

Express



Express est un projet de serveur web Node.js :

- *Un routage + une couche de “sucre” sur le serveur HTTP de Node.js*
- *Un routage déclaratif*
- Modèle de base pour les middlewares

Routes avec express



- Mécanisme principale pour le développement du serveur
 - **Route** : Association d'un *handler* à un certain type de requête
 - **app.method(url, handler)**
 - Méthode : Permet de définir la méthode HTTP de la requête (ex: get, post, put, delete, etc...)
 - Handler : la fonction appelée lors de la réception de la requête conforme à la route. Elle possède toujours 2 arguments, typiquement appelés **req** et **res** :
 - **req** : la requête HTTP
 - **res** : la réponse à envoyer
- L'ordre de déclaration des routes est **important**. Toujours mettre le chemin racine en dernier.

Routes avec express



```

● ● ●

// Import du module express
const express = require('express');
// Création d'une objet express (le serveur)
const app = express();

// Répond avec "Hello World!" quand on reçoit une requête GET
app.get('/', (req, res) => {
  res.send('Hello World!');
});

const port = 5000;
// "Mise en ligne" du serveur
app.listen(port, () => {
  console.log(`Server is listening on port ${port}...`);
});
```

Quelques propriétés utiles de la requête HTTP



- `req.query` contient les paramètres de la requête (utile pour le traitement des requêtes GET)

Exemple :

// GET

`http://localhost:5000/?firstname=john&lastname=doe`

`req.query.firstname` // => "john"

`req.query.lastname` // => "doe"

Quelques propriétés utiles de la requête HTTP



- `req.body` contient les paramètres de la requête POST disponible uniquement si on a ajouté un *middleware d'express au serveur* :

```
● ● ●  
  
// Analyse les données envoyées par le formulaire HTML  
app.use(express.urlencoded({ extended: false }));  
  
// Analyse les données envoyées par le JS  
app.use(express.json());
```

Quelques propriétés utiles de la requête HTTP



Exemples :

```
// POST username=john&email=john@gmail.com
```

```
req.body.username // => "john"
```

```
req.body.email // => "john@gmail.com"
```

```
// POST { username= "john", email= "john@gmail.com" }
```

```
req.body.username // => "john"
```

Quelques propriétés utiles de la requête HTTP



- Une route peut être associée à un url contenant une partie variable
- La valeur “name” est disponible dans la variable `req.params.name` :

```
app.get('/user/:name', (req, res) => {  
  res.send(req.params.name);  
});
```

```
// GET /user/john
```

```
req.params.name // => “john”
```

Envoyer une réponse HTTP



- `res.send(data)` envoie la réponse HTTP avec pour contenu data
 - data peut être une chaîne de caractères, un objet, un tableau ou du contenu HTML
- `res.end()` termine la réponse HTTP sans envoyer de données
- `res.status(status-code)` modifie le code HTTP de la réponse
- `res.json()` Envoie une réponse JSON
- Et d'autres méthodes (cf. <http://expressjs.com/en/4x/api.html>)

Remarque : comme `end()`, `json()`, `send()` (et d'autres) envoient une réponse HTTP et terminent le cycle requête-réponse HTTP (pas besoin de `end()` explicite ensuite)

Utiliser un middleware



- Les fonctions qui ont comme argument *next* ou utilisées comme *handlers* pour les routes sont aussi appelées *middleware*
- On peut également en associer plusieurs dans la même route :

```
app.get('/', function1, function2, ...);
```

- On peut associer du *middleware* à toutes les requêtes (i.e. toute méthode, tout url) :

```
app.use(function3, function4, ...);
```

- Ou à toutes les requêtes vers un certain url (toute méthode)

```
app.use('/', function4, function5, ...);
```

Utiliser un middleware



- Chaque fonction de middleware dispose d'un argument en plus (en plus de req et res) : une référence à la prochaine fonction dans la pile d'exécution

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

- Si une fonction de *middleware* est exécutée et n'appelle pas la suivante avec next(), aucune des fonctions suivantes dans la pile ne sera exécutée

Utiliser un middleware



Il est également possible de définir des Middleware qui seront exécutés au début de chaque nouvelle requête entrante.

Il suffit simplement d'utiliser la fonction **use()** de l'objet **app**.

Plus de détails : <https://expressjs.com/fr/guide/using-middleware.html>

Rattacher des fichiers statiques



- Ces fichiers (html, js, jpg, ...) doivent se trouver dans un répertoire rendu accessible au serveur Express typiquement appelée *public*
- Pour rendre le repertoire accessible par le serveur express :

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

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

const port = 5000;
app.listen(port, () => {
  console.log(`Server is listening on port ${port}...`);
});
```

Rattacher des fichiers statiques



- Ensuite tout fichier dans le dossier *public* sera associé à l'url :
<http://localhost:5000/fichier>

Connexion à une base de données



- Un serveur express peut se connecter à une base de données et la manipuler
- La connexion à la base est gérée entièrement par un autre module Node.js, indépendant d'Express
- *npm* offre un module différent pour chaque SGBD majeur
- Pour PostgreSQL, installer : [npm install pg](#)

Connexion à une base de données



- Installer la librairie dotenv : `npm install dotenv`

```
// Utilisation du module dotenv
require('dotenv').config();
const express = require('express');
const app = express();

const port = 5000;
app.listen(port, () => {
  console.log(`Server is listening on port ${port}...`);
});
```

Connexion à une base de données



- Créer un fichier `.env` pour y mettre la informations de connexion :

A screenshot of a terminal window with a light gray background and a dark gray border. The window has three colored window control buttons (red, yellow, green) in the top-left corner. Inside the window, the following text is displayed in blue:

```
PGUSER=postgres  
PGHOST=localhost  
PGPASSWORD=secretpassword  
PGDATABASE=mydb  
PGPORT=5432
```

```
PGUSER=postgres  
PGHOST=localhost  
PGPASSWORD=secretpassword  
PGDATABASE=mydb  
PGPORT=5432
```

Ne pas oublier d'inclure le fichier `.env` dans le `.gitignore` !

Utiliser le module pg



- Documentation : <https://node-postgres.com/>



```
// Inclusion du module
const { Pool } = require('pg');

// Création d'un objet connexion
// pool utilisera des variables d'environnement
// pour les informations de connexion
const pool = new Pool();

// Export de la méthode query pour réaliser des requêtes
module.exports = {
  query: (text, params) => pool.query(text, params)
};
```

Utiliser le module pg



- Exécuter une requête :

```
require('dotenv').config();
const express = require('express');
const app = express();
const db = require('./db/index.js');

app.get('/', async (req, res) => {
  try {
    const res = await db.query('SELECT NOW()');
    console.log(res.rows[0].now);
  } catch (error) {
    console.log(error);
  }

  res.end();
});

const port = 5000;
app.listen(port, () => {
  console.log(`Server is listening on port ${port}...`);
});
```

Utiliser le module pg



- La méthode `query` reçoit 2 paramètres:
 - `text` : La requête SQL
 - `params`: Le tableau des paramètres de la requête pour éviter la concaténation de string dans le texte de la requête directement
- Plus de détails : <https://node-postgres.com/features/queries>

Créer un module Node.js



Pour rendre le code plus modulaire on peut créer des modules (un module = un fichier)

- Par défaut, tout est privé dans un module (variable, fonctions..). Pour qu'une variable ou fonction ne soit pas privée, il faudra clairement le spécifier.
- Chaque fichier JS possède un objet qui lui est propre nommé **module**
- Lorsqu'on fait appel à la fonction **require()**, c'est l'attribut **exports** de l'objet **module** du fichier JS qu'on import qui sera retourné. Il est vide par défaut.
- Si on ne spécifie pas le chemin, la fonction **require()** ira chercher le module dans le dossier **node_modules**

Créer un module Node.js



- Il suffit d'exporter des fonctions ou objets :



```
module.exports = (a, b) => a + b;
```

Créer un module Node.js



- Ensuite importer monModule.js, comme n'importe quel autre module, pour utiliser la fonction exportée :



```
const sum = require('./monModule.js');  
console.log(sum(2, 3)); // => 5
```

Créer un module Node.js



- On peut également exporter un objet :

```
module.exports = {  
  firstname: 'john',  
  lastname: 'doe',  
  fullname() {  
    return `${this.firstname} ${this.lastname}`;  
  }  
};
```

Créer un module Node.js



- Pour importer l'objet :



```
const john = require('./monModule.js');  
console.log(john.fullname()); // => "john doe"
```


Créer un module Node.js



- On peut remplacer *monModule.js* avec un dossier *monModule* contenant un fichier *index.js*

Créer un router avec Express



- Express a introduit la classe *Router* qui permet de :
 - Encapsuler un ensemble de routes “relatives”
 - Les exporter dans leur ensemble
 - Les attacher au serveur Express en le montant sur un chemin racine
- (cf. prochaine slide)

Créer un router avec Express



- Dans monRouter.js :

```
const express = require('express');  
const router = express.Router();  
  
// On attache des routes à router de la même façon  
// qu'à un serveur express()  
router.get('/', function1).post(function2);  
  
module.exports = router;
```

Créer un router avec Express



- Dans app.js :

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

// router
const mainRouter = require('./routes/monRouter.js');

// Les routes GET / et POST / seront gérées par
// mainRouter
app.use('/api/v1', mainRouter);

const port = 5000;
app.listen(port, () => {
  console.log(`Server is listening on port ${port}...`);
});
```

Authentication avec JSON Web Token



C'est un standard définissant une méthode légère et sécurisée pour transmettre une information à travers un objet JSON.

L'information étant transmise, on pourra aisément vérifier sa validité.

`header.payload.signature`

Plus de détails : <https://jwt.io/introduction>

Documentation de la librairie **jsonwebtoken** :

<https://www.npmjs.com/package/jsonwebtoken>