Dev Web

côté serveur

S3 - R3.01 - 2024/2025

Créer un projet VS Charger un projet

1. Créer un nouveau projet

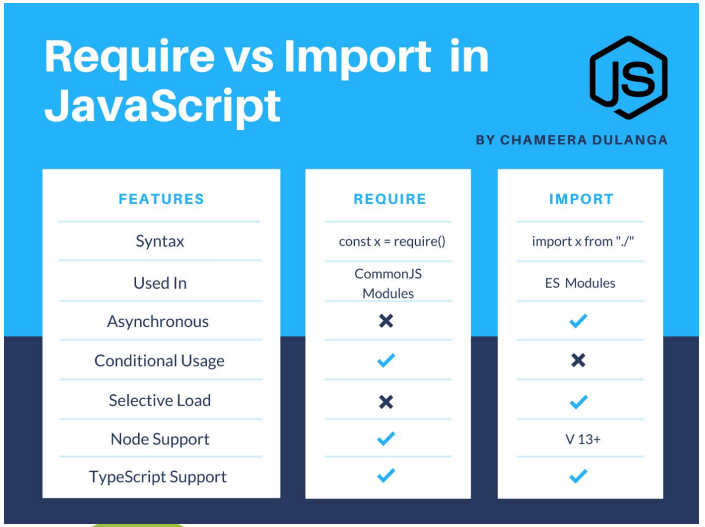
- **`npm init`** :
  Cette commande vous permet de créer un nouveau projet Node.js en interagissant avec vous pour configurer le fichier `package.json`.
  Vous serez invité à renseigner diverses informations telles que le nom du projet, la version, la description, l'auteur, etc.
- **`npm init -y`** :
  Crée un nouveau projet avec un fichier `package.json` par défaut sans vous demander de confirmer chaque paramètre. Idéal pour un démarrage rapide.
  Par défaut :
  - Nom du projet : Nom du dossier
  - Version : 1.0.0
  - Licence : ISC

2. Charger un projet existant

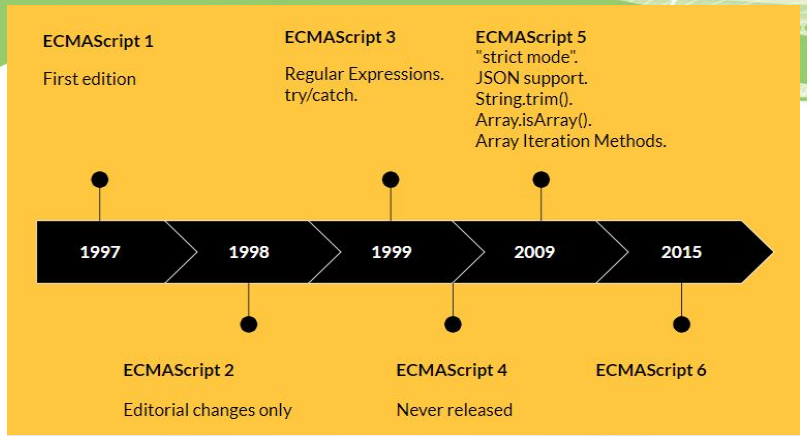
- **`npm install`** :
  Lorsque vous avez déjà un projet existant avec un fichier `package.json`, la première étape consiste à exécuter `npm install`. Cette commande va :
  - Installer toutes les dépendances listées dans le fichier `package.json`.
  - Créer un dossier `node_modules` contenant ces dépendances.
- Utilisez cette commande après avoir cloné ou téléchargé un projet Node.js pour rétablir l'environnement de travail.

CommonJS & ESM

- CommonJs est le système de module original et par défaut de Node.js qui utilise require et module.exports.
- Les modules ECMAScript sont relativement récents et utilisent import & export.



CommonJS & ESM



ECMAScript modules	CommonJS modules
Files ending in .js if there is a top-level field "type" with a value of "module" in the nearest parent package.json file	**Files ending in .js** f there is a top-level field "type" with a value of "commonjs" in the nearest parent package.json file
Files ending in .mjs	**Files ending in .cjs**
Strings passed in as an argument to --eval or --print, or piped to node via STDIN, with the flag --input-type=module	**Strings passed in as an argument to --eval or --print, or piped to node via STDIN, with the flag --input-type=commonjs**
	Files ending in .js if there is no top-level field "type" in the nearest parent package.json file
	Strings passed in as an argument to --eval or --print, or piped to node via STDIN, without the flag --input-type

CommonJS & ESM

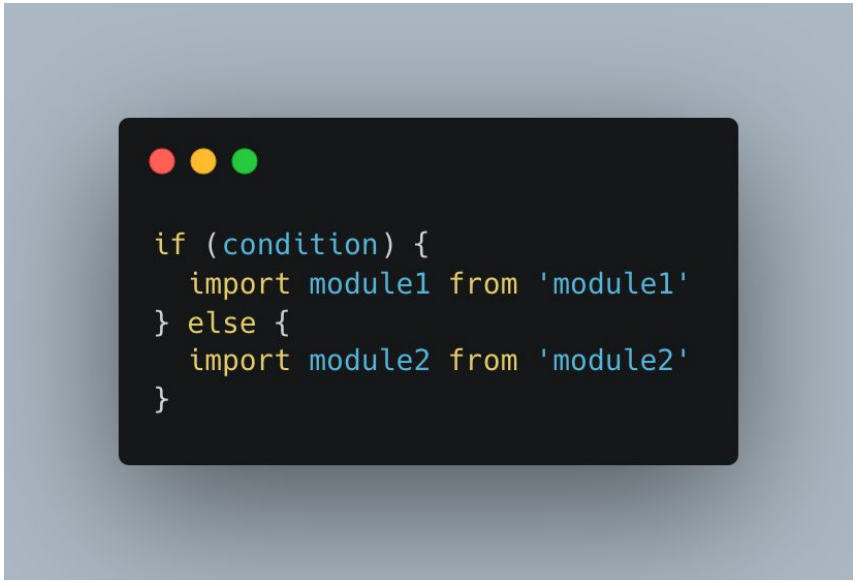
- Les modules ECMAScript (également appelés modules ES ou ESM) ont été introduits dans le cadre de la spécification ECMAScript 2015 dans le but de donner à JavaScript un système de modules officiel adapté à différents environnements d'exécution. La spécification ESM essaie de conserver quelques bonnes idées des systèmes de modules existants précédents comme CommonJS. La syntaxe est très simple et compacte. Il est possible de charger des modules de manière asynchrone.
- Le différenciateur le plus important entre ESM et CommonJS est que les modules ES sont statiques, ce qui signifie que les importations sont décrites au niveau supérieur de chaque module et en dehors de toute instruction de flux de contrôle.

CommonJS & ESM

Caractéristiques	CommonJS	ES Modules (ESM)
Système d'import/export	Utilise `require()` pour importer et `module.exports` pour exporter	Utilise `import` pour importer et `export` pour exporter
Fichier de configuration	Pas besoin de configuration spéciale	Nécessite `"type": "module"` dans `package.json`
Syntaxe d'import	`const express = require('express');`	`import express from 'express';`
Compatibilité	Utilisé par défaut dans Node.js jusqu'à la version 12	Nouveau standard, introduit dans ECMAScript 2015 (ES6)
Fonctionnement synchrone/asynchrone	Chargement synchrone des modules	Chargement asynchrone des modules
Interopérabilité	Peut importer des modules ES Modules avec des configurations supplémentaires (via `import()`)	Peut importer des modules CommonJS facilement, mais l'inverse nécessite des ajustements
Utilisation typique	Plus courant dans les projets Node.js anciens ou existants	Recommandé pour les projets modernes ou front-end
Avantages	- Simple et largement utilisé - Bien supporté dans l'écosystème Node.js	- Syntaxe standardisée et moderne - Support natif dans les navigateurs
Inconvénients	- Syntaxe non standard en dehors de Node.js	- Besoin de spécifier `"type": "module"` et de faire attention à la compatibilité avec les modules CommonJS

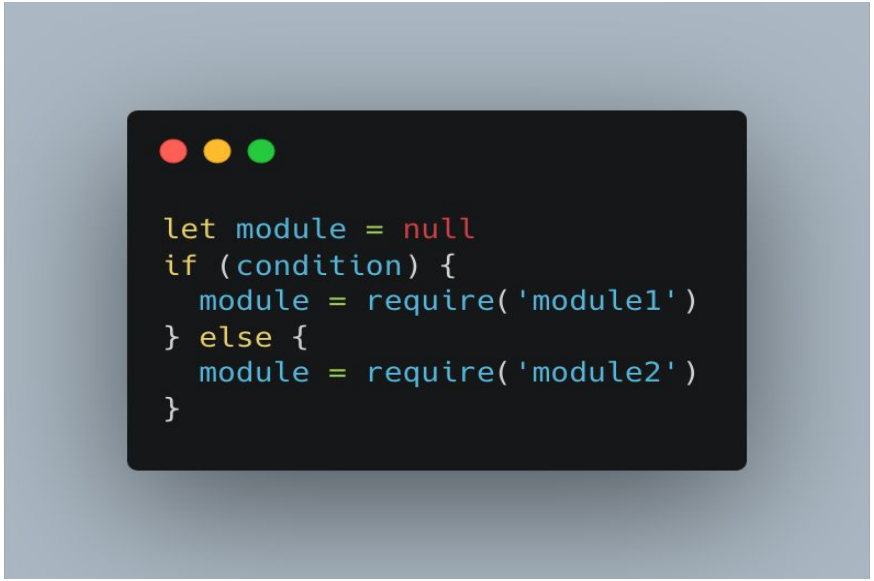
CommonJS & ESM

Par exemple, le code suivant ne serait pas valide lors de l'utilisation de modules ES :



```
if (condition) {
  import module1 from 'module1'
} else {
  import module2 from 'module2'
}
```

Alors que dans CommonJS, il est parfaitement bien d'écrire quelque chose comme ceci :



```
let module = null
if (condition) {
  module = require('module1')
} else {
  module = require('module2')
}
```

CommonJS

package.json

```
{
  "name": "express-hello-world",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

server.js

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

app.get('/', (req, res) => {
  res.send('Hello, World!');
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

ESM

package.json

```
{
  "name": "express-hello-world",
  "version": "1.0.0",
  "type": "module",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

server.js

```
import express from 'express';
const app = express();

app.get('/', (req, res) => {
  res.send('Hello, World!');
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Introduction à Express

Express.js : Pourquoi c'est le framework à apprendre

- **Express.js** accélère le développement et offre une structure stable pour construire des applications.
- Comme Node.js, **Express.js** est open-source et soutenu par une large communauté.
- Depuis sa sortie en 2010, **Express.js** est le framework le plus utilisé.

Autres frameworks Node.js

Framework	Description
Koa.js	Créé par les développeurs d'Express.js, se concentre sur une bibliothèque de méthodes non offertes par Express. (koa.js)
Hapi.js	Similaire à Express.js mais se concentre sur la réduction du code à écrire. (hapi.js)
Sails.js	Basé sur Express.js, offre plus de structure et une bibliothèque plus large. (sails.js)
Total.js	Axé sur la gestion de requêtes haute performance et de réponses rapides. (total.js)

Une première application Express

Ajout du module Express à votre application :

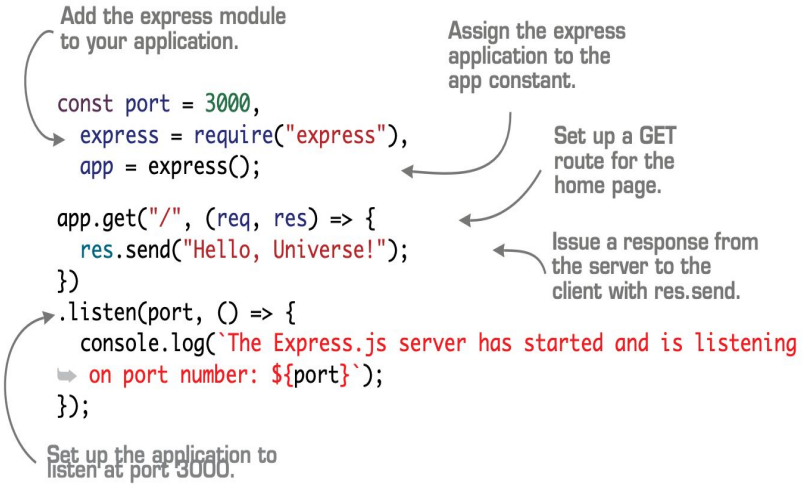
- Le module **express** est importé dans l'application à l'aide de la commande `require("express")`.
- Une instance d'Express est assignée à la constante `app`.

Définir une route GET pour la page d'accueil :

- La méthode `app.get()` crée une route pour la requête GET vers l'URL racine ("/").
- Lorsque cette route est appelée, le serveur répond avec le message `"Hello, Universe!"`, grâce à `res.send()`.

Démarrage du serveur sur le port 3000 :

- La méthode `app.listen()` permet au serveur de démarrer et d'écouter sur le port 3000.
- Une fois que le serveur est en marche, il affiche dans la console un message indiquant que le serveur a démarré, suivi du numéro de port (`3000`).



Installation et Utilisation de Nodemon

Pourquoi utiliser Nodemon ?

- Pour voir les modifications du code serveur de votre application, il faut généralement redémarrer le serveur manuellement.
- Cela devient fastidieux à mesure que vous faites plus de changements.
- **Nodemon** simplifie ce processus en redémarrant automatiquement votre serveur dès qu'un fichier est modifié.

Installer Nodemon en tant que dépendance de développement (devDependency) :

- Commande : `npm i nodemon --save-dev` ou `npm i nodemon -D`
- Avantage : Chaque projet a sa propre version de Nodemon, adaptée à la version la plus récente lors du développement.

Installer Nodemon globalement :

- Commande : `npm i nodemon -g`
- Vous devrez peut-être utiliser `sudo` ou avoir les droits d'administrateur.

Une fois Nodemon installé, il suffit de modifier votre **package.json** :

```
"scripts": {
  "start": "nodemon main.js"
}
```

Cela permet de lancer votre application avec la commande `npm start`.

Les méthodes de l'objet request dans Express.js

```
console.log(req.params);
console.log(req.body);
console.log(req.url);
console.log(req.query);
```

Access request parameters.

Les données de l'objet request :

Objet de la requête	Description
params	Permet d'extraire des ID ou tokens depuis l'URL. Très utile pour les routes RESTful.
body	Contient le contenu de la requête, notamment les données d'un formulaire soumis via POST.
url	Fournit des informations sur l'URL visitée.
query	Permet d'extraire des données soumises en tant que chaîne de requête dans l'URL.

Express.js offre des méthodes pour extraire et enregistrer les données provenant de la requête HTTP, facilitant l'accès aux informations de la requête dans votre application.

Création de routes avec Express.js

Définition d'une route POST avec Express.js

- En Express.js, une route est un point d'accès qui prend une méthode HTTP et un chemin (URL).
- La route est définie avec l'objet `app`, suivie de la méthode HTTP et du chemin.

La méthode POST prend une requête à l'URL `/contact` et renvoie un message de confirmation.

```
app.post("/contact", (req, res) => {
  res.send("Contact information submitted successfully.");
});
```

Handle requests with the Express.js post method.

Utilisation des paramètres de route

- Les paramètres de route dans Express.js permettent de capturer des données dans l'URL.
- Un paramètre de route est précédé de deux points `:` et peut être accédé via `req.params`.

```
app.get("/items/:vegetable", (req, res) => {
  res.send(req.params.vegetable);
});
```

Respond with path parameters.

- Cette route prend un paramètre `vegetable` dans l'URL et le renvoie dans la réponse.
- Exemple : Une requête à `/items/lettuce` renverra "lettuce".

Tester les APIs POST avec Postman ou Insomnia

Pourquoi tester vos APIs ?

- Lors du développement d'applications utilisant des routes **POST**, il est essentiel de s'assurer que les données envoyées par le client sont correctement reçues et traitées par le serveur.
- Des outils comme **Postman** et **Insomnia** permettent de tester facilement les requêtes API sans avoir besoin de construire un client complet.

Installer l'outil :

- **Postman** : Téléchargez depuis https://www.postman.com/downloads/.
- **Insomnia** : Téléchargez depuis https://insomnia.rest/download.

POSTMAN

Insomnia

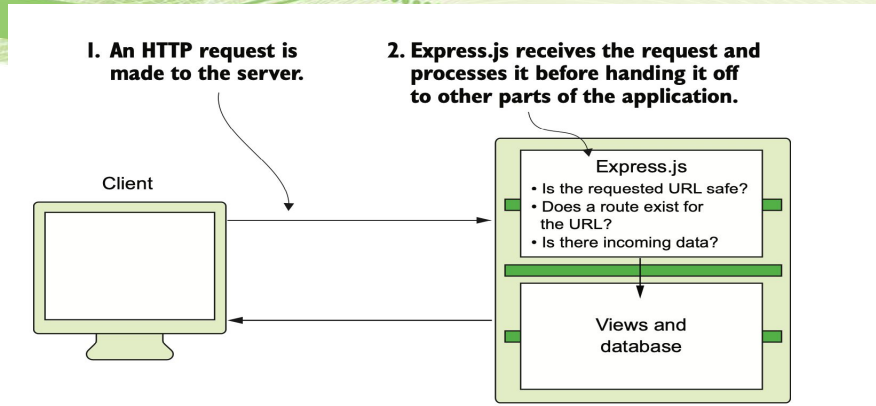
Comprendre le fonctionnement d'un framework web

Rôle d'un framework web

- Un **framework web** facilite le développement en gérant les tâches répétitives pour vous.
- **Express.js** offre une structure intuitive pour personnaliser vos applications, notamment en écoutant des requêtes HTTP à des URL spécifiques et en répondant via des fonctions de rappel.

Middleware dans Express.js

- Un **middleware** est un code qui s'interpose entre l'interaction HTTP et votre application Node.js. Il aide à écouter, analyser, filtrer et traiter les requêtes avant que les données n'interagissent avec la logique de l'application.
- Cela fonctionne comme un **bureau de poste** : avant que votre colis ne soit expédié, un employé vérifie la taille, la conformité et la sécurité du paquet.



Exemple de fonctionnement du middleware avec Express.js :

1. Une requête HTTP est envoyée par le client au serveur.
2. **Express.js** traite la requête avant de la passer aux autres parties de l'application :
   - Vérifie si l'URL demandée est sûre.
   - Vérifie si une route correspond à l'URL.
   - Analyse les données entrantes.

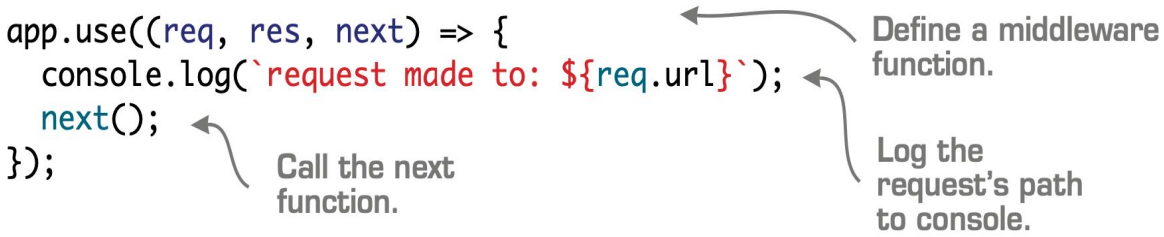
Utilisation d'un Middleware personnalisé dans Express.js

Pourquoi utiliser un middleware personnalisé ?

- **Express.js** permet d'ajouter une couche entre la réception d'une requête et son traitement.
- Vous pouvez utiliser des middlewares pour enregistrer le chemin des requêtes, gérer des erreurs, ou encore appliquer des règles de sécurité.

Fonctionnement d'un middleware personnalisé

- Le middleware se définit avec la méthode `app.use()`.
- Un middleware personnalisé prend trois arguments : **req**, **res**, et **next**.
- Le `next()` est essentiel pour continuer la chaîne de traitement des requêtes.

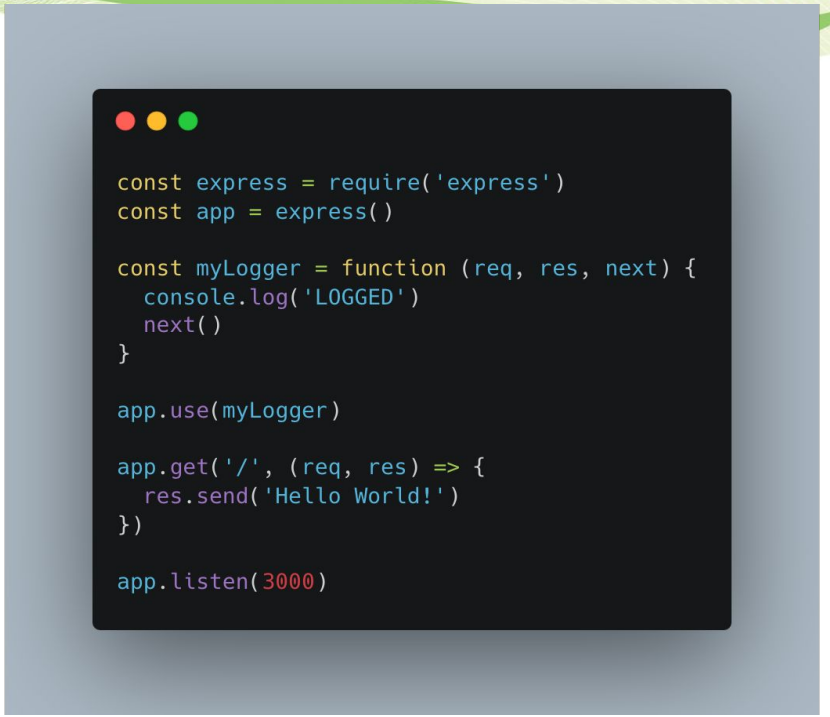


Importance du `next()`

- Il est important d'appeler `next()` pour indiquer à Express que le middleware a fini son traitement et que la requête peut continuer son chemin vers le prochain middleware ou route.
- Sans `next()`, la requête serait bloquée et le serveur ne répondrait pas.

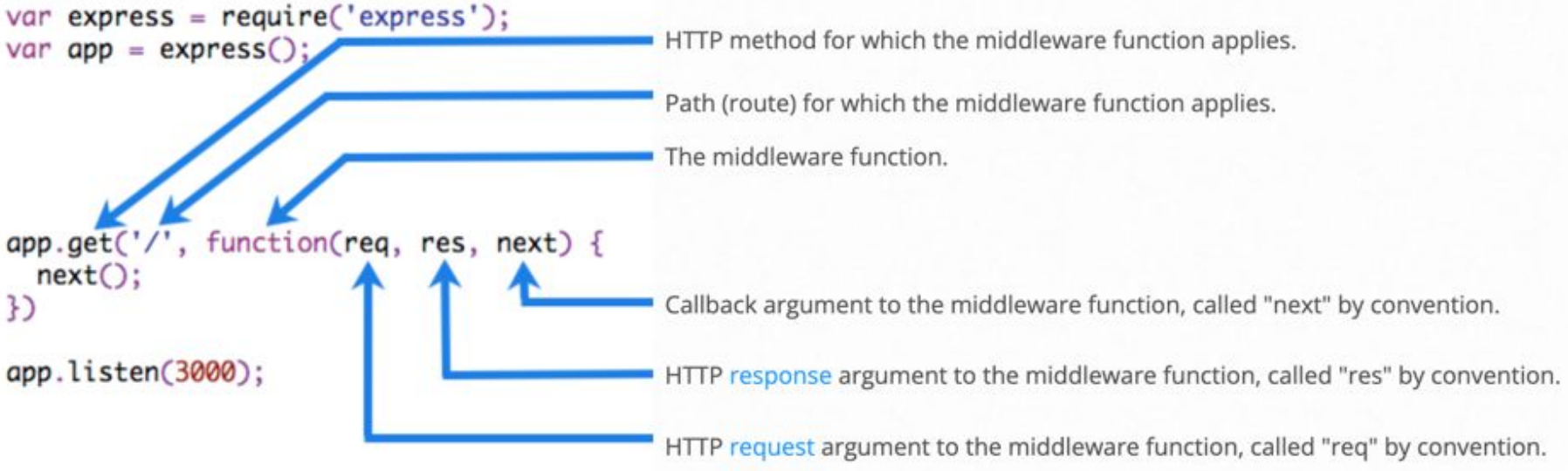
Middlewares

- La fonction **app.use** monte les fonctions middleware. Ceci est un élément important du jargon Express.
- Les fonctions du middleware sont impliquées dans le traitement des requêtes et l'envoi des résultats aux clients HTTP.
- Ils ont accès aux objets de requête et de réponse et sont censés traiter leurs données et éventuellement ajouter des données à ces objets.



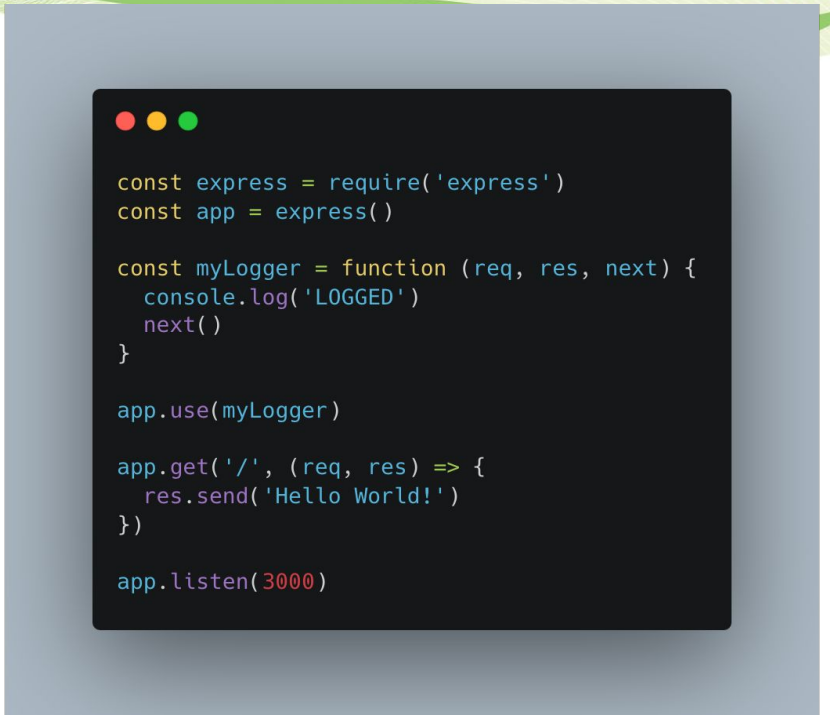
```
const express = require('express')
const app = express()

const myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}

app.use(myLogger)

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(3000)
```

Middlewares



Middlewares

- Les fonctions middleware prennent trois arguments. Les deux premiers (request et response) sont équivalents aux objets de requête et de réponse de l'objet HTTP request Node.js.
- Le dernier argument, **next**, est une fonction de rappel qui contrôle la fin du cycle requête-réponse, et il peut être utilisé pour envoyer des erreurs dans le pipeline middleware.



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

const myLogger = function (req, res, next) {
  console.log('LOGGED')
  next()
}

app.use(myLogger)

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(3000)
```

Les 5 types de Middleware dans Express.js

1. Middleware d'application

Ce type de middleware est relié directement à l'instance d'Express et s'exécute à chaque requête.

- Utilisé pour des tâches comme le **logging** ou le traitement commun à chaque requête.

```
const port = 3000,
express = require("express"),
app = express();

app.use((req, res, next) => {
    console.log(`Requête reçue à ${new Date()}`);
    next();
  });

// Créer une route pour la page d'accueil
app.get("/", (req, res) => {
  res.send("Hello, Universe!");
});

// Démarrer l'application Express.js sur le port 3000
app.listen(port, () => {
  console.log(`Le serveur Express.js a démarré et écoute sur le port : ${port}`);
});
```

Les 5 types de Middleware dans Express.js

2. Middleware du routeur

- Branché sur une instance de **router** plutôt que sur l'application globale. Utilisé pour organiser les routes dans des sous-groupes.
- Idéal pour gérer des **routes complexes** ou hiérarchiques.

```
// Créer un routeur
const router = express.Router();

// Middleware pour toutes les requêtes passant par le routeur
router.use((req, res, next) => {
  console.log('Middleware du routeur');
  next();
});

// Route GET pour l'URL /api/hello
router.get('/hello', (req, res) => {
  res.send('Hello World depuis /api/hello');
});

// Appliquer le routeur à la route de base /api
app.use('/api', router);

// Démarrer le serveur
const port = 3000;
app.listen(port, () => {
  console.log(`Le serveur écoute sur http://localhost:${port}`);
});
```

Les 5 types de Middleware dans Express.js

3. Middleware de traitement d'erreurs

- Ce middleware prend **4 arguments** et est utilisé pour gérer les erreurs dans une application Express.
- Sert à capturer et gérer les **erreurs** dans l'application.

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

// Route simple pour Hello World
app.get('/', (req, res) => {
  res.send('Hello World!');
});

// Middleware de capture des routes 404 (Non trouvées)
app.use('*',(req, res, next) => {
  const error = new Error('Route non trouvée');
  error.status = 404;
  next(error);  // Envoie l'erreur au middleware de gestion d'erreurs
});

// Middleware de gestion des erreurs (y compris les 404)
app.use((err, req, res, next) => {
  console.error(err.stack);  // Log de l'erreur dans la console
  res.status(err.status || 500).send(err.message || 'Erreur serveur!');  // Envoi de la réponse d'erreur
});

// Démarrer le serveur
const port = 3000;
app.listen(port, () => {
  console.log(`Le serveur écoute sur http://localhost:${port}`);
});
```

Les 5 types de Middleware dans Express.js

4. Middleware intégré

Express a un middleware intégré appelé **`express.static`**. Il sert des fichiers statiques comme des images ou des PDF.

app.use(express.static('public'));

Utilisé pour gérer les **documents statiques** de votre application.

```
const express = require('express');
const path = require('path');  // Pour gérer les chemins de fichiers
const app = express();

// Middleware pour servir des fichiers statiques (par exemple, des images, CSS, HTML) dans le dossier
"public"
app.use(express.static(path.join(__dirname, 'public')));

// Middleware pour analyser les données JSON envoyées dans le corps des requêtes POST
app.use(express.json());

// Route GET simple pour Hello World
app.get('/', (req, res) => {
  res.send('Hello World!');
});

// Route POST pour illustrer express.json() - elle reçoit des données JSON
app.post('/data', (req, res) => {
  console.log(req.body);  // Affiche les données envoyées en JSON
  res.send(`Données reçues : ${JSON.stringify(req.body)}`);
});

// Démarrer le serveur
const port = 3000;
app.listen(port, () => {
  console.log(`Le serveur écoute sur http://localhost:${port}`);
});
```

Les 5 types de Middleware dans Express.js

5. Middleware externe

Les middleware externes sont des modules JavaScript installables via `npm`, par exemple pour la gestion des **cookies** ou des **sessions**.

- Permet d'utiliser des **dépendances externes** comme `cookie-parser`, `body-parser`, etc.

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();

// Middleware body-parser pour traiter les données JSON et URL-encoded
app.use(bodyParser.json());  // Analyser les corps de requêtes au format JSON
app.use(bodyParser.urlencoded({ extended: true }));  // Analyser les données envoyées par formulaire
(URL-encoded)

// Route GET simple pour Hello World
app.get('/', (req, res) => {
  res.send('Hello World!');
});

// Route POST pour illustrer body-parser - reçoit et affiche des données JSON ou URL-encoded
app.post('/data', (req, res) => {
  console.log(req.body);  // Affiche les données envoyées dans le corps de la requête
  res.send(`Données reçues : ${JSON.stringify(req.body)}`);
});

// Démarrer le serveur
const port = 3000;
app.listen(port, () => {
  console.log(`Le serveur écoute sur http://localhost:${port}`);
});
```

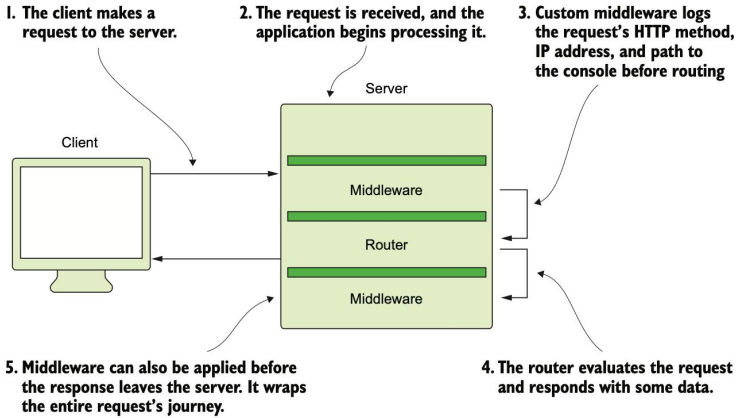
Analyser les données des requêtes dans Express.js

Capturer les données d'une requête

- Il est essentiel de savoir capturer les données des utilisateurs via les requêtes HTTP.
- **Deux manières principales** d'obtenir des données d'une requête utilisateur :
  1. Le corps de la requête avec une requête **POST**.
  2. La chaîne de requête (query string) dans l'URL.

Fonctionnement d'une requête et du middleware :

1. **Le client envoie une requête** au serveur.
2. **Le serveur** reçoit la requête et commence à la traiter.
3. Un **middleware personnalisé** journalise des informations comme la méthode HTTP, l'adresse IP, et le chemin de la requête dans la console avant que la requête ne soit routée.
4. **Le routeur** évalue la requête et répond avec des données.
5. **Un middleware peut également être appliqué** avant que la réponse ne quitte le serveur, en enveloppant l'ensemble du processus de requête.



Capturer les données postées depuis le corps de la requête dans Express.js

Configurer le parsing des données dans Express.js :

1. Utilisez **`express.urlencoded()`** pour analyser les données encodées en URL (comme celles soumises par un formulaire HTML).
2. Utilisez **`express.json()`** pour analyser les données envoyées en JSON dans le corps d'une requête.

```
app.use(
  express.urlencoded({
    extended: false
  })
);
app.use(express.json());

app.post("/", (req, res) => {
  console.log(req.body);
  console.log(req.query);
  res.send("POST Successful!");
});
```

Tell your Express.js application to parse URL-encoded data.

Create a new post route for the home page.

Log the request's body.

Comment tester :

1. **Envoyer une requête POST** à `http://localhost:3000/` avec des données encodées en URL (par exemple via un formulaire ou curl) :

```
curl --data "first_name=Joseph&last_name=Azar"
http://localhost:3000
```

Tester les chaînes de requête (query strings) :

- Visitez l'URL suivante dans votre navigateur :
  `http://localhost:3000/?cart=3&pagesVisited=4&utmcode=837623`

Router - Controller - Service - Model/DAO