**Introduction à Express/Node**

* [Aperçu : Express Web Framework (Node.js/JavaScript)](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs)
* [Suivant](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/development_environment)

Dans ce tout premier article consacré à Express, nous répondons aux questions « Qu'est-ce que Node ? » et « Qu'est-ce que Express ? », et vous donnons un aperçu de ce qui fait d'Express un framework web si spécial. Nous décrivons les principales fonctionnalités et montrons quelques-uns des principaux composants d'une application Express (bien que vous ne disposiez pas encore d'un environnement de développement pour le tester).

|  |  |
| --- | --- |
| **Prérequis :** | Une culture de base en informatique, une compréhension globale de la [programmation côté serveur](https://developer.mozilla.org/fr/docs/Learn/Server-side/First_steps) et, en particulier, les mécanismes d'[interactions client-serveur dans un site web.](https://developer.mozilla.org/fr/docs/Learn/Server-side/First_steps/Client-Server_overview) |
| **Objectif :** | Devenir familier avec ce qu'est Express et comment il s'intègre dans Node, les fonctionnalités qu'il apporte, et les principales étapes pour construire une application Express. |

[**Introduction à Node**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#introduction_%C3%A0_node)

[Node](https://nodejs.org/) (ou plus formellement *Node.js*) est un environnement d'exécution open-source, multi-plateforme, qui permet aux développeuses et développeurs de créer toutes sortes d'applications et d'outils côté serveur en [JavaScript](https://developer.mozilla.org/fr/docs/Glossary/JavaScript). Cet environnement est destiné à être utilisé en dehors du navigateur (il s'exécute directement sur son ordinateur ou dans le système d'exploitation du serveur). Aussi, Node ne permet pas d'utiliser les API JavaScript liées au navigateur mais des API plus traditionnellement utilisées sur un serveur dont notamment celles pour HTTP ou la manipulation de systèmes de fichier.

Dans une perspective de développement de serveur web, Node présente un certain nombre d'avantages :

* D'excellentes performances ! Node a été créé pour optimiser le rendement et l'évolution des applications web. Il s'agit d'une bonne solution à de nombreux problèmes de développement web (par exemple, les applications en temps réel).
* Le code est intégralement écrit en JavaScript ce qui signifie que l'on dépense moins d'énergie à basculer d'un langage à l'autre quand on code côté client et côté serveur.
* Le JavaScript est un langage de programmation plutôt récent et bénéficie encore d'améliorations dans sa conception en comparaison à d'autres langages web côté serveur (Python, PHP, etc.). Beaucoup d'autres langages nouveaux et populaires compilent/convertissent en JavaScript pour pouvoir utiliser TypeScript, CoffeeScript, ClojureScript, Scala, LiveScript, etc.
* Le gestionnaire de paquets (NPM) offre l'accès à des milliers de bibliothèques réutilisables. Il dispose d'une excellente résolution de dépendances et peut être utilisé pour automatiser la plupart des chaines de compilation.
* Node.js est portable. Il est disponible sous Microsoft Windows, macOS, Linux, etc. De plus, il est bien supporté par beaucoup d'hébergeurs web qui fournissent souvent une infrastructure spécifique et une documentation pour héberger des sites Node.
* Node possède une communauté et un écosystème très dynamiques avec beaucoup de gens désireux d'aider.

Vous pouvez utiliser Node.js pour créer un simple serveur web en utilisant l'API Node HTTP.

[**Hello Node.js**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#hello_node.js)

L'exemple qui suit crée un serveur web qui écoute toutes sortes de requêtes HTTP sur l'URL https://127.0.0.1:8000/. Quand une requête est reçue, le script répond avec la chaine « Salut tout le monde ». Si vous avez déjà installé Node, suivez les étapes de l'exemple suivant :

1. Ouvrez un terminal (sur Windows, ouvrez l'invite de commande (cmd)),
2. Créez le dossier où vous voulez sauvegarder le programme, appelez-le par exemple test-node et placez-vous dedans en utilisant la commande suivante dans votre console :

cd test-node

1. Dans votre éditeur de texte favori, créez un fichier nommé "hello.js" et collez ce qui suit dedans :

// Charge le module HTTP

const http = require("http");

const hostname = "127.0.0.1";

const port = 8000;

// Crée un serveur HTTP

const server = http.createServer((req, res) => {

 // Configure l'en-tête de la réponse HTTP

 // avec le code du statut et le type de contenu

 res.writeHead(200, { "Content-Type": "text/plain" });

 // Envoie le corps de la réponse « Salut tout le monde »

 res.end("Salut tout le monde\n");

});

// Démarre le serveur à l'adresse 127.0.0.1 sur le port 8000

// Affiche un message dès que le serveur commence à écouter les requêtes

server.listen(port, hostname, () => {

 console.log(`Le serveur tourne à l'adresse https://${hostname}:${port}/`);

});

1. Sauvegardez le fichier dans le dossier créé plus haut.
2. Retournez au terminal et tapez :

node hello.js

Puis saisissez l'URL "https://localhost:8000" dans votre navigateur. Vous devriez alors voir "**Salut tout le monde**" en haut à gauche d'une page web ne contenant rien d'autre que ce texte.

[**Les frameworks web**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#les_frameworks_web)

D'autres tâches de développement web ne sont pas directement prises en charge par Node de façon native. Si vous voulez ajouter différentes manipulations pour divers requêtes HTTP (GET, POST, DELETE, etc.), gérer différemment des requêtes vers plusieurs chemins URL ("routes"), servir des pages statiques ou utiliser des modèles pour créer dynamiquement la réponse, alors vous devrez écrire tout le code vous-même ou, pour éviter de réinventer la roue, vous servir des cadres applicatifs web (frameworks).

[**Introduction à Express**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#introduction_%C3%A0_express)

[Express](https://expressjs.com/) est le *framework* actuellement le plus populaire dans Node et est la bibliothèque sous-jacente pour un grand nombre d'autres [cadres applicatifs web pour Node](https://expressjs.com/fr/resources/frameworks.html). Il fournit des mécanismes pour :

* Écrire des fonctions de traitement pour différentes requêtes HTTP répondant à différentes URI (par le biais des *routes*).
* Intégrer avec les moteurs de rendu de « vues » dans le but de générer des réponses en insérant des données dans des modèles (« *templates* »). Configurer certains paramètres d'applications comme le port à utiliser à la connexion et l'emplacement des modèles nécessaires pour la mise en forme de la réponse.
* Ajouter des requêtes de traitement « *middleware* » (intergiciel) où vous le voulez dans le tunnel gestionnaire de la requête.

Bien qu'Express soit assez minimaliste, des *middlewares* (fonctions intermédiaires) compatibles ont été créés pour résoudre quasiment tous les problèmes de développement web. Il existe des bibliothèques pour se servir des cookies, gérer les sessions, la connexion de l'utilisateur, les paramètres de l'URL, les données POST, les entêtes de sécurité et d'autres encore. Vous trouverez une liste des paquets maintenus par l'équipe Express ici : [Express Middleware](https://expressjs.com/fr/resources/middleware.html) (ainsi que la liste de paquets tiers populaires).

**Note :** Cette flexibilité est à double tranchant. Il y a une multitude de paquets pour résoudre chaque problème mais trouver le bon paquet à utiliser peut vite devenir un défi. Il n'y a pas non plus de « bonne manière » pour structurer une application et beaucoup d'exemples que vous trouverez sur le net ne sont pas optimisés ou montrent seulement une infime partie de ce que vous devez faire pour développer une application web.

[**D'où viennent Node et Express ?**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#do%C3%B9_viennent_node_et_express)

À ses débuts en 2009, Node a été publié pour Linux uniquement. Le gestionnaire de paquets NPM est sorti en 2010, et le support natif de Windows fut ajouté en 2012. Ceci est un très court aperçu d'une aventure riche en rebondissements. Allez creuser ça sur [Wikipédia](https://fr.wikipedia.org/wiki/Node.js#Historique) si vous voulez en savoir plus.

Express est sorti pour la première fois en novembre 2010. Vous pouvez consulter la [liste des modifications](https://expressjs.com/en/changelog/4x.html) pour plus d'informations sur la version courante et [GitHub](https://github.com/expressjs/express/blob/master/History.md) pour plus de détails sur l'historique des publications.

[**Quelle popularité pour Node et Express ?**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#quelle_popularit%C3%A9_pour_node_et_express)

La popularité d'un *framework* web est importante car elle conditionne la maintenance dans le temps et les ressources qu'il est raisonnable de mettre à disposition dans la documentation, les bibliothèques d'extensions et le support technique.

Il n'existe pas d'échelle de mesures définitive et fiable pour l'estimation de la popularité des *frameworks* côté serveur, bien que des sites comme [Hot Frameworks](https://hotframeworks.com/) essaient d'estimer la popularité par le comptage du nombre de projets GitHub ou StackOverflow. La question est : « Est-ce que Node et Express sont suffisamment populaires pour pouvoir s'affranchir des plateformes non-populaires ? Continuent-ils à évoluer ? Pouvez-vous avoir de l'aide si besoin ? Existe-t-il une opportunité pour vous de gagner de l'argent si vous apprenez Express ? ».

Si on se réfère à la [liste des entreprises utilisant Express](https://expressjs.com/fr/resources/companies-using-express.html), la quantité de gens contribuant au code et le nombre de gens fournissant un support payant ou bien gratuit, alors oui, *Express* est un framework populaire !

[**Express est-il « dogmatique » ?**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#express_est-il_%C2%AB_dogmatique_%C2%BB)

Les cadres logiciels web se décrivent souvent comme ayant ou non des opinions données sur tel ou tel sujet au sens où ils sont orientés dans leur usage selon ces choix/opinions. En anglais, on dit d'un *framework* qu'il est *opinionated* ou non.

Les *frameworks* avec une opinion sont ceux qui ont un avis arrêté sur la « bonne manière » de gérer certaines tâches. Ils fournissent souvent un cadre permettant de développer rapidement *dans un domaine particulier* (résolvant des problèmes d'un type particulier) parce que la bonne manière de faire quoi que ce soit est généralement bien comprise et bien documentée. Toutefois, ils peuvent manquer de flexibilité pour la résolution de problèmes hors de leur portée et tendent à offrir peu de choix concernant les composants et approches qu'ils peuvent utiliser.

Les *frameworks* sans opinion, par contraste, ont beaucoup moins de restrictions sur la meilleure manière d'assembler des composants ensemble pour atteindre un objectif, ou encore sur les composants que vous devriez utiliser. Ils laissent aux développeurs la possibilité d'utiliser les outils les plus adaptés pour achever une tâche particulière, bien que cela nécessite que vous cherchiez et trouviez ces composants par vous-même.

Express n'est pas particulièrement dogmatique. Vous pouvez intégrer quasiment n'importe quelle fonction intermédiaire compatible voulue dans la pile de gestion des requêtes, dans quasiment n'importe quel ordre. Vous pouvez structurer l'application en un fichier comme en plusieurs, et utiliser n'importe quelle structure de dossiers. Vous pourrez même quelques fois vous sentir perdu⋅e par la liberté que vous avez de vous organiser comme vous le souhaitez !

[**À quoi ressemble du code Express ?**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#%C3%A0_quoi_ressemble_du_code_express)

Dans un site web utilisé pour traiter des données, une application web attend des requêtes HTTP du navigateur web (ou d'un autre client). Quand une requête est reçue, l'application cherche quelle action est requise en fonction du modèle de l'URL et des possibles informations associés contenues dans les données POST ou GET. Selon ce qui est requis, il pourra alors lire ou écrire des informations dans une base de données ou effectuer d'autre tâches pour satisfaire la requête. L'application va alors retourner une réponse au navigateur web, souvent une page HTML créée dynamiquement pour le navigateur, en intégrant les données récupérées dans un modèle HTML.

Express fournit des méthodes pour spécifier quelle fonction est appelée pour une méthode HTTP particulière (GET, POST, SET, etc.) et un modèle d'URL ("Route"), ainsi que des méthodes pour spécifier quel moteur de rendu de vues ("view") est utilisé, où sont les modèles de vues et quel modèle utiliser pour générer une réponse. Vous pouvez utiliser les fonctions intermédiaires d'Express pour prendre en charge les cookies, les sessions, les utilisateurs, obtenir les paramètres POST/GET, etc. Vous pouvez utiliser n'importe que système de base données supporté par Node (Express ne définit aucun comportement relatif aux bases de données).

Les sections suivantes expliquent quelques choses communes que vous verrez en travaillant avec du code *Express* et *Node*.

[**Hello World Express**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#hello_world_express)

Tout d'abord intéressons-nous à l'exemple [Hello World](https://expressjs.com/fr/starter/hello-world.html) standard d'Express (nous expliquons chaque partie de cet exemple ci-dessous, et dans les sections suivantes).

**Note :** Si vous avez déjà installé Node et Express (ou si vous les installez comme montré dans [l'article suivant](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/development_environment)), vous pouvez enregistrer ce code dans un fichier texte appelé **app.js** et l'exécuter via un terminal en tapant :

**node app.js**

const express = require("express");

const app = express();

const port = 3000;

app.get("/", (req, res) => {

 res.send("Hello World!");

});

app.listen(port, () => {

 console.log(`Application exemple à l'écoute sur le port ${port}!`);

});

Les deux premières lignes importent (avec require()) le module express et créent une [application express](https://expressjs.com/en/4x/api.html#app). Cet objet, qui est traditionnellement nommé app, possède des méthodes pour acheminer les requêtes HTTP, configurer les intergiciels, rendre les vues HTML, enregistrer un moteur de modèles et modifier les [paramètres de l'application](https://expressjs.com/en/4x/api.html#app.settings.table) qui contrôlent le comportement de l'application (par exemple, le mode d'environnement, si les définitions de route sont sensibles à la casse, etc.).

La partie centrale du code (les trois lignes commençant par app.get) montre une *définition de route*. La méthode app.get() spécifie une fonction de rappel qui sera invoquée chaque fois qu'il y aura une requête HTTP GET avec un chemin ('/') relatif à la racine du site. La fonction de rappel prend une requête et un objet réponse comme arguments, et appelle simplement [send()](https://expressjs.com/en/4x/api.html%22%20%5Cl%20%22res.send%22%20%5Ct%20%22_blank) sur la réponse pour renvoyer la chaîne de caractères "Hello World!".

Le dernier bloc démarre le serveur sur le port 3000 et affiche un commentaire de journal dans la console. Avec le serveur en cours d'exécution, vous pouvez aller sur localhost:3000 dans votre navigateur pour voir l'exemple de réponse renvoyé.

[**Créer et importer des modules**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#cr%C3%A9er_et_importer_des_modules)

Un module est une bibliothèque/fichier JavaScript que vous pouvez importer dans un autre code en utilisant la fonction require() de Node. *Express* lui-même est un module, tout comme les bibliothèques de *middleware* et de base de données que nous utilisons dans nos applications *Express*.

Le code ci-dessous montre comment nous importons un module par son nom, en utilisant le framework *Express* comme exemple. Tout d'abord, nous invoquons la fonction require(), en spécifiant le nom du module sous forme de chaîne ('express'), et en appelant l'objet retourné pour créer une [applicationExpress](https://expressjs.com/en/4x/api.html%22%20%5Cl%20%22app%22%20%5Ct%20%22_blank). Nous pouvons alors accéder aux propriétés et fonctions de l'objet application.

const express = require("express");

const app = express();

Vous pouvez également créer vos propres modules qui peuvent être importés de la même manière.

**Note :** Vous *voudrez* créer vos propres modules, car cela vous permet d'organiser votre code en parties maniables — une application monolithique à fichier unique est difficile à comprendre et à maintenir. L'utilisation de modules vous aide également à gérer votre espace de noms, car seules les variables que vous exportez explicitement sont importées lorsque vous utilisez un module.

Pour rendre les objets disponibles en dehors d'un module, il suffit de les affecter à l'objet exports. Par exemple, le module **square.js** ci-dessous est un fichier qui exporte les méthodes area() et perimeter() :

exports.area = function (width) {

 return width \* width;

};

exports.perimeter = function (width) {

 return 4 \* width;

};

Nous pouvons importer ce module en utilisant require(), puis appeler la ou les méthodes exportées comme indiqué :

var square = require("./square"); // Ici, nous demandons le nom du fichier sans l'extension de fichier .js (facultative).

console.log(

 "L'aire d'un carré dont la largeur est de 4 est la suivante " +

 square.area(4),

);

**Note :** Vous pouvez également spécifier un chemin absolu vers le module (ou un nom, comme nous l'avons fait initialement).

Si vous souhaitez exporter un objet complet en une seule affectation au lieu de le construire une propriété à la fois, affectez-le à module.exports comme indiqué ci-dessous (vous pouvez également procéder ainsi pour faire de la racine de l'objet exports un constructeur ou une autre fonction) :

module.exports = {

 area: function (width) {

 return width \* width;

 },

 perimeter: function (width) {

 return 4 \* width;

 },

};

**Note :** L'objet exports peut être vu comme un [raccourci](https://nodejs.org/api/modules.html#modules_exports_shortcut) vers module.exports au sein d'un module donné. En fait, exports est simplement une variable qui est initialisée avec la valeur de module.exports avant que le module soit évalué. Cette valeur est une référence vers un objet. Cela signifie que exports référence le même objet que celui référencé par module.exports. Cela signifie également qu'affecter une autre valeur à exports le détachera complètement de module.exports.

Pour de plus amples informations sur les modules, voir [Modules](https://nodejs.org/api/modules.html#modules_modules) (documentation Node).

[**Utilisation des API asynchrones**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#utilisation_des_api_asynchrones)

Le code JavaScript utilise fréquemment des API asynchrones plutôt que synchrones pour les opérations qui peuvent prendre un certain temps à se terminer. Une API synchrone est une API dans laquelle chaque opération doit être terminée avant que l'opération suivante puisse commencer. Par exemple, les fonctions d'enregistrement suivantes sont synchrones et impriment le texte dans la console dans l'ordre (Premier, Second).

console.log("Premier");

console.log("Second");

En revanche, une API asynchrone est une API qui lance une opération et revient immédiatement (avant que l'opération ne soit terminée). Une fois l'opération terminée, l'API utilisera un mécanisme quelconque pour effectuer des opérations supplémentaires. Par exemple, le code ci-dessous imprimera « Second, Premier » car même si la méthode setTimeout() est appelée en premier, et revient immédiatement, l'opération ne se termine pas avant plusieurs secondes.

setTimeout(function () {

 console.log("Premier");

}, 3000);

console.log("Second");

L'utilisation d'API asynchrones non bloquantes est encore plus importante sur Node que dans le navigateur, car *Node* est un environnement d'exécution événementiel avec un seul *thread*. Cela signifie que toutes les requêtes adressées au serveur sont exécutées sur le même *thread* (plutôt que d'être fractionnées en *threads* distincts). Ce modèle est extrêmement efficace en termes de vitesse et de ressources du serveur, mais il signifie que si l'une de vos fonctions appelle des méthodes synchrones qui prennent beaucoup de temps pour se terminer, elle bloquera non seulement la demande actuelle, mais aussi toutes les autres demandes traitées par votre application Web.

Il existe plusieurs façons pour une API asynchrone d'informer votre application de la fin d'une opération. La méthode la plus courante consiste à enregistrer une fonction de rappel lorsque vous invoquez l'API asynchrone, qui sera rappelée lorsque l'opération sera terminée. C'est l'approche utilisée ci-dessus.

**Note :** L'utilisation des rappels (« *callbacks* ») peut être assez « désordonnée » si vous avez une séquence d'opérations asynchrones dépendantes qui doivent être exécutées dans l'ordre, car cela entraîne de multiples niveaux de rappels imbriqués. Ce problème est communément appelé « l'enfer des callbacks ». Ce problème peut être réduit par de bonnes pratiques de codage dont l'utilisation des [promesses](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Promise) ou de [async](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Statements/async_function)/[await](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Operators/await).

**Note :** Une convention courante pour Node et Express est d'utiliser des callbacks de type error-first. Dans cette convention, la première valeur de vos *fonctions de rappel* est une valeur d'erreur, tandis que les arguments suivants contiennent des données de succès. Il y a une bonne explication de l'utilité de cette approche dans ce blog : [The Node.js Way - Comprendre les callbacks de type « Error First ».](https://fredkschott.com/post/2014/03/understanding-error-first-callbacks-in-node-js) (fredkschott.com).

[**Créer des gestionnaires de route**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#cr%C3%A9er_des_gestionnaires_de_route)

Dans notre exemple *Hello World* d'Express (voir ci-dessus), nous avons défini une fonction de gestion de route (un *callback*) pour les requêtes HTTP GET vers la racine du site ('/').

app.get("/", (req, res) => {

 res.send("Hello World!");

});

La fonction de rappel prend une requête et un objet réponse comme arguments. Dans ce cas, la méthode appelle simplement [send()](https://expressjs.com/en/4x/api.html%22%20%5Cl%20%22res.send%22%20%5Ct%20%22_blank) sur la réponse pour renvoyer la chaîne de caractères « Hello World ! ». Il existe un [nombre d'autres méthodes de réponse](https://expressjs.com/fr/guide/routing.html#response-methods) pour terminer le cycle requête/réponse, par exemple vous pourriez appeler [res.json()](https://expressjs.com/en/4x/api.html%22%20%5Cl%20%22res.json%22%20%5Ct%20%22_blank) pour envoyer une réponse JSON ou [res.sendFile()](https://expressjs.com/en/4x/api.html%22%20%5Cl%20%22res.sendFile%22%20%5Ct%20%22_blank) pour envoyer un fichier.

**Note :** Vous pouvez utiliser les noms d'arguments de votre choix dans les fonctions de rappel ; lorsque le rappel est invoqué, le premier argument sera toujours la requête et le second sera toujours la réponse. Il est judicieux de les nommer de telle sorte que vous puissiez identifier l'objet avec lequel vous travaillez dans le corps du *callback*.

L'objet *Express* application fournit également des méthodes permettant de définir des gestionnaires de route pour tous les autres verbes HTTP, qui sont pour la plupart utilisés exactement de la même manière :

checkout(), copy(), **delete()**, **get()**, head(), lock(), merge(), mkactivity(), mkcol(), move(), m-search(), notify(), options(), patch(), **post()**, purge(), **put()**, report(), search(), subscribe(), trace(), unlock(), unsubscribe().

Il existe une méthode de routage spéciale, app.all(), qui sera appelée en réponse à toute méthode HTTP. Ceci est utilisé pour charger les fonctions *middleware* à un chemin particulier pour toutes les méthodes de requête. L'exemple suivant (tiré de la documentation d'Express) montre un gestionnaire qui sera exécuté pour les requêtes vers /secret indépendamment du verbe HTTP utilisé (à condition qu'il soit supporté par le [module http](https://nodejs.org/api/http.html#http_http_methods)).

app.all("/secret", (req, res, next) => {

 console.log("Accès à la section secrète ...");

 next(); // passe le contrôle au gestionnaire suivant

});

Les routes vous permettent de faire correspondre des modèles particuliers de caractères dans une URL, d'extraire certaines valeurs de l'URL et de les transmettre comme paramètres au gestionnaire de la route (en tant qu'attributs de l'objet de la demande transmis comme paramètre).

Il est souvent utile de regrouper les gestionnaires de route pour une partie particulière d'un site et d'y accéder en utilisant un préfixe de route commun (par exemple, un site avec un Wiki pourrait avoir toutes les routes liées au wiki dans un seul fichier et les faire accéder avec un préfixe de route de */wiki/*). Dans *Express*, ceci est réalisé en utilisant l'objet [express.Router](https://expressjs.com/fr/guide/routing.html%22%20%5Cl%20%22express-router%22%20%5Ct%20%22_blank). Par exemple, nous pouvons créer notre route wiki dans un module nommé **wiki.js**, puis exporter l'objet Router, comme indiqué ci-dessous :

// wiki.js - Module route du wiki

const express = require("express");

const router = express.Router();

// Route vers la page d'accueil

router.get("/", (req, res) => {

 res.send("Page d'accueil du wiki");

});

// Route vers la page à propos

router.get("/about", (req, res) => {

 res.send("À propos de ce wiki");

});

module.exports = router;

**Note :** L'ajout de routes à l'objet Router est identique à l'ajout de routes à l'objet app (comme indiqué précédemment).

Pour utiliser le routeur dans notre fichier d'application principal, nous devrions alors require() le module route (**wiki.js**), puis appeler use() sur l'application *Express* pour ajouter le routeur au chemin de manipulation du middleware. Les deux routes seront alors accessibles depuis /wiki/ et /wiki/about/.

const wiki = require("./wiki.js");

// ...

app.use("/wiki", wiki);

Nous vous montrerons beaucoup plus en détails comment travailler avec les routes, et en particulier comment utiliser le Router, plus tard dans la section liée [Routes et contrôleurs](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes).

[**Utilisation d'un middleware/intergiciel**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#utilisation_dun_middlewareintergiciel)

L'intergiciel (aussi appelé « *middleware* ») est largement utilisé dans les applications d'Express, pour des tâches allant du service de fichiers statiques à la gestion des erreurs, en passant par la compression des réponses HTTP. Alors que les fonctions de route terminent le cycle requête-réponse HTTP en renvoyant une réponse au client HTTP, les fonctions d'intergiciel effectuent *typiquement* une opération sur la demande ou la réponse, puis appellent la fonction suivante dans la « pile », qui peut être un autre intergiciel ou un gestionnaire de route. L'ordre dans lequel les intergiciels sont appelés dépend du code de l'application.

**Note :** L'intergiciel peut effectuer n'importe quelle opération, exécuter n'importe quel code, apporter des modifications à l'objet requête et réponse, et il peut *aussi mettre fin au cycle requête-réponse*. S'il ne met pas fin au cycle, alors il doit appeler next() pour passer le contrôle à la fonction suivante de l'intergiciel (ou la requête sera laissée en suspens).

La plupart des apps utiliseront des *programmes intermédiaires tiers* afin de simplifier les tâches courantes de développement web comme le travail avec les cookies, les sessions, l'authentification des utilisateurs, l'accès aux données POST et JSON des requêtes, la journalisation, etc. Vous pouvez trouver une [liste des paquets *middleware* maintenus par l'équipe Express](https://expressjs.com/fr/resources/middleware.html) (qui inclut également d'autres paquets populaires de tiers). D'autres paquets Express sont disponibles sur le gestionnaire de paquets NPM.

Pour utiliser un *middleware* tiers, vous devez d'abord l'installer dans votre application à l'aide de NPM. Par exemple, pour installer l'intergiciel de journalisation des requêtes HTTP [morgan](https://expressjs.com/fr/resources/middleware/morgan.html%22%20%5Ct%20%22_blank), vous devez procéder comme suit :

bashCopy to Clipboard

npm install morgan

Vous pourriez alors appeler use() sur l'objet *Express application* pour ajouter l'intergiciel à la pile :

const express = require('express');

const logger = require('morgan');

const app = express();

app.use(logger('dev'));

...

**Note :** Les fonctions d'intergiciel et de routage sont appelées dans l'ordre où elles sont déclarées. Pour certains intergiciels, l'ordre est important (par exemple, si l'intergiciel de session dépend de l'intergiciel de cookie, alors le gestionnaire de cookie doit être ajouté en premier). Il est presque toujours nécessaire d'appeler l'intergiciel avant de définir les routes, sinon vos gestionnaires de routes n'auront pas accès aux fonctionnalités ajoutées par votre intergiciel.

Vous pouvez écrire vos propres fonctions intergiciels, et vous serez probablement amené à le faire (ne serait-ce que pour créer un code de gestion des erreurs). La **seule** différence entre une fonction *middleware* et un *callback* de gestionnaire de route est que les fonctions *middleware* ont un troisième argument next, que les fonctions *middleware* sont censées appeler si elles ne sont pas celle qui termine le cycle de requête (lorsque la fonction *middleware* est appelée, cela contient la fonction *next* qui doit être appelée).

Vous pouvez ajouter une fonction d'intergiciel à la chaîne de traitement avec app.use() ou app.add(), selon que vous voulez appliquer l'intergiciel à toutes les réponses ou aux réponses avec un verbe HTTP particulier (GET, POST, etc). Vous spécifiez les routes de la même manière dans les deux cas, bien que la route soit facultative lors de l'appel à app.use().

L'exemple ci-dessous montre comment vous pouvez ajouter la fonction *middleware* en utilisant les deux méthodes, et avec/sans route.

const express = require("express");

const app = express();

// Un exemple de fonction middleware

let a\_middleware\_function = function (req, res, next) {

 // ... effectuer certaines opérations

 next(); // Appelez next() pour qu'Express appelle la fonction middleware suivante dans la chaîne.

};

// Fonction ajoutée avec use() pour toutes les routes et verbes

app.use(a\_middleware\_function);

// Fonction ajoutée avec use() pour une route spécifique

app.use("/uneroute", a\_middleware\_function);

// Une fonction middleware ajoutée pour un verbe et une route HTTP spécifiques

app.get("/", a\_middleware\_function);

app.listen(3000);

**Note :** Ci-dessus, nous déclarons la fonction *middleware* séparément, puis nous la définissons comme fonction de rappel. Dans notre précédente fonction de gestion de route, nous avons déclaré la fonction de rappel lorsqu'elle a été utilisée. En JavaScript, les deux approches sont valables.

La documentation d'Express contient beaucoup d'autres excellents documents sur [l'utilisation](https://expressjs.com/fr/guide/using-middleware.html) et [l'écriture](https://expressjs.com/fr/guide/writing-middleware.html) d'intergiciels Express.

[**Servir les fichiers statiques**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#servir_les_fichiers_statiques)

Vous pouvez utiliser l'intergiciel [express.static](https://expressjs.com/en/4x/api.html%22%20%5Cl%20%22express.static%22%20%5Ct%20%22_blank) pour servir des fichiers statiques, notamment vos images, CSS et JavaScript (static() est la seule fonction de l'intergiciel qui fait réellement **partie** d'*Express*). Par exemple, vous utiliserez la ligne ci-dessous pour servir des images, des fichiers CSS et des fichiers JavaScript à partir d'un répertoire nommé **'public'** au même niveau que celui où vous appelez node :

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

Tous les fichiers du répertoire public sont servis en ajoutant leur nom de fichier (*relatif* au répertoire "public" de base) à l'URL de base. Ainsi, par exemple :

https://localhost:3000/images/dog.jpg

https://localhost:3000/css/style.css

https://localhost:3000/js/app.js

https://localhost:3000/about.html

Vous pouvez appeler static() plusieurs fois pour servir plusieurs répertoires. Si un fichier ne peut pas être trouvé par une fonction middleware, alors il sera simplement transmis au *middleware* suivant (l'ordre dans lequel le *middleware* est appelé est basé sur votre ordre de déclaration).

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

app.use(express.static("media"));

Vous pouvez également créer un préfixe virtuel pour vos URL statiques, plutôt que de voir les fichiers ajoutés à l'URL de base. Par exemple, ici nous [spécifions un chemin de montage](https://expressjs.com/en/4x/api.html#app.use) pour que les fichiers soient chargés avec le préfixe « /media » :

app.use("/media", express.static("public"));

Maintenant, vous pouvez charger les fichiers qui se trouvent dans le répertoire public à partir du préfixe du chemin /media.

https://localhost:3000/media/images/dog.jpg

https://localhost:3000/media/video/cat.mp4

https://localhost:3000/media/cry.mp3

**Note :** Voir également [Servir des fichiers statiques dans Express](https://expressjs.com/fr/starter/static-files.html).

[**Traitement des erreurs**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#traitement_des_erreurs)

Les erreurs sont traitées par une ou plusieurs fonctions spéciales du *middleware* qui ont quatre arguments, au lieu des trois habituels : (err, req, res, next). Par exemple :

app.use(function (err, req, res, next) {

 console.error(err.stack);

 res.status(500).send("Quelque chose s'est cassé !");

});

Ceux-ci peuvent retourner tout contenu nécessaire, mais doivent être appelés après tous les autres app.use() et les appels de routes afin qu'ils soient le dernier *middleware* dans le processus de traitement des requêtes !

Express est livré avec un gestionnaire d'erreurs intégré, qui prend en charge toutes les erreurs restantes qui pourraient être rencontrées dans l'application. Cette fonction *middleware* de gestion des erreurs par défaut est ajoutée à la fin de la pile de fonctions middleware. Si vous passez une erreur à next() et que vous ne la gérez pas dans un gestionnaire d'erreurs, elle sera traitée par le gestionnaire d'erreurs intégré ; l'erreur sera écrite au client avec la trace de la pile.

**Note :** La trace de la pile n'est pas incluse dans l'environnement de production. Pour exécuter une application serveur Express, la variable d'environnement NODE\_ENV doit être définie avec la valeur production.

**Note :** Les codes d'état HTTP 404 et autres « erreurs » ne sont pas traités comme des erreurs. Si vous voulez les gérer, vous pouvez ajouter une fonction *middleware* pour le faire. Pour plus d'informations, consultez la [FAQ](https://expressjs.com/fr/starter/faq.html#how-do-i-handle-404-responses).

Pour plus d'informations, voir [Gestion des erreurs](https://expressjs.com/fr/guide/error-handling.html) (docs Express).

[**Utilisation des bases de données**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#utilisation_des_bases_de_donn%C3%A9es)

Les apps *Express* peuvent utiliser tout mécanisme de base de données pris en charge par *Node* (*Express* lui-même ne définit aucun comportements/exigences supplémentaire spécifique pour la gestion des bases de données). Il existe de nombreuses options, notamment PostgreSQL, MySQL, Redis, SQLite, MongoDB, etc.

Pour les utiliser, vous devez d'abord installer le pilote de base de données à l'aide de NPM. Par exemple, pour installer le pilote de la populaire base de données NoSQL MongoDB, vous devez utiliser la commande suivante :

bashCopy to Clipboard

npm install mongodb

La base de données elle-même peut être installée localement ou sur un serveur en nuage. Dans votre code Express, vous avez besoin du pilote, vous vous connectez à la base de données, puis vous effectuez des opérations de création, lecture, mise à jour et suppression (en anglais, on utilise l'acronyme CRUD qui signifie *Create, Read, Update, Delete*). L'exemple ci-dessous (tiré de la documentation d'Express) montre comment vous pouvez trouver des enregistrements « mammifères » en utilisant MongoDB.

// cela fonctionne avec les anciennes versions de mongodb version ~ 2.2.33

const MongoClient = require("mongodb").MongoClient;

MongoClient.connect("mongodb://localhost:27017/animals", function (err, db) {

 if (err) throw err;

 db.collection("mammals")

 .find()

 .toArray(function (err, result) {

 if (err) throw err;

 console.log(result);

 });

});

// pour mongodb version 3.0 et supérieure

const MongoClient = require("mongodb").MongoClient;

MongoClient.connect(

 "mongodb://localhost:27017/animals",

 function (err, client) {

 if (err) throw err;

 let db = client.db("animals");

 db.collection("mammals")

 .find()

 .toArray(function (err, result) {

 if (err) throw err;

 console.log(result);

 client.close();

 });

 },

);

Une autre approche populaire consiste à accéder à votre base de données de manière indirecte, via un mappeur objet-relationnel (« ORM »). Dans cette approche, vous définissez vos données en tant qu'objets ou modèles et l'ORM les met en correspondance avec le format de base de données sous-jacent. L'avantage de cette approche est qu'en tant que développeur, vous pouvez continuer à penser en termes d'objets JavaScript plutôt qu'en termes de sémantique de base de données, et qu'il existe un endroit évident pour effectuer la validation et la vérification des données entrantes. Nous parlerons davantage des bases de données dans un article ultérieur.

Pour plus d'informations, voir [Intégration de base de données](https://expressjs.com/fr/guide/database-integration.html) (docs Express).

[**Rendu des données (vues)**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#rendu_des_donn%C3%A9es_vues)

Les moteurs de modèles (appelés « moteurs de vue » par *Express*) vous permettent de spécifier la *structure* d'un document de sortie dans un modèle, en utilisant des espaces réservés pour les données qui seront remplies lorsqu'une page sera générée. Les modèles sont souvent utilisés pour créer du HTML, mais peuvent également créer d'autres types de documents. Express prend en charge [un certain nombre de moteurs de modèles](https://github.com/expressjs/express/wiki#template-engines), et il existe une comparaison utile des moteurs les plus populaires ici : [Comparaison des moteurs de création de modèles JavaScript : Jade, Mustache, Dust et plus](https://strongloop.com/strongblog/compare-javascript-templates-jade-mustache-dust/).

Dans le code des paramètres de votre application, vous définissez le moteur de modèles à utiliser et l'emplacement où Express doit rechercher les modèles à l'aide des paramètres « views » et « view engines », comme indiqué ci-dessous (vous devrez également installer le paquet contenant votre bibliothèque de modèles !)

const express = require("express");

const path = require("path");

const app = express();

// Définir le répertoire contenant les modèles ('views')

app.set("views", path.join(\_\_dirname, "views"));

// Définir le moteur d'affichage à utiliser, dans ce cas 'some\_template\_engine\_name'.

app.set("view engine", "some\_template\_engine\_name");

L'apparence du modèle dépendra du moteur que vous utilisez. En supposant que vous ayez un fichier de modèle nommé « index.<template\_extension> » qui contient des espaces réservés pour des variables de données nommées « title » et « message », vous appelleriez [Response.render()](https://expressjs.com/en/4x/api.html%22%20%5Cl%20%22res.render%22%20%5Ct%20%22_blank) dans une fonction de gestionnaire de route pour créer et envoyer la réponse HTML :

app.get("/", function (req, res) {

 res.render("index", {

 title: "À propos des poules",

 message: "Elles sont où ?",

 });

});

Pour plus d'informations, voir [Utilisation des moteurs de modèles avec Express](https://expressjs.com/fr/guide/using-template-engines.html) (docs Express).

[**Structure du fichier**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#structure_du_fichier)

Express ne fait aucune supposition en termes de structure ou de composants que vous utilisez. Les routes, les vues, les fichiers statiques et toute autre logique spécifique à l'application peuvent vivre dans un nombre quelconque de fichiers avec n'importe quelle structure de répertoire. Bien qu'il soit parfaitement possible d'avoir l'ensemble de l'application *Express* dans un seul fichier, il est généralement judicieux de diviser votre application en fichiers basés sur la fonction (par exemple, gestion de compte, blogs, forums de discussion) et le domaine de problème architectural (par exemple, modèle, vue ou contrôleur si vous utilisez une [architecture MVC](https://developer.mozilla.org/fr/docs/Glossary/MVC)).

Dans une prochaine rubrique, nous utiliserons le *Générateur d'applications express*, qui crée un squelette d'application modulaire que nous pouvons facilement étendre pour créer des applications web.

[**Résumé**](https://developer.mozilla.org/fr/docs/Learn/Server-side/Express_Nodejs/Introduction#r%C3%A9sum%C3%A9)

Félicitations, vous avez terminé la première étape de votre voyage Express/Node ! Vous devriez maintenant comprendre les principaux avantages d'Express et de Node, et savoir à quoi ressemblent les principales parties d'une application Express (routes, intergiciels, gestion des erreurs et code modèle). Vous devez également comprendre qu'Express étant un framework non autonome, la manière dont vous assemblez ces éléments et les bibliothèques que vous utilisez dépendent largement de vous !

Bien sûr, Express est délibérément un cadre d'application web très léger, et une grande partie de ses avantages et de son potentiel provient de bibliothèques et de fonctionnalités tierces. Nous les examinerons plus en détail dans les articles suivants. Dans notre prochain article, nous nous pencherons sur la configuration d'un environnement de développement Node, afin que vous puissiez commencer à voir du code Express en action.