

Node.js

Node.js

Introduction NodeJs

Installer environnement Developpement.

Node.js (“express”) (routing)

Node.js (“express”) (middleware)

Node.js (“express”) (router)

Introduction NodeJs

NodeJs, C'est quoi?



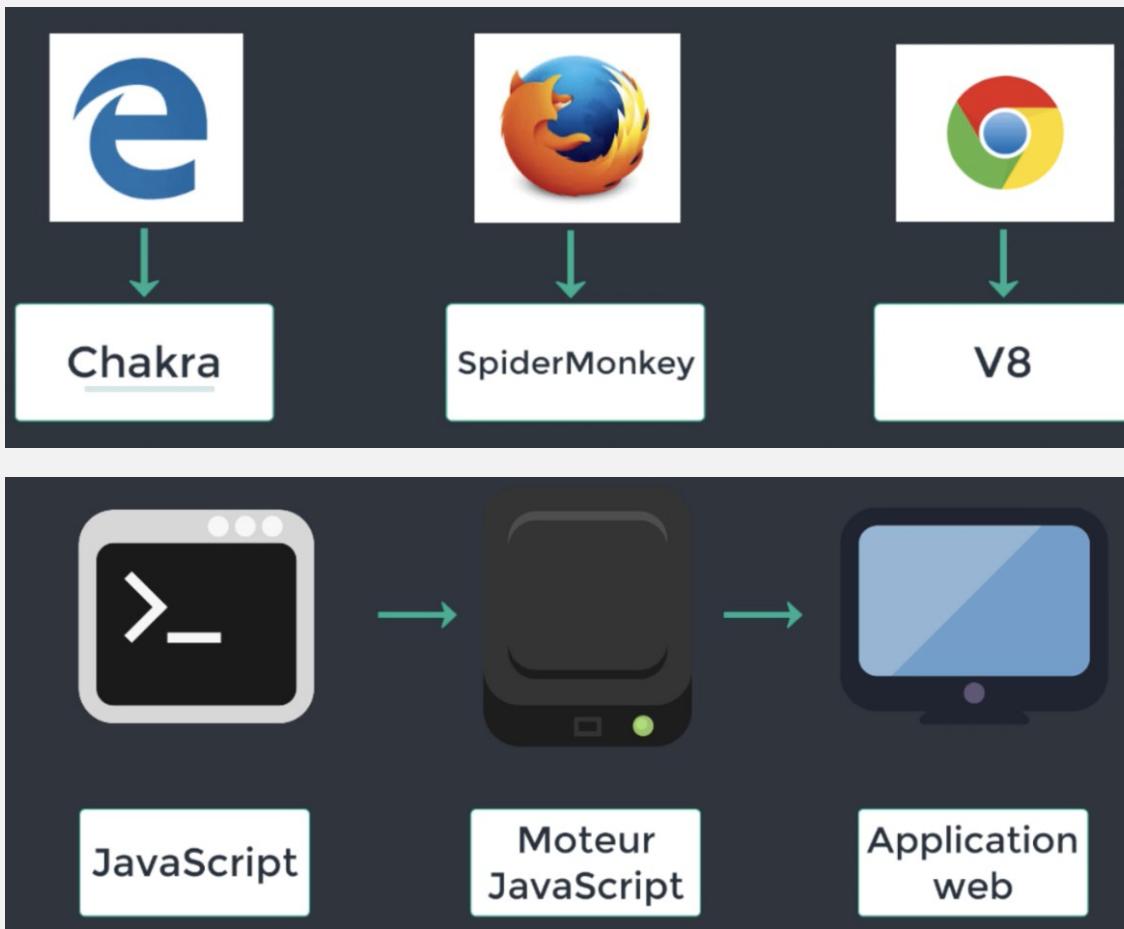
Node.js est un environnement bas niveau permettant l'exécution de JavaScript côté serveur.

Node.js est un outil JavaScript open-source basé sur le moteur JavaScript de Google Chrome.

Node.js a été créé par Ryan Dahl en 2009. Son développement et sa maintenance sont effectués par l'entreprise Joyent.

Node.js est utilisé pour créer des applications réseaux évolutives avec une architecture d'E/S non bloquante, rapide et efficace, pilotée par les événements.

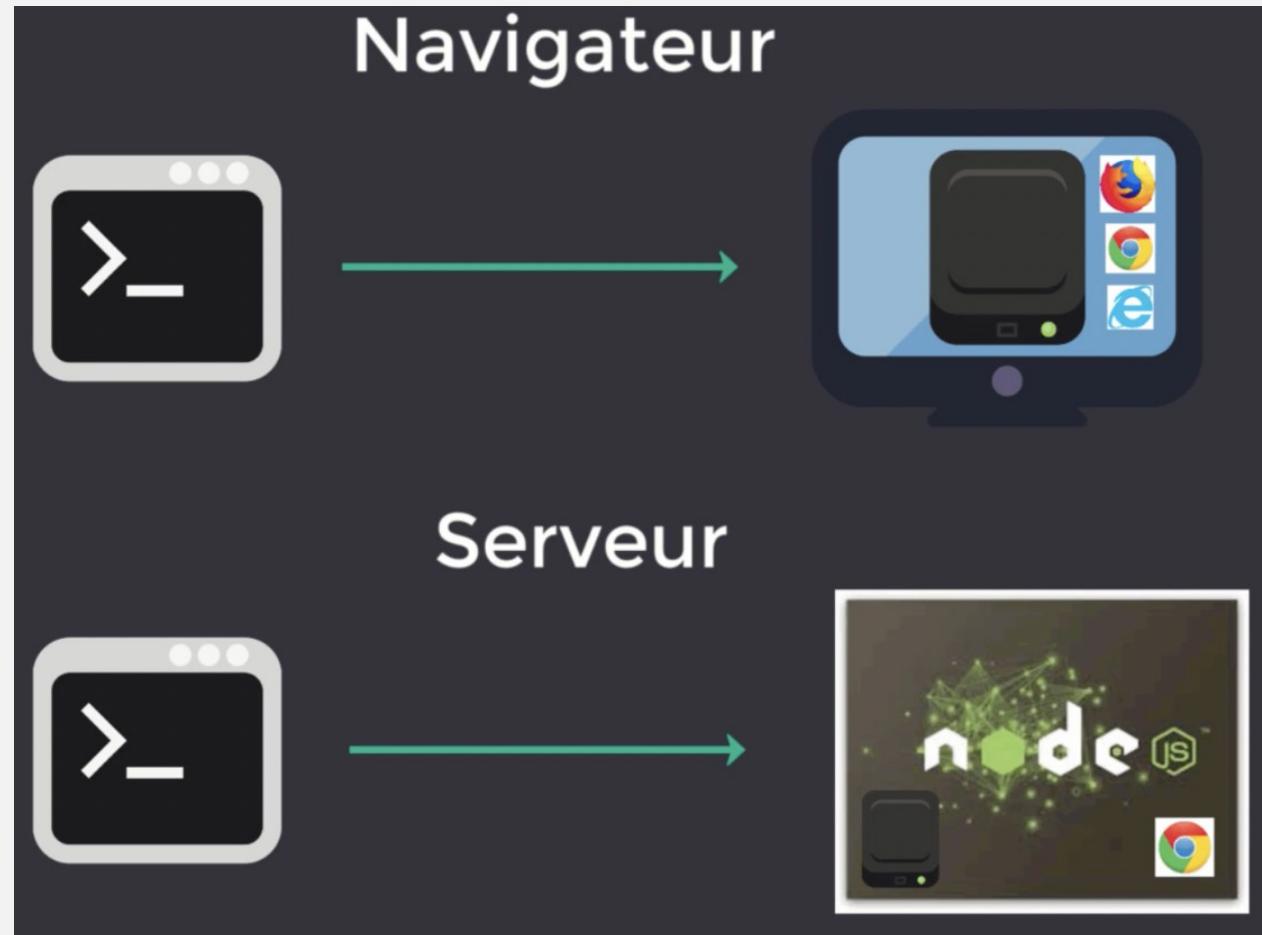
Navigateur vs Serveur



Environnement Web
=

Navigateur + Moteur
JavaScript

Navigateur vs Serveur



NodeJs vs Javascript



Définition	C'est un langage de programmation. Vous tapez du code, qui sera ensuite interprété par un moteur JavaScript, et vous obtenez une application utilisable par des utilisateurs.	C'est un environnement pour le code JavaScript, qui fonctionne en dehors du navigateur.
------------	---	---



Rôle	Utiliser pour n'importe quelle action à implémenter dans une application web : navigation, récupération de données, affichage d'informations, etc.	Utiliser pour effectuer des opérations sur n'importe quel système d'exploitation.
------	--	---

NodeJs vs Javascript



Moteur
JavaScript

Utilise le moteur JavaScript du navigateur pour être interprété.

Utilise le moteur JavaScript V8
du navigateur Chrome.

Les avantages de NodeJs

Avantage n°1
Node.js utilise le langage JavaScript

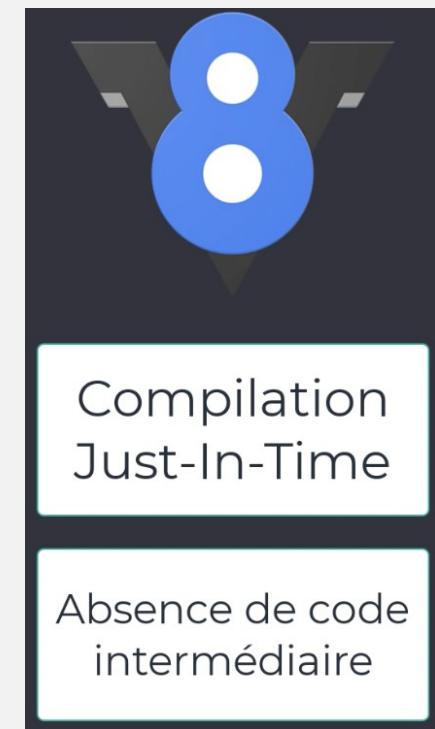
Avantage n°2
Node.js est très rapide

Avantage n°3
Node.js est flexible

Avantage n°4
Node.js est populaire

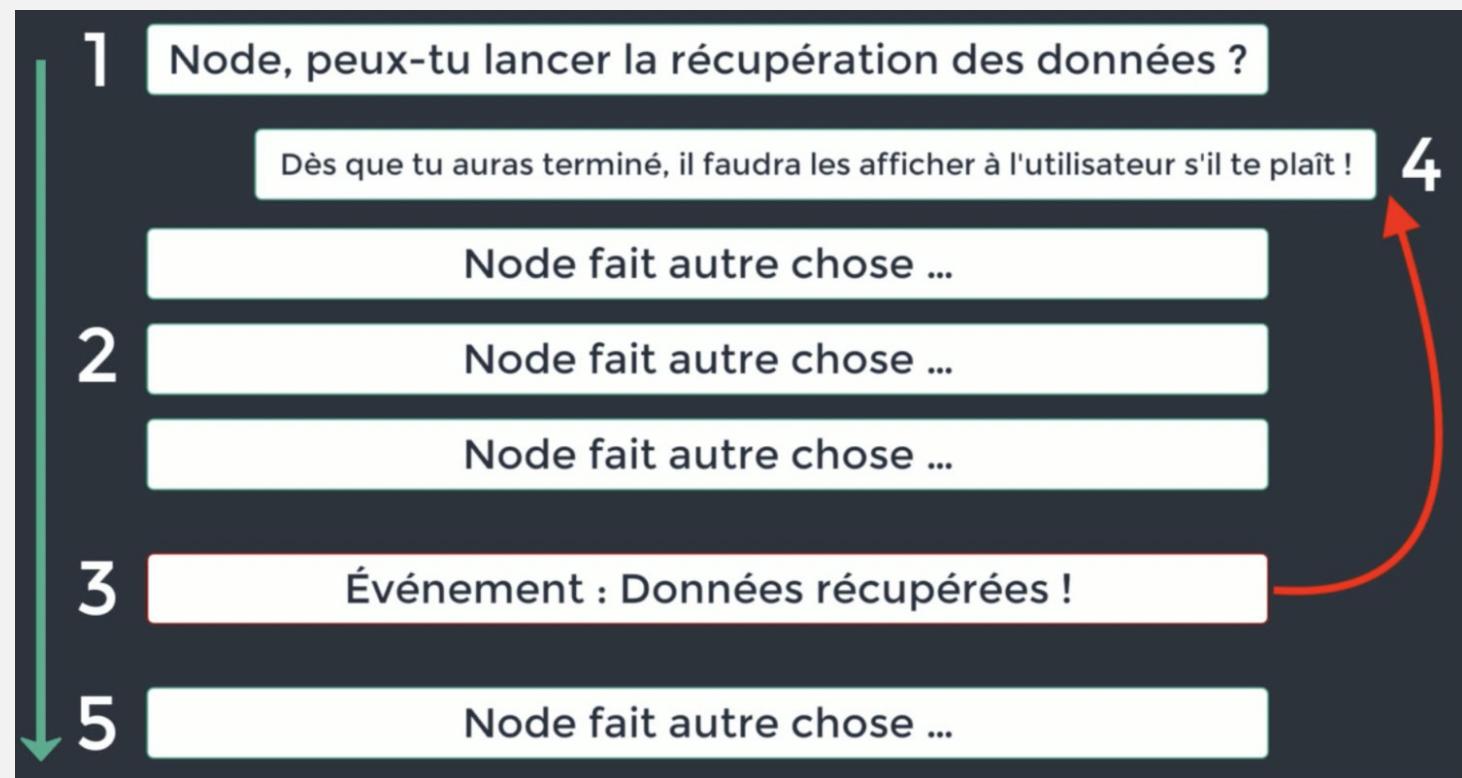
Les principes de NodeJs

Le Moteur
JavaScript V8



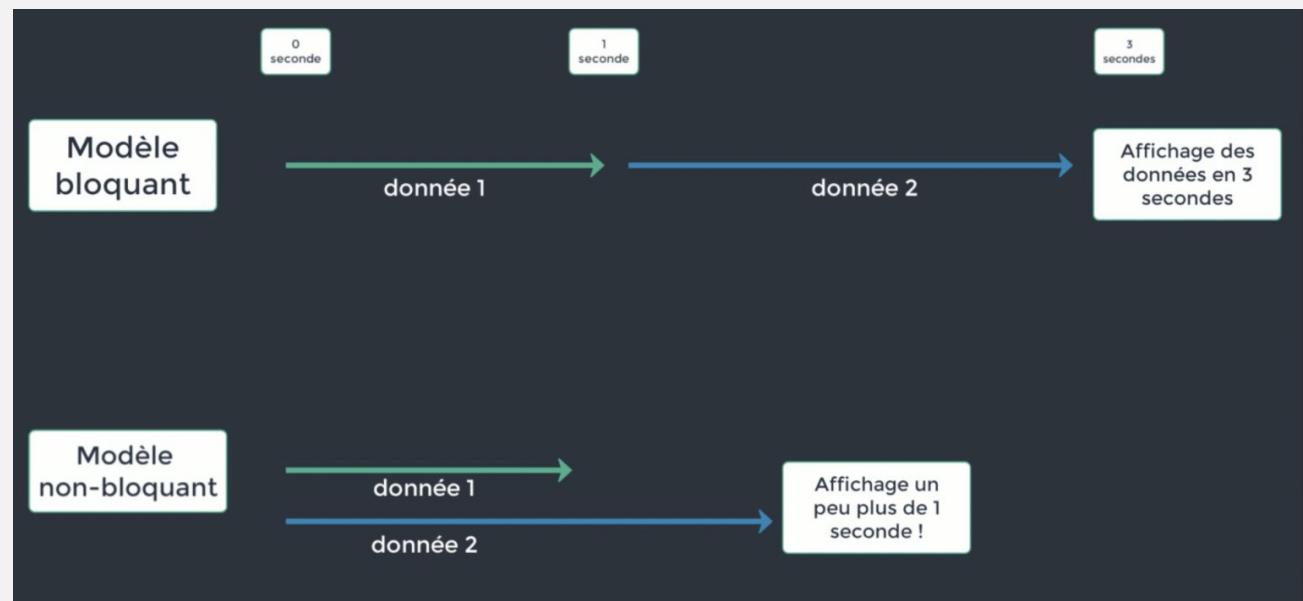
Les principes de NodeJs

Le Modèle Non-Bloquant

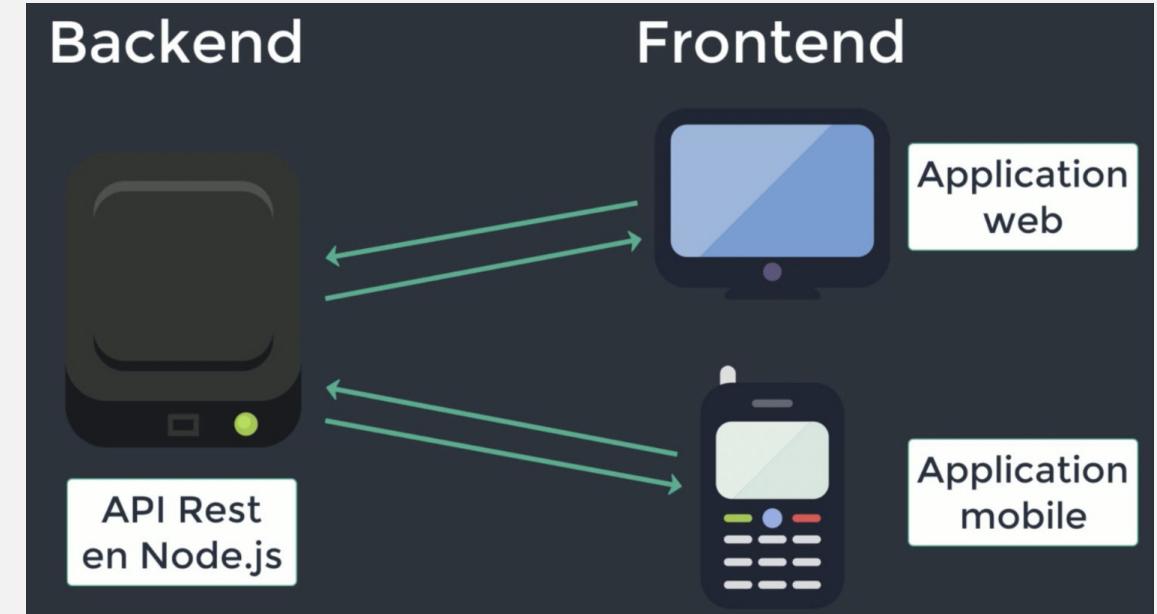
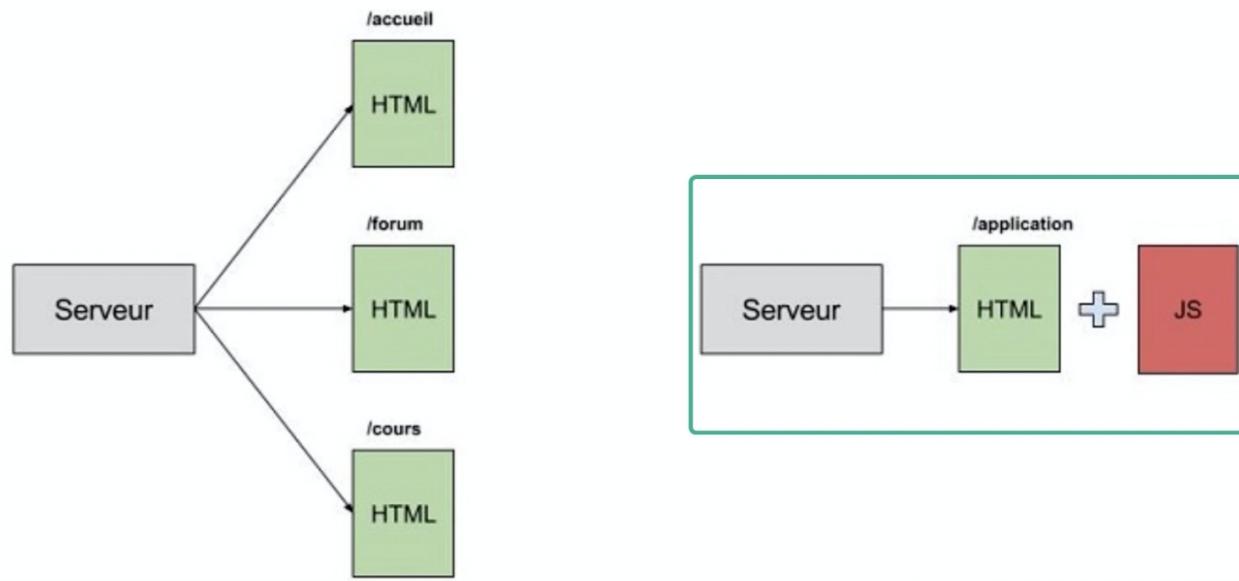


Les principes de NodeJs

Le Modèle
Non-Bloquant



Les Types d'applications avec NodeJs



Installer l'environnement NodeJS

Installer l'environnement NodeJS

```
node -v  
npm -v  
npm init  
npm install express  
npm install nodemon --save-dev  
npm run start
```

NodeJs sans Framework

Node.js – fs (Async Read)

```
var fs = require('fs');
var in_data;

fs.readFile('./fn_input.txt', function (err, data) {
    if (err) return console.error(err);
    in_data = data;
    console.log('Async input file content: ' + in_data);
});

console.log('Program Ended.');
```

Node.js – fs (Async Write)

```
var fs = require('fs');
var out_data = 'Output line 1.\r\nOutput line 2.\r\nOutput last line.';

fs.writeFile('./async_output.txt', out_data, function (err) {
    if (err) console.error(err);
    console.log('Async output file content: ' + out_data);
});

console.log('Program Ended.');
```

Node.js – fs (Sync Read)

```
● ● ●

var fs = require('fs');

var in_data = fs.readFileSync('./fn_input.txt');

console.log('Sync input file content: ' + in_data);

console.log('Program Ended.');
```

Node.js – fs (Sync Write)

```
var fs = require('fs');
var out_data = 'Output line 1.\r\nOutput line 2.\r\nOutput last line.';

fs.writeFileSync('./sync_output.txt', out_data);
console.log('Sync output file content: ' + out_data);

console.log('Program Ended.');
```

Node.js – net (Network Server)

```
var net = require("net");
var server = net.createServer(function(connection) {
  console.log('Client connected.');
  connection.on('end', function() {
    console.log('Client disconnected.');
  });
  connection.write('Hello World!\n');
  connection.pipe(connection);
// renvoie les données à l'objet de connexion qui est le client
});
server.listen(8080, function() {
  console.log('Server is listening.');
});
console.log('Server Program Ended.');
```

Node.js – net (Network Client)

```
var client = net.connect({port: 8080}, 'localhost', function() {
    console.log('Connected to Server.');
});

client.on('data', function (data) {
    console.log(data.toString());
    client.end();
});

client.on('end', function () {
    console.log('Disconnected from server.');
});
console.log('Client Program Ended.');
```

Node.js (Modules)

Module name	Description
buffer	Buffer module can be used to create Buffer class.
console	Le module console est utilisé pour imprimer des informations sur stdout et stderr.
dns	Le module dns est utilisé pour effectuer la recherche DNS réelle et les fonctionnalités de résolution de noms o/s sous-jacentes.
domain	Le module de domaine permet de gérer plusieurs opérations d'E/S différentes en tant que groupe unique.
fs	Le module fs est utilisé pour les E/S de fichiers.
net	net module fournit des serveurs et des clients sous forme de flux. Agit comme un wrapper réseau.
os	Os Module fournit des fonctions utilitaires de base liées aux o/s.
path	Le module path fournit des utilitaires pour la gestion et la transformation des chemins d'accès aux fichiers.
process	Le module de processus est utilisé pour obtenir des informations sur le processus en cours.

Node.js – web module

Le **server Web** est une application logicielle qui traite les demandes à l'aide du protocole HTTP et renvoie des pages Web en réponse aux clients.

Le **server Web** fournit généralement des documents HTML avec des images, des feuilles de style et des scripts.

La plupart des **servers Web** prennent également en charge les scripts côté serveur utilisant un langage de script ou redirigent vers le serveur d'applications qui effectuent la tâche spécifique d'obtenir des données à partir de la base de données, d'effectuer une logique complexe, etc.

Le **server Web** renvoie ensuite la sortie du serveur d'applications au client.

Le **server Web** Apache est l'un des serveurs Web les plus couramment utilisés. C'est un projet open source.

Web server et Path

Le serveur Web mappe le chemin d'accès d'un fichier à l'aide de l'URL, Uniform Resource Locator. Il peut s'agir d'un système de fichiers local ou d'un programme externe/interne.

Un client effectue une demande à l'aide du navigateur, URL : <http://www.abc.com/dir/index.html>

Le navigateur fera la demande comme suit:

GET /dir/index.html HTTP/1.1

HOST www.abc.com

Web Architecture

Les applications Web sont normalement divisées en quatre couches :

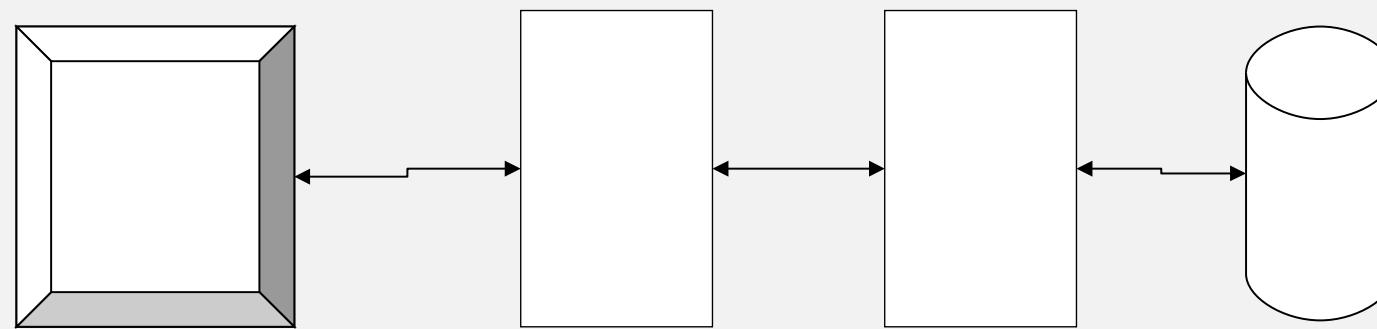
Client: cette couche se compose de navigateurs Web, de navigateurs mobiles ou d'applications qui peuvent faire une requête HTTP au serveur.

Serveur: cette couche est constituée d'un serveur Web qui peut intercepter la demande faite par les clients et leur transmettre la réponse.

Business: cette couche se compose d'un serveur d'applications qui est utilisé par le serveur Web pour effectuer des tâches dynamiques. Cette couche interagit avec la couche de données via une base de données ou certains programmes externes.

Données : cette couche est constituée de bases de données ou de toute source de données.

Web Architecture



Client

**Web
server**

**Application
server**

Database

Création d'un serveur Web avec de Node (module http)

Créez un serveur HTTP à l'aide de la méthode `http.createServer`. Passez-lui une fonction avec des paramètres de demande et de réponse.

Écrivez l'exemple d'implémentation pour renvoyer une page demandée.

Passez un port 8081 pour écouter la méthode.

`http_server.js`
`test.html`
`http_client.js`

http_server.js

```
var http = require('http');
var fs = require('fs');
var url = require('url');
http.createServer(function (request, response) {
  var pathname = url.parse(request.url).pathname;
  console.log('Request for ' + pathname + ' received.');
  fs.readFile(pathname.substr(1), function (err, data) {
    if (err) { console.log(err.stack);
               response.writeHead(404, {'Content-Type' : 'text/html'});
               // HTTP status: 404 : NOT FOUND
             } else { response.writeHead(200, {'Content-Type' : 'text/html'});
               // HTTP status: 200 : OK
               response.write(data.toString());
             }
    response.end();           // send the response body
  });
}).listen(8081);
console.log('Server running at http://127.0.0.1:8081/test.html');
console.log('Server Program Ended.');
```

test.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>wk4_01_test</title>
  </head>
  <body>
    Hello World!
  </body>
</html>
```

http_client.js

```
var http = require('http');
var options = {
    host: 'localhost', port: '8081', path: '/test.html'
};
var callback = function (response) {
    // callback function is used to deal with response
    var body = '';
    response.on('data', function (data) {
        body += data;
    });
    response.on('end', function () {
        console.log(body);
    });
    response.on('error', (error) => {
        console.error(error);
    });
};
var req = http.request(options, callback);
req.end();
```

Framework Express

Node.js – Express

Express js est un framework d'application Web très populaire conçu pour créer des applications Web Node.js.

Il fournit un environnement intégré pour faciliter le développement rapide d'applications Web basées sur des nœuds.

Le **framework Express** est basé sur le moteur middleware Connect et utilise le framework de modèle html **Jade** pour les modèles HTML.

Principales caractéristiques du framework Express :

Permet de configurer des middlewares pour répondre aux requêtes HTTP.

Définit une table de routage qui est utilisée pour effectuer différentes actions en fonction de la méthode HTTP et de l'URL.

Permet de restituer dynamiquement des pages HTML en fonction de la transmission d'arguments aux modèles.

Installing Express

À partir de la nodejs_workspace (répertoire de travail), désinstallez express si vous installez express localement à partir du chapitre précédent.

npm uninstall express

Créez un répertoire myapp qui se trouve sous le répertoire nodejs_workspace.

(mkdir myapp, puis cd myapp). [<http://expressjs.com/en/starter/installing.html>]

Utilisez la commande npm init sous le nouveau dossier myapp pour créer un fichier package.json pour votre application – myapp.

npm init

REMARQUE: vous pouvez appuyer sur RETOUR pour accepter les valeurs par défaut pour la plupart d'entre eux, sauf que le point d'entrée est l'application.js:

entry point: app.js

Installez express localement dans le répertoire nodejs_workspace et enregistrez-le dans la liste des dépendances :

npm install express --save

Installez l'infrastructure Express globalement à l'aide de npm si vous souhaitez créer une application Web à l'aide du terminal de nœud.

npm install express -g --save

Node.js – Web Application avec express

<http://expressjs.com/en/starter/hello-world.html>

Dans le répertoire myapp, créez un fichier nommé app.js et ajoutez les codes suivants :

Exécutez l'application.js sur le serveur : **node app.js**

Ensuite, chargez <http://localhost:3000/> dans un navigateur pour voir la sortie.



The image shows a terminal window with a dark background and light-colored text. At the top, there are three small colored dots (red, yellow, green). Below them, the Node.js application code is displayed:

```
var express = require('express');
  var app = express();
  app.get('/', function (req, res) {
    res.send('Hello World!');
  });
  app.listen(3000, function () {
    console.log('app.js listening to
http://localhost:3000/');
  })
```

The code defines an Express application that listens on port 3000 and responds with "Hello World!" to requests made to the root URL. A log message is also printed to the console when the server starts.

Web application – get/post form (server.js)

```
var express = require('express');
var app = express();
app.get('/', function (req, res) { // To display index.html
    res.sendFile(__dirname + "/index.html");
});
app.get('/process_get', function (req, res) { // To process get method
    var response = { fname: req.query.fname,
        lname: req.query.lname }; // preparing the output in JSON format
    console.log(response);
    res.json(response);
})
```

Web application – get/post form (server.js)

```
● ● ●

var bodyParser = require('body-parser');          // To process post method
var urlenParser = bodyParser.urlencoded({ extended: false});    // creating the application/x-www-form-urlencoded parser
app.post('/process_post', urlenParser, function (req, res) {
  var response = { fname: req.body.fname,
    lname: req.body.lname };    // preparing the output in JSON format
  console.log(response);
  res.end(JSON.stringify(response));
});

var server = app.listen(8081, function () {
  console.log('server.js is listening at http://127.0.0.1:8081/index.html or http://localhost:8081/index.html');
});

console.log('End of program');
```

Framework Express - Routing

Basic Routing

Le routage fait référence à la détermination de la façon dont une application répond à une demande client à un point de terminaison particulier, qui est un URI (ou chemin) et une méthode de requête HTTP spécifique (GET, POST, etc.).

Chaque itinéraire peut avoir une ou plusieurs fonctions de gestionnaire, qui sont exécutées lorsque l'itinéraire est mis en correspondance.

La définition de l'itinéraire prend la structure suivante :

```
appli. MÉTHODE(CHEMIN, GESTIONNAIRE);
```

Où:

app est une instance d'express.

METHOD est une méthode de requête HTTP, en minuscules.

PATH est un chemin sur le serveur

- HANDLER est la fonction exécutée lorsque l'itinéraire est mis en correspondance.

Route methods

Une méthode route est dérivée de l'une des méthodes HTTP et est attachée à une instance de la classe express.

Le code suivant est un exemple d'itinéraires définis pour les méthodes GET et POST vers la racine de l'application.

GET method route:

```
app.get('/', function (req, res) {
  res.send('Get request to the homepage');
});
```

Route methods

POST method route:

```
app.post('/', function (req, res) {
    res.send('Post request to the homepage');
});
```

Route methods

Il existe une méthode de routage spéciale, `app.all()`, qui n'est dérivée d'aucune méthode HTTP. Cette méthode est utilisée pour charger des fonctions middleware sur un chemin pour toutes les méthodes de requête.

Dans l'exemple suivant, le gestionnaire sera exécuté pour les requêtes à « /secret », que vous utilisez ***GET, POST, PUT, DELETE*** ou toute autre méthode de requête HTTP prise en charge dans le module http :

```
app.all('/secret', function (req, res, next) {
  console.log('Accessing the secret section ...');
  next(); // pass control to the next handler
});
```

Node.js - routing (hkbu_app_route.js)

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

// This route path will match requests to the root route, /
app.get('/', function (req, res) {
    res.send('Hello World! get method');
});

// This route path will match requests to /user
app.all('/user', function (req, res) {
    console.log('Accessing /user...');
    res.send('Hello World! /user request...');
});
```

Node.js - routing (hkbu_app_route.js)

```
// This route path will match requests to /random.txt
app.get('/random.txt', function (req, res) {
    res.send('Hello World! get /random.txt');
});
// To define routes with route parameters, simply specify the route parameters in the
// path of the route as shown below.
app.get('/users/:userId/books/:bookId', function(req, res) {
    res.send(req.params);
});
app.get('/floats/:digit.:decimal', function(req, res) {
    res.send(req.params);
});
app.listen(3000, function () {
    console.log('app.js listening to http://127.0.0.1:3000/ or http://localhost:3000/');
});
```

Route paths

Les chemins d'itinéraire, en combinaison avec une méthode de requête, définissent les points de terminaison auxquels les demandes peuvent être effectuées. Les chemins d'itinéraire peuvent être des chaînes, des modèles de chaîne ou des expressions régulières.

Les caractères ?, +, * et () sont des sous-ensembles de leurs homologues d'expression régulière. Le trait d'union (-) et le point(.) sont interprétés littéralement par des chemins basés sur des chaînes.

REMARQUE : Express utilise path-to-regexp pour faire correspondre les chemins d'itinéraire ; voir la documentation path-to-regexp pour toutes les possibilités de définition des chemins d'itinéraire. Express Route Tester est un outil pratique pour tester les itinéraires Express de base, bien qu'il ne prenne pas en charge la correspondance de modèles.

Remarque : Les chaînes de requête ne font pas partie du chemin d'itinéraire.

Route paths based on strings

Ce chemin d'itinéraire fera correspondre les demandes à l'itinéraire racine, /.

```
app.get('/', function (req, res) {  
    res.send('root');});
```

Ce chemin d'itinéraire correspondra aux demandes à /about.

```
app.get('/about', function (req, res) {  
    res.send('about');});
```

Ce chemin d'itinéraire fera correspondre les demandes à /random.text.

```
app.get('/random.text', function (req, res) {  
    res.send('random.text');});
```

Route paths basés sur des modèles de chaîne

Ce chemin d'itinéraire correspondra à acd et abcd.

```
app.get('/ab?cd', function(req, res) { res.send('ab?cd'); });
```

Ce chemin d'itinéraire correspondra à abcd, abbcd, abbbcd, etc.

```
app.get('/ab+cd', function(req, res) { res.send('ab+cd'); });
```

Ce chemin d'itinéraire correspondra à abcd, abxcd, abRANDOMcd, ab123cd, etc.

```
app.get('/ab*cd', function(req, res) { res.send('ab*cd'); });
```

Ce chemin d'itinéraire correspondra à /abe et /abcde.

```
app.get('/ab(cd)?e', function(req, res) { res.send('ab(cd)?e'); });
```

Route paths based on regular expressions

Ce chemin d'itinéraire correspondra à n'importe quoi avec un « a » dans le nom de l'itinéraire.

```
app.get(/a/, function(req, res) {  
    res.send('/a/');  
});
```

Route paths based on regular expressions

Ce chemin d'itinéraire correspondra au “butterfly” et à “dragonfly”, mais pas à “butterflyman” ou “dragonfly man”, etc.

```
app.get('.*fly$', function(req, res) {  
    res.send('/.*fly$/');  
});
```

Route Parameters

Les paramètres d'itinéraire (route) sont nommés segments d'URL qui sont utilisés pour capturer les valeurs spécifiées à leur position dans l'URL. Les valeurs capturées sont renseignées dans l'objet `req.params`, avec le nom du paramètre route spécifié dans le chemin d'accès comme clés respectives.

```
Route path: /users/:userId/books/:bookId
Request URL: http://localhost:3000/users/34/books/8989  req.params:
{ "userId": "34", "bookId": "8989" }
```

Route Parameters

Pour définir des itinéraires avec des paramètres d'itinéraire, spécifiez simplement les paramètres d'itinéraire dans le chemin de l'itinéraire comme indiqué ci-dessous.

```
app.get('/users/:userId/books/:bookId', function(req, res) {  
    res.send(req.params);});
```

Route parameters (continue...)

Étant donné que le trait d'union (-) et le point (.) sont interprétés littéralement, ils peuvent être utilisés avec des paramètres d'itinéraire à des fins utiles.

Route path: /flights/:from-:to

Request URL: http://localhost:3000/flights/LAX-SFO

req.params: { "from": "LAX", "to": "SFO" }

Route path: /floats/:digit.:decimal

Request URL: http://localhost:3000/floats/123.45

req.params: { "digit": "123", "decimal": "45" }

REMARQUE : Le nom des paramètres d'itinéraire doit être composé de « caractères de mot » ([A-Za-z0-9_]).

Route handlers

Vous pouvez fournir plusieurs fonctions de rappel qui se comportent comme un middleware pour gérer une demande. La seule exception est que ces rappels peuvent appeler `next('route')` pour contourner les rappels de route restants.

Vous pouvez utiliser ce mécanisme pour imposer des conditions préalables à un itinéraire, puis passer le contrôle aux itinéraires suivants s'il n'y a aucune raison de poursuivre l'itinéraire actuel. Les gestionnaires d'itinéraires peuvent prendre la forme d'une fonction, d'un tableau de fonctions ou de combinaisons des deux.

Route handlers samples

Une seule fonction de rappel peut gérer un itinéraire. Par exemple:

```
app.get('/example/a', function (req, res) {  
    res.send('Hello from A!');});
```

Route handlers samples

Plusieurs fonctions de rappel peuvent gérer un itinéraire (assurez-vous de spécifier l'objet suivant).
Par exemple:

```
app.get('/example/b', function (req, res, next) {
  console.log('the response will be sent by the next function ...');
  next();
}, function (req, res) { res.send('Hello from B!'); });
```

Route handlers samples (continue...)

Un tableau de fonctions de rappel peut gérer un itinéraire. Par exemple:

```
var cb0 = function (req, res, next) { console.log('CB0'); next(); }
var cb1 = function (req, res, next) { console.log('CB1'); next(); }
var cb2 = function (req, res) { res.send('Hello from C!'); }
app.get('/example/c', [cb0, cb1, cb2]);
```

Route handlers samples (continue...)

Une combinaison de fonctions indépendantes et de tableaux de fonctions peut gérer un itinéraire. Par exemple:

```
var cb0 = function (req, res, next) { console.log('CB0'); next(); }
var cb1 = function (req, res, next) { console.log('CB1'); next(); }
app.get('/example/d', [cb0, cb1], function (req, res, next) {
  console.log('the response will be sent by the next function ...');
  next();
}, function (req, res) { res.send('Hello from D!'); });
```

Framework Express - Response

Response Methods

Les méthodes sur l'objet response (res) du tableau suivant peuvent envoyer une réponse au client et mettre fin au cycle demande-réponse.

Si aucune de ces méthodes n'est appelée à partir d'un gestionnaire de routage, la demande du client reste en suspens.

Response Methods (continue...)

Method	Description
res.download()	Demander le téléchargement d'un fichier.
res.end()	Mettez fin au processus de réponse.
res.json()	Envoyez une réponse JSON.
res.jsonp()	Envoyez une réponse JSON avec la prise en charge JSONP.
res.redirect()	Rediriger une demande (requête)
res.render()	Afficher un modèle de révision.
res.send()	Envoyez une réponse de différents types.
res.sendFile()	Envoyez un fichier en tant que flux d'octets.
res.sendStatus()	Définissez le code d'état de la réponse et envoyez sa représentation sous forme de chaîne en tant que corps de la réponse.

Framework Express – Les « Middleware »

Express Middleware

*Les fonctions **middleware** sont des fonctions qui ont accès à l'objet de requête (req), à l'objet de réponse (res) et à la fonction middleware suivante dans le cycle requête-réponse de l'application. La fonction middleware suivante est généralement désignée par une variable nommée next.*

Les fonctions **middleware** peuvent effectuer les tâches suivantes :

Exécutez n'importe quel code.

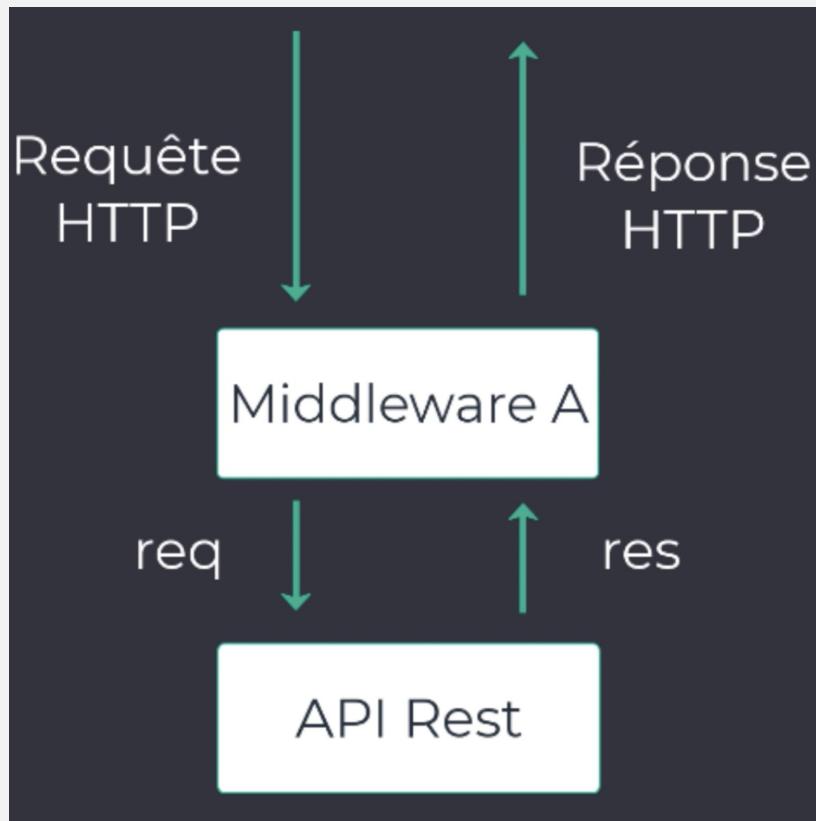
Apportez des modifications aux objets de demande et de réponse.

Terminez le cycle demande-réponse.

Appelez le prochain middleware de la pile.

Si la fonction middleware actuelle ne met pas fin au cycle requête-réponse, elle doit appeler `next()` pour passer le contrôle à la fonction middleware suivante. Sinon, la demande sera laissée en suspens.

Express Middleware



```
1 | const middleware = (req, res, next) => {
2 |   // Traitement quelconque,
3 |   // On peut intervenir sur les objets 'req' et 'res'.
4 |   // Puis on indique à Express que le traitement est terminé:
5 |   next()
6 | }
```

Express Middleware (continue...)

Voici les éléments d'un appel de fonction middleware :

```
var express = require('express');
var app = express();
app.get('/', function(req, res, next) {
    next();
});
app.listen(3000);
```

Express Middleware (continue...)

Get : méthode HTTP pour laquelle la fonction middleware s'applique.

- **'r'**: Path (route) pour lesquels la fonction middleware s'applique.
- **function ()**: La fonction middleware.
- **req**: Argument de requête HTTP à la fonction middleware.
- **res**: Argument de réponse HTTP à la fonction middleware.
- **next**: Argument de rappel à la fonction middleware.

app.use() method

app.use([path,] function [, function...])

Monte la ou les fonctions middleware spécifiées sur le chemin spécifié. Si le chemin d'accès n'est pas spécifié, il est défini par défaut sur '/'.

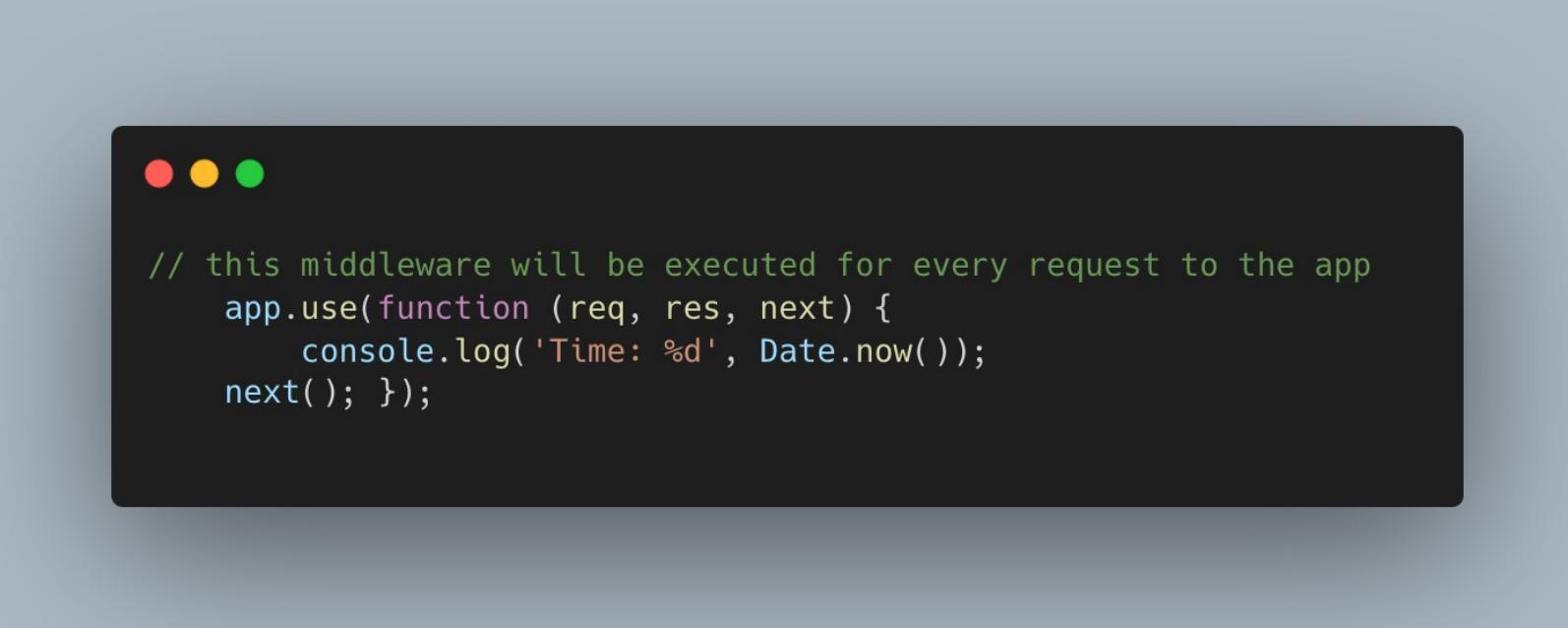
REMARQUE : Un itinéraire correspond à tout chemin qui suit immédiatement son chemin avec un « / ». Par exemple, app.use('/apple', ...) correspondra à « /apple », « /apple/images », « /apple/images/news », etc.

Le montage d'une fonction middleware sur un chemin entraîne l'exécution de la fonction middleware chaque fois que la base du chemin demandé correspond au chemin.

```
app.use('/admin', function(req, res, next) {
  // GET 'http://www.example.com/admin/new'
  console.log(req.originalUrl); // '/admin/new'
  console.log(req.baseUrl); // '/admin'
  console.log(req.path); // '/new'
  next();
});
```

app.use() (continue...)

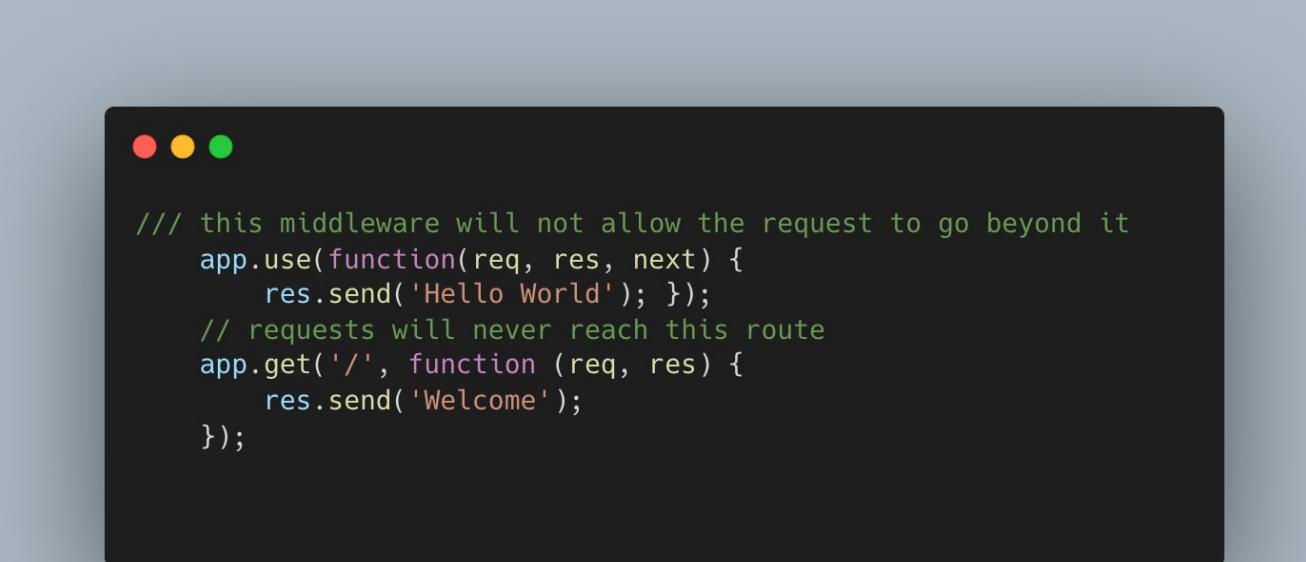
Étant donné que le chemin est par défaut « / », un middleware monté sans chemin sera exécuté pour chaque requête à l'application.



```
// this middleware will be executed for every request to the app
app.use(function (req, res, next) {
  console.log('Time: %d', Date.now());
  next();
});
```

app.use() (continue...)

Les fonctions middleware sont exécutées séquentiellement, par conséquent, l'ordre d'inclusion du middleware est important:



```
/// this middleware will not allow the request to go beyond it
app.use(function(req, res, next) {
    res.send('Hello World');
}); // requests will never reach this route
app.get('/', function (req, res) {
    res.send('Welcome');
});
```

Utiliser un Middleware

Express est un framework Web de routage et de middleware qui a une fonctionnalité minimale qui lui est propre : une application Express est essentiellement une série d'appels de fonction middleware.

Les fonctions middleware sont des fonctions qui ont accès à l'objet de requête (`req`), à l'objet de réponse (`res`) et à la fonction middleware suivante dans le cycle requête-réponse de l'application. La fonction middleware suivante est généralement désignée par une variable nommée `next`.

Les fonctions middleware peuvent effectuer les tâches suivantes:

Exécutez n'importe quel code.

Apportez des modifications aux objets de demande et de réponse.

Terminez le cycle demande-réponse.

Appelez la fonction middleware suivante dans la pile.

Utiliser un Middleware

Si la **fonction middleware** actuelle ne met pas fin au cycle requête-réponse, elle doit appeler `next()` pour passer le contrôle à la fonction middleware suivante. Sinon, la demande sera laissée en suspens.

[Une application Express peut utiliser les types de middleware suivants:](#)

[Middleware au niveau de l'application](#)[Middleware au niveau du routeur](#)[Middleware de gestion des erreurs](#)[Middleware intégré](#)
[Middleware tiers](#)

REMARQUE: Vous pouvez charger un middleware au niveau de l'application et du routeur avec un chemin de montage facultatif. Vous pouvez également charger une série de fonctions middleware ensemble, ce qui crée une sous-pile du système middleware à un point de montage.

Les types de Middleware

1/5. Le Middleware
d'application

2/5. Le Middleware
du routeur

3/5. Le Middleware de
traitement d'erreurs

4/5. Le Middleware
intégré

5/5. Les
Middlewares tiers

Le Middleware d'application

1/5. Le Middleware d'application

```
1 | var app = express();
2 |
3 | // On passe un middleware en paramètre de la méthode use():
4 | app.use(function (req, res, next) {
5 |   console.log('Time:', Date.now());
6 |   next();
7 | });
```

Le Middleware du Router

```
const express = require('express')
const app = express()
const router = express.Router()

// a middleware function with no mount path. This code is executed for every request to the router
router.use((req, res, next) => {
  console.log('Time:', Date.now())
  next()
})

// a middleware sub-stack shows request info for any type of HTTP request to the /user/:id path
router.use('/user/:id', (req, res, next) => {
  console.log('Request URL:', req.originalUrl)
  next()
}, (req, res, next) => {
  console.log('Request Type:', req.method)
  next()
})
```

Le Middleware de traitement des erreurs

3/5. Le Middleware de traitement d'erreurs

```
1 app.use(function(err, req, res, next) {  
2   console.error(err);  
3   res.send('Erreur !');  
4 });
```

Le Middleware intégré

À partir de la version 4.x, Express ne dépend plus de Connect. Les fonctions middleware qui étaient auparavant incluses avec Express sont maintenant dans des modules séparés; voir la liste des fonctions middleware.

Express dispose des fonctions middleware intégrées suivantes :

express.static sert des ressources statiques telles que des fichiers HTML, des images, etc.

express.json analyse les demandes entrantes avec des charges utiles JSON. REMARQUE:

Disponible avec Express 4.16.0+

express.urlencoded analyse les demandes entrantes avec des charges utiles codées par

URL. REMARQUE: Disponible avec Express 4.16.0+

Le Middleware intégré

À partir de la version 4.x, Express ne dépend plus de Connect. Les fonctions middleware qui étaient auparavant incluses avec Express sont maintenant dans des modules séparés; voir la liste des fonctions middleware.

Express dispose des fonctions middleware intégrées suivantes :

express.static sert des ressources statiques telles que des fichiers HTML, des images, etc.

express.json analyse les demandes entrantes avec des charges utiles JSON. REMARQUE:

Disponible avec Express 4.16.0+

express.urlencoded analyse les demandes entrantes avec des charges utiles codées par

URL. REMARQUE: Disponible avec Express 4.16.0+

Le Middleware Tiers

Utilisez un middleware tiers pour ajouter des fonctionnalités aux applications Express.

Installez le module Node.js pour les fonctionnalités requises, puis chargez-le dans votre application au niveau de l'application ou au niveau du routeur.

L'exemple suivant illustre l'installation et le chargement de la fonction middleware d'analyse des cookies cookie-parser.

```
const express = require('express')
const app = express()
const cookieParser = require('cookie-parser')

// load the cookie-parsing middleware
app.use(cookieParser())
```

Express Routers

Un **objet routeur** est une instance isolée de middleware et d'itinéraires. Vous pouvez le considérer comme une « mini-application », capable uniquement d'exécuter des fonctions de middleware et de routage. Chaque application Express dispose d'un routeur d'application intégré.

Un **routeur** se comporte comme un middleware lui-même, vous pouvez donc l'utiliser comme argument pour `app.use()` ou comme argument pour la méthode `use()` d'un autre routeur.

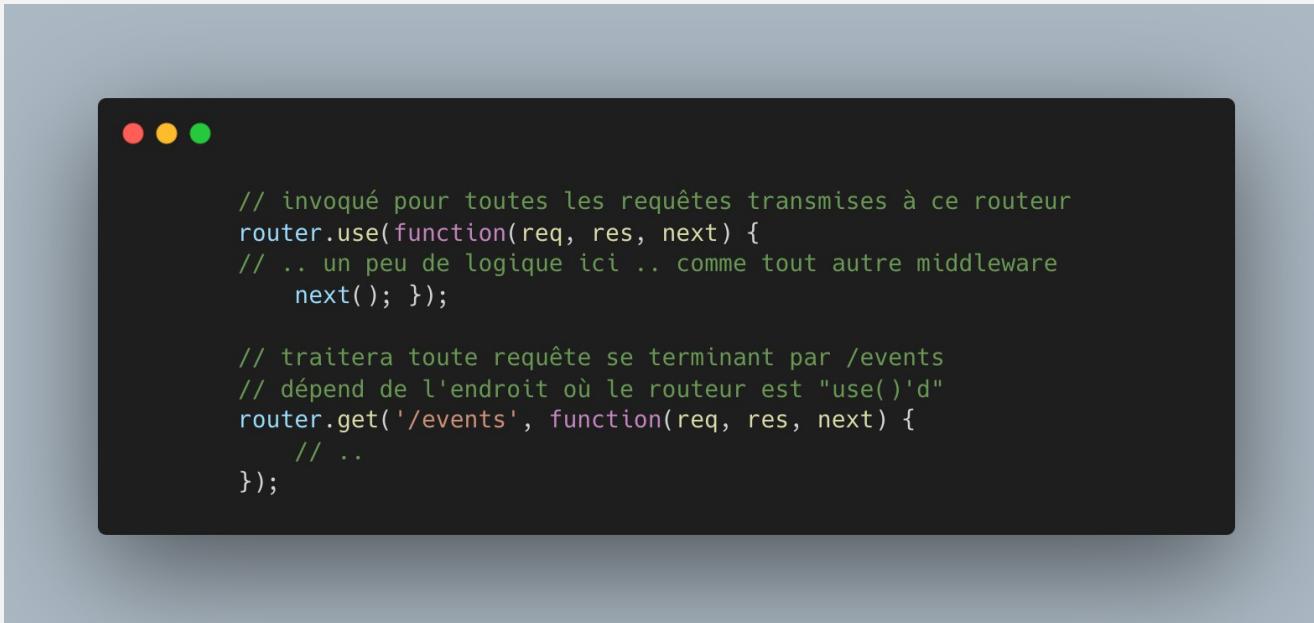
L'objet **express** de niveau supérieur possède une méthode **Router()** qui crée un nouvel objet routeur.



```
var express = require('express');
var app = express();
var router = express.Router();
```

Express Routers 2

Une fois que vous avez créé un objet routeur, vous pouvez y ajouter des itinéraires de méthode middleware et HTTP (tels que get, put, post, etc.) comme une application.

A screenshot of a terminal window with a dark background and light-colored text. The window has three colored window control buttons (red, yellow, green) at the top left. The terminal shows a snippet of JavaScript code for setting up an Express router. It includes a middleware function using the 'use' method and an HTTP endpoint using the 'get' method.

```
// invoqué pour toutes les requêtes transmises à ce routeur
router.use(function(req, res, next) {
  // .. un peu de logique ici .. comme tout autre middleware
  next();
});

// traitera toute requête se terminant par /events
// dépend de l'endroit où le routeur est "use()'d"
router.get('/events', function(req, res, next) {
  // ..
});
```

Express Routers 2

Vous pouvez ensuite utiliser un routeur pour une URL racine particulière en séparant ainsi vos itinéraires en fichiers ou même en mini-applications.



```
// seules les requêtes vers /calendar/* seront envoyées à notre "routeur"  
app.use('/calendar', router);
```

Router Methods

router.all(path, [callback, ...] callback)

Cette méthode est extrêmement utile pour mapper une logique « globale » pour des préfixes de chemin spécifiques ou des correspondances arbitraires. Par exemple, si vous placez l'itinéraire suivant en haut de toutes les autres définitions d'itinéraire, il faudrait que tous les itinéraires à partir de ce point nécessitent une authentification et chargent automatiquement un utilisateur.



```
router.all('*', requireAuthentication, loadUser);
```

Router Methods

Un autre exemple de ceci est la fonctionnalité « globale » de la liste blanche. Ici, l'exemple ressemble beaucoup à ce qu'il était auparavant, mais il ne restreint que les chemins préfixés par « /api »:

```
router.all('/api/*', requireAuthentication);
```

Router Methods 2

router.METHOD(path, [callback, ...] callback)

Le routeur. Les méthodes METHOD() fournissent la fonctionnalité de routage dans Express, où METHOD est l'une des méthodes HTTP, telles que GET, PUT, POST, etc., en minuscules. Ainsi, les méthodes réelles sont router.get(), router.post(), router.put(), etc.

Vous pouvez fournir plusieurs rappels, et tous sont traités de manière égale, et se comportent comme des intergiciels, sauf que ces rappels peuvent appeler next('route') pour contourner le(s) rappel(s) de route restant(s).

Vous pouvez utiliser ce mécanisme pour effectuer des conditions préalables sur un itinéraire, puis passer le contrôle aux itinéraires suivants lorsqu'il n'y a aucune raison de poursuivre l'itinéraire correspondant.

```
router.get('/', function(req, res){  
  res.send('hello world');  
});
```

Router Methods 3

router.use([path], [function, ...] function)

Utilise la ou les fonctions middleware spécifiées, avec chemin de montage facultatif, qui est par défaut « / ».

Cette méthode est similaire à app.use().

Le middleware est comme un tuyau de plomberie : les requêtes commencent à la première fonction middleware définie et se frayent un chemin « vers le bas » du traitement de la pile middleware pour chaque chemin qu'elles correspondent.

L'ordre dans lequel vous définissez le middleware avec router.use() est très important. Ils sont appelés séquentiellement, donc l'ordre définit la priorité du middleware. Par exemple, un enregistreur est généralement le tout premier middleware que vous utiliseriez, de sorte que chaque demande est enregistrée.

Router-level Middleware

Le middleware au niveau du routeur fonctionne de la même manière que le middleware au niveau de l'application, sauf qu'il est lié à une instance d'express.Router().

```
var router = express.Router();
```

Chargez le middleware au niveau du routeur à l'aide du routeur.use() et du routeur.METHOD() fonctions.

L'exemple de code suivant réplique le système middleware illustré ci-dessus pour le middleware au niveau de l'application (voir app5.js), à l'aide d'un middleware au niveau du routeur.

Error-handling Middleware

Définissez les fonctions middleware de gestion des erreurs de la même manière que les autres fonctions middleware, sauf avec quatre arguments au lieu de trois, en particulier avec la signature (err, req, res, next) :

```
app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

Error-handling Middleware

REMARQUE : Le middleware de gestion des erreurs prend toujours quatre arguments. Vous devez fournir quatre arguments pour l'identifier en tant que fonction middleware de gestion des erreurs. Même si vous n'avez pas besoin d'utiliser l'objet suivant, vous devez le spécifier pour conserver la signature. Sinon, l'objet suivant sera interprété comme un middleware normal et ne parviendra pas à gérer les erreurs.

Third-party Middleware

Utilisez un middleware tiers pour ajouter des fonctionnalités aux applications Express.

Installez le module Node.js pour les fonctionnalités requises, puis chargez-le dans votre application au niveau de l'application ou au niveau du routeur.

L'exemple suivant illustre l'installation et le chargement de la fonction middleware d'analyse des cookies cookie-parser.

```
$ npm install cookie-parser

var express = require('express');
var app = express();
var cookieParser = require('cookie-parser');
// charge le middleware d'analyse des cookies
app.use(cookieParser());
```