

Cette traduction fournie par StrongLoop / IBM.
Il se peut que ce document soit obsolète par rapport à la documentation en anglais. Pour connaître les mises à jour les plus récentes, reportez-vous à la [Documentation en anglais](#).

Utilisation de middleware

Express est une infrastructure web middleware et de routage, qui a des fonctions propres minimales : une application Express n'est ni plus ni moins qu'une succession d'appels de fonctions middleware.

Les fonctions de **middleware** sont des fonctions qui peuvent accéder à l'**objet Request** (`req`), l'**objet response** (`res`) et à la fonction middleware suivant dans le cycle demande-réponse de l'application. La fonction middleware suivant est couramment désignée par une variable nommée `next`.

Les fonctions middleware effectuent les tâches suivantes :

- Exécuter tout type de code.
- Apporter des modifications aux objets de demande et de réponse.
- Terminer le cycle de demande-réponse.
- Appeler la fonction middleware suivant dans la pile.

Si la fonction middleware en cours ne termine pas le cycle de demande-réponse, elle doit appeler la fonction `next()` pour transmettre le contrôle à la fonction middleware suivant. Sinon, la demande restera bloquée.

Une application Express peut utiliser les types de middleware suivants :

- Middleware niveau application
- Middleware niveau routeur
- Middleware de traitement d'erreurs
- Middleware intégré
- Middleware tiers

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

Middleware niveau application

Liez le middleware niveau application à une instance de l'objet `app object` en utilisant les fonctions `app.use()` et `app.METHOD()`, où `METHOD` est la méthode HTTP de la demande que gère la fonction middleware (par exemple `GET`, `PUT` ou `POST`) en minuscules.

Cet exemple illustre une fonction middleware sans chemin de montage. La fonction est exécutée à chaque fois que l'application reçoit une demande.

```
var app = express();

app.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});
```

Cet exemple illustre une fonction middleware montée sur le chemin `/user/:id`. La fonction est exécutée pour tout type de demande HTTP sur le chemin `/user/:id`.

```
app.use('/user/:id', function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});
```

Cet exemple illustre une route et sa fonction de gestionnaire (système de middleware). La fonction gère les demandes `GET` adressées au chemin `/user/:id`.

```
app.get('/user/:id', function (req, res, next) {
  res.send('USER');
});
```

Voici un exemple de chargement d'une série de fonctions middleware sur un point de montage, avec un chemin de montage. Il illustre une sous-pile de middleware qui imprime les infos de demande pour tout type de demande HTTP adressée au chemin `/user/:id`.

```
app.use('/user/:id', function(req, res, next) {
  console.log('Request URL:', req.originalUrl);
  next();
}, function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});
```

Les gestionnaires de routage vous permettent de définir plusieurs routes pour un chemin. L'exemple ci-dessous définit deux routes pour les demandes `GET` adressées au chemin `/user/:id`. La deuxième route ne causera aucun problème, mais ne sera jamais appellée puisque la première route boucle le cycle demande-réponse.

Cet exemple illustre une sous-pile de middleware qui gère les demandes `GET` adressées au chemin `/user/:id`.

```
app.get('/user/:id', function (req, res, next) {
  console.log('ID:', req.params.id);
  next();
}, function (req, res, next) {
  res.send('User Info');
});

// handler for the /user/:id path, which prints the user ID
app.get('/user/:id', function (req, res, next) {
  res.end(req.params.id);
});
```

Pour ignorer les fonctions middleware issues d'une pile de middleware de routeur, appelez `next('route')` pour passer le contrôle à la prochaine route. **REMARQUE** : `next('route')` ne fonctionnera qu'avec les fonctions middleware qui ont été chargées via les fonctions `app.METHOD()` ou `router.METHOD()`.

Cet exemple illustre une sous-pile de middleware qui gère les demandes `GET` adressées au chemin `/user/:id`.

```
app.get('/user/:id', function (req, res, next) {
  // if the user ID is 0, skip to the next route
  if (req.params.id == 0) next('route');
  // otherwise pass the control to the next middleware function in this stack
  else next();
}, function (req, res, next) {
  // render a regular page
  res.render('regular');
});

// handler for the /user/:id path, which renders a special page
```

```
app.get('/user/:id', function (req, res, next) {
  res.render('special');
});
```

Middleware niveau routeur

Le middleware niveau routeur fonctionne de la même manière que le middleware niveau application, à l'exception près qu'il est lié à une instance de `express.Router()`.

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

Chargez le middleware niveau routeur par le biais des fonctions `router.use()` et `router.METHOD()`.

Le code d'exemple suivant réplique le système de middleware illustré ci-dessus pour le middleware niveau application, en utilisant un middleware niveau routeur :

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

// a middleware function with no mount path. This code is executed for every request to the router
router.use(function (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', function(req, res, next) {
  console.log('Request URL:', req.originalUrl);
  next();
}, function (req, res, next) {
  console.log("Request Type:", req.method);
  next();
});

// a middleware sub-stack that handles GET requests to the /user/:id path
router.get('/user/:id', function (req, res, next) {
  // if the user ID is 0, skip to the next router
  if (req.params.id == 0) next('route');
  // otherwise pass control to the next middleware function in this stack
  else next(); //
}, function (req, res, next) {
  // render a regular page
  res.render('regular');
});

// handler for the /user/:id path, which renders a special page
router.get('/user/:id', function (req, res, next) {
  console.log(req.params.id);
  res.render('special');
});

// mount the router on the app
app.use('/', router);
```

Middleware de traitement d'erreurs

Le middleware de traitement d'erreurs comporte toujours **quatre** arguments. Vous devez fournir quatre arguments pour l'identifier comme une fonction middleware de traitement d'erreurs. Même si vous n'avez pas besoin d'utiliser l'objet `next`, vous devez le spécifier pour maintenir la signature. Sinon, l'objet `next` sera interprété comme un middleware ordinaire et n'arrivera pas à gérer les erreurs.

Définissez les fonctions middleware de traitement d'erreurs de la même façon que d'autres fonctions middleware, à l'exception près qu'il faudra 4 arguments au lieu de 3, et plus particulièrement avec la signature (`err, req, res, next`) :

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

Pour obtenir des détails sur le middleware de traitement d'erreurs, reportez-vous à : [Traitement d'erreurs](#).

Middleware intégré

Depuis la version 4.x, Express ne dépend plus de [Connect](#). A l'exception de `express.static`, toutes les fonctions middleware précédemment incluses à Express font désormais partie de modules distincts. Veuillez vous reporter à la [liste des fonctions middleware](#).

`express.static(root, [options])`

La seule fonction middleware intégrée dans Express est `express.static`. Cette fonction est basée sur `serve-static` et a la responsabilité de servir les actifs statiques d'une application Express.

L'argument `root` spécifie le répertoire racine à partir duquel servir les actifs statiques.

L'objet `options` facultatif peut avoir les propriétés suivantes :

Propriété	Description	Type	Valeur par défaut
<code>dotfiles</code>	Option pour servir les fichiers dotfiles. Les valeurs possibles sont "allow", "deny" et "ignore"	Chaîne	"ignore"
<code>etag</code>	Activer ou désactiver la génération etag	Booléen	true
<code>extensions</code>	Définit l'extension de fichier de recharge.	Tableau	[]
<code>index</code>	Envole le fichier d'index du répertoire. Utilisez <code>false</code> pour désactiver l'indexation de répertoire.	Mix	"index.html"
<code>lastModified</code>	Définit l'en-tête <code>Last-Modified</code> sur la date de dernière modification du fichier dans le système d'exploitation. Les valeurs possibles sont <code>true</code> ou <code>false</code> .	Booléen	true
<code>maxAge</code>	Définit la propriété max-age de l'en-tête Cache-Control, en millisecondes ou par une chaîne au format ms format	Numérique	0
<code>redirect</code>	Réapplique les barres obliques "/" lorsque le chemin d'accès est un répertoire.	Booléen	true
<code>setHeaders</code>	Fonction pour définir les en-têtes HTTP à servir avec le fichier.	Fonction	

Voici un exemple d'utilisation de la fonction middleware `express.static` avec un objet options élaboré :

```
var options = {
  dotfiles: 'ignore',
  etag: false,
  extensions: ['htm', 'html'],
  index: false,
  maxAge: '1d',
  redirect: false,
  setHeaders: function (res, path, stat) {
    res.set('x-timestamp', Date.now());
  }
};

app.use(express.static('public', options));
```

Vous pouvez avoir plusieurs répertoires statiques par application :

```
app.use(express.static('public'));
app.use(express.static('uploads'));
app.use(express.static('files'));
```

Pour obtenir plus de détails sur la fonction `serve-static` et ses options, reportez-vous à la documentation [serve-static](#).

Middleware tiers

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

Installez le module Node.js pour la fonctionnalité requise, puis chargez-le dans votre application au niveau application ou au niveau router.

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

```
$ npm install cookie-parser
```

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

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

Pour obtenir une liste non exhaustive des fonctions middleware tiers utilisées couramment avec Express, reportez-vous à :

[Middleware tiers](#).

Star | 58,580 Express est un projet de la [Fondation OpenJS](#). Consultez le site Web GitHub. Copyright © StrongLoop, Inc., et autres contributeurs expressjs.com.



Ce(tte) oeuvre est mise à disposition selon les termes de la [Licence Creative Commons Attribution - Partage dans les Mêmes Conditions 3.0 États-Unis](#).