarrage automatique similaires à **Nodemon**. Vous pouvez l'installer en tapant la commande suivante dans votre terminal : `npm install -g pm2`
3. **Forever** : Forever est un autre outil de surveillance et de redémarrage automatique pour Node.js. Il peut être installé en tapant la commande suivante dans votre terminal : `npm install -g forever`

Ces outils vous permettront de modifier votre application Node.js en temps réel sans avoir à arrêter et redémarrer manuellement votre serveur à chaque fois que vous effectuez une modification.

Que veut dire **middleware**

En informatique, le terme "middleware" (logiciel intermédiaire) désigne un logiciel qui se situe entre deux autres logiciels pour faciliter leur communication et leur interaction.

En développement web, un **middleware** est un composant logiciel qui est placé entre la **requête du client** et la **réponse du serveur** dans une application web. Il peut être utilisé pour effectuer diverses tâches telles que la **validation des données de la requête**, la **gestion des sessions utilisateur**, **l'authentification et l'autorisation**, la **gestion des erreurs**, le **traitement de données**, le **cache**, etc.

En utilisant des **middlewares**, les développeurs peuvent facilement ajouter des fonctionnalités à leur application web sans avoir besoin de modifier directement le code principal de l'application. Les **middlewares** sont souvent utilisés dans les **frameworks web** tels que **Express.js** pour **Node.js** ou **Django pour Python**, où ils sont largement utilisés pour ajouter des fonctionnalités supplémentaires à une application web.

- 1) **npm install body-parser --save**
- 2) **npm install cookie-parser --save**
- 3) **npm install multer --save**

Ces commandes sont des commandes pour **installer des packages Node.js** à l'aide de **npm** (Node Package Manager). Voici ce que chacune d'entre elles fait :

1) **npm install body-parser --save :**

Cette commande installe le package **body-parser** pour Node.js. **Body-parser** est un **middleware** qui permet de **traiter les données des requêtes HTTP entrantes dans une application Node.js**. Il peut être utilisé pour parser des **données** en **JSON**, en **XML**, en **texte brut**, en données d'un **formulaire HTML**, etc. L'option **---save** ajoute ce package dans le fichier **package.json** de l'application Node.js, ce qui permet à l'application de le retrouver facilement et de l'installer automatiquement lors du déploiement de l'application sur une autre machine.

2) **npm install cookie-parser --save :**

Cette commande installe le package **cookie-parser** pour Node.js. **Cookie-parser** est un **middleware** qui permet de **traiter les cookies HTTP entrants dans une application Node.js**. Il peut être utilisé pour lire les **cookies**, pour les **parser** et pour les **modifier**. L'option **--save** ajoute ce package dans le fichier **package.json** de l'application Node.js, ce qui permet à l'application de le retrouver facilement et de l'installer automatiquement lors du déploiement de l'application sur une autre machine.

3) **npm install multer --save :**

Cette commande installe le package **multer** pour Node.js. **Multer** est un **middleware** qui permet de **traiter les données de formulaire multipart (par exemple, les fichiers) dans une application Node.js**. Il peut être

utilisé pour parser des **fichiers**, pour les stocker sur le **disque dur** ou pour les envoyer à un **service de stockage en ligne**. L'option `--save` ajoute ce package dans le fichier `package.json` de l'application Node.js, ce qui permet à l'application de le retrouver facilement et de l'installer automatiquement lors du déploiement de l'application sur une autre machine.

EXEMPLE

EX0

```
var express = require('express');
var app = express();
app.get('/', function (req, res) {
  res.send('Welcome to JavaTpoint');
})
var server = app.listen(8084, function () {
  var host = server.address().address
  var port = server.address().port
  console.log(host);
  console.log(port);
  console.log(`Example app listening at http://$host:$port`)
})
console.log("l'adresse : http://localhost:8084/ ");
```

3) L' Object Request

L'objet de requête (**request object**) dans **Express.js** est un objet qui représente la **requête HTTP faite par un client** vers **le serveur**. Il contient des informations sur la requête, telles que **l'URL**, la **méthode HTTP**, les **en-têtes** et **les données de corps**. L'objet de requête est passé en tant que **premier argument** à une **fonction** de **gestionnaire de route** dans une application Express.js.

Voici quelques-unes des **propriétés** et des **méthodes** disponibles sur **l'objet de requête** :

- **req.params**: Un objet contenant les propriétés mappées sur les paramètres de route nommés. Par exemple, si vous avez une route définie comme **/users/:userId**, **req.params.userId** contiendra la valeur de **:userId** dans l'URL de la requête.
- **req.query**: Un **objet** contenant **les paramètres d'interrogation analysés**.
- **req.body**: Un **objet** contenant le **corps de la requête analysé**. Cette propriété n'est disponible que si vous avez installé **un middleware** pour gérer l'analyse des corps de requête, comme **body-parser**.
- **req.method**: La méthode HTTP utilisée dans la requête (par exemple **GET**, **POST**, etc.).

- `req.headers`: Un objet contenant les en-têtes de la requête.
- `req.url`: L'URL de la requête.
- `req.path`: La partie de l'URL qui correspond à la route définie pour la requête.
- `req.protocol`: Le protocole utilisé pour la requête (par exemple `http` ou `https`).
- `req.cookies`: Un objet contenant les cookies envoyés avec la requête. Cette propriété est disponible si vous avez installé le middleware `cookie-parser`.
- `req.get(header)`: Une méthode pour récupérer la valeur d'un en-tête spécifique de la requête.
- `req.is(type)`: Une méthode pour vérifier le type de contenu de la requête.

Il est important de noter que la plupart des propriétés de l'objet de requête ne sont pas définies par défaut, mais sont ajoutées à l'objet de requête par les middlewares utilisés dans l'application Express.js.

Exemples pour chaque propriété de l'objet de requête :

Supposons que nous avons une application Express.js avec la route `/users/:userId`, qui renvoie les détails d'un utilisateur spécifique en fonction de son ID.

(1) `req.params`

EX1 : Si un client fait une requête GET à l'URL `/users/123`, `req.params.userId` sera `123`.

```
const express = require('express');
const app = express();
app.get('/users/:userId', (req, res) => {
  const userId = req.params.userId;
  // Récupère l'ID d'utilisateur à partir de la requête URL
  // userId sera '123' pour l'URL /users/123
  // Renvoie les détails de l'utilisateur correspondant à l'ID
  console.log('Utilisateur numero : ' + userId);
});

app.listen(8084, () => {
  console.log('Serveur démarré sur le port 8084');
});
console.log("l'adresse : http://localhost:8084/users/123");
```

(2) req.query

EX2 : Si un client fait une requête **GET** à l'URL : `/search?nom=Slimani&prenom=Brahim`, (`req.query.nom` sera `Slimani`) Et (`req.query.prenom` sera `Brahim`).

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

app.get('/search', (req, res) => {
  const nom = req.query.nom;
  const prenom = req.query.prenom;
  // searchQuery sera 'express' pour l'URL
  /search?nom=Slimani&prenom=Brahim
  // Effectue la recherche en utilisant la requête de recherche
  console.log("NOM: " + nom )
  console.log("PRENOM: " + prenom )
});

app.listen(8084, () => {
  console.log('Serveur démarré sur le port 8084');
});
console.log("l'adresse :
http://localhost:8084/search?nom=Slimani&prenom=Brahim" );
```

EX3 : On veut récupérer deux paramètres de requête via `req.query`. Dans cet exemple, nous supposons qu'on veut faire une recherche sur une `liste d'animaux` et que vous voulez que **l'utilisateur puisse spécifier** un **terme de recherche** (nom de l'animal) ainsi qu'un **nombre maximum de résultats** à renvoyer si le nombre limite n'est pas spécifié sa défaut de 10:

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

app.get('/recherche', (req, res) => {
  const termeRecherche = req.query.terme;
  const limiteResultats = req.query.limite || 10;
  // Si le paramètre limite n'est pas fourni,
  //on utilise une valeur par défaut de 10
```

```

// Recherchez la liste d'animaux en fonction du terme de recherche
const resultatsRecherche = rechercheAnimaux(termeRecherche);

// Renvoyer les premiers résultats jusqu'à la limite spécifiée
const resultatsLimites = resultatsRecherche.slice(0, limiteResultats);

res.send(resultatsLimites);
});

// Fonction de recherche d'animaux factice pour l'exemple
function rechercheAnimaux(termeRecherche) {
  const animaux = [
    'chat',
    'chien',
    'oiseau chat',
    'poisson',
    'souris chat ',
    'cheval',
    'mouton',
    'vache',
    'cochon vache chat ',
    'canard chat '
  ];

  return animaux.filter((animal) => animal.includes(termeRecherche));
}

// Démarrer le serveur
app.listen(8084, () => {
  console.log('Serveur démarré sur le port 8084');
});
console.log("l'adresse : http://localhost:8084/recherche?terme=chat&limite=2");

```

1. Exécutez le fichier en utilisant la commande `node app.js`.
2. Dans votre navigateur Web, accédez à l'URL `http://localhost:8084/recherche?terme=chat&limite=5` (ou n'importe quelle URL avec différents termes de recherche et limites).
3. Le résultat de la recherche s'affiche dans le navigateur, contenant les cinq premiers résultats de la recherche contenant le mot "chat" (puisque nous avons spécifié `limite=5` dans notre URL).

Si tout se passe bien, vous devriez voir une liste d'animaux contenant "chat" dans leurs noms, limitée à cinq résultats.

1. Importation d'Express et création de l'application

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

- `require('express')` importe le module **Express**.
- `express()` crée une application Express.

2. Création d'une route GET /recherche

```
app.get('/recherche', (req, res) => {
```

- Définit un **endpoint** qui répond aux requêtes GET envoyées à `/recherche`.

3. Récupération des paramètres de requête

```
const termeRecherche = req.query.terme;  
const limiteResultats = req.query.limite || 10;
```

- `req.query.terme` récupère la valeur du paramètre `terme` (exemple: `chat` dans `?terme=chat`).
- `req.query.limite` récupère `limite`, sinon utilise 10 comme valeur par défaut.

4. Exécution de la recherche

```
const resultatsRecherche = rechercheAnimaux(termeRecherche);
```

- Appelle `rechercheAnimaux(termeRecherche)`, qui filtre la liste des animaux pour ne garder que ceux contenant le terme recherché.

5. Limitation du nombre de résultats retournés

```
const resultatsLimites = resultatsRecherche.slice(0, limiteResultats);  
res.send(resultatsLimites);
```

- `slice(0, limiteResultats)` extrait les `limiteResultats` premiers résultats.
- `res.send(resultatsLimites)`; envoie ces résultats au client.

6. Fonction de recherche d'animaux

```
function rechercheAnimaux(termeRecherche) {  
  const animaux = [ 'chat', 'chien', 'oiseau chat', ... ];  
  return animaux.filter((animal) => animal.includes(termeRecherche));  
}
```

- Filtre la liste `animaux` pour ne garder que ceux contenant `termeRecherche`.

7. Démarrage du serveur

```
app.listen(8084, () => {  
  console.log('Serveur démarré sur le port 8084');  
});
```

- Démarre un serveur sur le port 8084.

8. Affichage de l'URL d'exemple

```
console.log("l'adresse : http://localhost:8084/recherche?terme=chat&limite=2");
```

- Affiche une URL d'exemple pour tester l'API.

Exemple d'utilisation

Si vous accédez à :

🔗 **`http://localhost:8084/recherche?terme=chat&limite=2`**

Le serveur retourne :

```
["chat", "oiseau chat"]
```

Seuls les **deux premiers** résultats contenant `"chat"` sont affichés

(3) req.body

npm install body-parser --save

EX4 : Si un client envoie une requête POST avec un corps JSON contenant les détails d'un nouvel utilisateur, les détails peuvent être récupérés à l'aide de `req.body`.

e4F.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Document</title>
</head>
<body>
  <form action="/submit" method="post">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" required>

    <label for="email">Email:</label>
    <input type="email" id="email" name="email" required>

    <button type="submit">Submit</button>
  </form>

</body>
</html>
```

e4.js

```
const express = require('express')
const bodyParser = require('body-parser')
const { Console } = require('console')

var name = ''
var email = ''
const app = express()
```

```
// Création d'une instance de Console
const myConsole = new Console(process.stdout, process.stderr);

// parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: false }))

// parse application/json
app.use(bodyParser.json())

// Route pour la page HTML
app.get('/e4F', (req, res) => {
    res.sendFile(__dirname + '/e4F.html');
});

app.post('/submit', (req, res) => {
    name = req.body.name
    email = req.body.email
    // faire quelque chose avec les données reçues...
    const Donnee = 'NOM :'+ name +' EMAIL : '+email
    res.send(Donnee);
    myConsole.log('NOM :'+ name +' EMAIL : '+email)
})

app.listen(8084, () => {
    console.log('Le serveur est en cours d\'exécution 8084...');
    console.log("l'adresse : http://localhost:8084/e4F' ");
});
```

[illegible]

(4) req.method

EX5 :

e5F.html pour method post

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Exemple de formulaire POST</title>
  </head>
  <body>
    <form action="http://localhost:8084/example2" method="POST">
      <input type="text" name="example" value="Exemple de donnée POST" />
      <input type="submit" value="Envoyer" />
    </form>
  </body>
</html>
```

e5FF.html pour method get

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Exemple de formulaire POST</title>
  </head>
  <body>
    <form action="http://localhost:8084/example1" method="GET">
      <input type="text" name="example" value="Exemple de donnée GET" />
      <input type="submit" value="Envoyer" />
    </form>
  </body>
</html>
```

e5.js pour method Get et Post

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

// Route pour la page HTML post
app.get('/e5F', (req, res) => {
  res.sendFile(__dirname + '/e5F.html');
});

// Route pour la page HTML get
app.get('/e5FF', (req, res) => {
  res.sendFile(__dirname + '/e5FF.html');
});

// Route pour la méthode GET
app.get('/example1', (req, res) => {
  console.log(req.method); // Affiche "GET" dans la console
  res.send('Exemple de requête GET');
});

// Route pour la méthode POST
app.post('/example2', (req, res) => {
  console.log(req.method); // Affiche "POST" dans la console
  res.send('Exemple de requête POST');
});

// Écouter le port 8084
app.listen(8084, () => {
  console.log('Serveur démarré sur le port 8084');
  console.log("l'adresse : http://localhost:8084/e5F' ");
  console.log("l'adresse : http://localhost:8084/e5FF' ");
});
```

(5) req.headers

EX6 : Supposons que nous ayons une route qui prend une **demande GET**:

Si nous envoyons une requête GET à cette route avec des en-têtes personnalisés, nous verrons la sortie de **req.headers** dans le **navigateur**:

Route 1 : (la **racine '/'**) affiche dans la **console** et dans le **navigateur** à l'aide de **res.json()** :

```
app.get('/', (req, res) => {  
  console.log(req.headers); //dans la console  
  res.json(req.headers); // affiche dans le navigateur  
});
```

Route 2 (une requête GET **'/exemple'**) affiche dans
- **navigateur** : **res.send()**;

```
app.get('/exemple', (req, res) => {  
  
  const message = 'Voici les en-têtes de votre requête:';  
  const headers = req.headers;  
  const output = `${message} </br> ${JSON.stringify(headers, null, 2)}`;  
  
  res.send(output);  
  
});
```

Cet objet contient tous les en-têtes de la requête, y compris l'hôte, la connexion, le contrôle de cache, l'agent utilisateur, etc. Les en-têtes personnalisés sont également inclus, tels que l'en-tête 'cookie' dans cet exemple.

Programme Complet :

```
const express = require('express');  
const app = express();  
  
app.get('/', (req, res) => {  
  res.json(req.headers);  
  console.log(req.headers);  
});  
  
app.get('/exemple', (req, res) => {
```

```

const message = 'Voici les en-têtes de votre requête: ';
const headers = req.headers;
const output = `${message} </br> ${JSON.stringify(headers, null, 2)} `;

res.send(output);

});

app.listen(8084, () => {

  console.log('Serveur démarré sur le port 8084');
});
console.log("l'adresse : http://localhost:8084/exemple ");

```

(6) req.url

EX7:

Dans cet exemple, nous créons une application Express qui définit 3 routes: '/', '/contact' et '/etudiant'. Chaque route a une fonction de rappel qui gère la requête HTTP correspondante.

Lorsqu'un client accède à l'application à la racine ('/'), la fonction de rappel de la route '/' est exécutée. Cette fonction de rappel reçoit deux arguments: req (la requête HTTP) et res (la réponse HTTP). Dans la fonction de rappel, nous utilisons req.url pour obtenir l'URL demandée par le client et nous affichons cette URL dans le navigateur.

De même, lorsque le client accède à '/contact', la fonction de rappel de la route '/contact' est exécutée, et nous utilisons à nouveau req.url pour obtenir l'URL demandée et l'afficher dans la navigateur.

De même, lorsque le client accède à '/etudiant', la fonction de rappel de la route '/etudiant' est exécutée, et nous utilisons à nouveau req.url pour obtenir l'URL demandée et l'afficher dans la navigateur.

Notez que req.url renvoie l'URL demandée par le client, y compris la partie du chemin (par exemple, '/contact') et la partie de la requête (par exemple, '/contact?subject=feedback'), le cas échéant.

En résumé, `req.url` est une propriété de l'objet de requête `req` dans Express.js qui contient l'URL demandée par le client. Elle est souvent utilisée pour déterminer quelle route doit être utilisée pour gérer la requête.

```
const express = require('express');
const app = express();
var url;
var output;
var message

app.get('/', (req, res) => {
  message = 'Page d\'accueil: ';
  url = req.url;
  output = `${message} </br> ${JSON.stringify(url, null, 2)}`;
  res.send(output);
});

app.get('/contact', (req, res) => {
  message = 'Page de contact: ';
  url = req.url;
  output = `${message} </br> ${JSON.stringify(url, null, 2)}`;
  res.send(output);
});

app.get('/etudiant', (req, res) => {
  message = 'Page de etudiant: ';
  url = req.url;
  const output = `${message} </br> ${JSON.stringify(url, null, 2)}`;
  res.send(output);
});

app.listen(8084, () => {
  console.log("Serveur lancé sur l'adresse : http://localhost:8084");
});
```


(7) req.path

EX8:

Dans cet exemple, **req.path** est utilisé dans plusieurs des routes définies :

`'/', '/about', '/contact'.`

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

// Route pour la page d'accueil
app.get('/', function(req, res) {
  res.send(`Bienvenue sur la page d'accueil !:</br> ${req.path}`);
});

// Route pour la page "À propos"
app.get('/about', function(req, res) {
  res.send(`À propos de nous:</br> ${req.path}`);
});

// Route pour la page "Contact"
app.get('/contact', function(req, res) {
  res.send(`Nous contacter :</br> ${req.path}`);
});

// Route pour toutes les autres pages
app.get('*', function(req, res) {
  res.send(`Page introuvable: ${req.path}`);
});

// Écoute du serveur
app.listen(8084, function() {
  console.log("Serveur lancé sur l'adresse : http://localhost:8084");
});
```

(8) req.protocol

EX9 : Dans cet exemple, nous avons créé une application Express simple qui écoute sur le port **8084**. Lorsqu'un utilisateur accède à la page d'accueil (`/`), nous récupérerons le protocole utilisé (HTTP ou HTTPS) en utilisant la propriété **protocol** de l'objet **req** (la requête).

Ensuite, nous renvoyons une réponse à l'utilisateur en utilisant **res.send()** avec une chaîne de caractères qui inclut le protocole récupéré.

```

const express = require('express');

const app = express();

// Route pour récupérer le protocole utilisé
app.get('/', (req, res) => {
  const protocol = req.protocol;
  res.send(`Le protocole utilisé est : ${protocol}`);
});

// Écoute du serveur
app.listen(8084, function() {
  console.log("Serveur lancé sur l'adresse : http://localhost:8084");
});

```

(9) req.cookies

EX10: utilisation de **req.cookies** avec Express.js :

Tout d'abord, vous devez installer le module **cookie-parser** via npm :

```
npm install cookie-parser
```

Dans cette exemple on utilise **req.cookies** pour stocker et afficher un nom d'utilisateur :

views/index.html :

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Exemple de cookies avec Express.js</title>
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">

  </head>
  <body>
    <header>
      <h1>Bienvenue sur notre site web !</h1>
      <nav>
        <ul>
          <li><a href="/">Accueil</a></li>
          <li><a href="/about">À propos</a></li>
          <li><a href="/contact">Contact</a></li>
          <li><a href="/logout">log out</a></li>
        </ul>
      </nav>
    </header>

```

```

<main>
  <section>
    <h2>Connexion</h2>

    <form action="/login" method="post">
      <label for="username">Nom d'utilisateur :</label>
      <input type="text" id="username" name="username">
      <button type="submit">Se connecter</button>
    </form>
  </section>

</main>
<footer>
  <p>Tous droits réservés © 2023</p>
</footer>

</body>
</html>

```

ex10.js

```

const express = require('express');
const cookieParser = require('cookie-parser');

const app = express();

app.use(cookieParser());
app.use(express.static('public'));
app.use(express.urlencoded({ extended: true }));

app.get('/logout', (req, res) => {
  res.clearCookie('username');
  res.redirect('/');
});

app.get('/', (req, res) => {
  const username = req.cookies.username;
  if (username) {
    res.send(`Bonjour ${username} !`);
  } else {
    res.sendFile('views/index.html', { root: __dirname });
  }
});

app.post('/login', (req, res) => {
  const { username } = req.body;

```

```
res.cookie('username', username);
res.redirect('/');
});

// Écoute du serveur
app.listen(8084, function() {
  console.log("Serveur lancé sur l'adresse : http://localhost:8084");
});
```

`const express = require('express');` : Importe le module Express.js pour créer et exécuter un serveur web.

`const app = express();` : Crée une instance d'application Express.js.

`const cookieParser = require('cookie-parser');` : Importe le module `cookie-parser` pour traiter les cookies HTTP dans Express.js.

`app.use(cookieParser());` : Cette ligne utilise le middleware **cookie-parser** pour gérer les cookies. Les cookies sont des petits fichiers de données stockés sur le navigateur du client. En utilisant ce middleware, les cookies peuvent être analysés et manipulés facilement dans l'application

`app.use(express.static('public'));` : Cette ligne permet de servir les fichiers statiques du dossier public sur le serveur. Les fichiers statiques incluent des fichiers tels que des images, des fichiers CSS et des fichiers JavaScript qui sont nécessaires pour afficher une page web.

`app.use(express.urlencoded({ extended: true }));` : Cette ligne utilise le middleware body-parser pour traiter les données envoyées via une méthode POST à partir d'un formulaire HTML. L'option `extended` permet de définir si l'analyse doit être étendue ou non. Si cette option est définie sur `true`, l'analyse prend en charge les données URL encodées complexes.

`app.get('/', (req, res) => {...});` : Configure une route HTTP GET pour l'URL racine (/) de l'application Express.js. Cette route utilise une fonction de rappel pour gérer la demande HTTP.

`const username = req.cookies.username;` : Récupère la valeur du cookie nommé `username` de la demande HTTP.

`if (username) {...}` : Vérifie si le cookie `username` existe. Si c'est le cas, affiche un message de bienvenue personnalisé avec le nom d'utilisateur extrait du cookie.
`res.sendFile('views/index.html', { root: __dirname });` : Si le cookie `username` n'existe pas, envoie la page HTML d'accueil (`index.html`) au client.

`app.post('/login', (req, res) => {...});` : Configure une route HTTP POST pour l'URL `/login` de l'application Express.js. Cette route utilise une fonction de rappel pour traiter les données POST envoyées par le client.

`res.cookie('username', username);` : Crée un nouveau cookie HTTP nommé `username` avec la valeur du nom d'utilisateur entré dans le formulaire de connexion. Le cookie est envoyé dans l'en-tête de la réponse HTTP.

`res.redirect('/');` : Redirige le client vers l'URL racine (/) de l'application Express.js après avoir créé le cookie `username`.

`<form action="/login" method="post">` : Crée un formulaire HTML qui envoie une demande HTTP POST à l'URL `/login` lorsqu'il est soumis.

`<input type="text" id="username" name="username">` : Crée un champ de saisie de texte pour le nom d'utilisateur.

`<button type="submit">Se connecter</button>` : Crée un bouton de soumission pour envoyer le formulaire.

En résumé, ce code configure un serveur Express.js pour gérer les cookies HTTP et fournit une page HTML avec un formulaire de connexion et un message dynamique personnalisé en fonction de l'état actuel

```
(10) req.get(header)
```

EX11:

V1

Dans cet exemple, nous créons une application Express.js qui écoute sur le port **8084**. Nous avons également créé un **middleware de démonstration** appelé **demoMiddleware**. Ce middleware est une fonction qui est appelée pour chaque requête entrante avant que la route correspondante ne soit appelée.

La fonction middleware nommée **demoMiddleware**. Les middlewares sont des fonctions qui sont exécutées par Express.js avant que les routes de l'application soient appelées. Le but de cette fonction middleware est d'afficher les en-têtes de la requête sur la console et de passer la main au middleware suivant en appelant la fonction **next()**.

Voici une explication ligne par ligne :

- **const demoMiddleware = (req, res, next) => {** : Cette ligne définit une fonction nommée **demoMiddleware** qui prend trois paramètres : **req**, **res** et **next**. **req** représente l'objet requête HTTP entrant, **res** représente l'objet réponse HTTP sortant et **next** est une fonction qui doit être appelée pour passer le contrôle au middleware suivant dans la chaîne de middlewares.
- **console.log('Headers de la requête:', req.headers);** : Cette ligne affiche les en-têtes de la requête HTTP sur la console en utilisant la méthode **console.log()**. Les en-têtes de la requête contiennent des informations telles que le type de contenu, la langue et les informations de connexion.
- **next();** : Cette ligne appelle la fonction **next()** pour passer le contrôle au middleware suivant. Si cette fonction n'est pas appelée, la réponse HTTP ne sera jamais renvoyée au client.

On définit une route d'application en utilisant la méthode **app.get()** d'Express.js. La route définit une réponse pour la requête HTTP GET à la racine de l'application. Elle utilise également un middleware nommé **demoMiddleware** pour traiter la requête avant que la réponse ne soit renvoyée au client.

Voici une explication ligne par ligne :

- `app.get('/', demoMiddleware, (req, res) => {})` : Cette ligne définit une route GET à la racine de l'application en utilisant la méthode `app.get()`. La route prend deux arguments : la chaîne `"/"` pour la racine de l'application et une fonction de rappel qui est appelée lorsque la route est appelée. Cette ligne utilise également le middleware `demoMiddleware` en tant que deuxième argument pour la méthode `app.get()`. Cela signifie que la fonction `demoMiddleware` sera exécutée avant la fonction de rappel de la route.
- `res.send('Exemple d\'utilisation de req.headers');` : Cette ligne envoie la réponse HTTP au client en utilisant la méthode `res.send()`. La réponse est une chaîne de caractères qui indique l'exemple d'utilisation des en-têtes de la requête HTTP enregistrés par le middleware `demoMiddleware`.

Dans le middleware `demoMiddleware`, nous utilisons la propriété `req.headers` pour accéder aux en-têtes de la requête. Les en-têtes sont un ensemble de **paires clé-valeur** qui sont envoyées avec chaque requête HTTP. Les en-têtes peuvent inclure des informations telles que le type de contenu accepté par le client, la langue préférée, les cookies et bien plus encore.

Nous utilisons simplement la fonction `console.log()` pour afficher les en-têtes dans la console à des fins de démonstration.

Enfin, nous avons créé une **route de démonstration** qui appelle le middleware `demoMiddleware` **avant de renvoyer une réponse**. Cette route est configurée pour gérer les requêtes GET vers la racine du site (`/`).

Lorsque vous exécutez cet exemple et accédez à `http://localhost:3000/` dans votre navigateur, vous devriez voir les en-têtes de la requête affichés dans la console de votre terminal. Ces en-têtes peuvent varier en fonction du navigateur que vous utilisez et des paramètres de requête que vous avez envoyés.

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

// Middleware de démonstration
const demoMiddleware = (req, res, next) => {
  console.log('Headers de la requête:', req.headers);
  next();
}

// Route de démonstration
app.get('/', demoMiddleware, (req, res) => {
  res.send('Exemple d\'utilisation de req.headers');
});
```

```
// Écoute du serveur
app.listen(8084, function() {
  console.log("Serveur lancé sur l'adresse : http://localhost:8084");
});
```

V2

Pour afficher `req.headers` dans la réponse HTTP envoyée au **navigateur** plutôt que dans la **console**, vous pouvez simplement remplacer `console.log()` par `res.send()`.

Dans cet exemple, nous avons modifié le middleware `demoMiddleware` pour envoyer une réponse HTTP contenant les en-têtes de la requête en format JSON, mais avec une mise en forme plus lisible grâce à l'utilisation de la balise HTML `<pre>`.

Nous avons également supprimé la deuxième fonction callback de la route de démonstration, car elle n'était pas nécessaire pour cet exemple.

Lorsque vous exécutez cet exemple et accédez à `http://localhost:3000/` dans votre navigateur, vous devriez voir une liste détaillée des en-têtes de la requête affichée dans votre navigateur.

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

// Middleware de démonstration
const demoMiddleware = (req, res, next) => {
  const headers = req.headers;
  res.send(`<pre>${JSON.stringify(headers, null, 2)}</pre>`);
  //next();
}

// Route de démonstration
app.get('/', demoMiddleware);
// Écoute du serveur
app.listen(8084, function() {
  console.log("Serveur lancé sur l'adresse : http://localhost:8084");
});
```


4) Object Response

En développement web, l'objet response (ou réponse en français) est un objet qui est renvoyé par le serveur en réponse à une requête HTTP. Cet objet est utilisé pour définir les propriétés de la réponse, telles que le **statut** de la réponse, les en-têtes de la réponse et le corps de la réponse.

Les propriétés les plus couramment utilisées de l'objet response incluent :

- **status**: la propriété **status** définit le code de statut HTTP de la réponse, qui indique si la requête a été réussie, redirigée ou s'il y a eu une erreur. Les codes de statut HTTP courants incluent 200 (OK), 404 (Not Found), 500 (Internal Server Error), etc.

```
// Définir le code de statut HTTP 404
res.status(404).send('Page not found');
```

- **headers**: la propriété headers est utilisée pour définir les en-têtes HTTP de la réponse, tels que Content-Type, Cache-Control, Set-Cookie, etc.

```
// Définir le type de contenu de la réponse en tant que JSON
res.setHeader('Content-Type', 'application/json');
```

- **send**: la méthode send est utilisée pour envoyer la réponse au client. Cette méthode peut prendre en charge divers types de données, tels que des chaînes de caractères, des objets JSON, des tableaux et des fichiers.

```
// Envoyer une réponse JSON au client
res.send({ message: 'Hello world!' });
```

- **redirect**: la méthode redirect est utilisée pour rediriger la requête vers une autre URL.

```
// Rediriger l'utilisateur vers une autre page
res.redirect('/login');
```

- **cookie**: la méthode cookie est utilisée pour définir des cookies HTTP qui seront envoyés au client avec la réponse.

```
// Définir un cookie
res.cookie('username', 'john_doe', { maxAge: 900000, httpOnly: true
});
```

- **download**: la méthode download est utilisée pour télécharger un fichier en tant que réponse.

```
// Télécharger un fichier
res.download('/path/to/file.pdf');
```

L'objet response est utilisé en conjonction avec l'objet request (ou requête) pour créer une communication bidirectionnelle entre le serveur et le client.

EX21

Cet exemple définit une route qui renvoie une liste de produits en format JSON. Il utilise les propriétés **status**, **headers** et **send** de l'objet **response** pour renvoyer la **réponse au client**

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

// Définir une route pour la liste des produits
app.get('/produits', (req, res) => {
  // Simulation d'une liste de produits
  const products = [
    { id: 1, name: 'Product 1', price: 10.99 },
    { id: 2, name: 'Product 2', price: 20.99 },
    { id: 3, name: 'Product 3', price: 30.99 }
  ];

  // Définir le type de contenu de la réponse en tant que JSON
  res.setHeader('Content-Type', 'application/json');

  // Définir le code de statut HTTP 200
  res.status(200);
```

```
// Envoyer la liste des produits au client  
res.send(products);  
});  
  
// Écoute du serveur  
app.listen(8084, function() {  
  console.log("Serveur lancé sur l'adresse : http://localhost:8084");  
});
```

Dans cet exemple, la route `/produits` renvoie une liste de produits sous forme de tableau JSON. L'objet **response** est utilisé pour définir le type de contenu de la réponse en tant que JSON avec la méthode **setHeader**, définir le code de statut HTTP 200 avec la propriété **status**, et envoyer la liste des produits au client avec la méthode **send**.

Série EXERCICES :

1. Créez une application Express.js qui affiche "Hello World!" lorsque vous accédez à la racine ("/") de l'application.
2. Créez une application Express.js qui affiche la date et l'heure actuelles lorsque vous accédez à la racine ("/").
3. Créez une application Express.js qui affiche un message personnalisé lorsque vous accédez à une URL spécifique ("/message").
4. Créez une application Express.js qui affiche le contenu d'un tableau JSON lorsque vous accédez à une URL spécifique ("/users").

```
const users = [  
  { id: 1, name: 'Alice' },  
  { id: 2, name: 'Bob' },  
  { id: 3, name: 'Charlie' },  
];
```

5. Créez une application Express.js qui affiche le contenu d'un objet JSON lorsque vous accédez à une URL spécifique ("/user").

```
user = {  
  id: 1,  
  name: 'Alice',  
  age: 25,  
  email: 'alice@example.com'  
};
```

6. Créez une application Express.js qui affiche les informations d'un utilisateur spécifique lorsque vous accédez à une URL spécifique ("/user/:id").

```
users = [  
  { id: 1, name: 'Ali' },  
  { id: 2, name: 'Omar' },  
  { id: 3, name: 'Fatima' },  
];
```

7. Créez une application Express.js qui affiche une page HTML personnalisée de votre choix lorsque vous accédez à une URL spécifique ("/page").
8. Créez une application Express.js qui récupère des données à partir d'un formulaire HTML et les affiche sur une autre page lorsque vous envoyez le formulaire ("/submit").

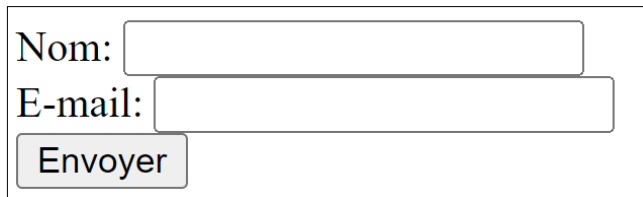
Formulaire de soumission

Nom:

E-mail:

9. Créez une application Express.js qui redirige l'utilisateur vers une autre URL lorsque vous accédez à une URL spécifique de votre choix ("/redirect").
10. Créez une application Express.js qui affiche les informations du navigateur de l'utilisateur lorsque vous accédez à une URL spécifique ("/user-agent").
11. Créez une application Express.js qui affiche les en-têtes de la requête HTTP lorsque vous accédez à une URL spécifique ("/headers").
12. Créez une application Express.js qui affiche les cookies envoyés avec la requête HTTP lorsque vous accédez à une URL spécifique ("/cookies").
13. Créez une application Express.js qui stocke des données dans un cookie et les affiche sur une autre page lorsque vous accédez à une URL spécifique ("/store").
14. Créez une application Express.js qui efface un cookie lorsque vous accédez à une URL spécifique ("/clear").
15. Créez une application Express.js qui affiche les paramètres de l'URL lorsque vous accédez à une URL spécifique ("/params/:id").
16. Créez une application Express.js qui affiche les paramètres de la requête lorsque vous accédez à une URL spécifique ("/query").

17. Créez une application Express.js qui affiche les données envoyées via la méthode POST lorsque vous accédez à une URL spécifique ("/post") de page html :



A simple HTML form with a light gray border. It contains two text input fields. The first is labeled 'Nom:' and the second is labeled 'E-mail:'. Below these fields is a button labeled 'Envoyer'.

18. Créez une application Express.js qui renvoie un code d'état 404 lorsque vous accédez à une URL non définie.

19. Créez une application Express.js qui renvoie un code d'état 500 lorsqu'il y a une erreur interne du serveur.

20. Créez une application Express.js qui gère les erreurs d'URL incorrecte et renvoie un message d'erreur approprié.

21. Créez une application Express.js qui renvoie une page d'erreur personnalisée lorsque vous accédez à une URL inexistante.

22. Créez une application Express.js qui redirige l'utilisateur vers une page d'erreur personnalisée lorsque vous accédez à une URL inexistante.

23. Créez une application Express.js qui permet de créer un nouvel utilisateur en envoyant une requête POST avec les données de l'utilisateur dans le corps de la requête ("req.body").

24. Créez une application Express.js qui permet de mettre à jour les informations d'un utilisateur existant en envoyant une requête PUT avec les données mises à jour dans le corps de la requête ("req.body").

25. Créez une application Express.js qui permet de supprimer un utilisateur en envoyant une requête DELETE avec l'ID de l'utilisateur dans les paramètres de l'URL ("req.params").

26. Créez une application Express.js qui affiche le nombre de requêtes HTTP reçues depuis le lancement de l'application.

- 27.** Créez une application Express.js qui ajoute un en-tête personnalisé à chaque réponse envoyée par l'application ("res.setHeader").
- 28.** Créez une application Express.js qui utilise la méthode "res.status" pour renvoyer un code d'état HTTP spécifique.
- 29.** Créez une application Express.js qui utilise la méthode "res.send" pour renvoyer différents types de données en fonction du type de la requête HTTP (JSON, HTML, texte, etc.).
- 30.** Créez une application Express.js qui utilise la méthode "res.redirect" pour rediriger l'utilisateur vers une autre URL en fonction des données de la requête HTTP.