Introduction et fonctionnement d'Express

Qu'est-ce qu'`Express` ?

`Express` est un framework pour `Node.js`.

Il s'agit du framework `Node.js` le plus utilisé au monde, et ce de très loin. Il s'agit également du premier framework car il est sorti en novembre 2010.

Il est utilisé en production par des entreprises comme **IBM**, **accenture**, **yandex** ou **Uber**.

Il offre plusieurs fonctionnalités intéressantes pour créer une `API` de manière rapide et performante.

Il permet d'utiliser un **router** très facilement en écrivant des fonctions appelées `handlers` et en les liant à différentes méthodes et URL (constituant une route).

Il permet de **servir des ressources statiques** en s'intégrant très facilement avec les `template engines` comme `pug` ou `ejs` que nous étudierons.

Il permet d'ajouter des fonctions effectuant des opérations supplémentaires grâce aux `middleware`. Il existe beaucoup de librairies prévues pour être utilisées comme `middleware express`. Un nombre conséquent de ces `middleware` sont maintenus par l'équipe d'`express`.

Le fonctionnement d'`Express`

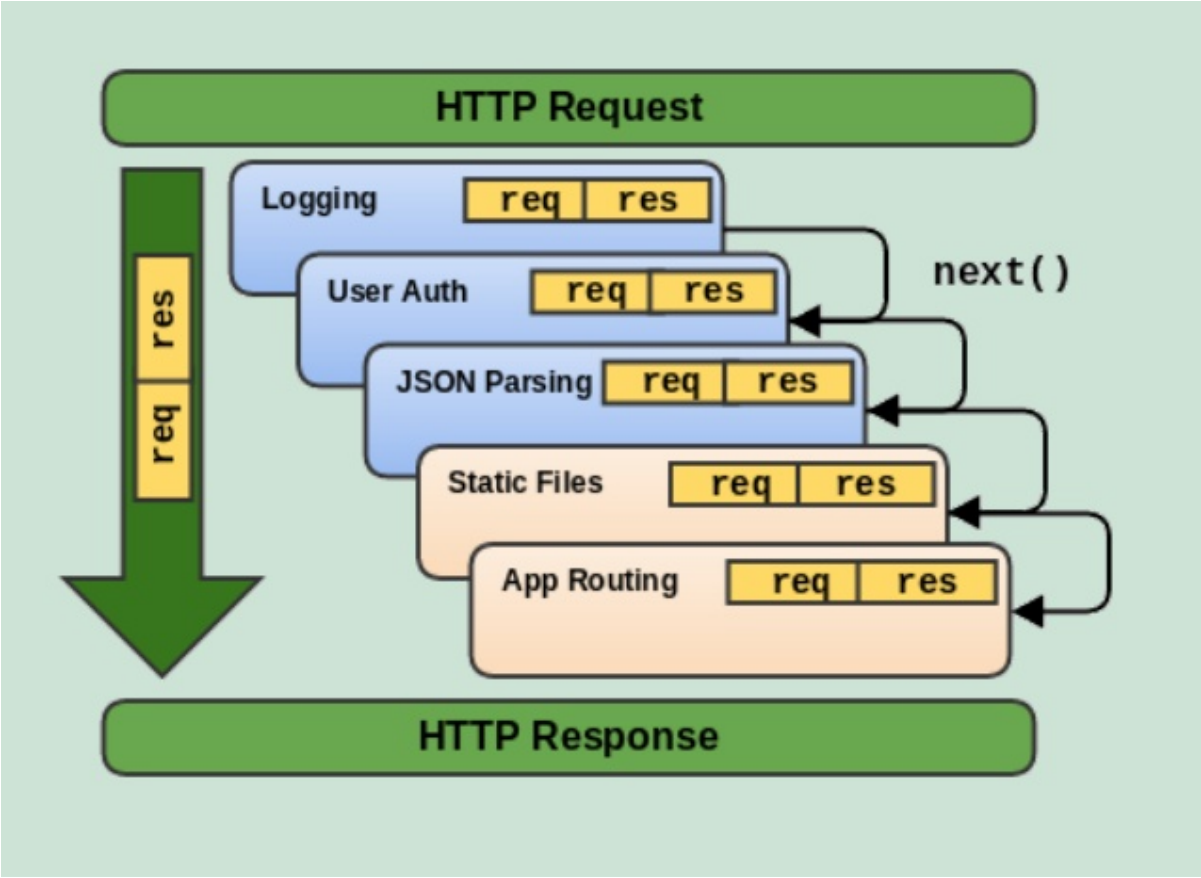
`Express` utilise un modèle de piles, appelées `stack` en anglais.

Il utilise une pile de `layers` appelés `middleware` et des fonctions de `routing` qui font référence aux fonctions spécifiques qui répondent à la requête client.

Les `middleware`

Un `middleware` est une fonction qui a accès aux objets `request`, `response` et `next` qui est le `middleware` suivant dans la pile des `middleware`.

Le modèle de la `stack` de `middleware` permet de définir une pile d'actions à effectuer pour chaque requête :



Un `middleware` peut effectuer toute opération, exécuter du code et effectuer des changements sur les objets `req` et `res`.

Il peut soit passer au `middleware` suivant dans la pile en utilisant `next()` (ce qu'il fera le plus souvent), ou il peut terminer le cycle en envoyant une réponse.

L'utilisation de `app.use()`

La méthode **`app.use()` permet de définir un (ou plusieurs) `middleware` qui sera appelé pour toutes les requêtes si aucune URL ne lui est passé** :

```
const express = require('express')
const app = express()
const monLogger = (req, res, next) => {
  console.log(req.url)
  next()
};
app.use(monLogger)
```

Ne vous focalisez pas trop sur le code, nous reverrons tout en détails.

Retenez que `app.use(middleware)` permet d'utiliser une fonction ayant accès à `(req, res, next)` pour **toutes les requêtes**.

L'utilisation de `app.use(url)`

Lorsque vous passez en premier argument une URL, le `middleware` ne s'activera que pour les routes qui matcheront cette URL.

Par exemple :

```
const express = require('express')
const app = express();
const monAPILogger = (req, res, next) => {
  console.log(req.url)
  next()
};
app.use('/api', monAPILogger)
```

Ce `middleware` ne s'activera que pour les routes commençant par `/api` .

Utilisation de `middleware` externes

Nous le reverrons, mais nous pouvons déjà vous montrer comment utiliser des `middleware` externes :

```
...
const bodyParser = require('body-parser');
app.use(bodyParser.json())
const monAPILogger = (req, res, next) => {
  console.log(req.url)
  next()
};
app.use('/api', monAPILogger)
```

Toutes les requêtes seront parsées par la librairie externe `body-parser` qui permet de ne pas devoir concaténer toutes les `chunk` du `body` comme nous l'avions vu avec `Node.js` sans framework.

Elles passeront ensuite en deuxième lieu par notre fonction `middleware` que nous avions créée pour afficher la méthode utilisée par la requête, uniquement sur les routes commençant par `/api` .

Les fonctions de `routing`

L'unique différence entre un `middleware` et une fonction de `routing`, également appelé `route handler`, et que cette dernière a vocation à répondre à l'agent utilisateur pour une route spécifique : c'est-à-dire une méthode donnée et une URL donnée.

Utilisation de `app.METHOD()`

Cette méthode permet de router une requête :

```
app.METHOD(path, callback)
```

La méthode peut être `get()` ou `post` ou `put()` par exemple :

```
app.put(...)
```

Le `path` peut être une URL ou une regex qui va matcher l'URL de la requête `Http`.

Le `callback` est une fonction `middleware` ou un tableau de fonctions `middleware`, ou plusieurs `middleware` à la suite séparés par des virgules.

Utilisation de `app.route().METHOD()`

Une méthode permet de faciliter l'écriture de routes avec `Express`.

Il s'agit de `app.route()`.

Cette méthode permet de chaîner plusieurs méthodes app.METHOD() pour la même route :

```
app.route('/exemple')
.get( (req, res, next) => {
  ...
})
.post( (req, res, next) => {
  ...
});
```

Ne vous en faites pas nous reverrons tout en détails. Mais cela vous permet de vous faire une idée sur comment fonctionne `Express`.

Ce qu'il y a à retenir est le **modèle en pile ou `stack`** qui permet de facilement chaîner de nombreuses fonctions et de router les requêtes `Http` entrantes.