Traitement d'erreurs

Définissez les fonctions middleware de traitement d'erreurs de la même manière que les autres fonctions middleware, à l'exception près que les fonctions de traitement d'erreurs se composent de quatre arguments et non de trois : `(err, req, res, next)`. Par exemple :

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

Définissez le middleware de traitement d'erreurs en dernier, après les autres appels `app.use()` et de routes ; par exemple :

```
var bodyParser = require('body-parser');
var methodOverride = require('method-override');

app.use(bodyParser());
app.use(methodOverride());
app.use(function(err, req, res, next) {
  // logic
});
```

Les réponses issues d'une fonction middleware peuvent être au format de votre choix, par exemple une page d'erreur HTML, un simple message ou une chaîne JSON.

A des fins organisationnelles (et d'infrastructure de niveau supérieur), vous pouvez définir plusieurs fonctions middleware de traitement d'erreurs, tout comme vous le feriez avec d'autres fonctions middleware ordinaires. Par exemple, si vous vouliez définir un gestionnaire d'erreurs pour les demandes réalisées avec `XHR` et pour celles réalisées sans, vous pourriez utiliser les commandes suivantes :

```
var bodyParser = require('body-parser');
var methodOverride = require('method-override');

app.use(bodyParser());
app.use(methodOverride());
app.use(logErrors);
app.use(clientErrorHandler);
app.use(errorHandler);
```

Dans cet exemple, les erreurs `logErrors` génériques pourraient écrire des informations de demande et d'erreur dans `stderr`, par exemple :

```
function logErrors(err, req, res, next) {
  console.error(err.stack);
  next(err);
}
```

Egalement dans cet exemple, `clientErrorHandler` est défini comme suit ; dans ce cas, l'erreur est explicitement transmise à la fonction suivante :

```
function clientErrorHandler(err, req, res, next) {
  if (req.xhr) {
    res.status(500).send({ error: 'Something failed!' });
  } else {
    next(err);
  }
}
```

La fonction "catch-all" `errorHandler` peut être mise en oeuvre comme suit :

```
function errorHandler(err, req, res, next) {
  res.status(500);
  res.render('error', { error: err });
}
```

Si vous transmettez tout à la fonction `next()` (sauf la chaîne `'route'`), Express considère la demande en cours comme étant erronée et ignorera tout routage de gestion non lié à une erreur et toute fonction middleware restants. Si vous voulez gérer cette erreur de quelque façon que ce soit, vous devrez créer une route de gestion d'erreur tel que décrit dans la section suivante.

Si vous disposez d'un gestionnaire de routage avec plusieurs fonctions callback, vour pouvez utiliser le paramètre `route` pour passer au gestionnaire de routage suivant. Par exemple :

```
app.get('/a_route_behind_paywall',
  function checkIfPaidSubscriber(req, res, next) {
    if(!req.user.hasPaid) {

      // continue handling this request
      next('route');
    }
  }, function getPaidContent(req, res, next) {
    PaidContent.find(function(err, doc) {
      if(err) return next(err);
      res.json(doc);
    });
  });
```

Dans cet exemple, le gestionnaire `getPaidContent` sera ignoré, mais tous les gestionnaires restants dans `app` pour `/a_route_behind_paywall` continueront d'être exécutés.

Les appels `next()` et `next(err)` indiquent que le gestionnaire en cours a fini et quel est son état. `next(err)` ignorera tous les gestionnaires restants dans la chaîne, sauf ceux définis pour gérer les erreurs tel que décrit ci-dessus.

Le gestionnaire de traitement d'erreurs par défaut

Express propose un gestionnaire d'erreurs intégré, qui traite toutes les erreurs qui pourraient survenir dans l'application. Cette fonction middleware de traitement d'erreurs par défaut est ajoutée à la fin de la pile de fonctions middleware.

Si vous transmettez l'erreur à `next()` et que vous ne voulez pas la gérer dans un gestionnaire d'erreurs, elle sera gérée par le gestionnaire d'erreurs intégré ; l'erreur sera alors écrite dans le client avec la trace de pile. La trace de pile n'est pas incluse dans l'environnement de production.

Définissez la variable d'environnement `NODE_ENV` sur `production` afin d'exécuter l'application en mode production.

Si vous appelez `next()` avec une erreur après avoir démarré l'écriture de la réponse (par exemple, si vous rencontrez une erreur lors de la diffusion en flux de la réponse au client) le gestionnaire de traitement d'erreurs par défaut Express ferme la connexion et met la demande en échec.

De ce fait, lorsque vous ajoutez un gestionnaire d'erreurs personnalisé, vous devriez déléguer les mécanismes de gestion d'erreur par défaut à Express, lorsque les en-têtes ont déjà été envoyés au client :

```
function errorHandler(err, req, res, next) {
  if (res.headersSent) {
    return next(err);
  }
  res.status(500);
  res.render('error', { error: err });
}
```